## ==========JVM========

## rt.jar被什么类加载器加载，什么时间加载 √

Bootstrap 类加载器（Bootstrap是C++实现的并没有一个Class和它对应）

XX.java 文件 javac 编译成XX.class，ClassLoader将XX.class装载到内存。此时也会将用到的java的类库相关的类（String、Object等）也装载到内存（存在于rt.jar）。

## 自己写的类被什么加载，什么时间加载；√

被AppClassLoader加载

（自己类的类加载器：System.***out***.println(TestClassLoaderLevel.**class**.getClassLoader());

打印：sun.misc.Launcher$AppClassLoader@73d16e93）

当我们主动使用类的时候，类才会被加载

主动使用有一下6种情况：

1.创建类的实例。例如：new Class();

2.访问某个类或接口的静态变量，或者给静态变量赋值

3.调用类的静态方法

4.反射：Class.forName("java.lang.String");

5.初始化一个类的子类

6.Java虚拟机启动时被标明为启动类的类（包含Main方法）

## 类加载的过程||类是如何加载到JVM中的 √

一个class文件怎么从硬盘到内存，开始执行的



分三步：

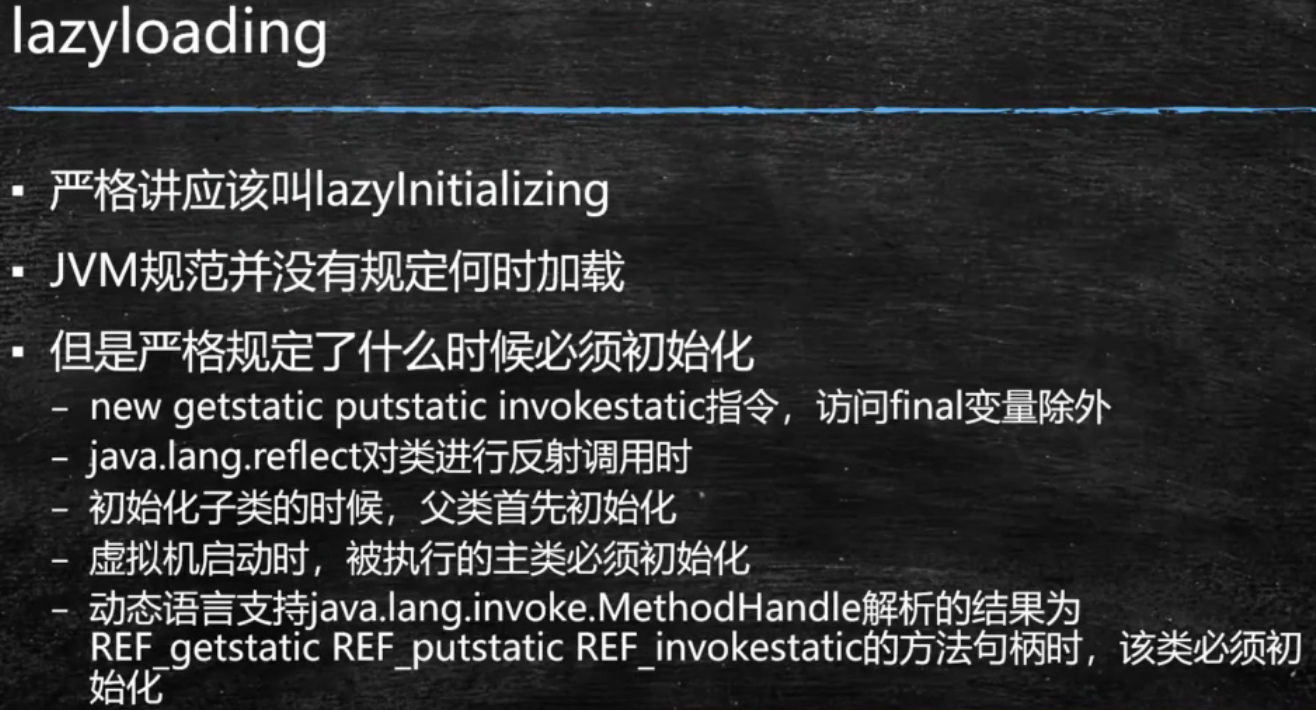
1. Loading ：把一个class文件（一个个的二进制，一个个的字节）load到内存，
2. Linking
3. Verification ：校验装进来的class文件符不符合class文件的标准（cafebabe开头）
4. **Preparation：给class静态成员变量赋默认值（\*不是初始值，如int 赋为0，引用赋为null）**
5. Resolution（解析）：
   * 1. class文件里类、方法、属性等符号引用（字符串），解析为直接引用，直接指向内存
     2. 常量池里中的各种符号引用解析为指针、偏移量等内存地址的直接引用）（直接能够访问到的内容）
6. Initializing：
   * 1. 调用类初始化代码，给静态成员变量赋初始值，执行静态语句块

## 自己写的两个不同的类是被同一个类加载器加载的吗？为什么？ √

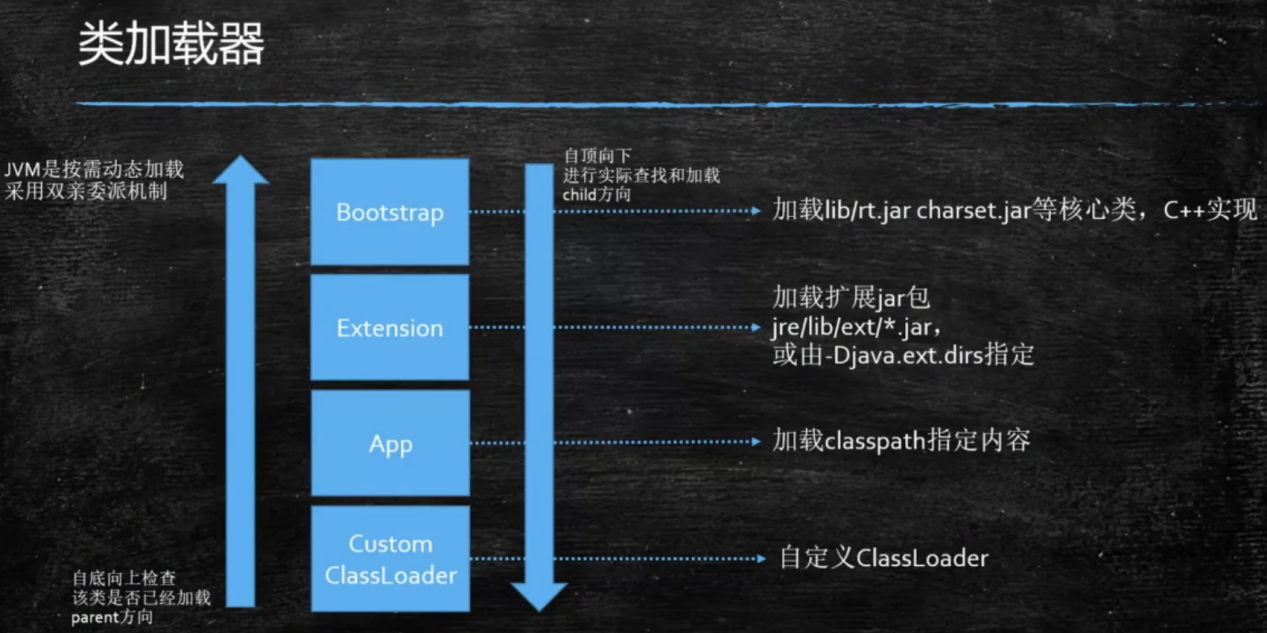
是，AppClassLoader应用类加载器,又称为系统类加载器,负责在JVM启动时,加载来自在命令java中的classpath或者java.class.path系统属性或者CLASSPATH操作系统属性所指定的JAR类包和类路径.

## 逃逸分析是什么，作用是什么，用途是什么;

## 什么情况下会触发类加载； √

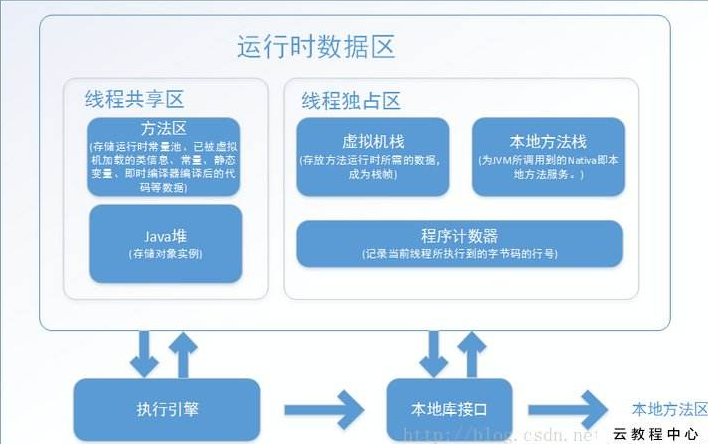


## JAVA 类加载器 √



## ==========JMM ==========

## JVM运行时内存区域划分 && Java内存模型JMM √



### PC（程序计数器）

记录当前线程所执行到的字节码的行号

### 虚拟机栈

存放方法运行时所需的数据，里面存的是栈帧（栈帧请看JVM学习笔记部分）

### 本地方法栈

JNI调用java虚拟机内部的C、C++的DLL

### 堆

存储对象实例

### 方法区

方法区是逻辑上的概念，装每个Class的结构，常量池

\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*面试题JDK1.7 & 1.8区别\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

方法区是逻辑上的概念，具体的实现：

Perm Space (<1.8指的就是永久区）：字符串常量位于PermSpace FGC不会被清理

Meta Space (>1.8 元数据区) ：字符串常量位于堆，会触发FGC

### 直接内存

JVM可以直接访问的内核空间的内存（OS管理的内存），用于NIO，提高效率，实现零拷贝

（在JDK1.8中，永久代被移除，取而代之的是元空间概念，也就是使用本地内存）

## 直接内存如何管理的 √

JDK1.4中加入新的NIO（New Input/OutPut）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，可以通过Native函数库直接分配堆外内存，然后通过Java堆中的DirectByteBuffer对象来对这块内存的引用进行操作，避免数据在Java堆与Native堆中数据的来回复制。

本机的直接内存是有限制的，因此肯定也会由于内存不足造成OutOfMemoryError异常。

在JDK规范中存在的是方法区，但是HotSpot是通过PermGen Space（永久代）实现的。永久代仍存在于JDK1.7中，并没完全移除，譬如符号引用(Symbols)转移到了native heap；字面量(interned strings)转移到了java heap，String.intern()方法的实现也有变化；类的静态变量(class statics)转移到了java heap；在JDK1.8中，永久代被移除，取而代之的是元空间概念，也就是使用本地内存。

不断的使用String.intern()方法，在JDK1.6中会产生java.lang.OutOfMemoryError: PermGen space这个错误，在JDK1.7中就会报java.lang.OutOfMemoryError: Java Heap space，而在JDK1.8 中，也是java.lang.OutOfMemoryError: Java Heap space，但是还会多出warning：对参数PermSize以及MaxPermSize的设置已经从1.8中移除。

具体从永久代向元空间的转换原因如下：

1、字符串存在永久代中，容易出现性能问题和内存溢出。

2、类及方法的信息等比较难确定其大小，因此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，太大则容易导致老年代溢出。

3、永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

## 讲下JVM的大页模式

## 对象内存布局 √

### 普通对象



|  |  |  |
| --- | --- | --- |
| 名称 | 解释 | 内存大小 |
| MarkWord  标记字段 | 哈希码、分代年龄、锁标志位、偏向线程ID、偏向时间戳等信息 | 8个字节 |
| ClassPointer类型指针 | 即指向当前对象的类的元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例 | -XX:+UseCompressedClassPointers 为4字节 不开启为8字节 |
| 实例数据 | 成员变量，int m=2 ，int n=1；String就引用指向别处 | 引用类型：-XX:+UseCompressedOops 为4字节 不开启为8字节 Oops Ordinary Object Pointers |
| Padding对齐  8的倍数 | 64位的机器读是按块来读的，并不是按字节，来提高效率。比如15个字节，就填充成16字节 | 8的倍数 |

### 数组对象

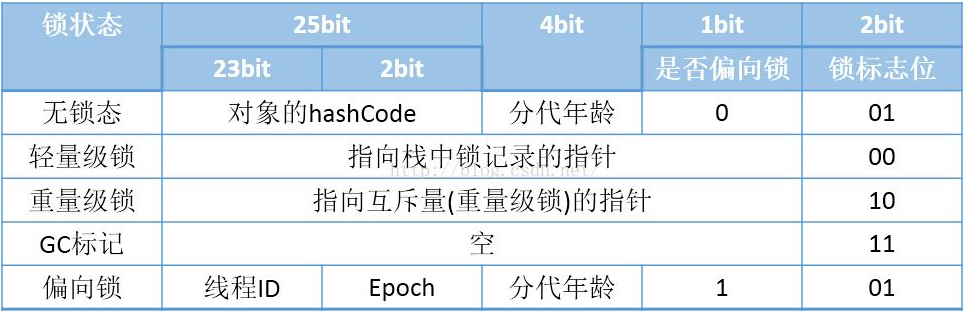
1. .对象头：markword 8
2. ClassPointer指针同上，数组装的class类型
3. 数组长度：4字节
4. 数组数据
5. 对齐 8的倍数

## 对象头 || Markword里面装的是什么？

至少说出来2点

1. **锁定信息：**有两位（严格说3位，还有偏向锁）代表对象有没有被锁定，锁定就是synchronized ( this 、o、p、...)这个对象，当所谓的synchronized这个对象的时候，实际上是对象脑袋上的两位，来代表锁的标志位。
2. **GC的标志位：**我这个对象被回收了多少次了，分代的年龄。

（这两项说出来，面试就八九不离十了）



## JMM里边的原子性、可见性、有序性是如何体现出来的（其实就是问JMM里的数据一致性）√

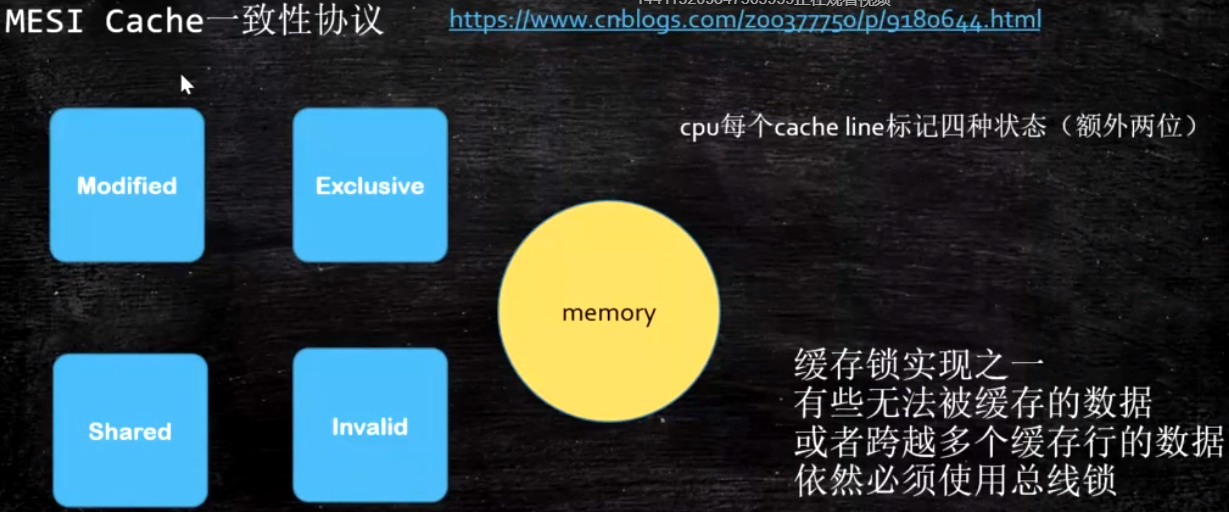
### 原子性解决（硬件层）：

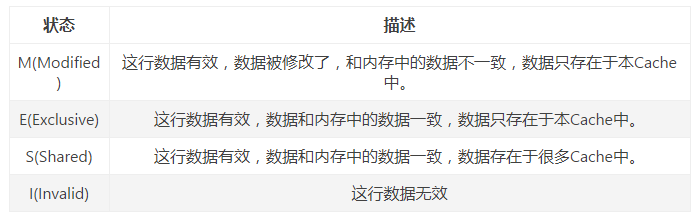
#### 总线锁（老CPU）：

在CPU和内存之间的通道，有一把总线锁，多个CPU访问主内存的时候，给这条通道上锁。因此效率低

#### MESI 缓存一致性协议

（Intel的CPU用的是这个）缓存锁。现代CPU的数据一致性实现 = 缓存锁（MESI...）+总线锁





### 可见性、有序性解决：

#### 硬件层：

内存屏障、原子指令。x86上的”lock +[其他指令]”指令 是一个Full Barrier，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。Software Locks通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序

#### 软件层（JVM级别）：

虚的东西，具体的实现是由JVM自己去实现的，具体的实现依赖于硬件的实现。

**LoadLoad屏障，StoreStore，LoadStore，StoreLoad四种屏障**

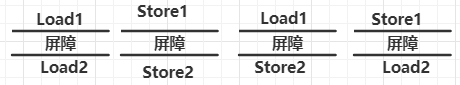
## JMM中内存屏障是什么意思 √

CPU为了提高指令执行效率，会打乱多条指令的执行顺序。会有**乱序读，合并写问题，**

特定的位置插进去内存屏障，将指令分开。指令不能重排

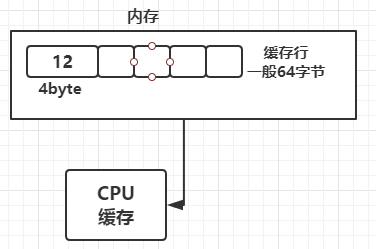
JVM级别是虚的东西，具体的实现是由JVM自己去实现的，具体的实现依赖于硬件的实现

理解：四种组合，是 屏障两端是Load 、 Store的四种组合，如下图：



## 缓存行是什么？ √

概念：CPU读取内存数据到自己缓存的时候，它不会只把这一个数据放进去，比如一个int类型12，他不会只把12所在的4个字节读进去，而是把后面的一堆内容一块读进去，这一块是一个【基本的缓存单位】，称之为缓存行，多数长度为64字节。



## 伪共享是什么？ √

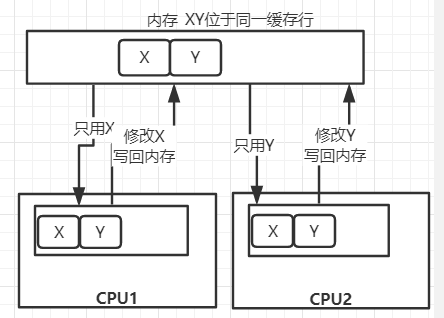
概念：位于同一缓存行的两个不同数据，被两个不同CPU锁定，产生互相影响的伪共享问题。

举例：X、Y位于同一缓存行，CPU1只会用到X，CPU2只会用到Y。

1. CPU1将缓存行从内存读到自己的高速缓存
2. CPU2将缓存行从内存读到自己的高速缓存
3. CPU1修改X的值，并写回内存，并通知CPU2，你的缓存行现在是Invalid状态了，你更新吧！
4. CPU2更新缓存行。
5. CPU2修改Y的值，并将缓存行写回内存，并通知CPU1，你的缓存行现在是Invalid状态了，你更新吧！
6. CPU1更新缓存行。

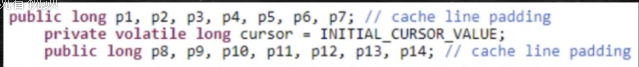
结果：两个位于同一缓存行的互相【无关的值】，变来变去的时候，会产生缓存行的互相影响问题。这就是伪共享。

产生不好的影响，需要重新load缓存行。



伪共享问题解决：缓存行对齐，能够提高效率，会浪费一定空间。

Disruptor框架源码：



## ==========GC ==========

## 如何判断对象是否可以回收或存活||如何定位垃圾||引用计数法与GC Root可达性分析法区别；√

### 引用计数法：

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:

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。

## 根对象(GC Roots)有哪些？

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：

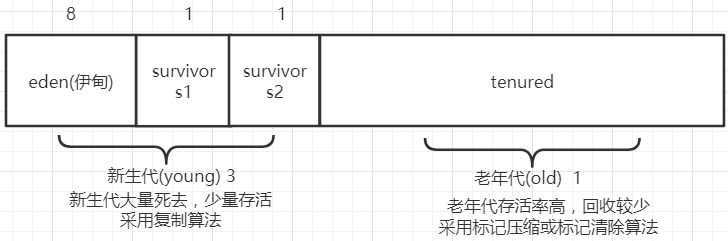
调用C、C++本地方法要用到的类对象

## 堆分为哪几块，比如说新生代老生代，那么新生代又分为什么？√

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。



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 | 占用JVM内存，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 | 直接使用物理内存，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

|  |
| --- |
| 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 |

|  |  |  |
| --- | --- | --- |
| 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 |  |

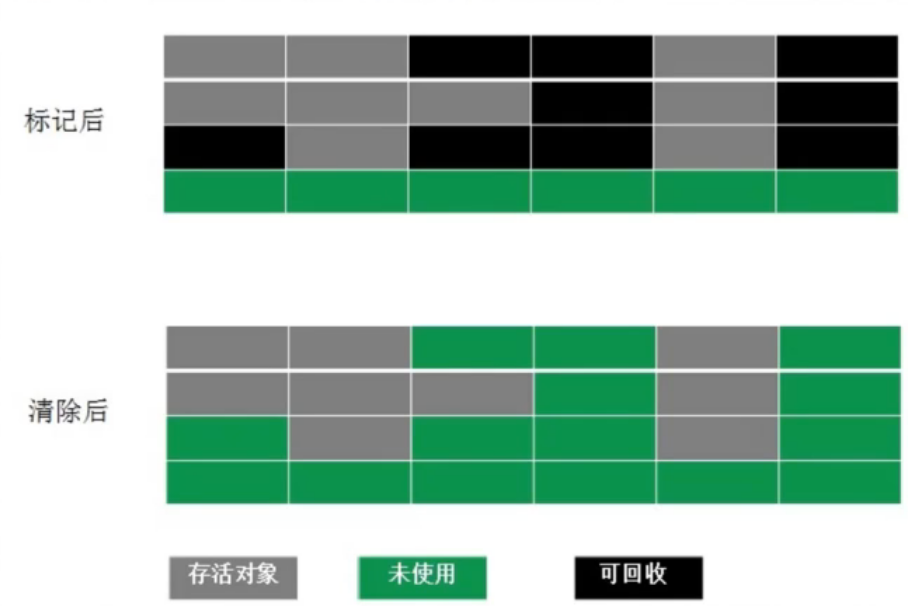
## 常见的GC回收算法及其含义 √

### 标记清除法

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

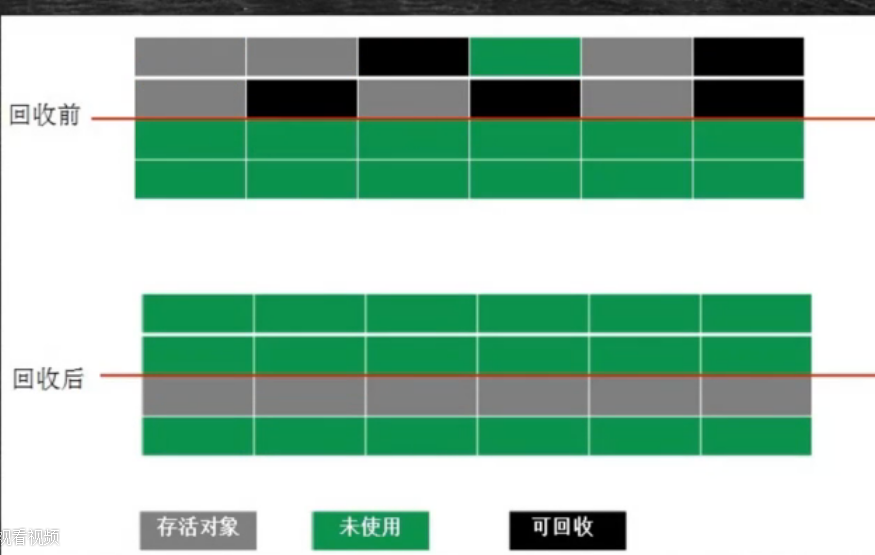


### 拷贝算法

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。

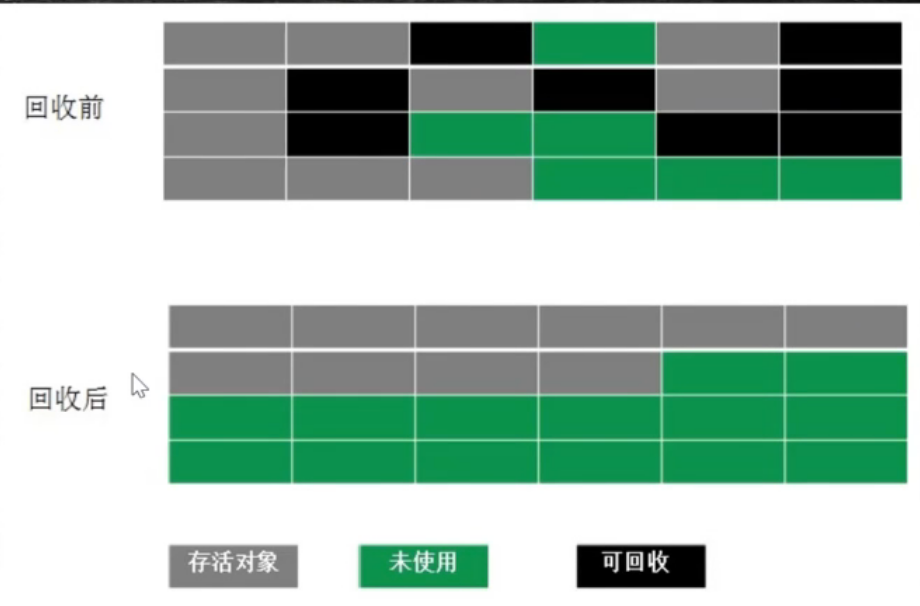


### 标记压缩

有用的，都往前面走，剩下的空间就都清理出来了。

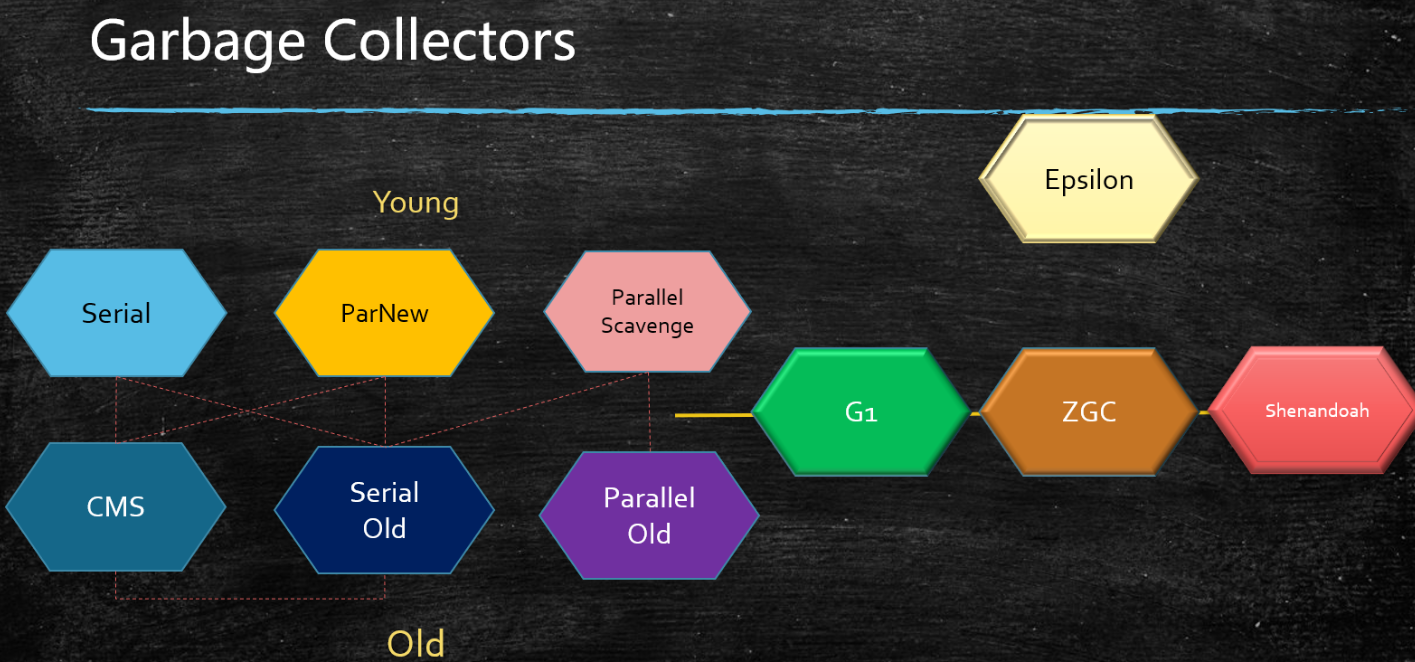
没有碎片

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



## 常见的垃圾回收器有哪些？\*\*\*\*\* √

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



|  |  |  |
| --- | --- | --- |
| 年轻代 | Serial | 串行回收 |
| ParNew | 配合CMS的并行回收 |
| Parallel Scavenge  发音 [ˈpærəlel] [ˈskævɪndʒ] | 并行回收 |
| 老年代 | Serial Old | 单线程回收 算法放在old区 |
| Parallel Old  发音 [ˈpærəlel] | 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 | JDK11 JDK自己调试用的 |

最常见的垃圾回收器组合：

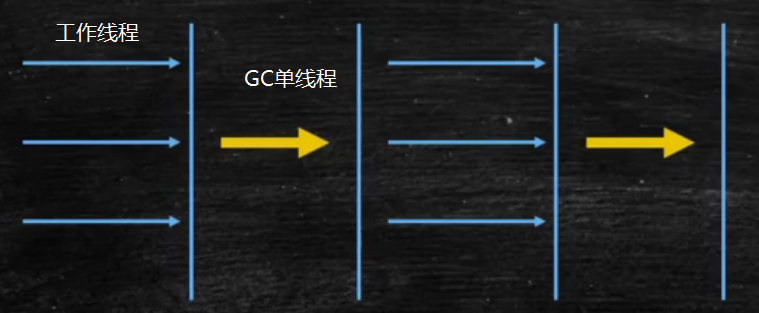
1. Serial + Serial Old
2. Parallel Scavenge + Parallel Old
3. ParNew + CMS

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。大多数的调优是调的这俩

##### 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。

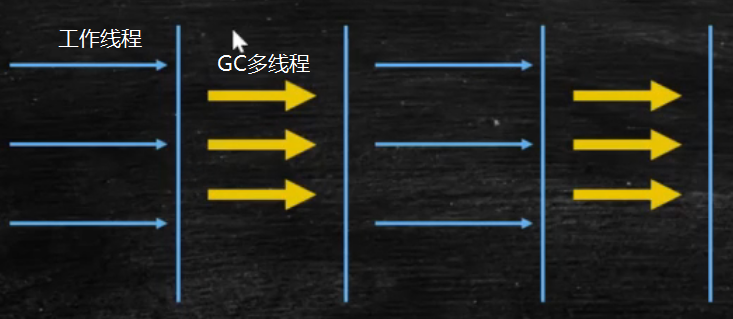


##### Serial Old：

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

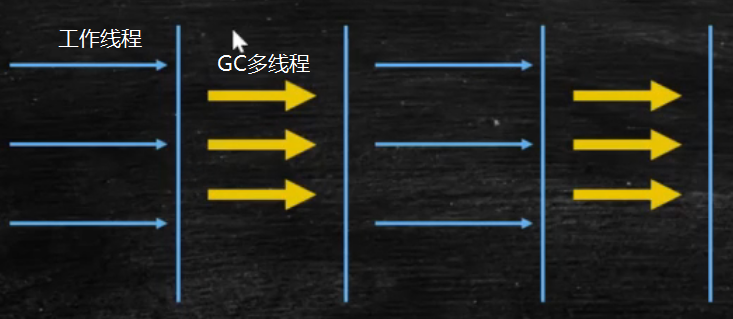
##### Parallel Scavenge：

STW（stop -the -world）问题，复制算法，多线程清理垃圾



##### Parallel old：

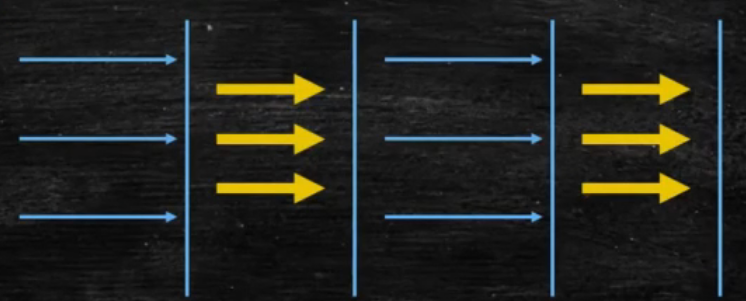
多线程 压缩算法



##### 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收集器与G1收集器的特点。√

CMS看上边。

G1收集器：三色标记 STAB

## CMS 和 G1 的区别知道吗？使用场景分别是？你项目中用的是哪个？√

|  |  |  |
| --- | --- | --- |
| 年轻代 | Serial | 串行回收 |
| ParNew | 配合CMS的并行回收 |
| Parallel Scavenge  发音 [ˈpærəlel] [ˈskævɪndʒ] | 并行回收 |
| 老年代 | Serial Old | 单线程回收 算法放在old区 |
| Parallel Old  发音 [ˈpærəlel] | 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 | JDK11 JDK自己调试用的 |

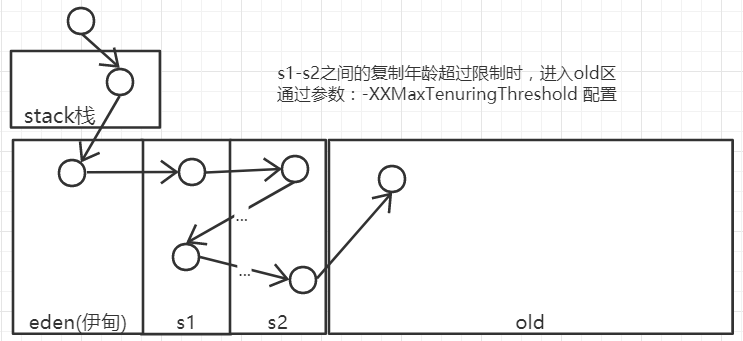
我们项目用的默认的：

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。大多数的调优是调的这俩

**最常见的垃圾回收器组合：**

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

## 为什么新生代内存需要有两个Survivor区？√



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

## Java在什么时候会出现内存泄漏；

## G1停顿吗，

## 为什么JVM调优经常会将-Xms和-Xmx参数设置成一样；√

-Xms：最小堆大小，刚开始堆的大小

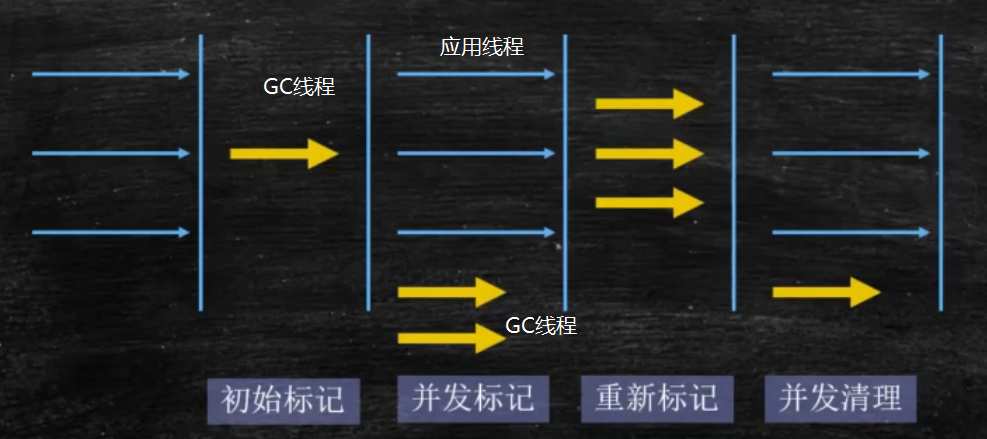
-Xmx：最大堆大小。最好把这里设置成一样的大小，否则堆会根据对象的占用的大小进行扩大或者压缩，浪费了系统计算的资源。

## CMS垃圾回收过程； \*\*\*\*非常重要\*\*\*\*\* ||CMS回收步骤，CMS为什么会停顿，停顿时间；√

ConcurrentMarkSweep 并发标记清除

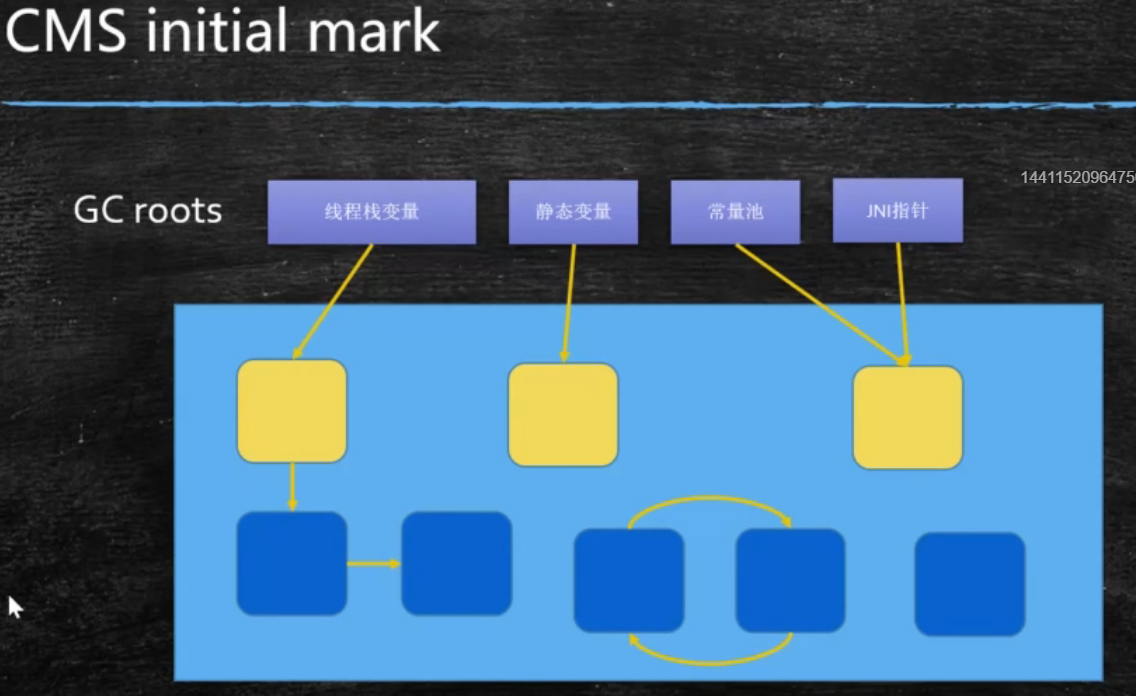
一个通常是并发，低停顿的垃圾回收器

###### 线程角度：

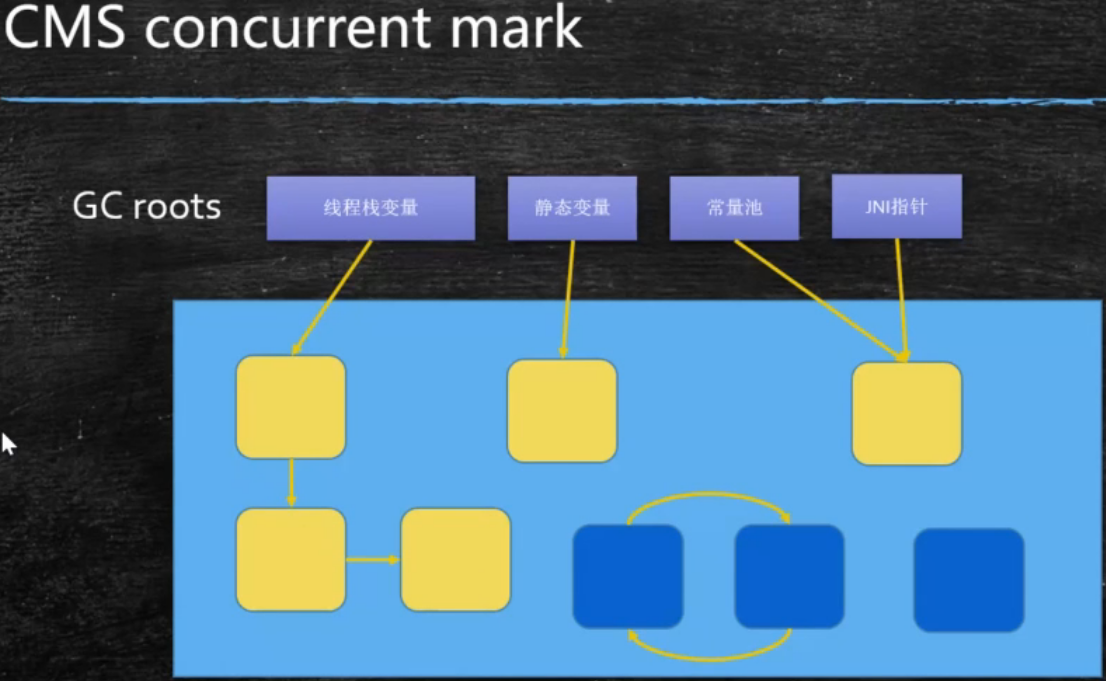


###### 四个阶段：

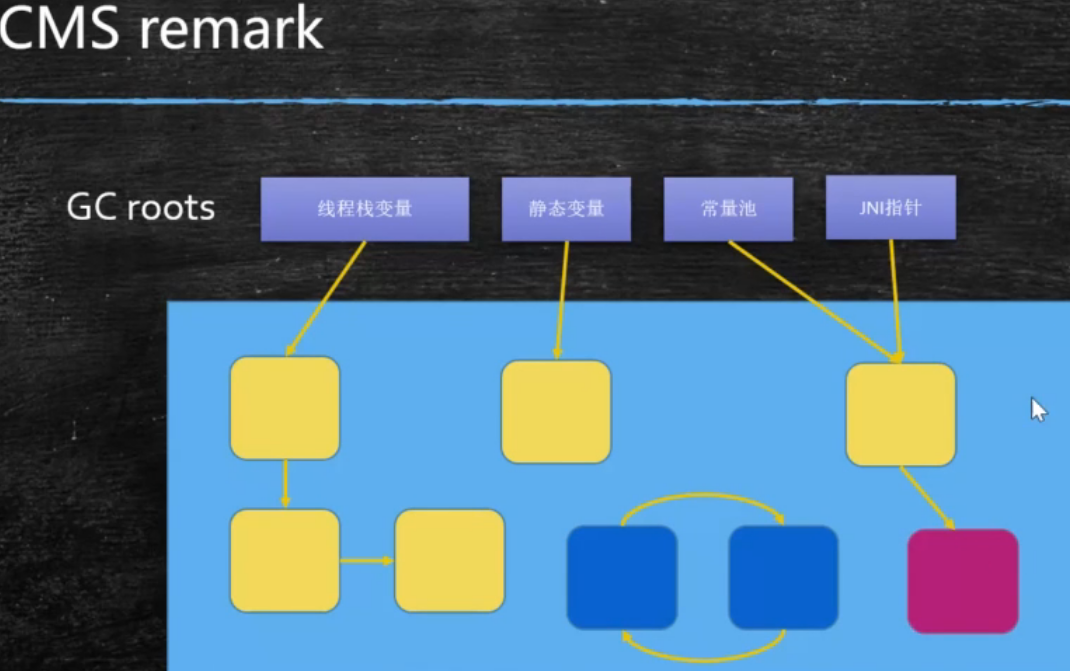
初始标记：这阶段是STW，只标记根对象那些不是垃圾的对象，单线程



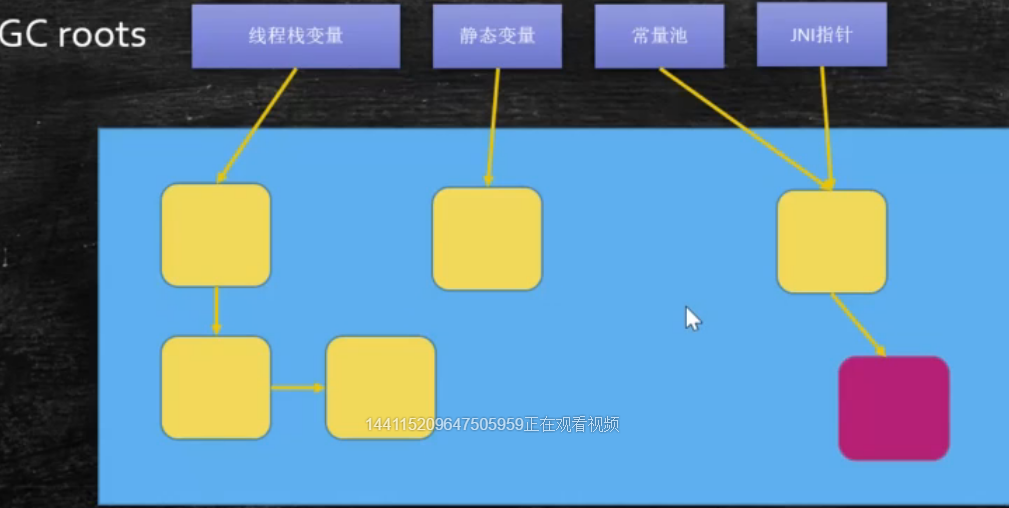
并发标记：应用线程执行者，GC线程同时执行，多线程



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。

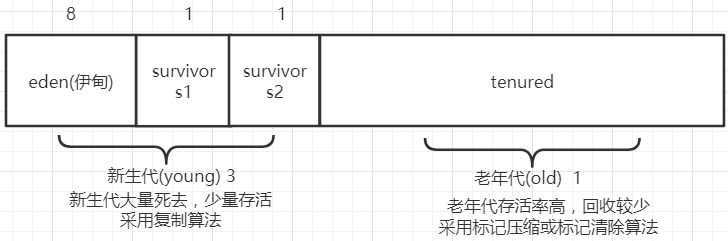
|  |
| --- |
| 如果你用的CMS，如果在日志里发现了Concurrent Mode Failure或者PromotionFailed，这两者基本就是说你的内存碎片太多了，对象分配不下了，怎么办呢？老奶奶（Serial Old）请出来。  所以CMS天然的一个问题就是，它本来设计出来是为了让垃圾回收的停顿时间比较短，但是一旦发生Concurrent Mode Failure或者PromotionFailed现象，它停顿的时间就会巨长巨长的，（一个老太太拿着小笤帚在扫天安门广场） |

## ==========GC 调优 ==========

## GC调优：调的是什么？√

GC调优：尽量减少FGC

## Minor GC与Full GC分别在什么时候发生？什么时候触发Full GC;√



#### 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();

## 然后讲下对象的死亡过程？

## Full GC次数太多了，如何优化；

## 如果你的项目出现了内存泄露，怎么监控这个问题呢；

## 标记清除和标记整理的区别和优缺点，为何标记整理会发生stop the world；

## 开闭原则，解析工厂方法模式，建造者模式，区别。手撸出来。

## 写个后缀表达式，为什么要设计后缀表达式，有什么好处？然后写下中缀。

## 我看你做过性能优化，比如你怎么分析项目里面的OOM的，内存泄露呢？详细说思路;

## 涉及OOM、JVM优化、源码问题、数据库优化、多线程等问题;

## CPU高？什么情况CPU高？解决什么问题？

## 开闭原则说一下

## Java内存抖动严重，优化的思路；

## 怎么判断对象是否可 GC？Java 对象有哪些引用类型？有什么区别？

## OOM 出现的有哪些场景？为什么会发生？

## 分析过 GC 日志吗？

## 你还知道哪些 JVM 调优参数？

## 假如线上服务发生 OOM，有哪些措施可以找到问题？

## 假如线上服务 CPU 很高该怎么做？有哪些措施可以找到问题？

## 假如线上应用频繁发生 Full GC，有哪些措施可以找到问题？

## 一般线上环境遇到 JVM 问题，你会使用哪些工具来分析？找到问题后又该如何去解决呢？

## 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

## 软引用和弱引用的使用场景（软引用可以实现缓存，弱引用可以用来在回调函数中防止内存泄露）；

## 强引用、软引用、弱引用、虚引用

## JVM如何设置参数

## JVM性能调优

## 常见的JVM性能监控和故障处理工具类：jps、jstat、jmap、jinfo、jconsole等

## 几种常用的内存调试工具：jmap、jstack、jconsole；