# JVM调优理论知识

这是块窗户纸，捅破就很简单了-------马士兵

# 1 GC的基础知识

## 1.1 什么是垃圾

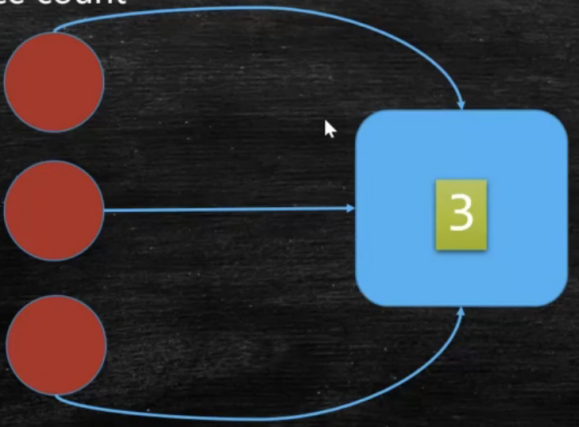
#### 概念：

没有任何引用指向的【一个对象】 或者 【多个对象】（这多个对象之间是循环引用）

|  |  |  |  |
| --- | --- | --- | --- |
| 语言 | 申请内存 | 释放内存 | 优缺点 |
| C | malloc | free | 忘记回收：这叫内存泄露，泄露多了内存溢出  多次回收：非法访问  开发效率低，执行效率高 |
| C++ | new | delete |
| Java | new | 自动回收 | 编程简单系统不容易出错  GC处理垃圾，开发效率高，执行效率偏低 |

## 1.2如何定位垃圾

#### 引用计数 （reference count）

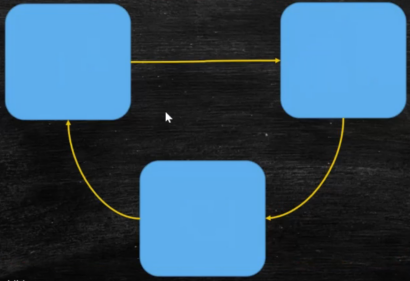


有几个引用指向一个对象，在它对象头上记个数。当计数变成0，就是垃圾

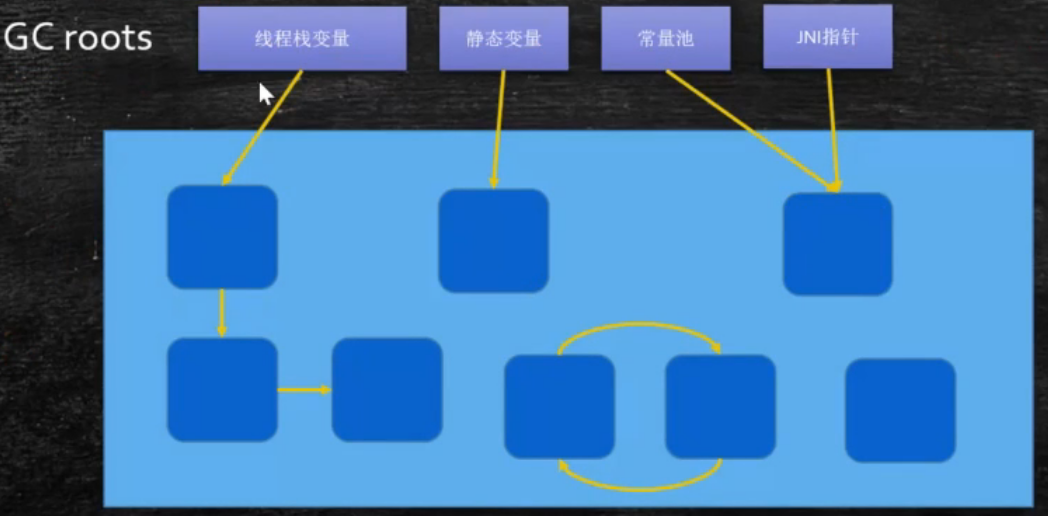
**存在问题：**

不能解决循环引用：三个对象相互引用，A-->B-->C--A ，但是没有对象指向它们三个，它们就是一堆垃圾。

如果用引用计数法，这块垃圾就找不到了，容易发生内存泄露



#### 根可达算法（Root Searching）



从根上对象开始顺着一根线往外捋，找到那些是有用对象，找不着的都是垃圾。

**========== 面试题，想进大厂就背过 ==============**

##### 2.1 根对象：（GC Roots）

程序启动后马上要用到的那些对象

###### 线程栈变量：

Main方法会启动main线程，main线程会有线程栈，里面会有的栈帧，从main栈帧里开始的对象，都是根对象，这些叫线程栈变量。（JVM学习笔记里有栈帧的图）

###### 静态变量：

一个T.class，T.class load到内存后马上要对静态变量初始化，所以静态变量能访问到的对象叫根对象

###### 常量池：

一个class要用到其他class的那些对象，这些是根对象

JNI：调用C、C++本地方法要用到的类对象

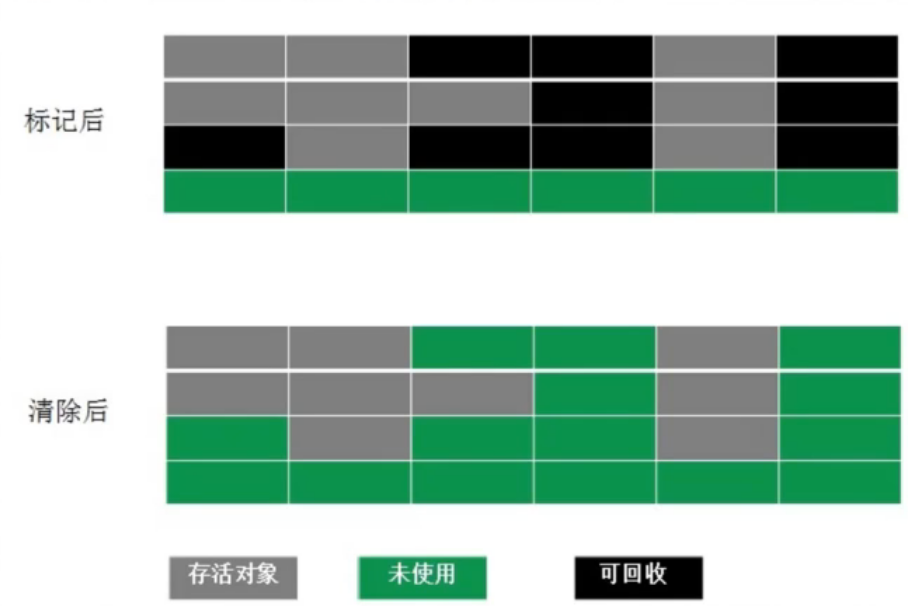
## 1.3 常见的垃圾回收算法 【背过】

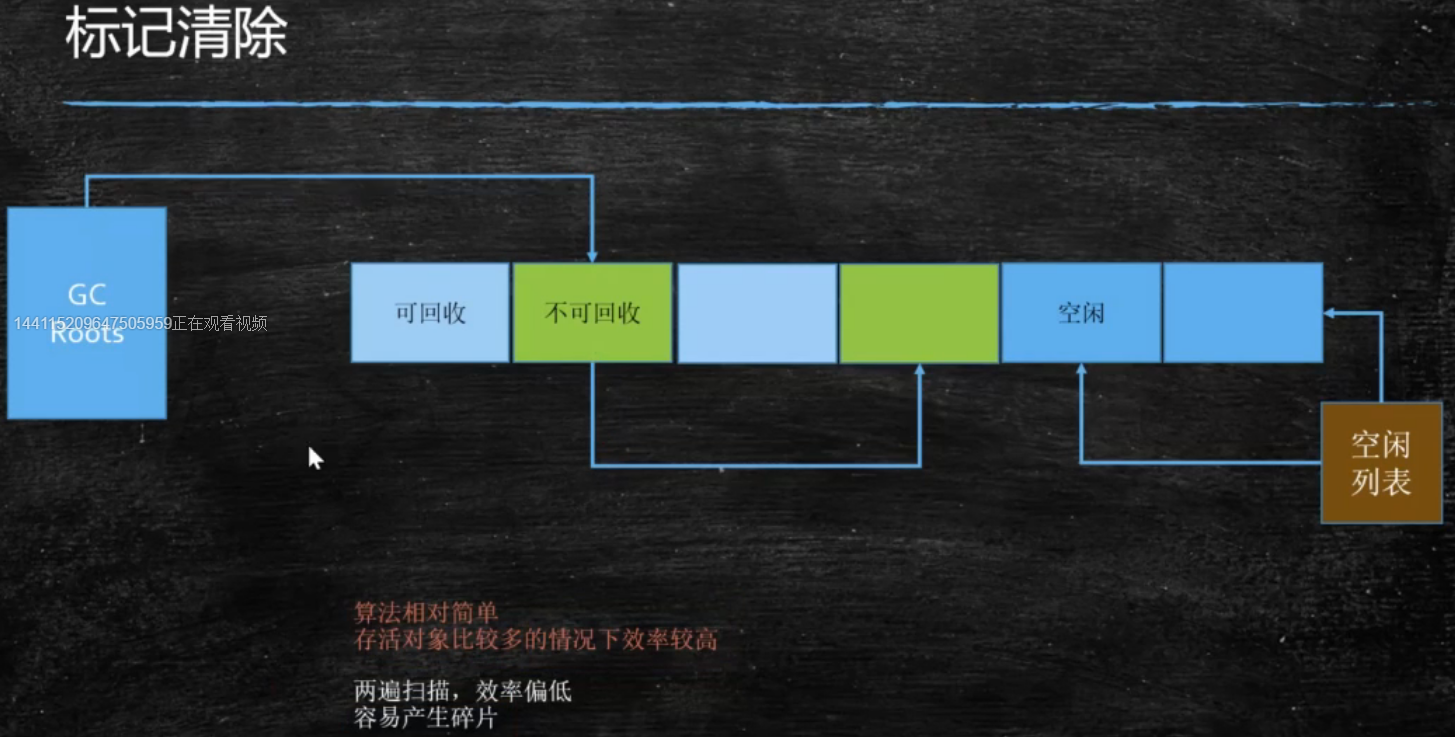
#### 1.标记清除法（mark sweep）：

没用的标记，然后清除掉（位置不用动）

位置不连续 产生碎片 效率偏低（两遍扫描，先找出有用的，再找出没用的清理）

适用于【存活对象比较多】的情况，效率高



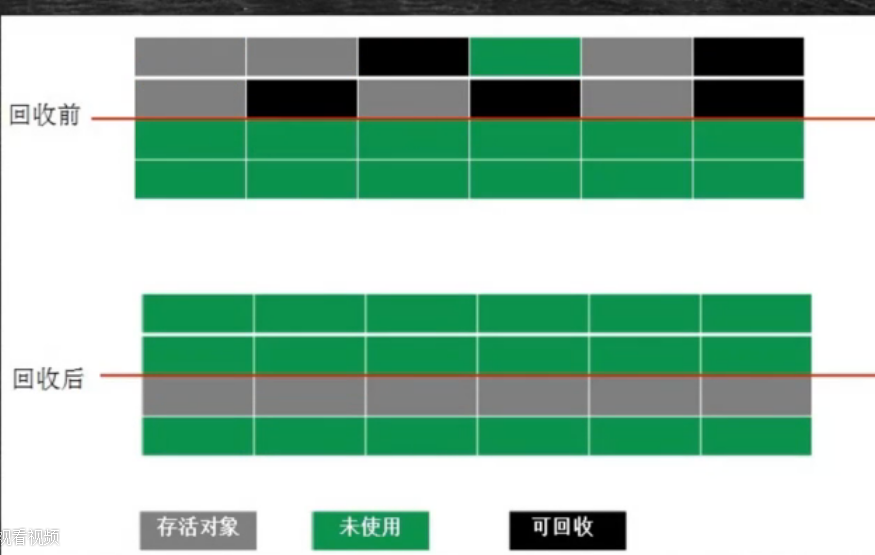


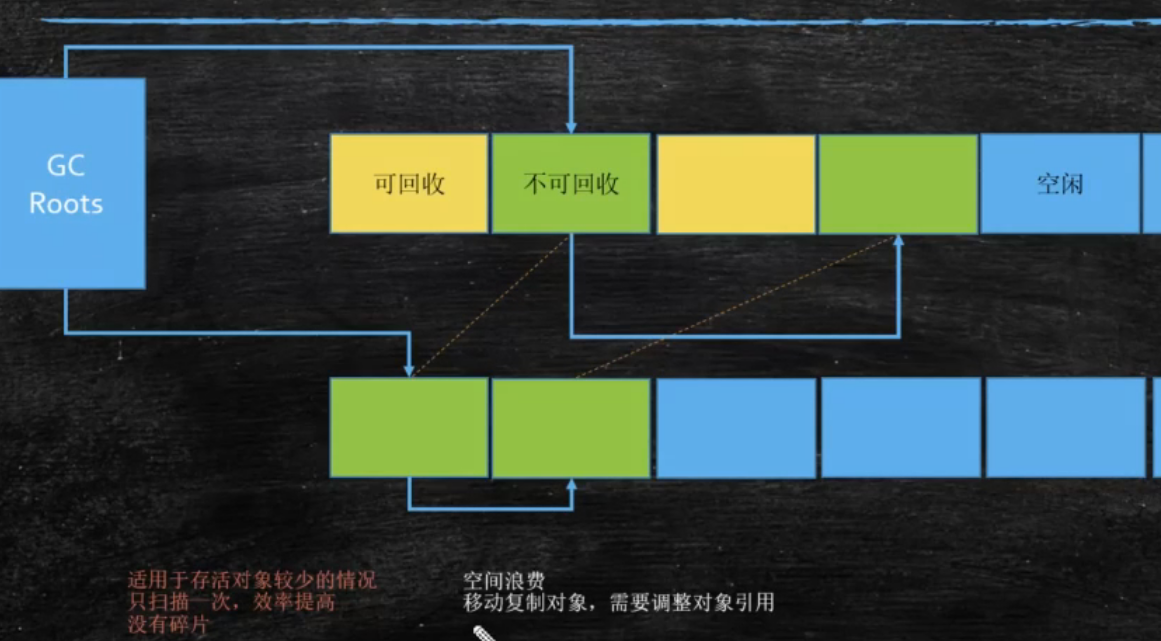
#### 2.拷贝算法（copying）：

将内存一分为二，有用的copy到一半，没用的在另一半，进行清除。

需要移动复制对象，引用需要调整。Copy其实很快

适用于【存活对象较少】的情况，只扫描一次，没有碎片 ，浪费空间。



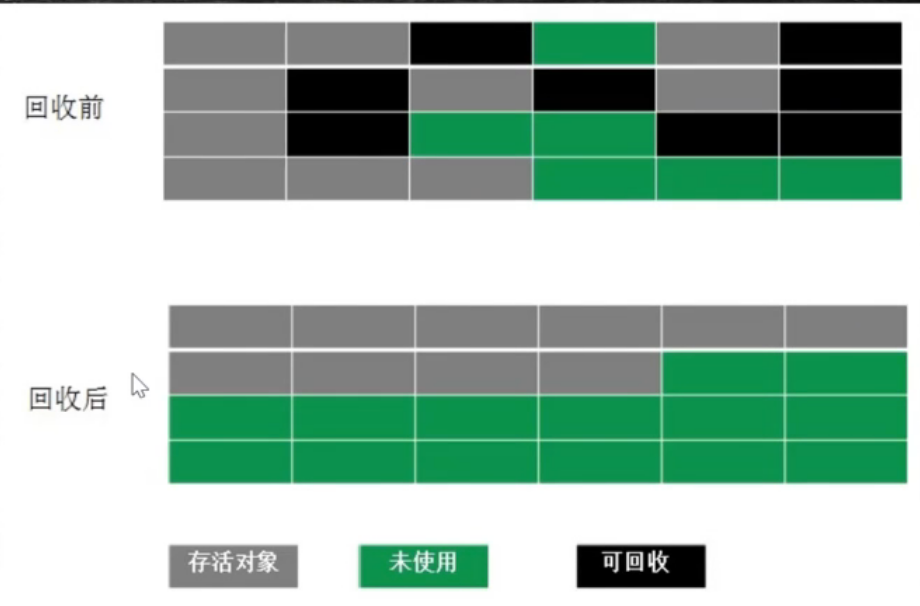


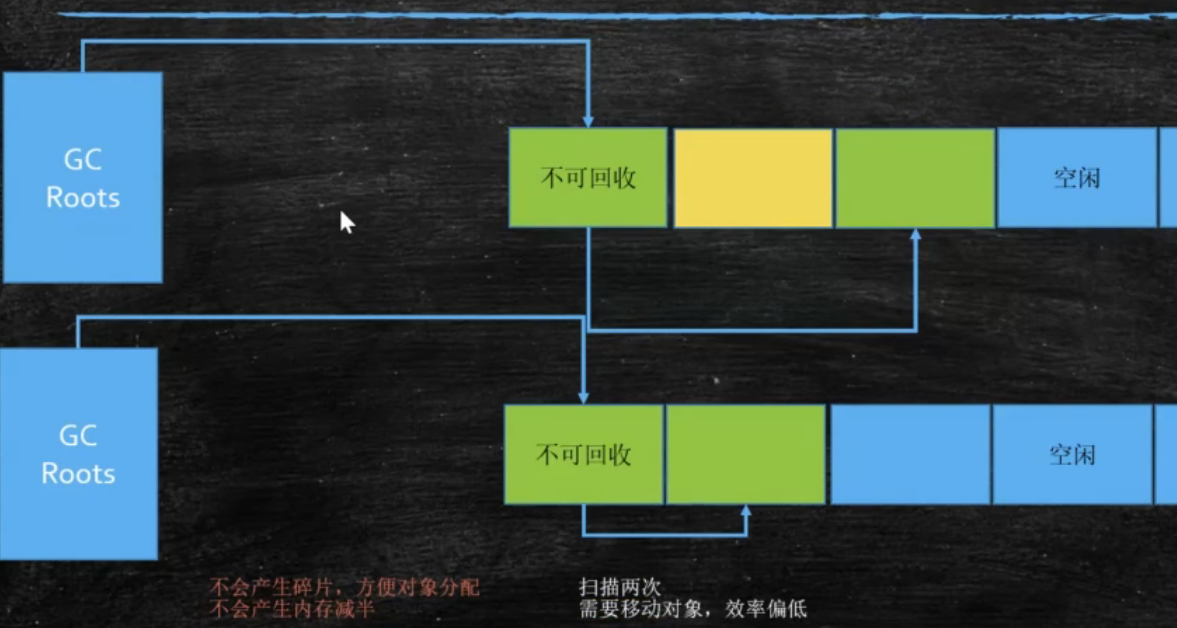
#### 3.标记压缩（mark compact）：

有用的，都往前面走，剩下的空间就都清理出来了。

没有碎片

效率偏低（两遍扫描【找不可回收的、移动】，指针需要调整）任何内存移动的时候，如果是多线程，都需要线程同步，如果是单线程，效率低

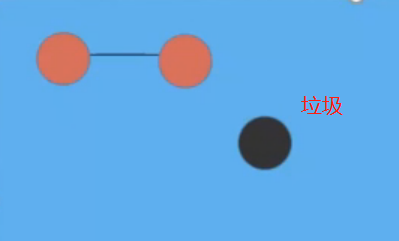




#### 垃圾回收算法形象例子

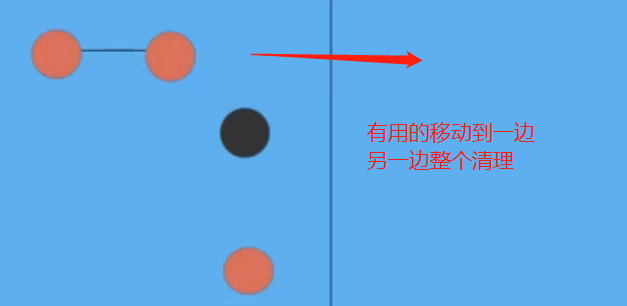
##### 标记清除：

在一房间扯线团，那些被剪断的，孤立的线团就是垃圾，把它们清除了。容易产生碎片



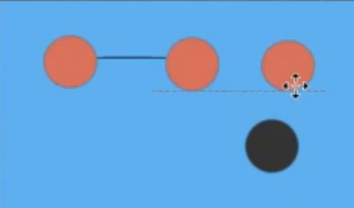
##### 拷贝算法

房间分两半，左边垃圾太多玩不开了，把有用的都copy到右边，左边的全部清除。清理效率高



##### 标记压缩

有用的都靠在一起，没用的清除。找垃圾，还要整理空间压缩，效率低。剩下的空间连续



## 1.4 JVM内存分代模型（用于分代垃圾回收算法，以JDK1.8为准）

### 1.4.1【部分】垃圾回收器使用的模型

-----------解释：一些最新的垃圾回收，如Epsilon （JDK11的垃圾回收器）、ZGC（JDK11）、 Shenandoah（JDK12）不再区分年轻代和老年代了，除此之外的GC都是使用逻辑分代模型。所以这里说讲解部分垃圾回收器模型

G1是逻辑分代，物理不分代

除此之外不仅逻辑分代，而且物理分代

### 1.4.2堆内存逻辑分代（不适用不分代垃圾收集器）

新生代 + 老年代 + 永久代(JDK1.7)Perm Generation / 元数据区(JDK1.8 )Metaspace

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 逻辑概念 | 具体实现 | 版本 | 大小限制 | 装什么 |
| Method Area  方法区 | 永久代  Perm Generation | JDK1.7 | 必须指定大小限制，这个限制会成为系统的限制。比如说你指定了500M，当你是用了动态代理（Spring），会动态创建很多Class文件，会发生永久代的内存溢出 | 装一个个的Class对象的信息，代码编译完的信息，字节码，，，（有可能溢出） |
| 元数据区  Metaspace | JDK1.8 | 可以设置  也可以不设置，无上限（受限于物理内存） |

**字符串常量：**

1.7 --永久代 1.8--堆里

新生代|老年代：堆内存

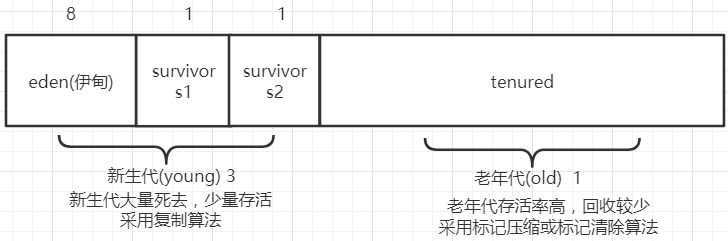
方法区：堆之外的空间

|  |
| --- |
| 随着JDK8的到来，JVM不再有PermGen。但类的元数据信息（metadata）还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的**本地内存**（操作系统内存）中 |

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类的实例对象

堆被划分成两个不同的区域：新生代 ( Young )、老年代 ( Old )。新生代 ( Young ) 又被划分为三个区域：Eden、From |S1 、To|S2。

这样划分的目的是为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。



PS：JDK 1.7还有个永久代PermGen

|  |  |  |
| --- | --- | --- |
| 新生代 | 垃圾回收使用复制算法 | 效率高 |
| 老年代 | 垃圾少，一般使用标记压缩，g1使用copy |  |

堆空间 = 新生代(1/3) + 老年代 (2/3)

新生代 = Eden(8/10) + From(1/10) + To(1/10)

PS：Eden区两个survivor区，称呼 S1/S2 | S0/S1 | From/To 都是一个意思

**新生代：= eden + 2个survivor区**

|  |
| --- |
| 举例：  一个for循环里头，每次都要new一个对象，就会有一堆的对象，出了for循环就没有人再引用他们了，经过一次YGC后这些对象全是垃圾，全被清掉。  新生代里产生的对象，多数情况下会被回收，80~90%的对象在进行垃圾回收时会被回收掉。这也是为什么eden区：S1：S2 = 8:1:1 |

### 1.4.3一个对象从出生到消亡

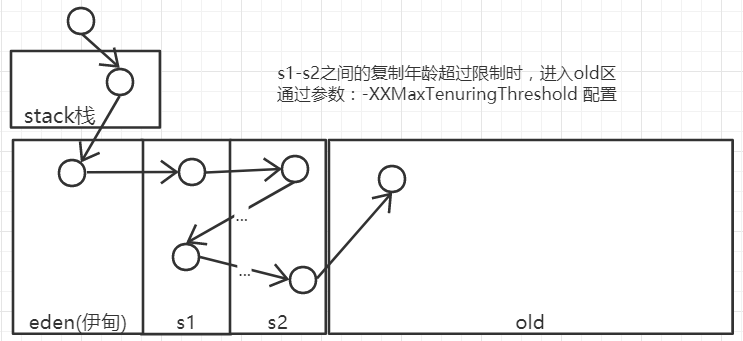
1. 一个对象产生之后首先尝试栈上分配
2. 栈上分配如果分配不下，进入eden区，
3. 第一次YGC：Eden区活着的对象进入s1，eden的垃圾全回收
4. 第二次YGC：S1 + Eden活着的对象--> 进入s2，S1+eden的垃圾全回收
5. 第三次YGC：eden + s2活着的对象--> 进入s1， eden + s2的垃圾全回收，
6. ........如此反复
7. 年龄够了，进入old区

（或者S1或S2装不下了，直接进入老年代；老的垃圾回收器年龄15岁，CMS6岁 就进old区）

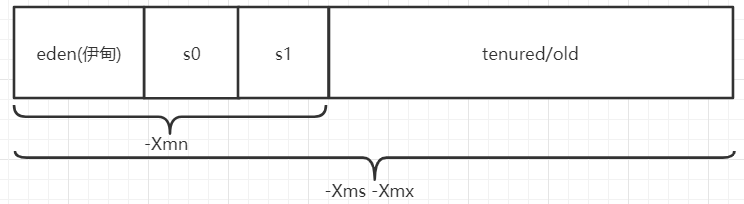
老年代也满了，触发一次FGC ---- Full GC

注：因为每次eden区都会有新的对象产生，故一直都是eden+S1/S2进行回收，有用的对象进入S1/S2

GC调优：调的是什么？尽量减少FGC



### 1.4.4 GC概念



#### MinorGC/YGC(Young)：

Eden区空间耗尽时触发。 （在年轻代发生的GC）

#### MajorGC/FullGC：

Old区空间不足时触发，新生代老年代同时进行回收。（在老年代或者整个区域回收） System.gc();

### 1.4.5 详解

#### 1.4.5.1 栈上分配--了解即可

##### 栈上分配

--什么对象会分配到栈上，调优无需调整，也不要动，没有意义

* 线程私有小对象
* 无逃逸（就在某一段代码里使用，出了这段代码没人认识它了，没人引用他）
* 支持标量替换（比如一个类T 里面就有俩属性int a ,int b ，就可以用普通的类型代替整个对象）

##### 线程本地分配TLAB（Thread Local Allocation Buffer）

----栈上分配不下了就会优先进行线程的本地分配，调优这块无需调整，

**概念：**在eden区，好多线程都往里头分配对象，多个线程会进行空间的争用，这个线程也想往这分，那个线程也想往这分，谁抢到算谁的。那么就用多线程的同步，一同步效率就会降低。所以设计了一种这样的机制 TLAB：

每个线程在eden区里取1%的空间，这块空间归这个线程独有，分配对象的时候，优先往我独有的线程里进行分配。这样就不会分其他线程产生争用，所以效率就会变高。

* 占用eden：默认1%
* 多线程的时候不用竞争eden就可以申请空间，提高效率
* 小对象

实验：默认情况下TLAB是打开的，下面的程序直接运行时间

|  |
| --- |
| **package** com.mashibing.jvm.c5\_gc;  *//-XX:-DoEscapeAnalysis -XX:-EliminateAllocations -XX:-UseTLAB -Xlog:c5\_gc\* // 逃逸分析 标量替换 线程专有对象分配* **public class** TestTLAB {  *//User u;* **class** User {  **int id**;  String **name**;   **public** User(**int** id, String name) {  **this**.**id** = id;  **this**.**name** = name;  }  }  *//调用一下new一个User* **void** alloc(**int** i) {  *//这个User new出来之后没有引用指向它，它逃不出alloc方法，无逃逸* **new** User(i, **"name "** + i);  *//u = new User(i, "name " + i);有逃逸* }   **public static void** main(String[] args) {  TestTLAB t = **new** TestTLAB();  **long** start = System.*currentTimeMillis*();  *//new 1000w个对象* **for**(**int** i=0; i<1000\_0000; i++) t.alloc(i);  **long** end = System.*currentTimeMillis*();  System.***out***.println(end - start);   *//for(;;);* } } |

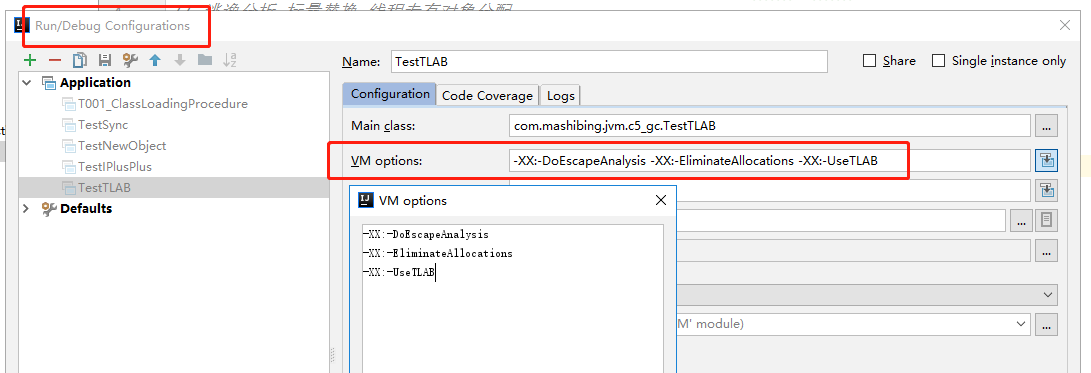
执行结果，大概500ms，下面关闭栈上分配 和TLAB

*-XX:-DoEscapeAnalysis 去掉逃逸分析*

*-XX:-EliminateAllocations 去掉标记替换*

*-XX:-UseTLAB --去掉TLAB*

用Idea Run---> Edit Configurations --> VM options ，输入上边的参数



运行 ，执行时间在800ms，比打开栈上分配和TLAB慢了300ms

栈上分配比堆上分配快很多，栈上分配不需要垃圾回收，栈一弹出就没了。跟垃圾回收没有什么关系，TLAB的话不要跟其他线程去争用，效率就会高。

#### 1.4.5.2 对象何时进入老年代

##### 超过XX:MaxTenuringThreshold 指定次数（YGC）

用法: -XX:MaxTenuringThreshold=3，主要是制新生代需要经历多少次GC晋升到老年代中的【最大阈值】。在JVM中用4个bit存储（放在对象头中），所以其最大值是15。但并非意味着，对象必须要经历15次YGC才会晋升到老年代中。例如，当survivor区空间不够时，便会提前进入到老年代中，但这个次数一定不大于设置的最大阈值。

===============================面试题=======================================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*--记住age最大是15，对象头4位记录着age \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

===============================面试题=======================================

* Parallel Scavenge垃圾回收器 默认年龄15
* CMS 垃圾回收器 默认年龄6
* G1垃圾回收器 默认年龄15

##### 动态年龄 --了解

<https://www.jianshu.com/p/989d3b06a49d>

垃圾回收的两个survivor区copy来copy去，比如从eden+s1 -->s2 ，对象个数已经超过s2对象个数的超过一般了，这个时候直接把s2中年龄最大对象的放入old区

* S1 --》 S2超过 50%
* 把年龄最大的放入Old区



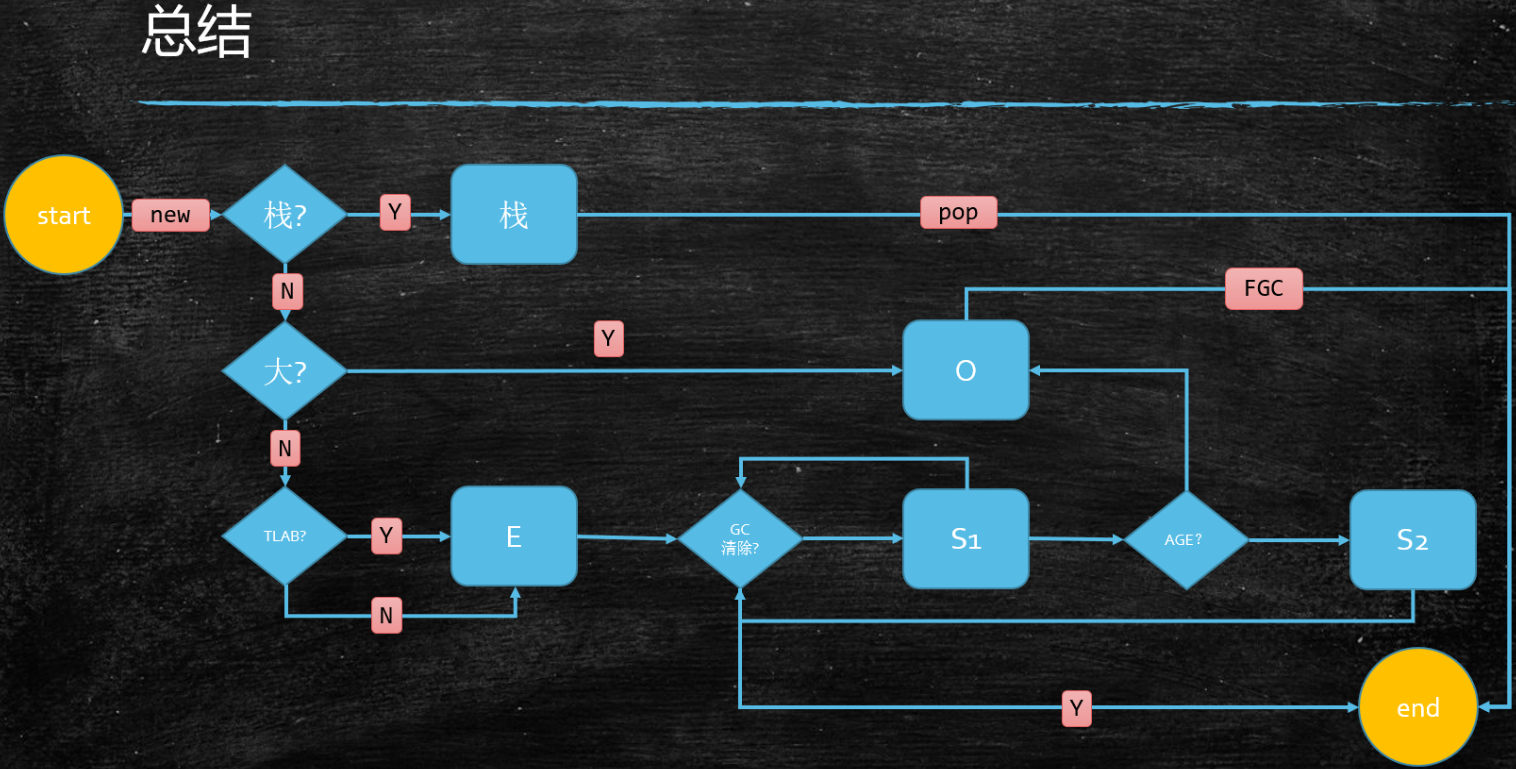
分配担保：（不重要）

正在进行YGC期间，有新的对象进来， survivor区空间不够了， 通过空间担保直接进入老年代

参考：<https://cloud.tencent.com/developer/article/1082730>

#### 1.4.5.3总结

对象分配过程详解



1. new一个对象
2. 首先尝试在栈上分配
3. 如果栈上能分配下，栈上有一好处，用完往外一弹，结束
4. 如果栈上分配不下，看对象大不大（用一个参数指定的）

如果对象特别大，会直接进入old区（FGC才被回收）

如果对象不够大，可能会进行TLAB，都是在Eden区，Eden区进行YGC：

如果清除完了，结束被回收。

如果没清除完，进S1区

如果年龄够了，进Old区，

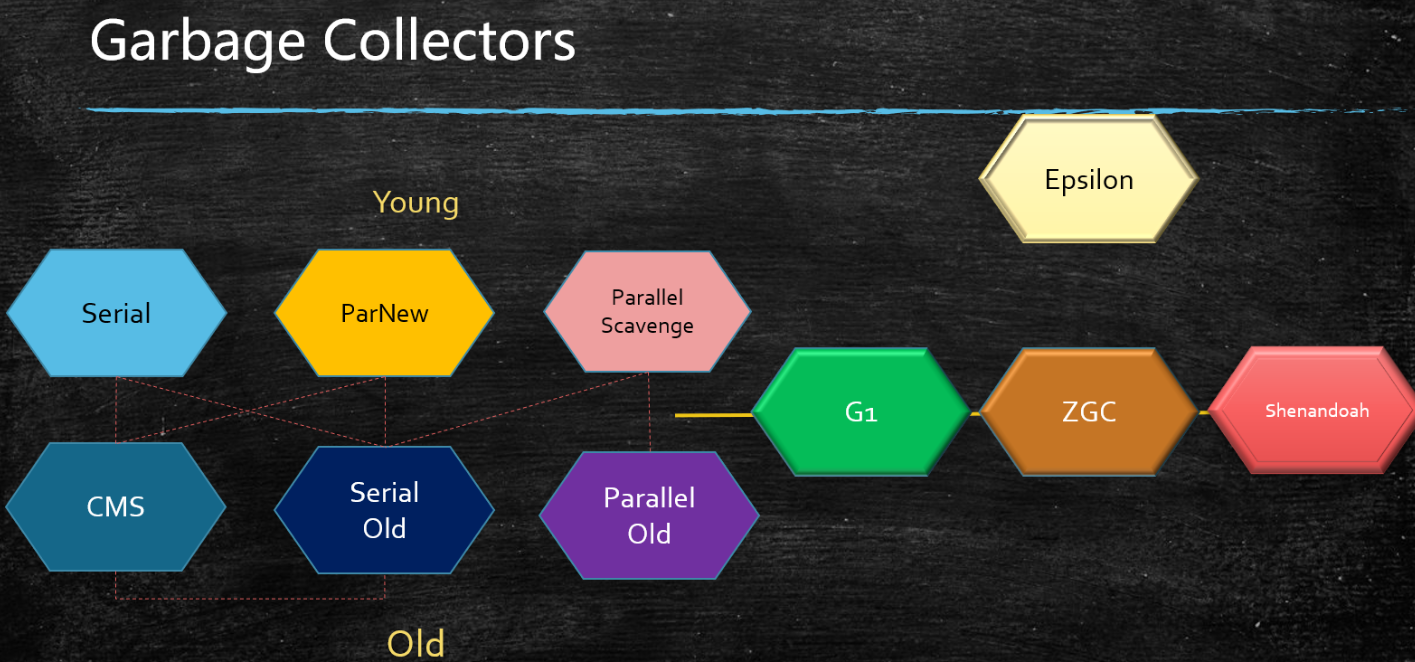
如果年龄不够，进S2区，【PS这里还有动态年龄判断】

S2经过YGC，再进入S1 或者进入Old

## 1.5垃圾回收器

#### 5.**1 常见的垃圾回收器**

==========到目前为止java产生的垃圾回收器 \*\*\*\*\*\*\*\*\* 二话不说，全背过========================



Serial：单线程的

Parallel：多线程的

|  |
| --- |
| JDK诞生 Serial追随 提高效率，诞生了PS，为了配合CMS，诞生了PN，CMS是1.4版本后期引入，CMS是里程碑式的GC，它开启了并发回收的过程，但是CMS毛病较多，因此目前任何一个JDK版本默认是CMS 并发垃圾回收是因为无法忍受STW |

|  |  |  |
| --- | --- | --- |
| 年轻代 | Serial | 串行回收 |
| ParNew | 配合CMS的并行回收 |
| Parallel Scavenge | 1. 并行回收 |
| 老年代 | Serial Old | 单线程回收 算法放在old区 |
| Parallel Old | 多线程的回收算法放在old区 |
| CMS  ConcurrentMarkSweep  JDK1.4以后 | 并发的， 垃圾回收和应用程序同时运行，降低STW的时间(200ms) CMS问题比较多，所以现在没有一个版本默认是CMS，只能手工指定 CMS既然是MarkSweep，就一定会有碎片化的问题，碎片到达一定程度，CMS的老年代分配对象分配不下的时候，使用SerialOld 进行老年代回收 想象一下： PS + PO -> 加内存 换垃圾回收器 -> PN + CMS + SerialOld（几个小时 - 几天的STW） 几十个G的内存，单线程回收 -> G1 + FGC 几十个G -> 上T内存的服务器 ZGC  算法：三色标记 + Incremental Update |
| 不分代 | G1  1.7才有，1.8稳定  1.9默认G1 | （只是逻辑上分年轻代老年代 物理上不分）  算法：三色标记 + SATB 10ms |
| ZGC | STW 1ms 算法：ColoredPointers + LoadBarrier |
| Shenandoah： | 算法：ColoredPointers + WriteBarrier |
|  | Eplison |  |

最常见的垃圾回收器组合：

1. Serial + Serial Old
2. Parallel Scavenge + Parallel Old
3. ParNew + CMS

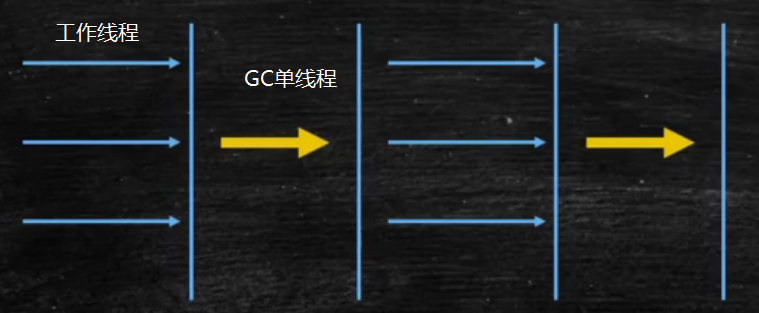
JDK1.8默认的垃圾回收器是Parallel Scavenge + Parallel Old ，如果上线之前没有进行任何的设置，就是这俩组合。大多数的调休是调的这俩

|  |
| --- |
| Java中Stop-The-World机制简称STW，是在执行垃圾收集算法时，[Java](http://www.jb51.net/list/list_207_1.htm" \t "https://www.cnblogs.com/williamjie/p/_blank)应用程序的其他所有线程都被挂起（除了垃圾收集帮助器之外）。Java中一种全局暂停现象，全局停顿，所有Java代码停止，native代码可以执行，但不能与JVM交互； |

##### Serial：

STW（stop -the -world）问题，采用复制算法，单线程清理垃圾。内存非常小的时候，JDK刚开始的时候 内存几十兆。其他程序全停止，进行垃圾回收。单CPU效率最高，虚拟机是Client模式的默认垃圾回收器。Safe point，保证其他线程的安全性，等一个操作完了再回收。

现在机器内存都非常大，回收停顿时间很长。

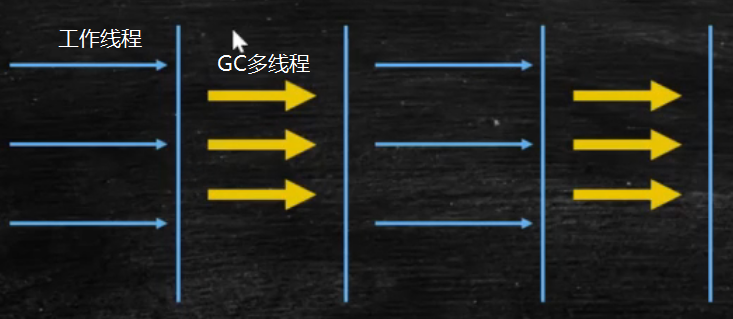


##### Serial Old：

STW（stop -the -world）问题，标记压缩算法，单线程清理垃圾。

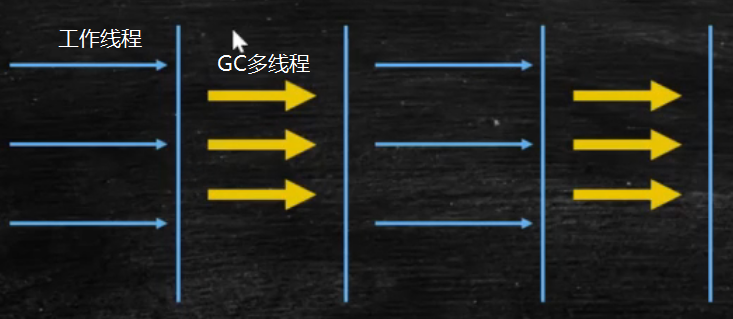
##### Parallel Scavenge：

STW（stop -the -world）问题，复制算法，多线程清理垃圾



##### Parallel old：

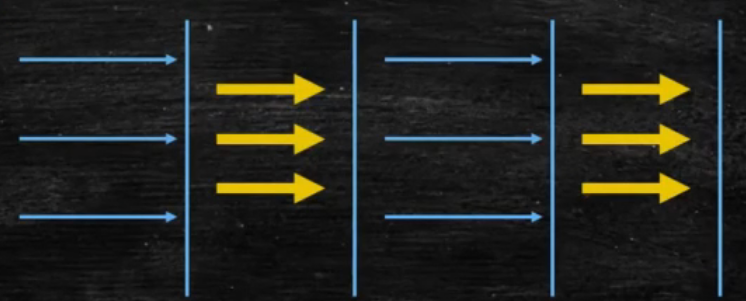
多线程 压缩算法



##### ParNew (Parallel New)

STW问题，复制算法 多线程，默认线程数为CPU核数。

和Parallel Scavenge的区别就是做了一些增强以配合CMS使用。CMS某个阶段的时候，ParNew可以同时运行。



1. PS 和 PN区别的延伸阅读： [https://docs.oracle.com/en/java/javase/13/gctuning/ergonomics.html#GUID-3D0BB91E-9BFF-4EBB-B523-14493A860E73](https://docs.oracle.com/en/java/javase/13/gctuning/ergonomics.html)

##### 垃圾收集器跟内存大小的关系

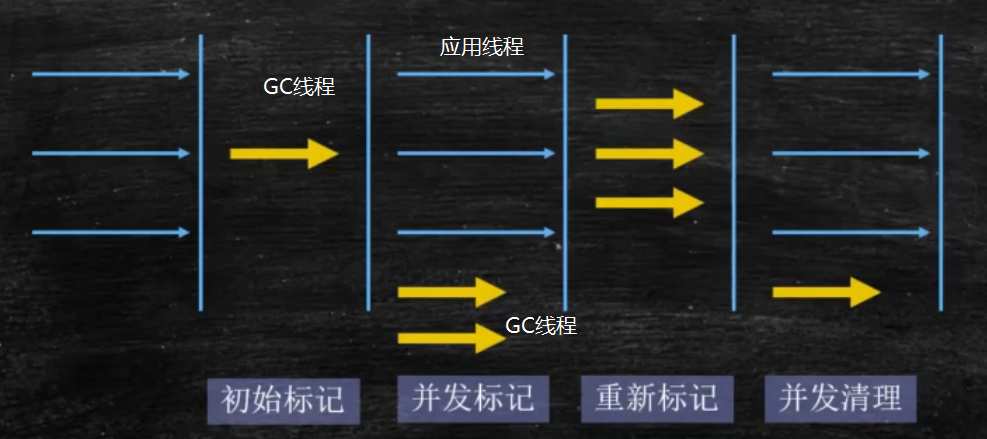
* Serial 几十兆
* PS 上百兆 - 几个G
* CMS - 20G
* G1 - 上百G
* ZGC - 4T - 16T（JDK13）

##### CMS： \*\*\*\*非常重要\*\*\*\*\*

ConcurrentMarkSweep 并发标记清除

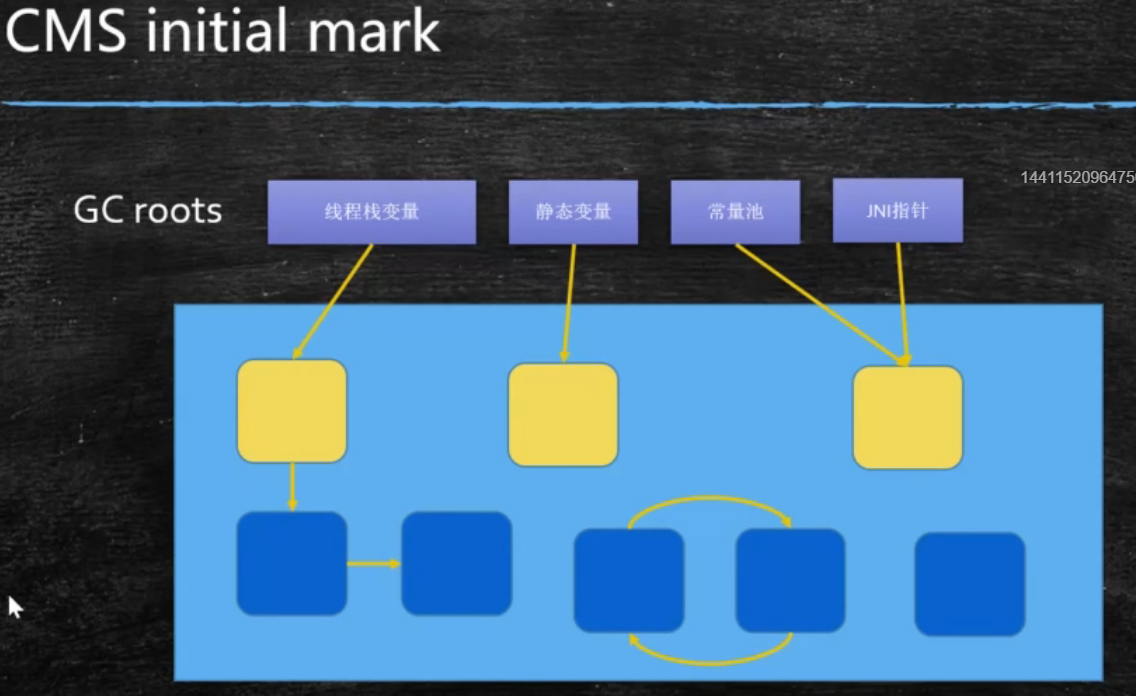
一个通常是并发，低停顿的垃圾回收器

###### 线程角度：

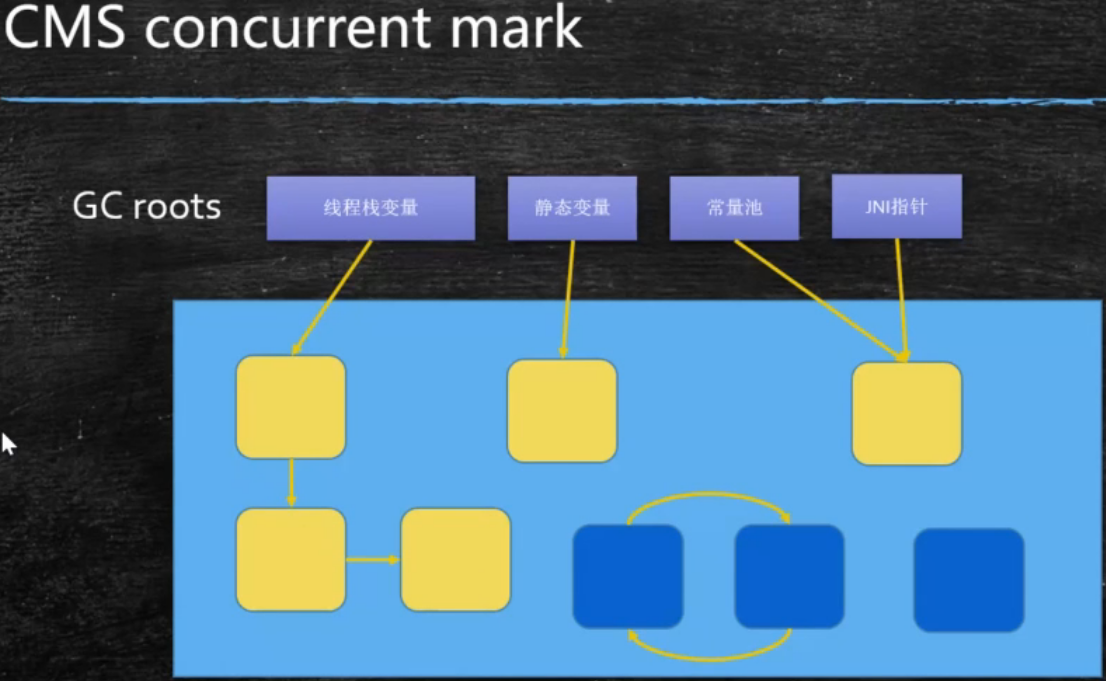


###### 四个阶段：

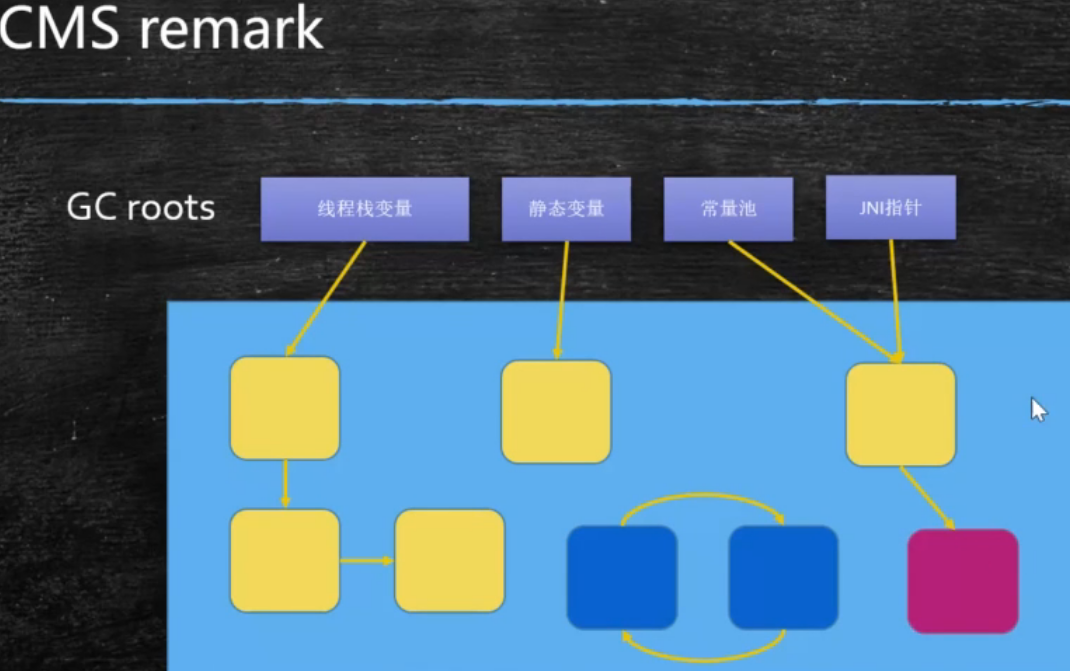
初始标记：这阶段是STW，只标记根对象那些不是垃圾的对象，单线程



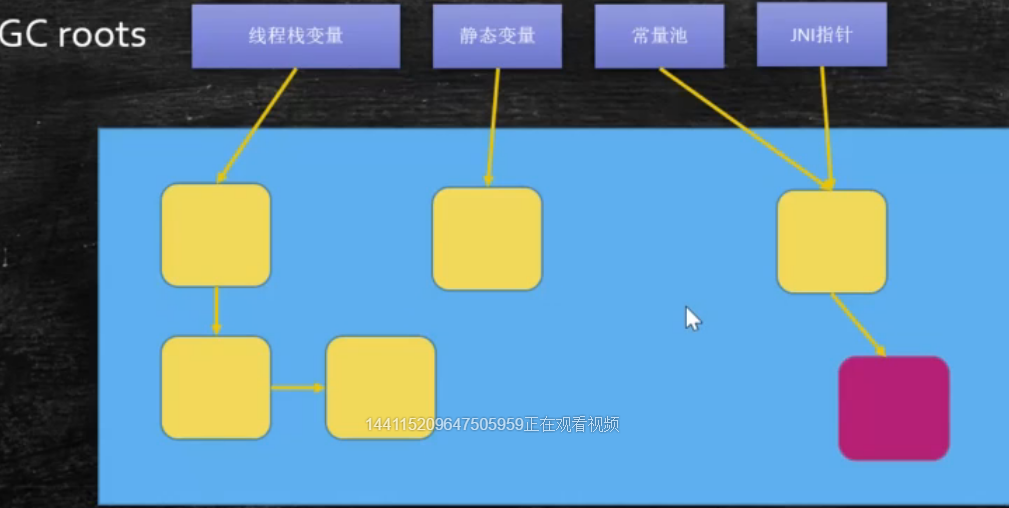
并发标记：应用线程执行者，GC线程同时执行，多线程



重新标记：这阶段是STW，标记那些并发标记期间，应用程序产生的垃圾，或者把垃圾变成了非垃圾。



并发清理：应用线程执行着（同时也会产生垃圾，叫浮动垃圾，下个阶段继续回收），一边清理着垃圾



##### CMS的问题：\*\*\*怼面试官\*\*\*\*

###### Memory Fragmentation内存碎片化

碎片化是比较严重的问题，CMS是MarkSweep （标记清除）就会产生碎片化，CMS从设计出来就是要对付几百M内存的，设计的时候就不是要应付很大内存的，现在很多人拿他来应付很大的内存，比如说32G内存。

如果拿CMS来对付32G内存，基本就会出问题。为什么会出问题呢？一旦从老年代产生很多很多碎片的时候，然后从年轻代过来的对象就找不着空间了，这叫PromotionFailed（升级找不着空间了），找不着空间后，CMS就会把老奶奶Serial Old 请出来（让老奶奶拿一小笤帚慢慢在里面扫垃圾），让Serial Old用一个线程在老年代做标记压缩。

###### Floating Garbage 浮动垃圾

Concurrent Mode Failure

解决：降低触发CMS的阈值

PromotionFailed

解决方案类似，保持老年代有足够的空间

**调整这个参数：**

–XX:CMSInitiatingOccupancyFraction 92% 可以降低这个值，让CMS保持老年代足够的空间

这个92%指的是：由于整块内存有浮动垃圾问题，所以得有一个阈值，92%的时候就会产生FGC，降低到68% ，50%，让它有足够的空间Promotion， 让它有足够的空间产生浮动来垃圾，回收的时候也能有空间让对象扔进来。

|  |
| --- |
| 如果你用的CMS，如果在日志里发现了Concurrent Mode Failure或者PromotionFailed，这两者基本就是说你的内存碎片太多了，对象分配不下了，怎么办呢？老奶奶（Serial Old）请出来。  所以CMS天然的一个问题就是，它本来设计出来是为了让垃圾回收的停顿时间比较短，但是一旦发生Concurrent Mode Failure或者PromotionFailed现象，它停顿的时间就会巨长巨长的，（一个老太太拿着小笤帚在扫天安门广场） |

### JVM调优第一步，了解生产环境下的垃圾回收器组合

JVM命令行参考 ：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

JVM参数分类：

标准： - 开头 ，所有的Hotspot都支持

非标准：-X 开头，特定版本Hotspot支持特定命令

不稳定：-XX 开头，下个版本可能取消

