04-JVM虚拟机

刘亚雄

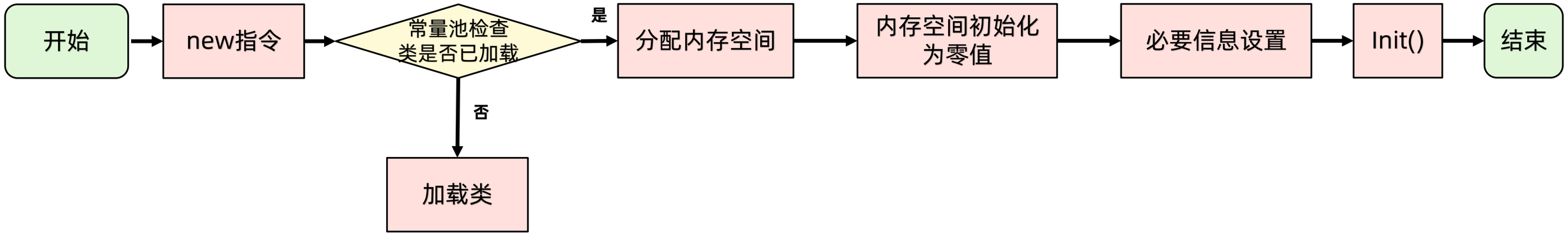
极客时间-Java 讲师

今日目标

1. 理解对象生命周期：创建，年轻代到老年代、销毁
2. 掌握对象内都有些什么
3. 掌握GC垃圾收集器基本原理：什么是垃圾、如何找到并清除垃圾
4. 掌握GC垃圾清除算法：Mark-Sweep、Copying、Mark-Compact
5. 掌握常见GC及其特点：Serial、Parallel、ParNew、CMS、G1、ZGC

四、对象创建流程与内存分配

4.1 对象的创建流程



常量池检查：检查new指令是否能在**常量池中定位到这个类的符号引用，**检查类之前是否被加载过

分配内存空间：

- 指针碰撞：GC不带压缩功能，Serial和ParNew
- 空闲列表：GC带压缩功能，CMS

必要信息设置：对象类的元数据，对象哈希码，GC分代年龄 → 对象头

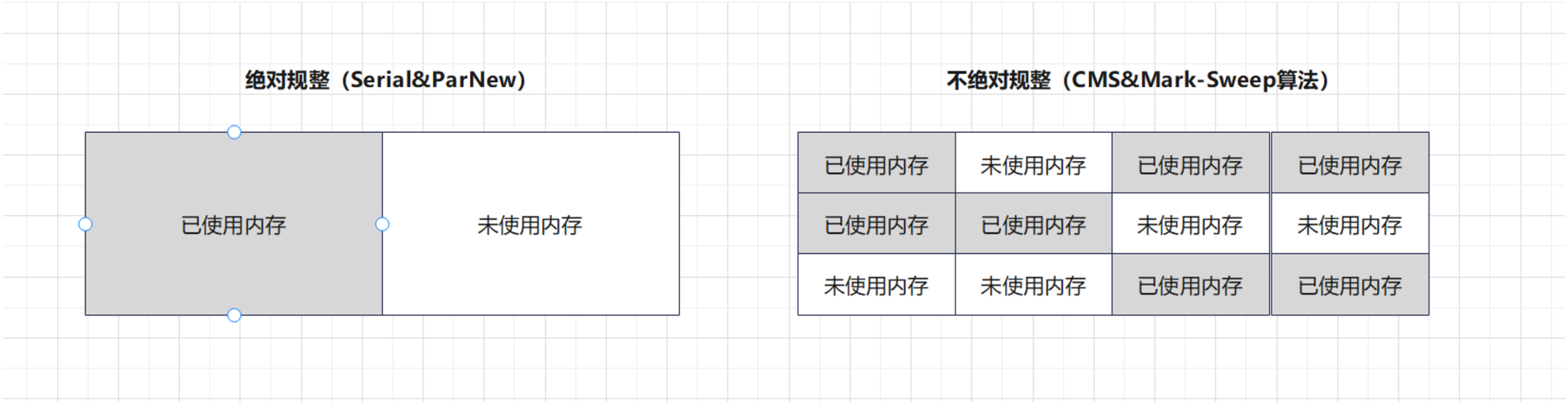
从哪里分配内存？

4.2 对象的内存分配方式

内存分配的方法有两种：

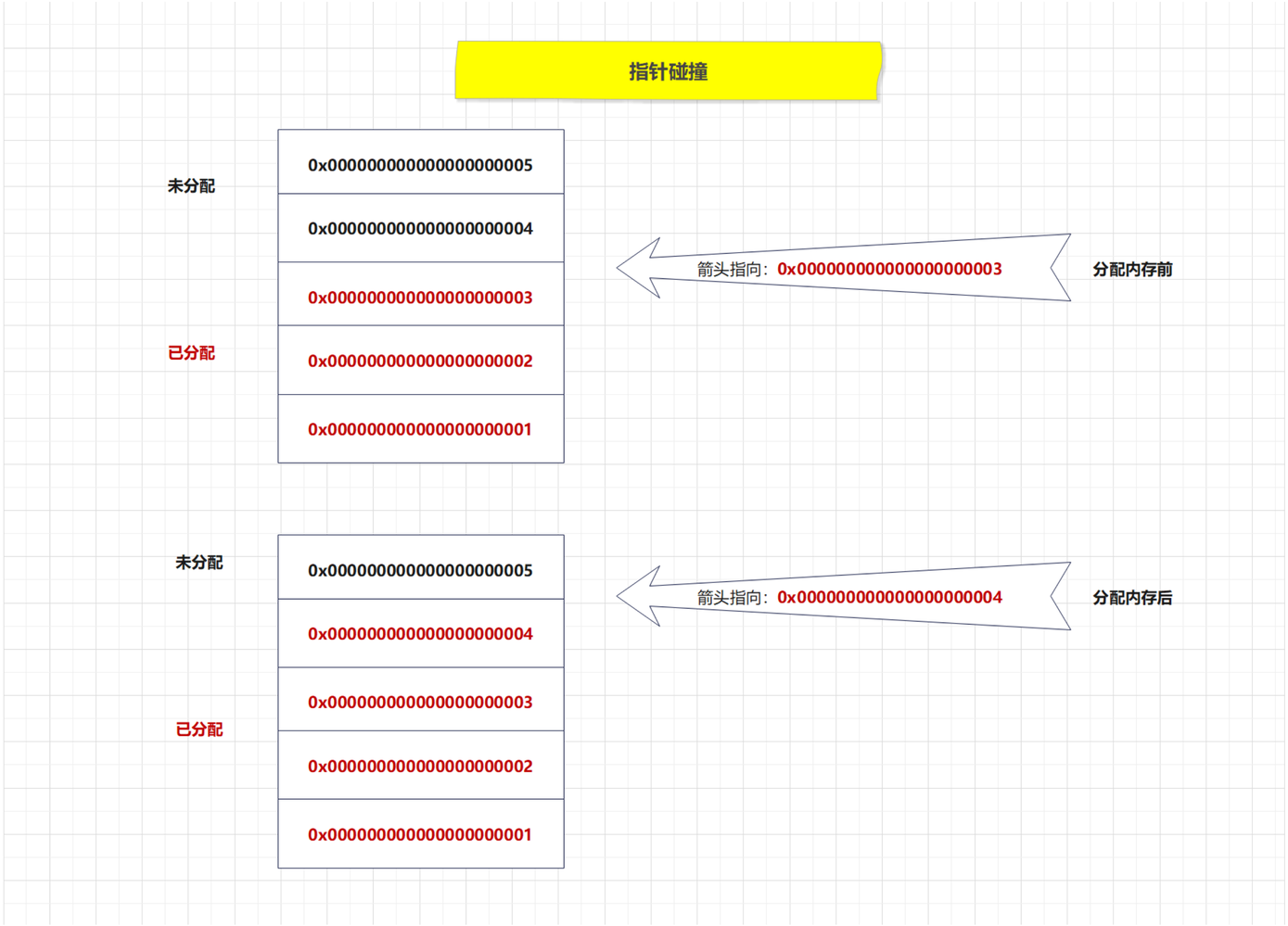
- 指针碰撞(Bump the Pointer)
- 空闲列表(Free List)

分配方法	说明	收集器
指针碰撞(Bump the Pointer)	内存地址是连续的（新生代）	`Serial`和`ParNew`收集器
空闲列表(Free List)	内存地址不连续（老年代）	`CMS`收集器和`Mark-Sweep`收集器



4.2 对象的内存分配方式

指针碰撞示意图：



内存分配存在线程安全问题吗？

4.2 对象的内存分配方式

内存分配安全问题：

- 虚拟机给A线程分配内存的过程中，指针未修改，此时B线程同时使用了该内存，是不是就出现问题了！

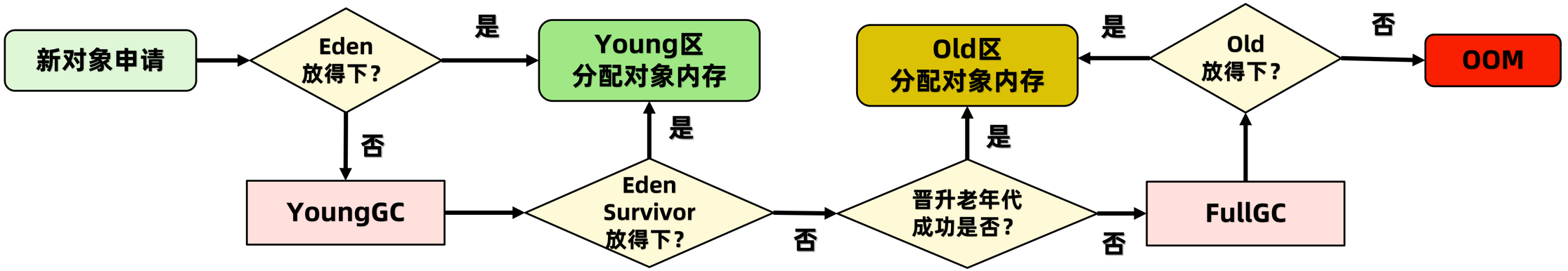
怎么办？

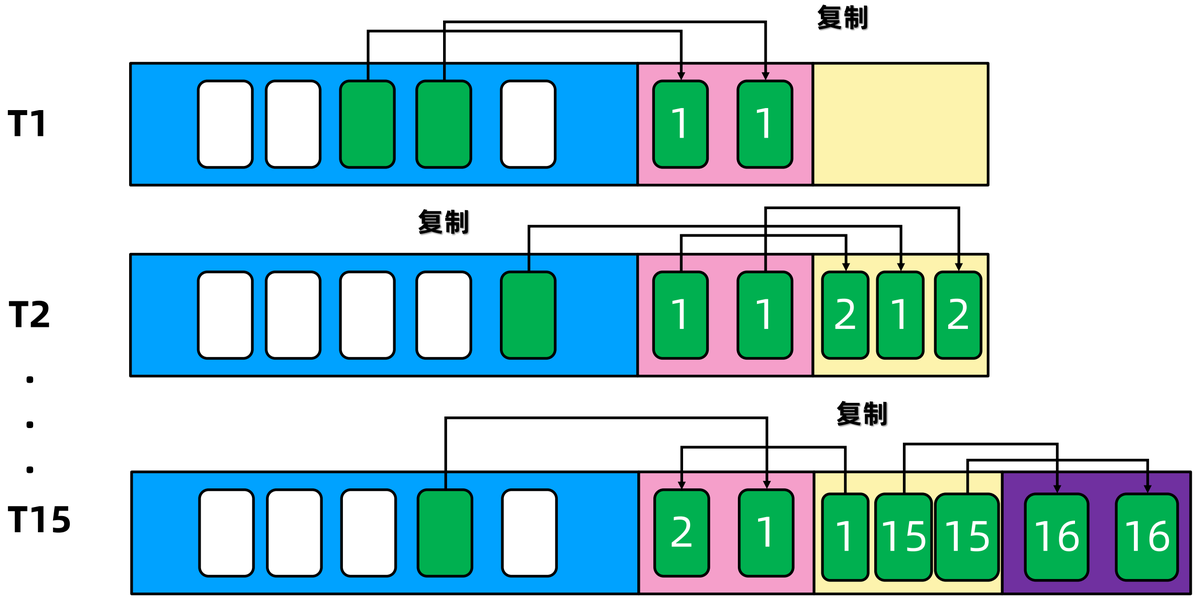
- **CAS乐观锁：**JVM虚拟机采用CAS失败重试的方式保证更新操作的原子性【并发编程CAS原理详解】
- TLAB （Thread Local Allocation Buffer） 本地线程分配缓存，预分配

分配主流程：首先从TLAB里面分配，如果分配不到，再使用CAS从堆里面划分

对象内存首先分配到哪里？对象怎么进入老年代的呢？

4.3 对象怎样才会进入老年代





对象怎样才会进入老年代？

- 新对象大多数默认都进入Eden
- 对象进入老年代的四种情况
  1. **年龄太大 MinorGC15次**【-XX:MaxTenuringThreshold】
  2. **动态年龄判断：**MinorGC后会动态判断年龄，将符合要求对象移入老年代
  3. **大对象**直接进入老年代**1M**【 -XX:PretenureSizeThreshold 】
  4. **MinorGC后存活对象太多无法放入Survivor**

案例：Talk is cheap Show me the code

4.4 对象内存布局

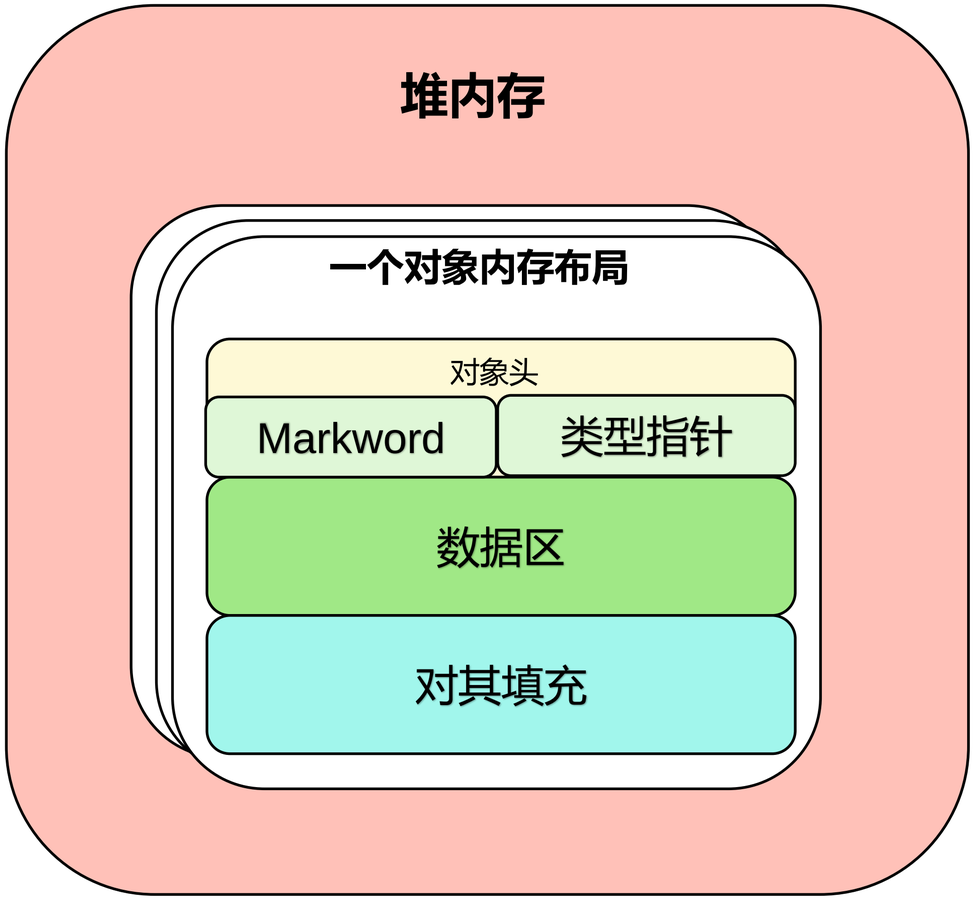
对象里的三个区：

01-对象头：8字节，如果是数组12字节

- 标记字段：存储对象运行时自身数据
  - 默认：对象Hashcode，GC分代年龄，锁状态
  - 存储数据结构并不是固定的
- 类型指针：对象指向类元数据的指针
  - 开启指针压缩占4字节，不开启8字节
- 数组长度：如果是数组，则记录数组长度，占4字节
- 对其填充：保证数组的大小永远是8字节的整数倍

02-实例数据：对象内部的成员变量

03-对其填充：8字节对象，保证对象大小是8字节的整数倍



标记字段的存储数据为什么会变化呢？

4.4 对象内存布局

Markword是可变的数据结构，对象头总大小固定8字节

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
无锁状态	对象Hashcode、对象GC分代年龄				01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空，不需要记录信息				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

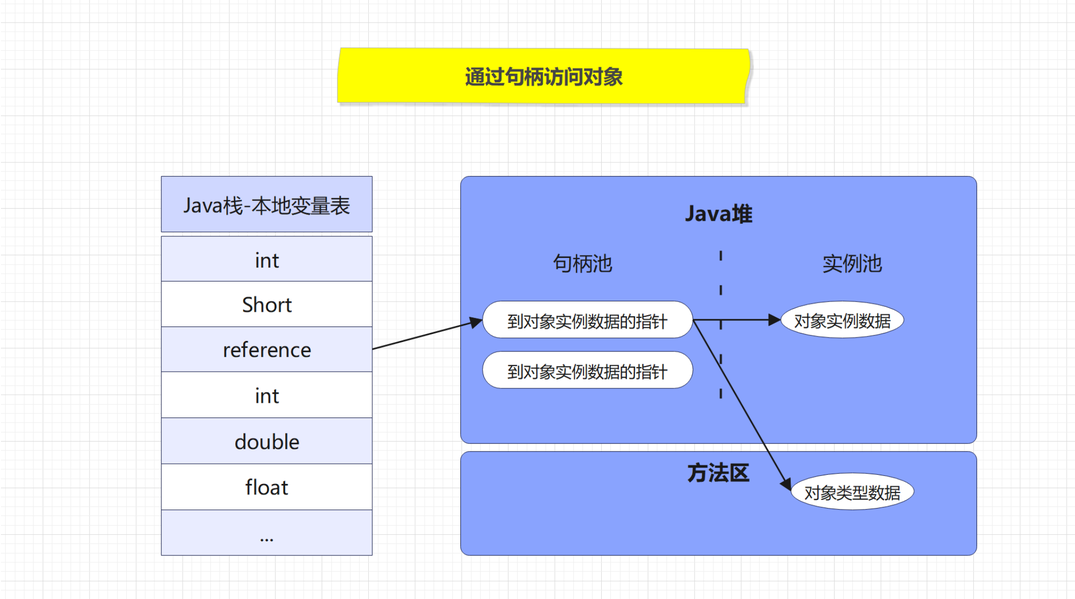
案例：打印对象内存布局信息

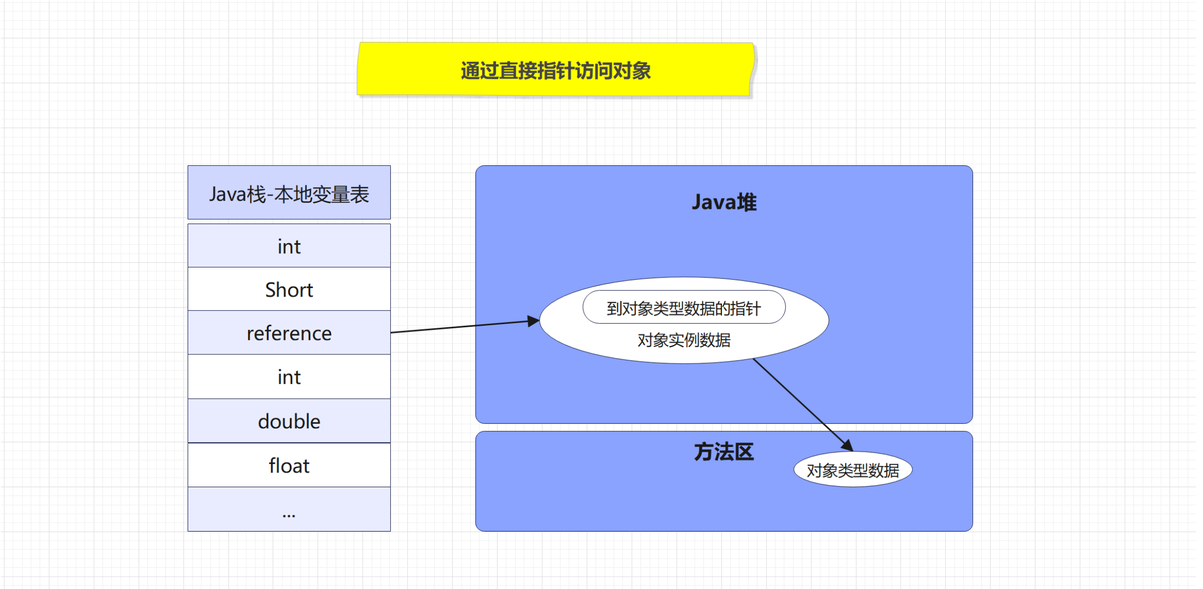
4.5 如何访问一个对象

有两种方式：

① 句柄

② 直接指针

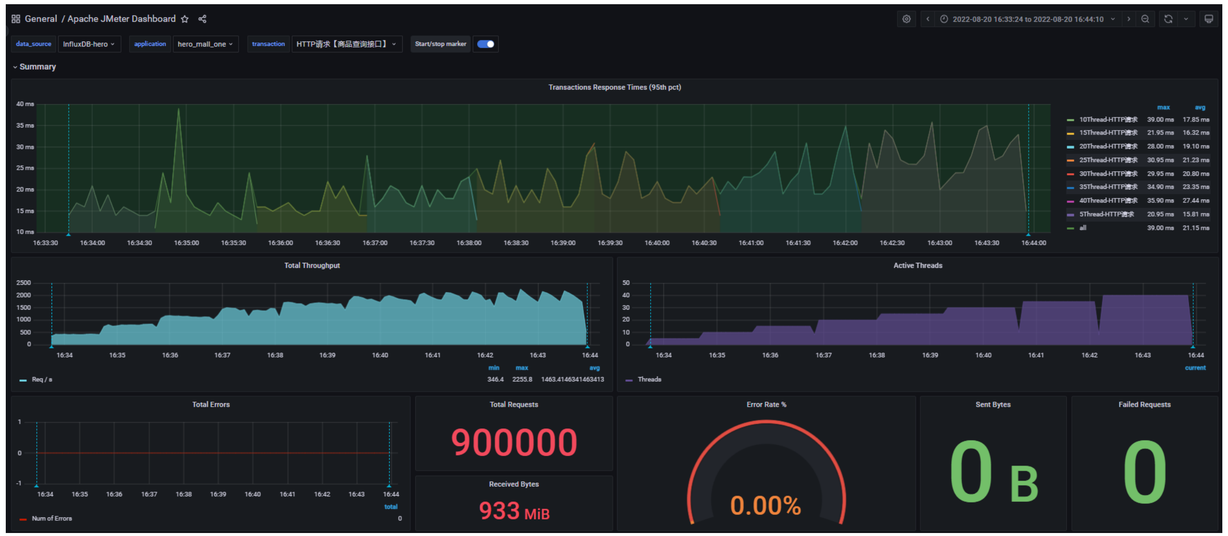




五、垃圾收集器

5.1 GC基本原理

01-为什么垃圾回收呢？



02-什么是垃圾呢？

- 没有被**引用**对象

03-如何找到这个垃圾呢？

- **引用计数法（Reference Counting）**
- **根可达分析算法（GCRooting Tracing）**

为什么会有两种找垃圾的方式？有什么区别呢

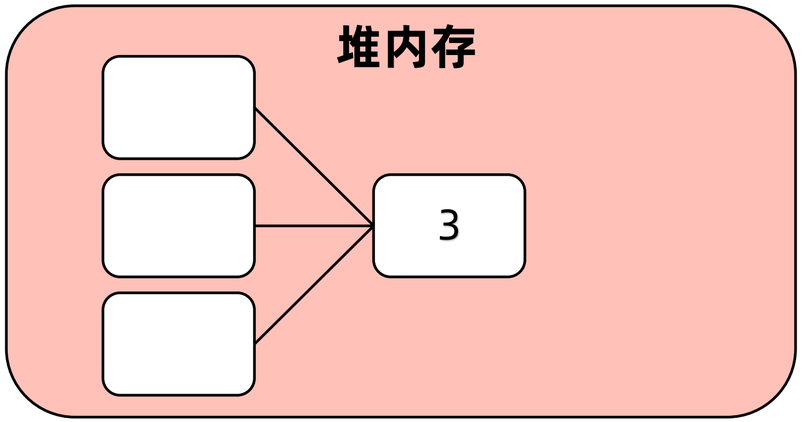
5.1 GC基本原理-如何找到垃圾？

01-引用计数法（Reference Counting）

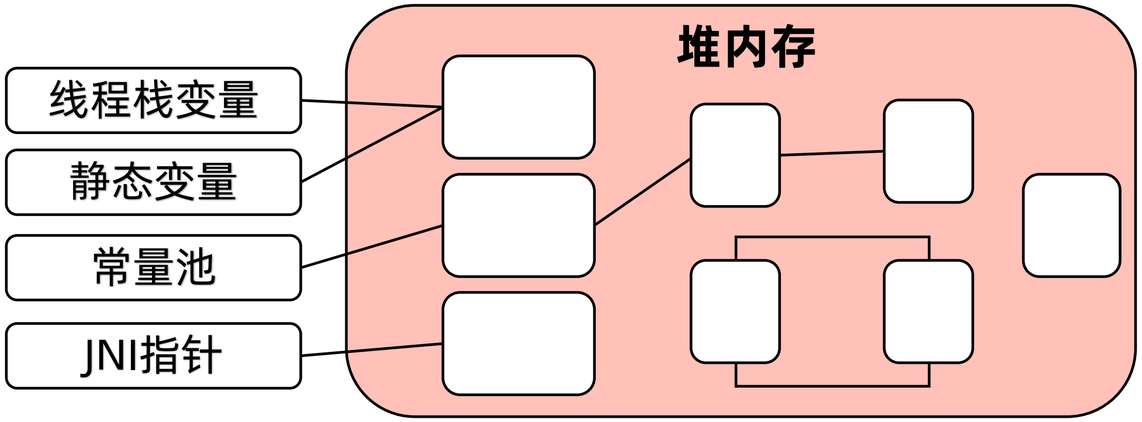
当对象引用消失，对象就称为垃圾

堆内存中主要存在三种引用关系：

- 单一引用
- 循环引用
- 无引用



引用计数法有无问题？



5.1 GC基本原理-如何找到垃圾？

02-根可达算法：

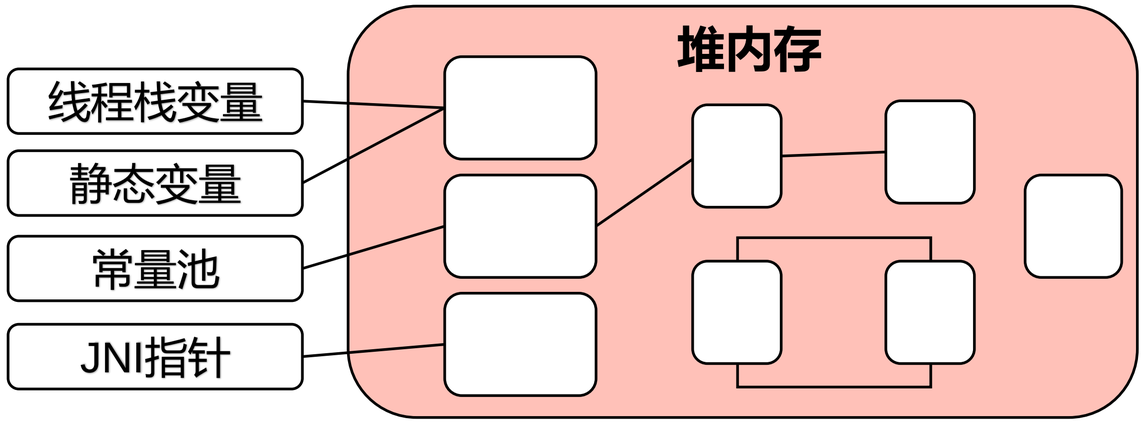
通过GCRoots作为对象起点向下搜索，当一个对象到GCRoots没有任何**引用链**时，**此对象是垃圾**。

引用链（ReferenceChain）：GCRoots搜索走过的路径

什么是GCRoots呢？

- 虚拟机栈中，栈帧本地变量表引用的对象
- 方法区中，类静态属性引用的对象
- 方法区中，常量引用对象
- 本地方法栈中，JNI引用的对象
- 虚拟机内部的引用

是垃圾就必须得立即死吗？



5.1 GC基本原理-如何找到垃圾？

03-垃圾对象死亡前至少经历两次标记：

- 第一次：可达性分析后，没有引用链对象会被第一次标记
- 第二次：标记后的对象会经历筛选，如果筛选不通过，则会被第二次标记

5.1 GC基本原理-如何找到垃圾？

04-对象引用有哪些？

- JDK1.2之后，Java对象的引用进行了扩充：**强引用，软引用**，弱引用，虚引用

引用类型	被垃圾回收时间	用途	生存时间
强引用	**从来不会**	**对象的一般状态**	**JVM停止时终止**
软引用	内存不足时	对象缓存	内存不足时终止
弱引用	正常GC	对象缓存	GC后终止
虚引用	正常GC	类似事件回调机制	GC后终止
无引用	**正常GC**	**对象的一般状态**	**GC后终止**

强引用：

```
1 Object obj = new Object();
```

软引用：

```
1 SoftReference<Object> sf = new SoftReference<Object>(obj);
```

弱引用：

```
1 WeakReference<Object> wf = new WeakReference<Object>(obj);
```

虚引用：

```
1 PhantomReference<Object> pf = new PhantomReference<Object>(obj, new ReferenceQueue<>());
```

这些引用类型有啥用？

5.2 GC基本原理-如何清除垃圾？

三种清除垃圾算法：

- 标记-清除算法（Mark-Sweep）
- 复制算法（ Copying ）
- 标记-整理算法（Mark-Compact）

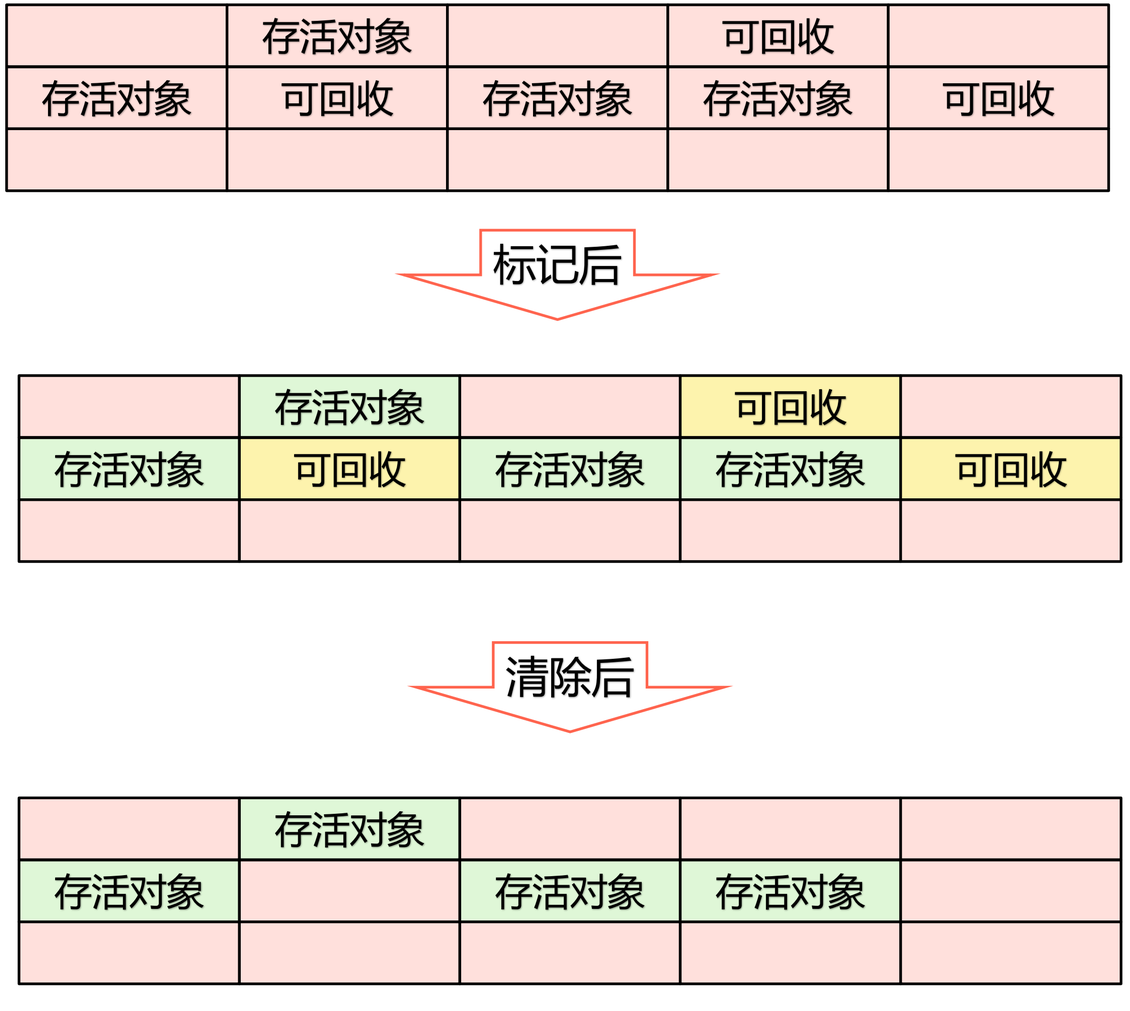
垃圾是找到了，那么咱们怎么干掉它呢？

5.2 GC基本原理-如何清除垃圾？

第一种：标记-清除算法（Mark-Sweep）

- 分为**标记**和**清除**两个阶段：
  - 标记：标记出所有需要回收对象
  - 清除：统一回收掉所有对象

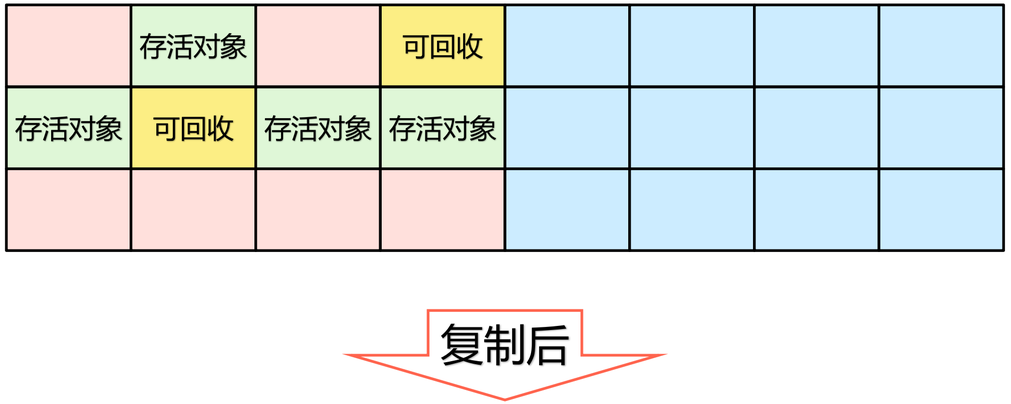
- **缺点：**
  - 执行效率不稳定
  - 空间碎片：会产生大量不连续内存碎片



5.2 GC基本原理-如何清除垃圾？

第二种：复制算法（Copying）

- 内存分为两块，清除垃圾时，将存活对象复制到另一块
- S0和S1区就是基于这个算法诞生的
- Eden:S = 8:2
- 不用担心S区不够，因为Old是担保人



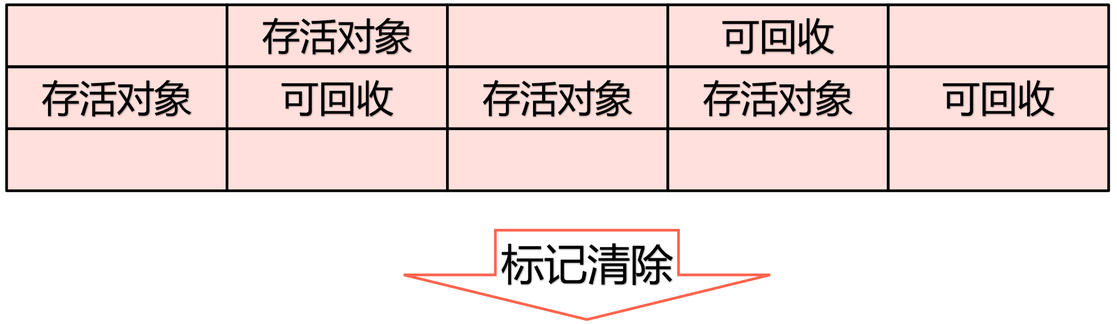
优缺点：

- 优点：没有内存空间碎片化
- 缺点：存在空间浪费

5.2 GC基本原理-如何清除垃圾？

第三种：标记-整理算法（Mark-Compact）

- 标记：标记出所有需要回收对象
- 清除：统一回收掉所有对象
- 整理：将所有存活对象向一端移动



优缺点：

- 优点：空间没有浪费，没有内存碎片化问题
- 缺点：性能较低

分代回收（Generational Collection）

- 新生代：选择**复制算法**，**弱分代假说**
- 老年代：选择**标记-清除**或**标记-整理**，**强分代假说**

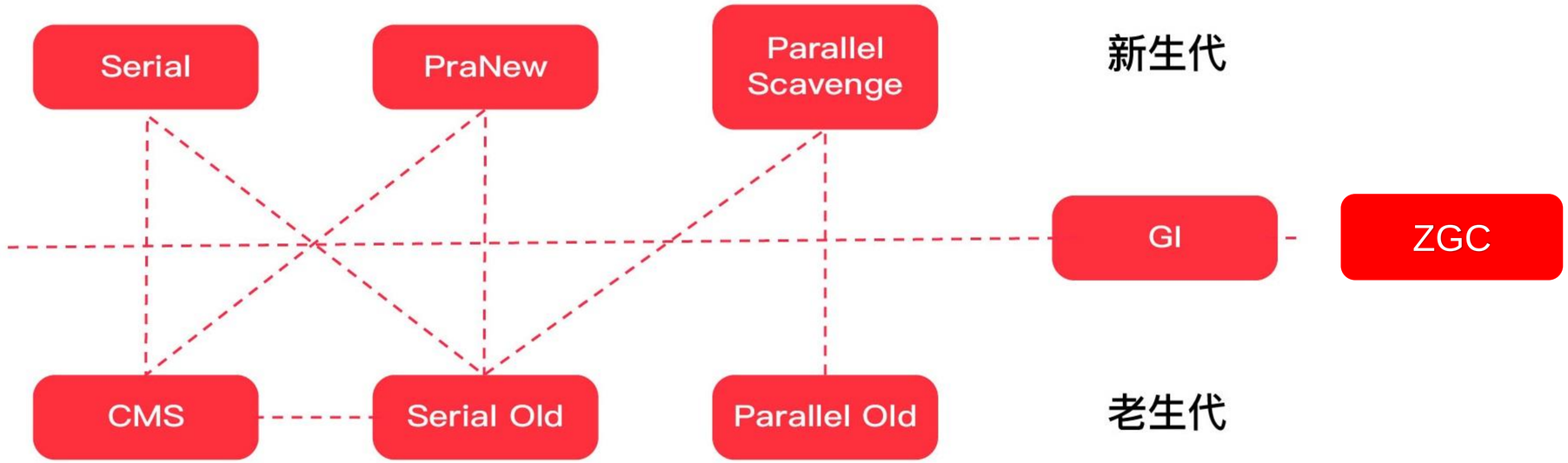
5.2 GC基本原理-用什么清除垃圾？

有 8 种不同的垃圾回收器，它们分别用于不同分代的垃圾回收

新生代（复制算法）：Serial，ParNew，Parallel Scavenge

老年代（标记-清除、标记-整理）：SerialOld，Parallel Old，CMS

整堆：G1，ZGC

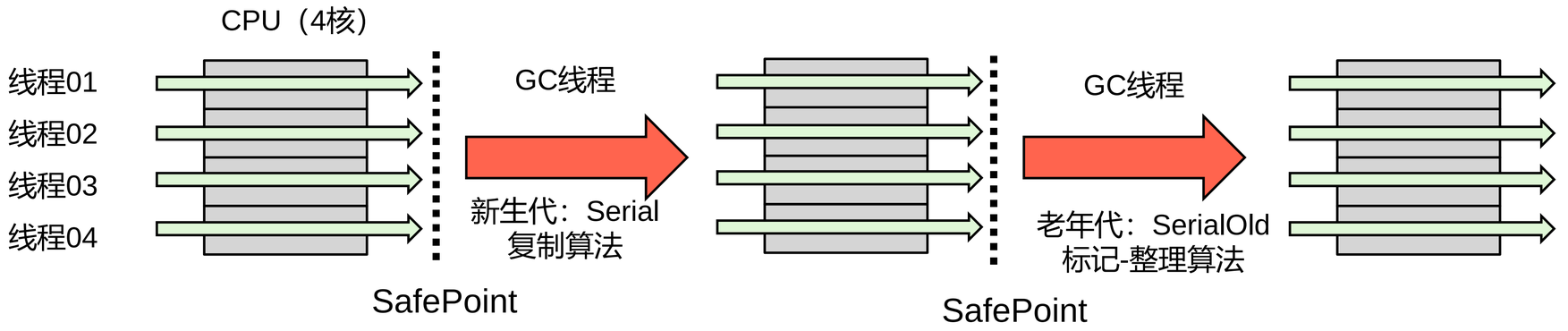


5.3 GC-串行收集器-Serial与SerialOld

配置参数： **-XX:+UseSerialGC**

特点：

- **Serial**新生代收集器，单线程执行，使用复制算法
- **SerialOld**老年代收集器，单线程执行，使用标记-整理算法
- 进行垃圾收集时，必须暂停用户线程



什么是SafePoint?

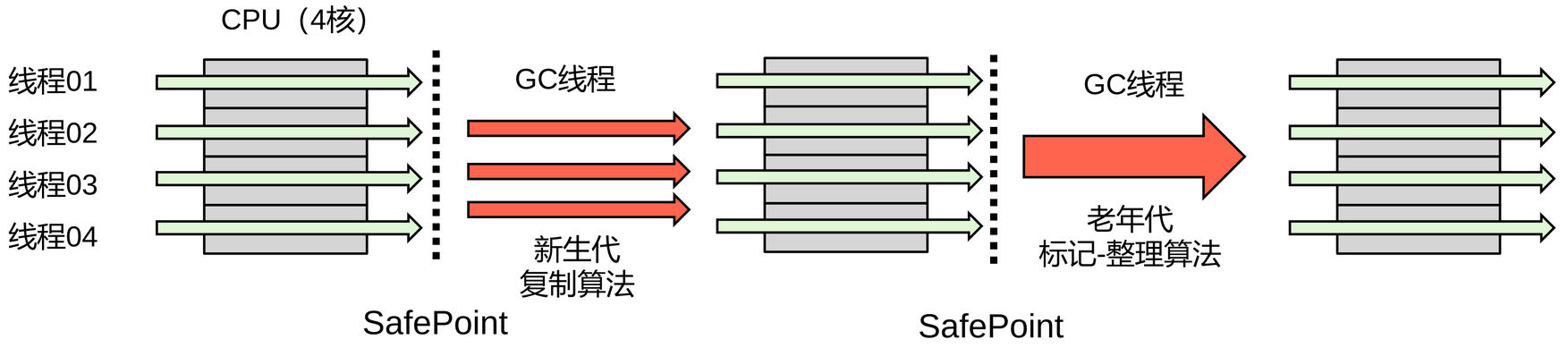
5.3 GC-并行收集器-Parallel Scavenge(Stop)

配置参数： **-XX:+UseParallelGC**

$$吞吐量=\frac{运行用户代码时间}{运行用户代码时间+运行垃圾收集时间}$$

特点：简称PS

- **吞吐量优先收集器**，垃圾收集需要暂停用户线程
- 新生代使用并行回收器，采用复制算法
- 老年代使用串行收集器，采用标记-整理算法



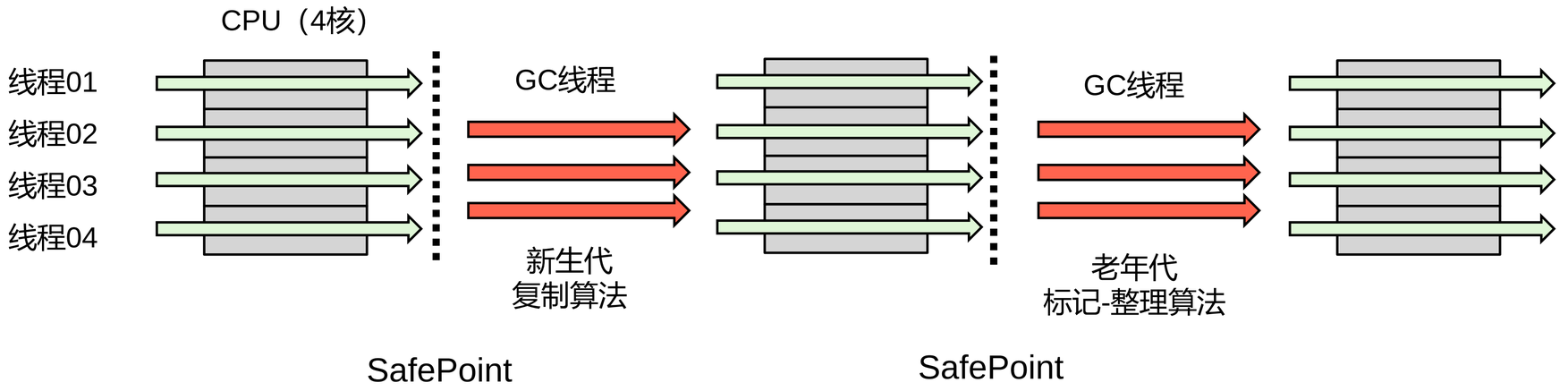
什么是Throughput?

5.3 GC-并行收集器-Parallel Old

配置参数： **-XX:+UseParallelOldGC**

特点：

- **PS收集器的老年代版本**
- **吞吐量优先收集器**，垃圾收集需要暂停用户线程，对CPU敏感
- 老年代使用并行收集器，采用标记-整理算法



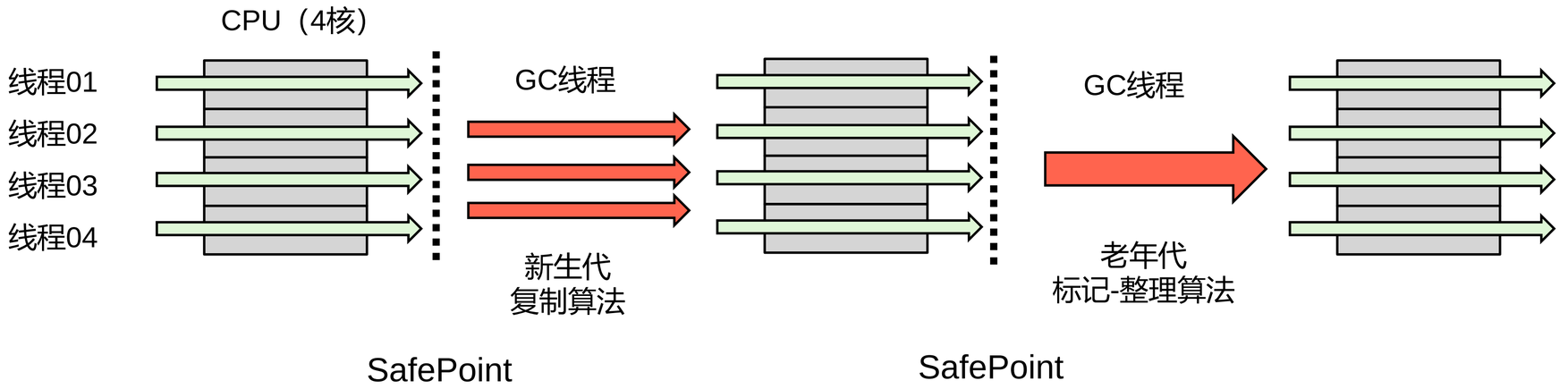
5.3 GC-并行收集器-ParNew

配置参数： -XX:+UseParNewGC

配置参数：-XX:ParallelGCThreads=n，垃圾收集线程数

特点：

- **新生代并行ParNew，老年代串行SerialOld**
- Serial的多线程版
- 单核CPU不建议使用

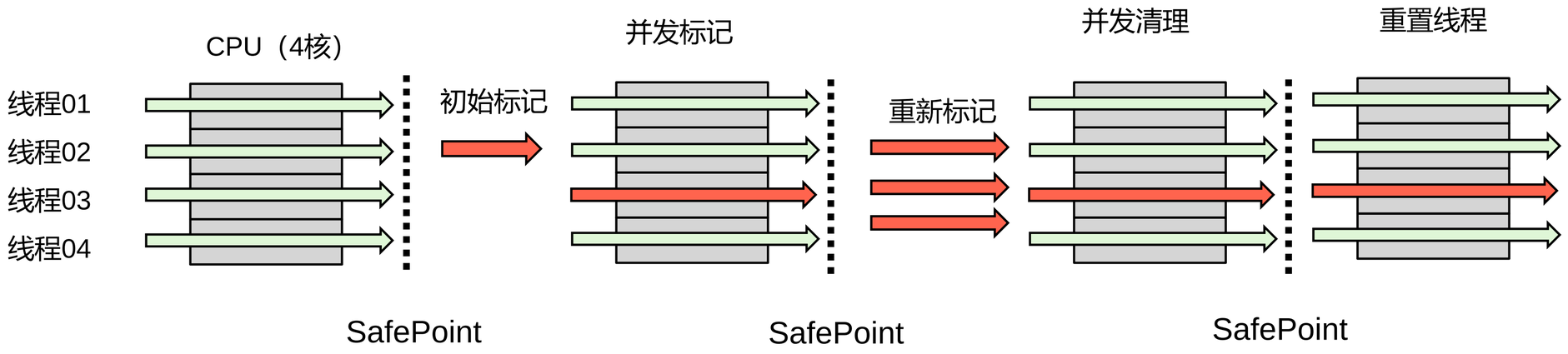


5.3 GC-并行收集器-CMS

配置参数： **-XX:+UseConcMarkSweepGC**

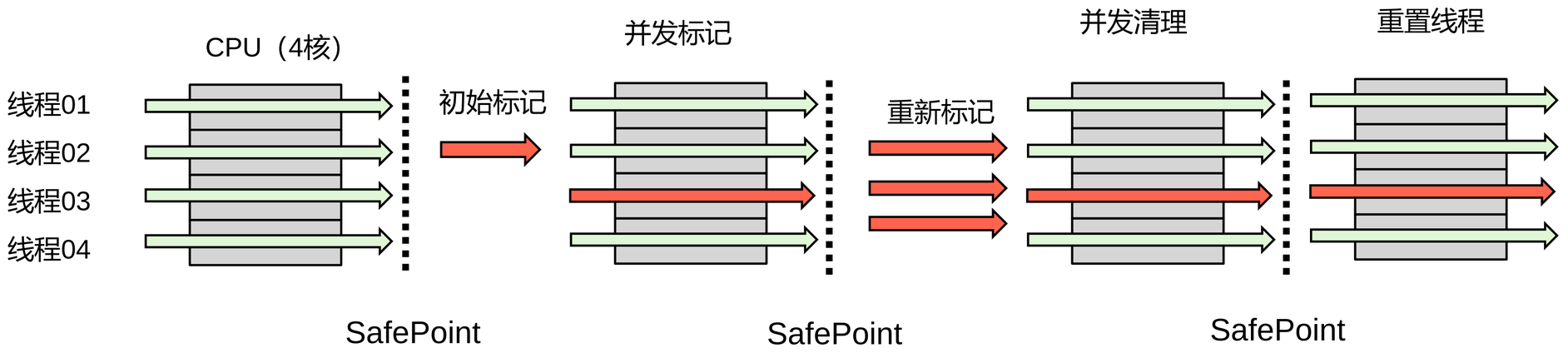
特点：

- **低延时，减少STW对用户的影响**
- 并发收集，用户线程与收集线程一起执行，对CPU资源敏感
- 不会等待堆填满再收集，到达阈值就开始收集。
- 采用标记-清除算法，所以会产生内存碎片



老年代：**标记-清除**算法

5.3 GC-并行收集器-CMS-详解



老年代：**标记-清除**算法

01-初始标记阶段：会STW，标记出GCRoots可以关联到的对象，关联对象较少，所以很快

02-并发标记阶段：不会STW，遍历GCRoots直接对象的引用链，耗时长

03-重新标记阶段：会STW，修正并发标记期间的新对象记录

04-并发清除阶段：不会STW，清除垃圾对象，释放内存空间

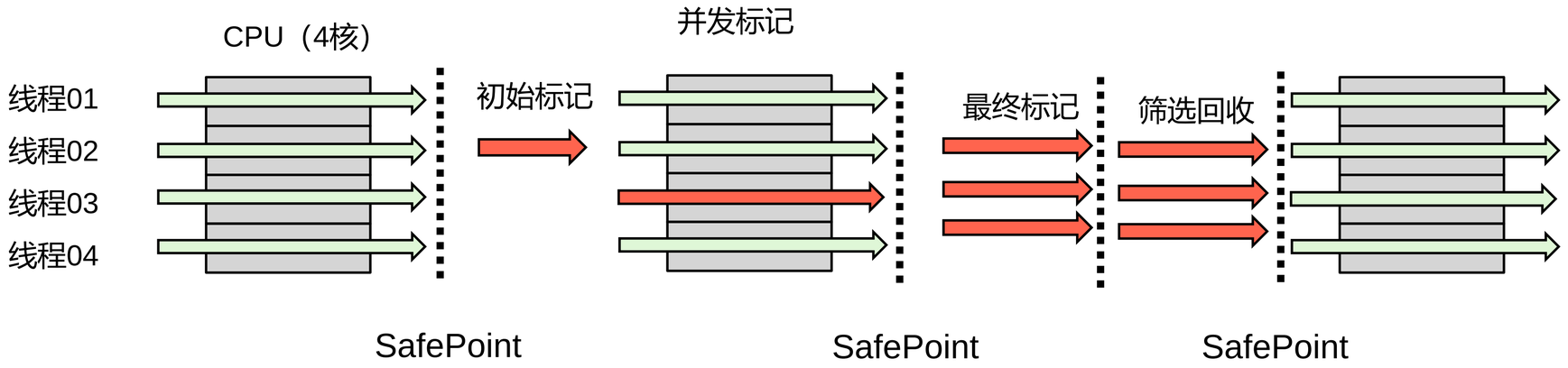
5.3 GC-并行收集器-Garbage-First

G1是一款面向服务端应用的全功能型垃圾收集器，大内存企业配置的主要是G1。

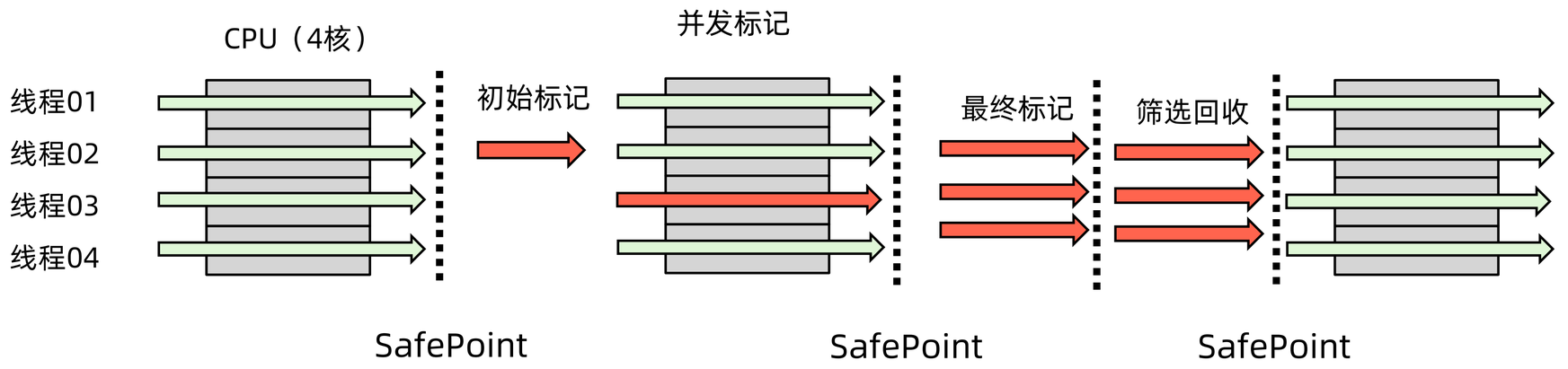
配置参数： -XX:+UseG1GC

特点：G1

- 吞吐量和低延时都行的**整堆垃圾收集器**
- **G1最大堆内存**32M*2048=64GB，**最小堆内存**1M*2048=2GB，低于此值不建议使用
- 全局使用**标记-整理算法**收集，局部采用**复制算法**收集
- 可预测的停顿：能让使用者指定GC消耗时间，默认是200ms



5.3 GC-并行收集器-Garbage-First-详解



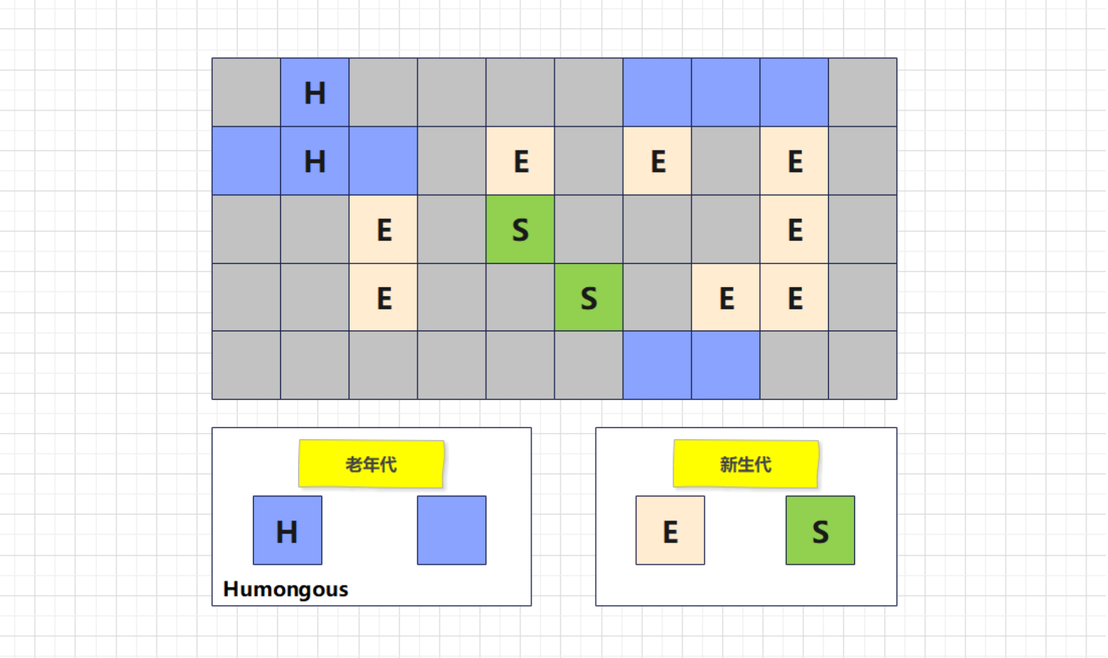
01-初始标记：会STW，标记出GCRoots可以关联到的对象，耗时短

02-并发标记：不会STW，遍历GCRoots直接对象的引用链，耗时长

03-最终标记：会STW，修正并发标记期间，标记产生变动的那部分

04-筛选回收：会STW，对各个Region的**回收价值和成本排序**，根据用户期望GC停顿时间确定回收计划

5.3 GC-并行收集器-Garbage-First-内存划分




```
-XX:+UseG1GC
# 使用 G1 垃圾收集器
-XX:MaxGCPauseMillis=
# 设置期望达到的最大GC停顿时间指标（JVM会尽力实现，但不保证达到），默认值是 200 毫秒。
-XX:G1HeapRegionSize=n
# 设置的 G1 区域的大小。值是 2 的幂，范围是 1 MB 到 32 MB 之间。
# 目标是根据最小的 Java 堆大小划分出约 2048 个区域。
# 默认是堆内存的1/2000。
-XX:ParallelGCThreads=n
# 设置并行垃圾回收线程数，一般将n的值设置为逻辑处理器的数量，建议最多为8。
-XX:ConcGCThreads=n
# 设置并行标记的线程数。将n设置为ParallelGCThreads的1/4左右。
-XX:InitiatingHeapOccupancyPercent=n
# 设置触发标记周期的 Java 堆占用率阈值。默认占用率是整个 Java 堆的 45%。
```

取消新生代与老年代的物理划分：采用若干个固定大小的Region

Region区类型：在逻辑上有Eden、Survivor、Old、Humongous

垃圾收集算法：全局采用标记-整理算法，局部采用复制算法

Humongous区域：当对象的容量超过了Region的50%，则被认为是巨型对象。

5.3 GC-并行收集器-ZGC

ZGC （ Z Garbage Collector ）在 JDK11中引入的一种**可扩展的低延迟垃圾收集器**，在 JDK15中发布稳定版。

配置参数： -XX:+UseZGC

特点：

- **< 1ms 最大暂停时间（JDK16-是10ms，JDK16+是<1ms ），不会随着堆内存增加而增加**
- **适合内存8MB，16TB**
- 并发，基于Region，压缩，NUMA感知，使用色彩指针，使用负载屏障
- 垃圾收集算法：标记-整理算法
- 主要目标：低延时

相关参数:

```
-XX:+UseZGC # 启用 ZGC
-Xmx        # 设置最大堆内存
-Xlog:gc    # 打印 GC日志
-Xlog:gc*   # 打印 GC 详细日志
```

THANKS

教育不是注满一桶水，而是点燃一把火