

# JVM 内存管理 & 实践

郭宁@猫眼电影

guoning02@maoyan.com

2017/03/07

# 分享内容

- JVM 内存结构
- JVM 内存回收
- HotspotVM
  - 内存回收，具体实现
  - 参数调优
- JVM 常见问题与排查方法

怎么**存放**数据？

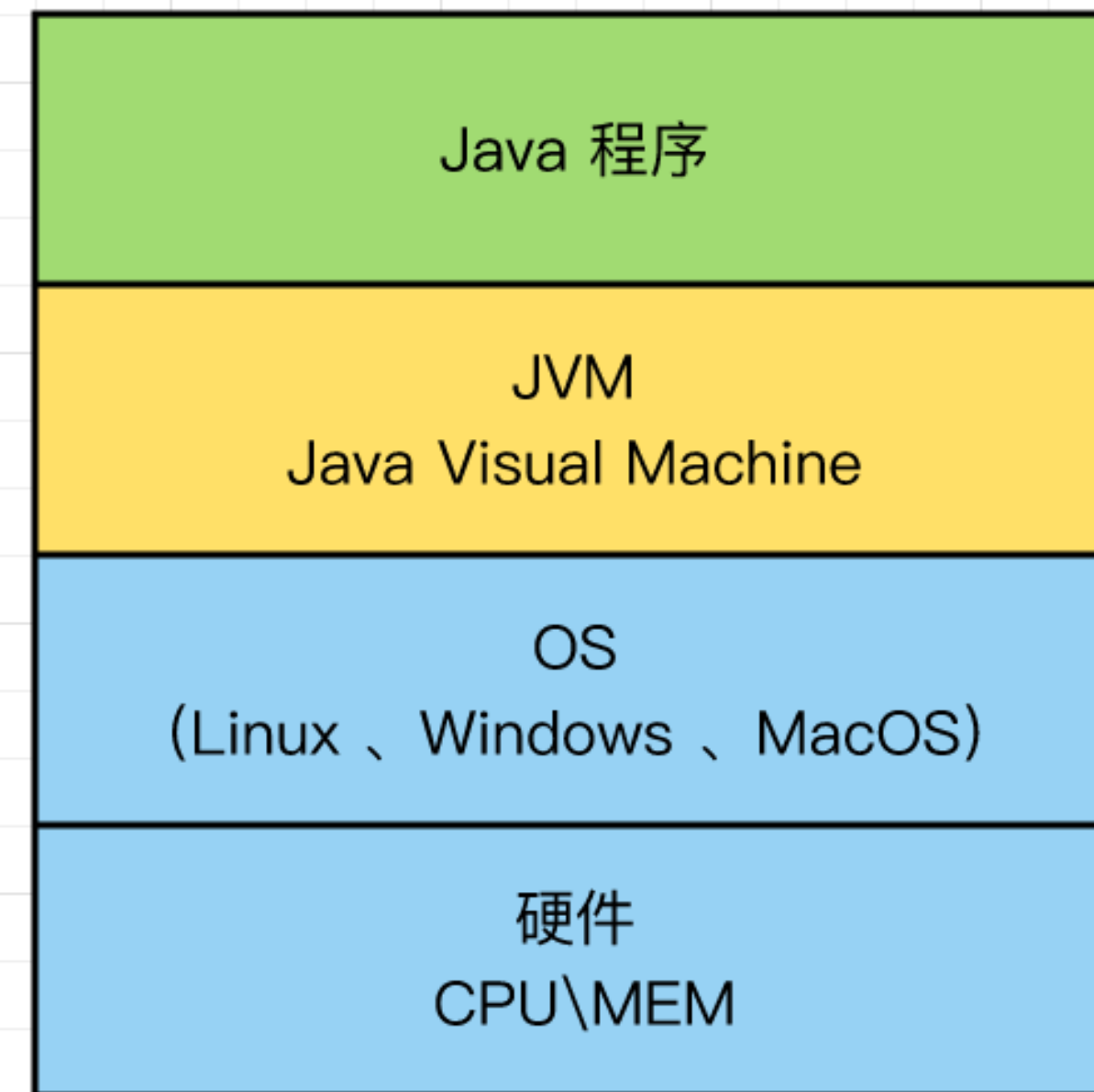
怎么**回收**数据？

具体**实现**，核心参数

**实践**

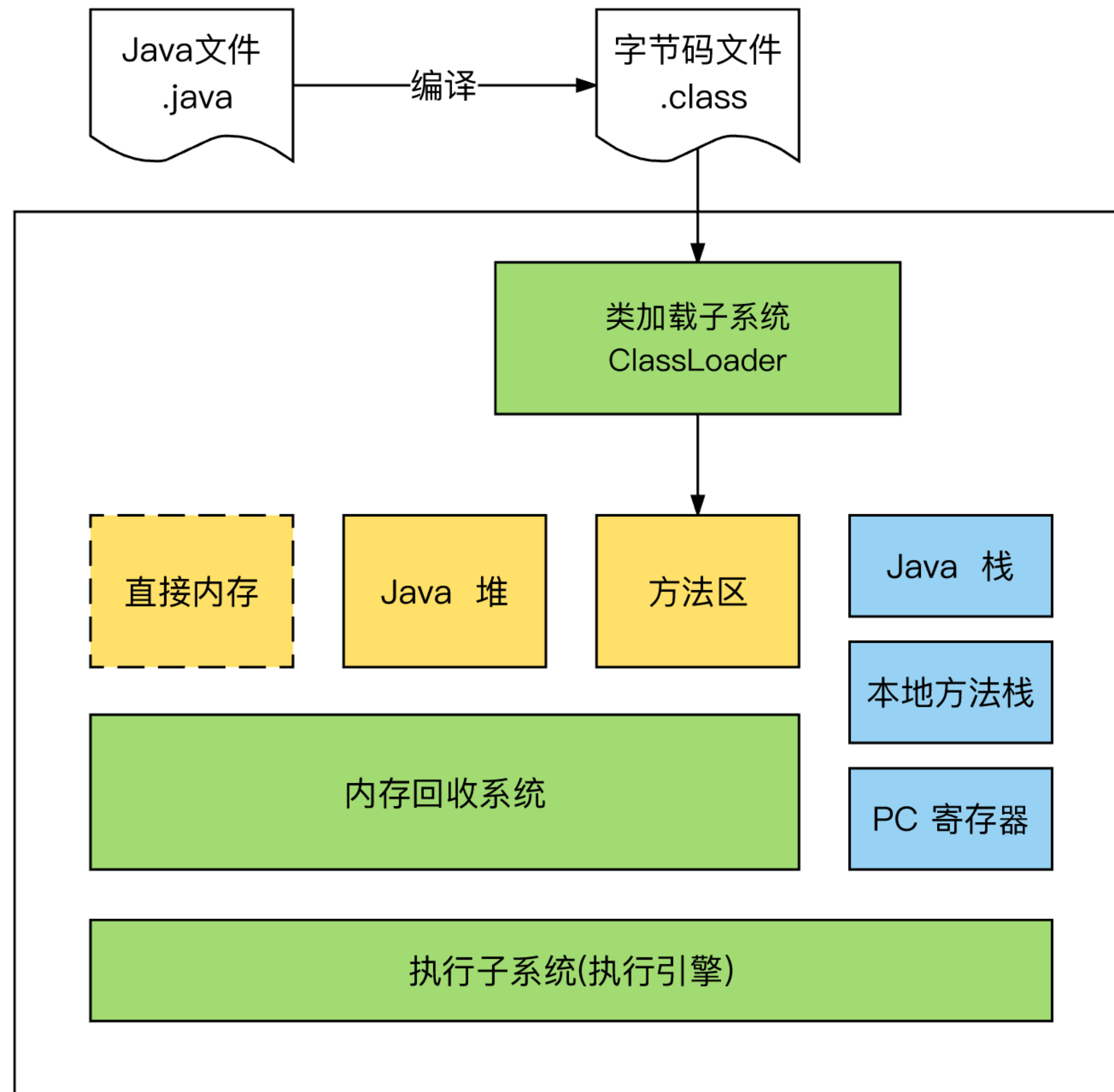
# JVM 内存结构：JVM 简介

- 什么是 JVM?
  - 对操作系统（OS）：JVM 是一个**应用程序**，一个**进程**
  - 对 **Java 代码**：Java 代码的**运行环境**，实现**跨平台**
- JVM 版本：
  - 不同厂商有不同版本的 JVM：Sun、IBM 等
  - **Hotspot** 是比较流行的版本（Oracle）
    - 其他 JVM：
      - Sun 早期的 Classic
      - IBM 的 J9
      - Oracle 的 JRockit



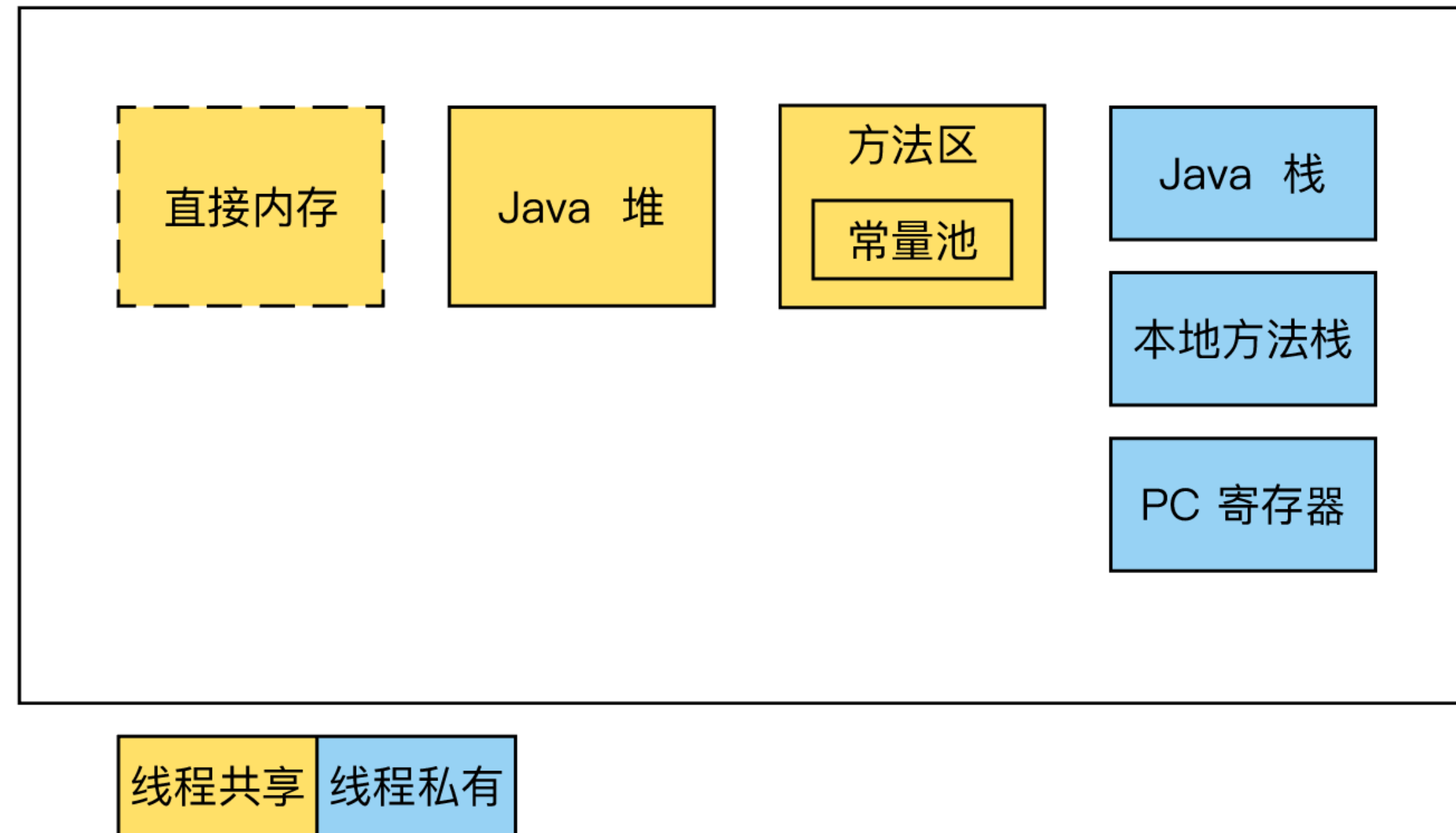
JVM 作用：  
1. Java 代码跨平台  
2. 内存回收

# JVM 内存结构：Java 代码执行过程



- JVM 整体由 4 部分组成：
  1. **加载**：类加载器 ClassLoader
  2. **执行**：执行引擎
  3. **内存**：运行时数据区，Runtime Date Area
  4. **内存回收**：垃圾回收

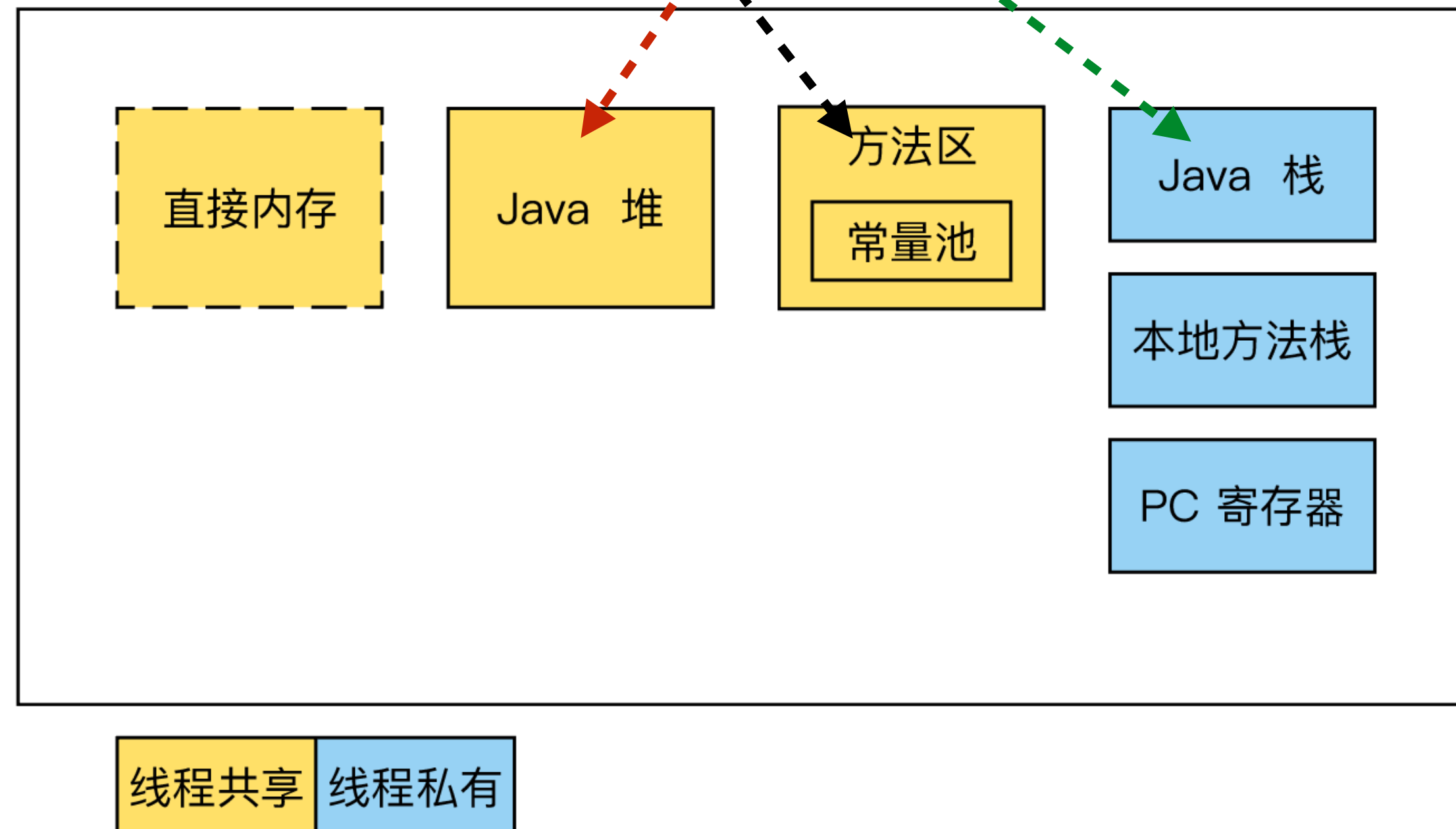
# JVM 内存结构：运行时数据区



- 方法区：
  1. 类：Class
  2. 静态变量
  3. 常量池（字符串常量、数字常量）
- Java 堆：
  1. 对象：Object
  2. 数组
- Java 栈：Java 方法调用过程
  1. 操作数栈
  2. 局部变量表
  3. 方法出口
- 本地方法栈：本地方法调用过程
- 程序计数器：Program Counter

# JVM 内存结构：运行时数据区

```
void sayHello(){  
    A a = new A();  
}
```



- JVM 内存空间：

1. **线程共享：**

- Java 堆
- 直接内存
- 方法区

2. **线程独占：**

- Java 栈
- 本地方案栈
- PC 寄存器

# JVM 内存结构：小结

- Tips:
  - JVM 是什么？用来做什么？
  - JVM 上，Java 代码执行的过程
  - JVM 上，进程、线程，对应内存空间

# 分享内容

- JVM 内存结构
- JVM 内存回收
- HotspotVM
  - 内存回收，具体实现
  - 参数调优
- JVM 常见问题与排查方法



怎么**存放**数据?



怎么**回收**数据?



# JVM 内存回收(GC): 为什么要回收?

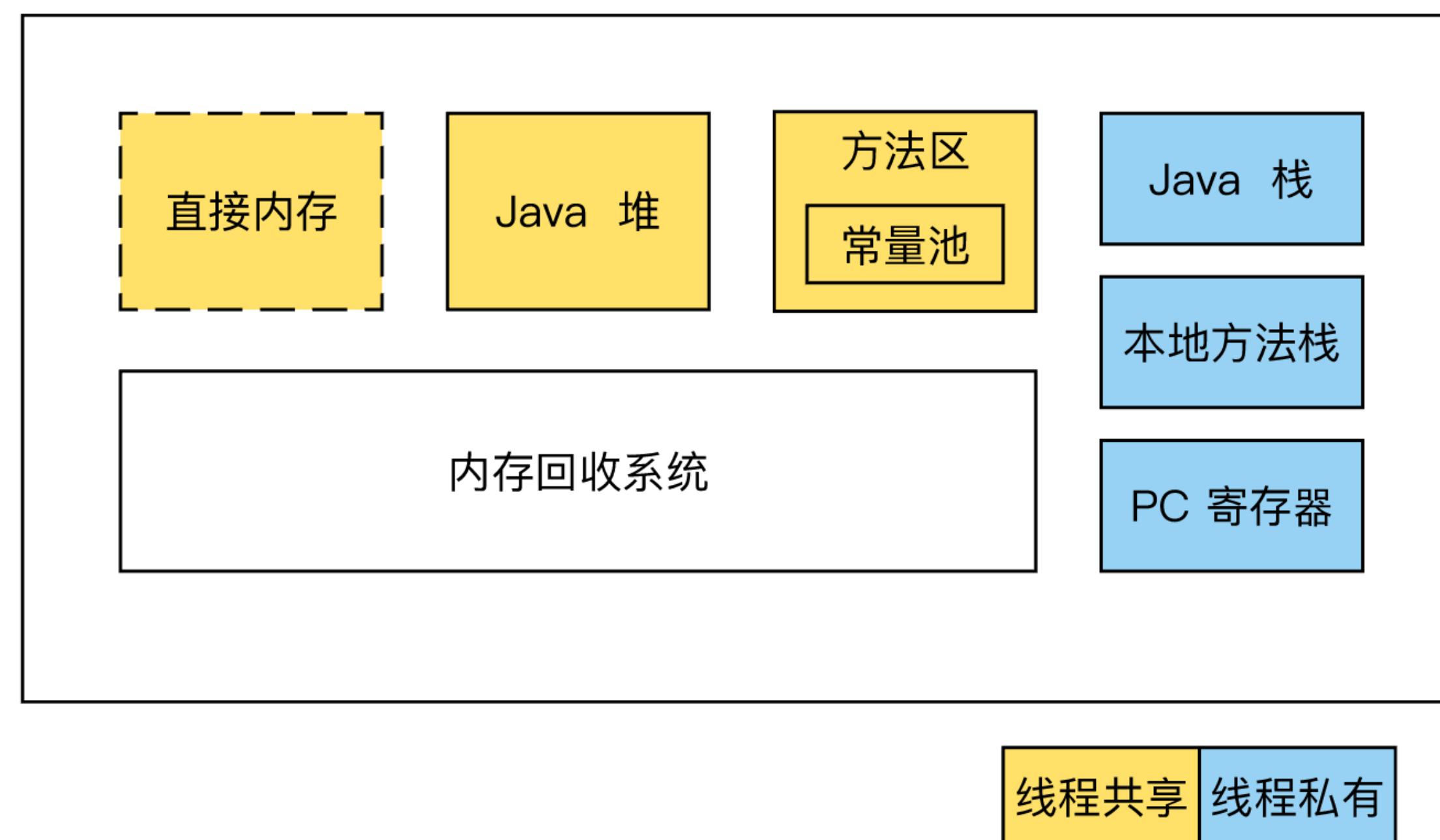
- 背景:
  - 「**已被占用**」的内存, 只有「**被释放**」, 才能再次使用
  - 不释放内存, **内存泄露**

- Java 代码, 运行在 JVM 上:
  - 由 JVM 负责内存回收, **自动回收**
  - Garbage Collection: 垃圾回收

- C 代码, 申请到的内存, 必须**手动释放**:
  - delete : 释放 new 分配的空间
  - free: 释放 malloc 分配的空间

# JVM 内存回收(GC)

- 内存回收，**目标**：回收**不再使用**的内存，释放空间，防止内存泄漏
- GC 的核心问题：
  1. 回收哪些内存？
  2. 如何回收内存？
  3. 回收内存时，是否需要暂停服务？



# 核心问题一：回收哪些内存

- **核心问题一：回收哪些内存？**
  - **标准：**已被占用，但**不会再被使用**的内存
  - JVM 内存**分配的粒度：****对象、基础类型**
- **判断策略：**
  - **引用计数**
  - **根结点可达（根搜索）：**哪些是根结点

# 核心问题一：回收哪些内存

- 引用计数：

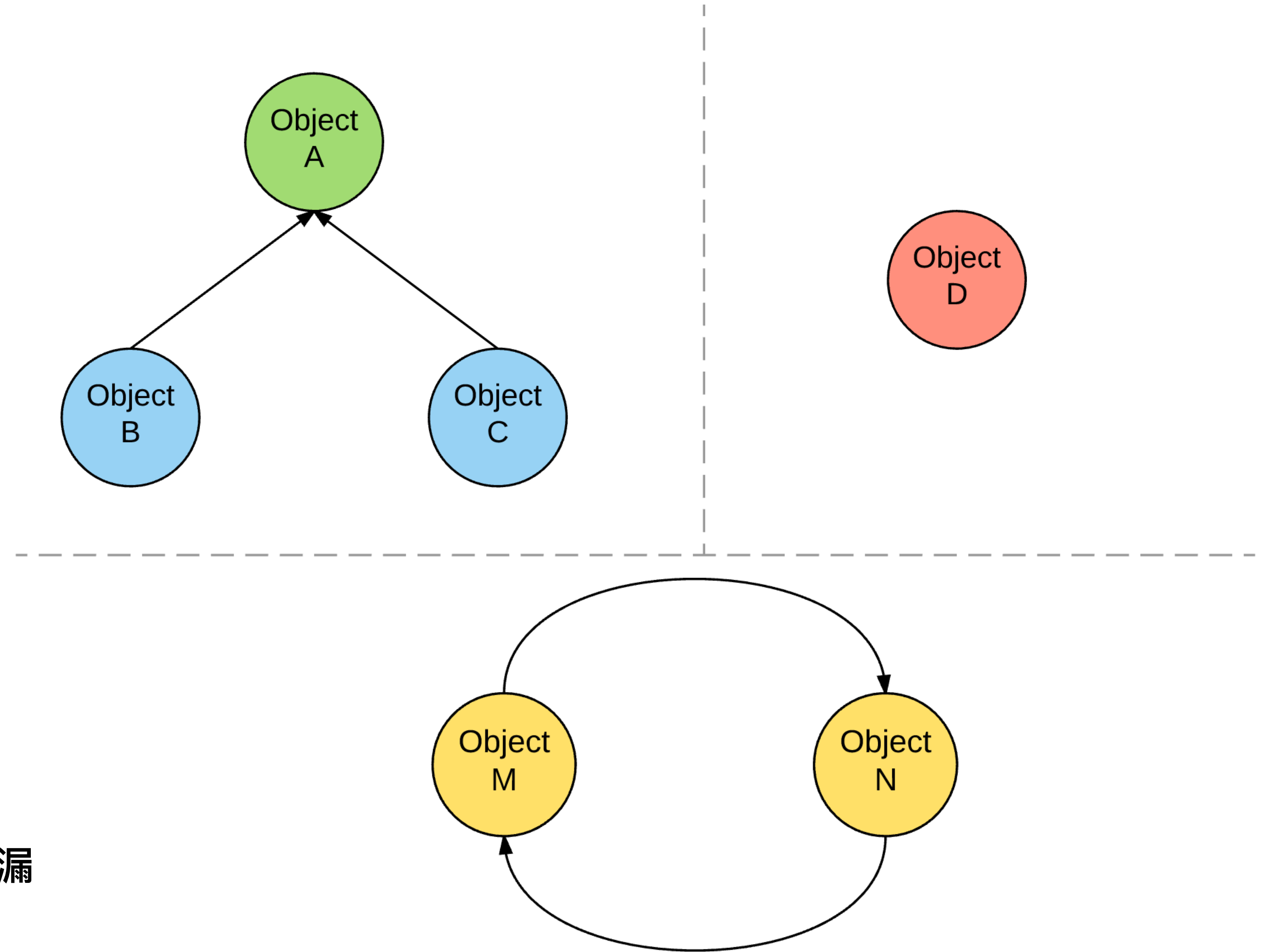
- Object 每次**被引用**，计数加「+1」
- Object 每次被**释放引用**，计数「-1」
- 判断 Object 的引用次数「=0」

- 优点：

- 判断简单
- 算法效率高

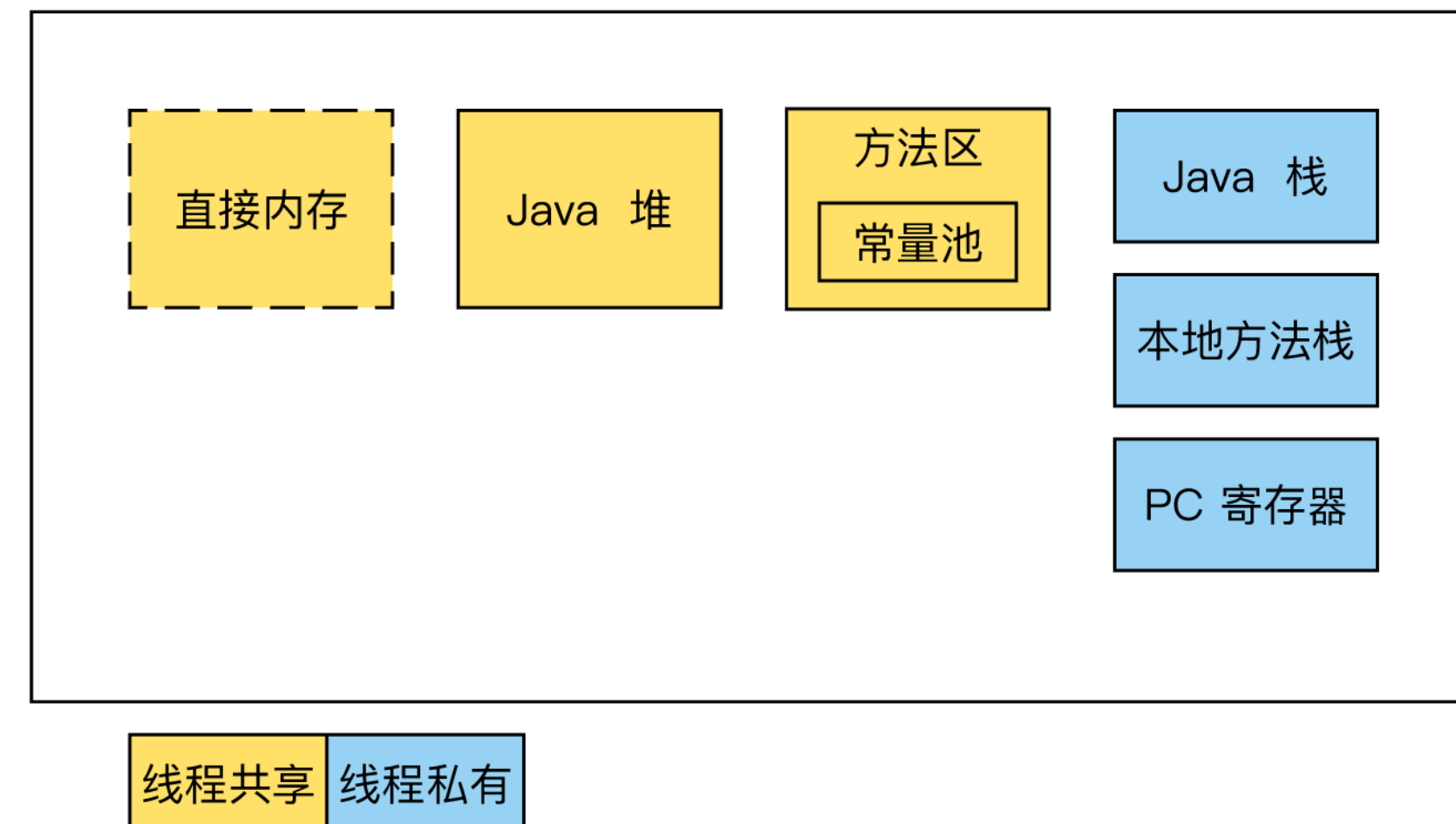
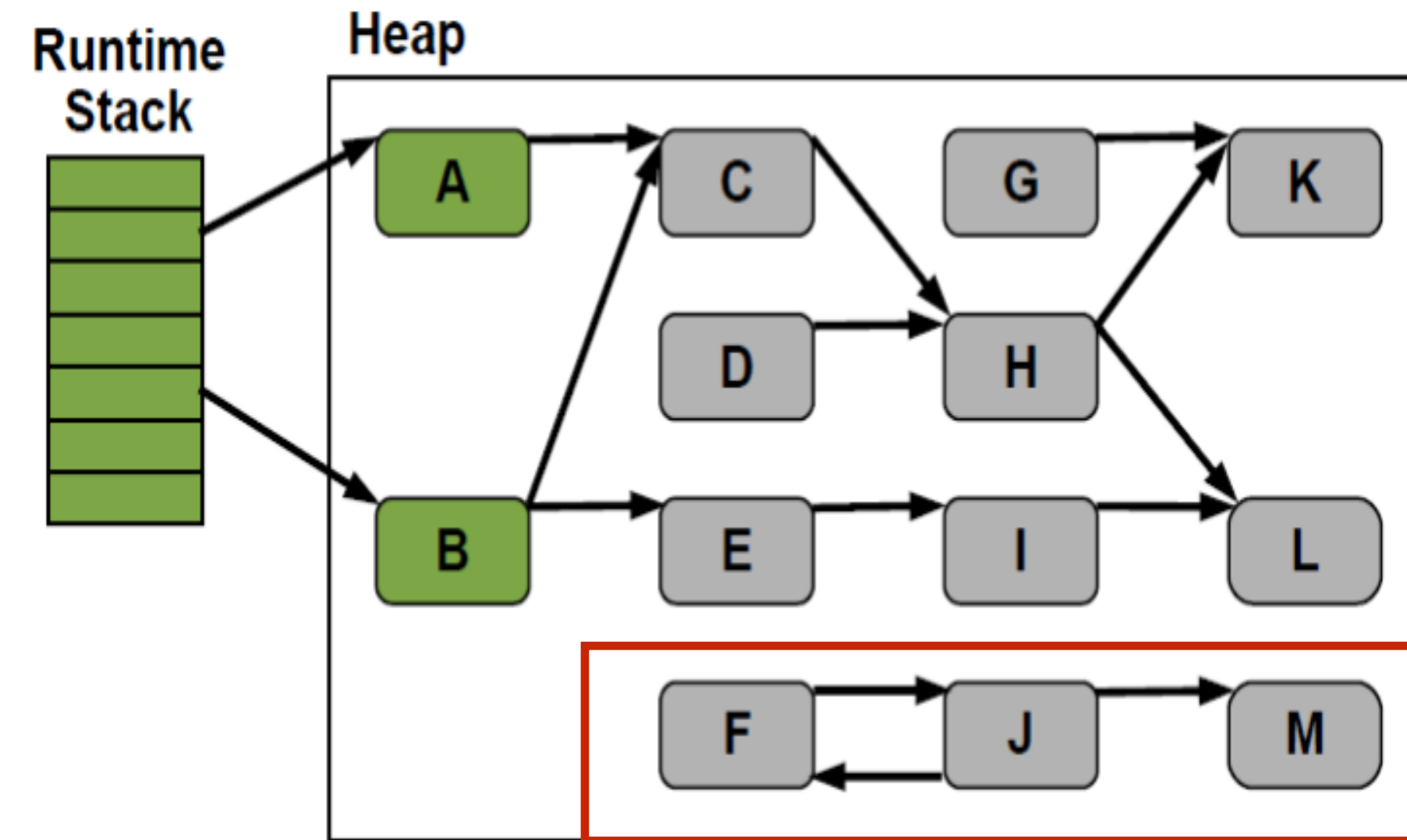
- 缺点：

- 多个 Object 之间，「**循环引用**」内存泄漏



# 核心问题一：回收哪些内存

- 根结点可达（根搜索）：
  - 从「**确定被使用**」的对象，出发
  - 遍历所有「**可到达的对象**」
  - 「可到达的对象」之外的内存，一律回收
- 优点：
  - 解决「**循环引用**」内存泄漏
- 根结点（Root Node）：
  1. **Java 栈**：引用的对象
  2. **本地方法栈**：引用的对象
  3. **方法区**：静态属性，引用的对象
  4. **方法区**：常量属性，引用的对象

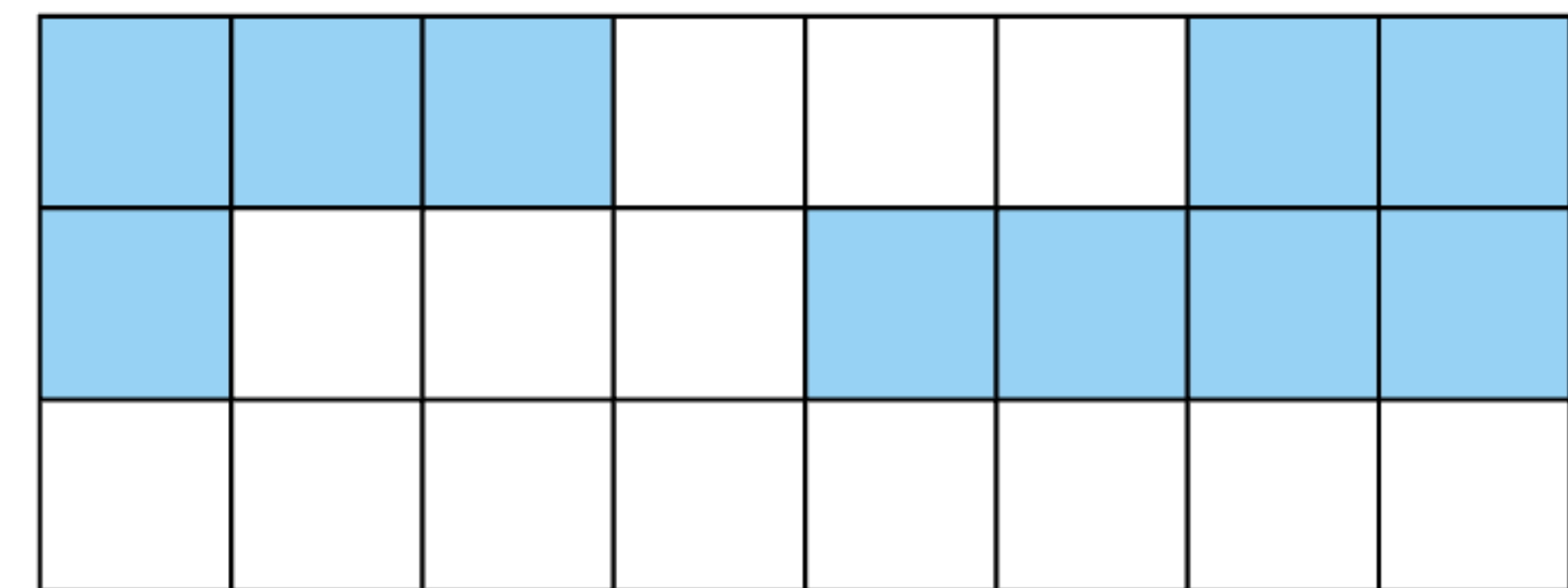
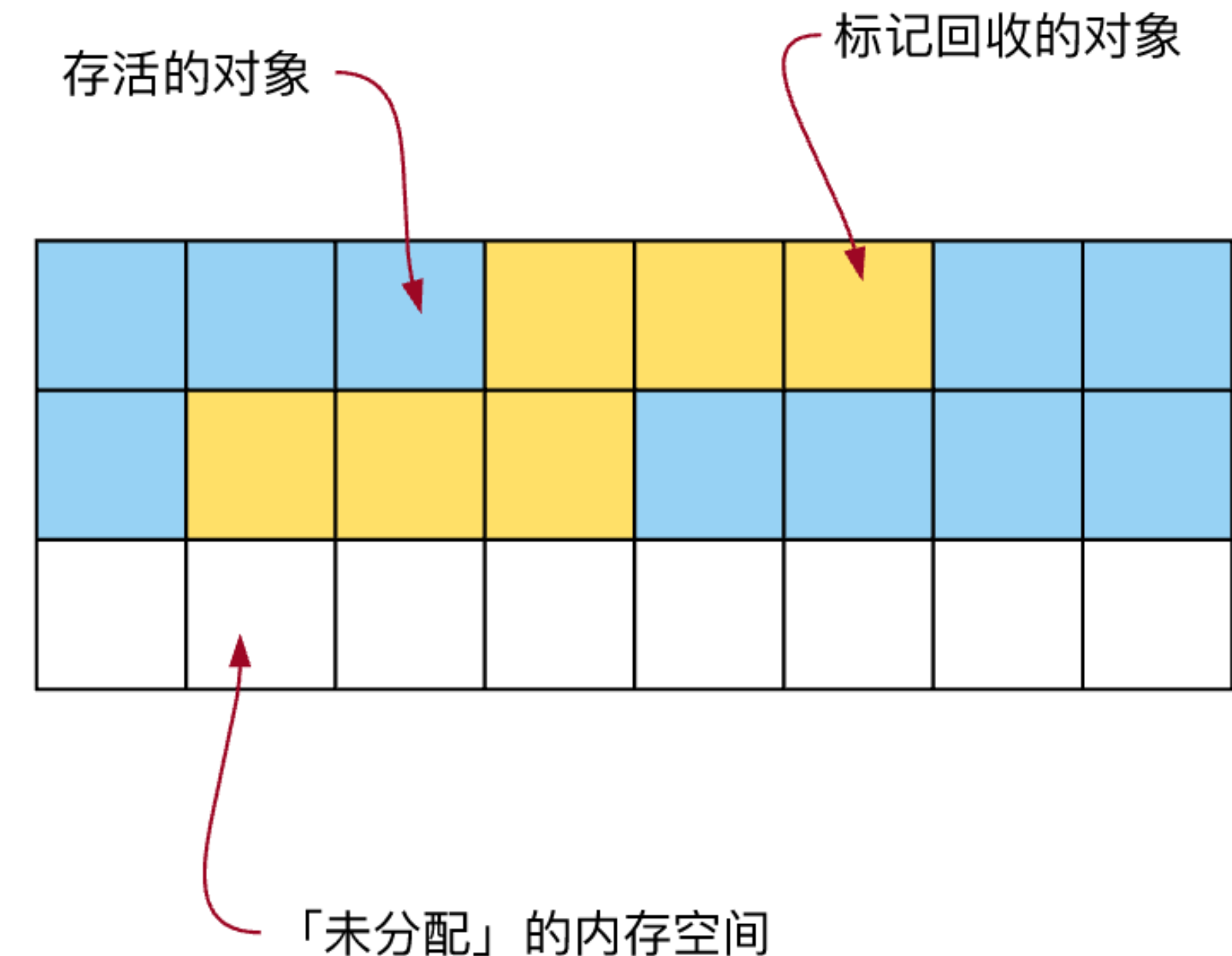


# 核心问题二： 如何回收？

- 核心问题二： 如何回收？
  1. 标记-清除
  2. 标记-清除-压缩（简称： 标记-整理）
  3. 标记-复制-清除（简称： 复制）
  4. 分代回收： 根据对象存活时间， 分级策略

## 核心问题二： 如何回收？

- **标记-清除：**
  - **标记：** 标记出「不再使用的对象」
  - **清除：** 释放对象占用的内存
- **优点：**
  - **简单直接**
  - 不会影响 JVM 进程正常运行
- **缺点：**
  - 产生「**内存碎片**」，不利于再次分配内存
  - **Note：** JVM 为 Object 分配内存时，分配**连续的内存空间**



# 核心问题二： 如何回收？

- **标记-清除-压缩：**（简称：**标记-整理**）

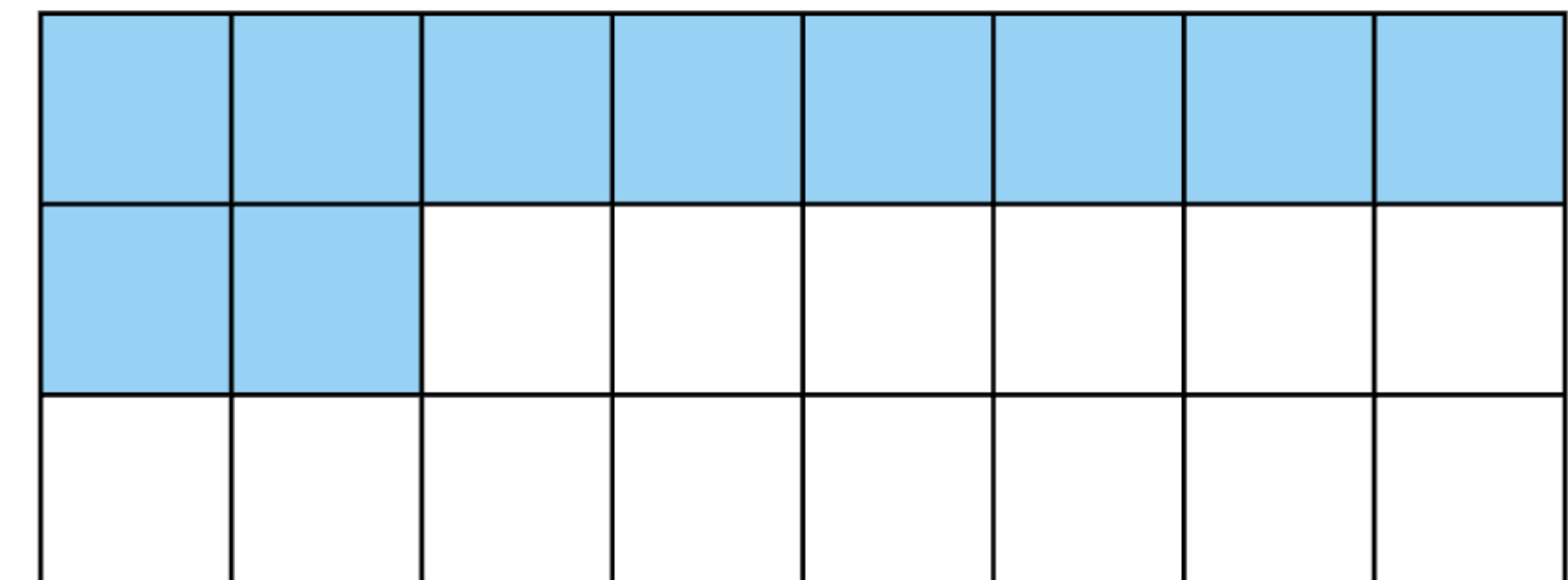
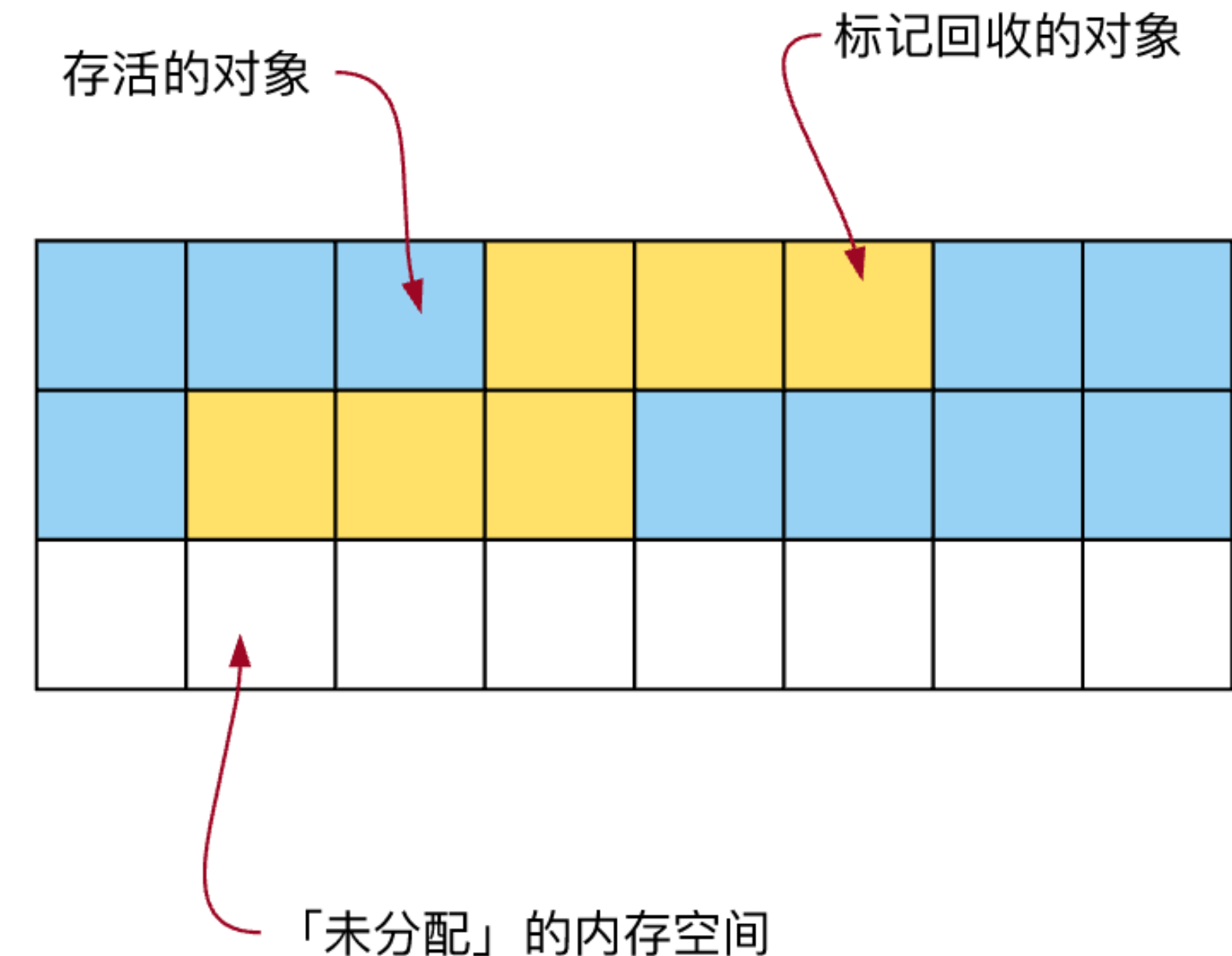
- **标记**「不再使用的对象」
- **清除：** 释放对象占用的内存
- **压缩：** 压缩内存碎片

- **优点：**

- 简单直接
- 不会产生「内存碎片」

- **缺点：**

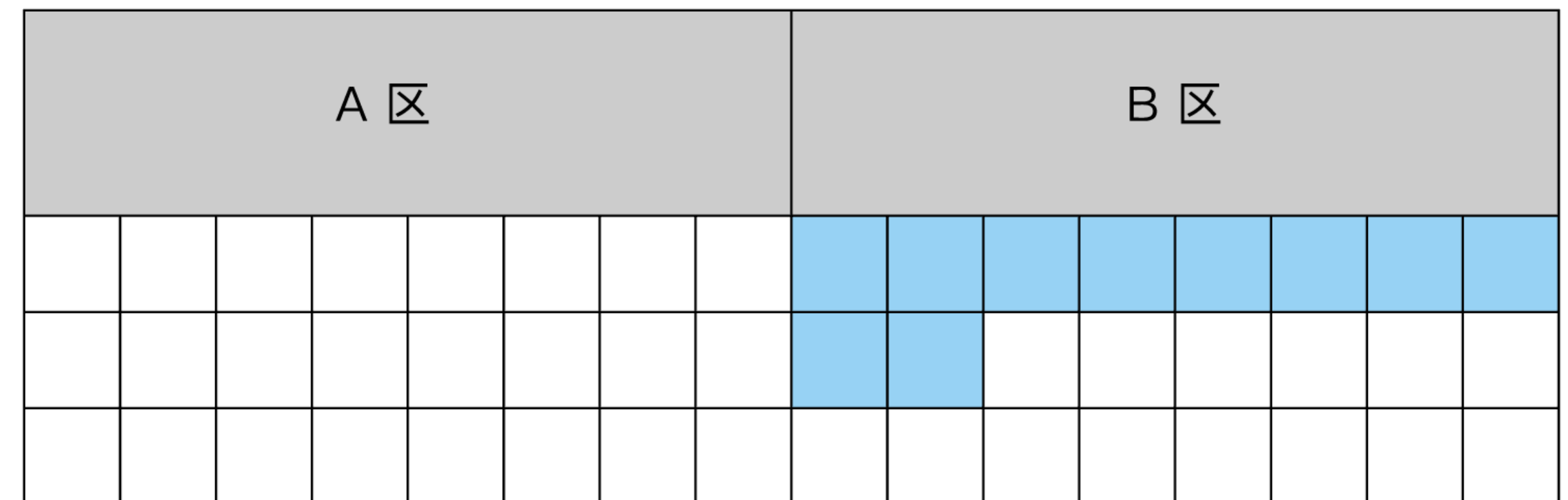
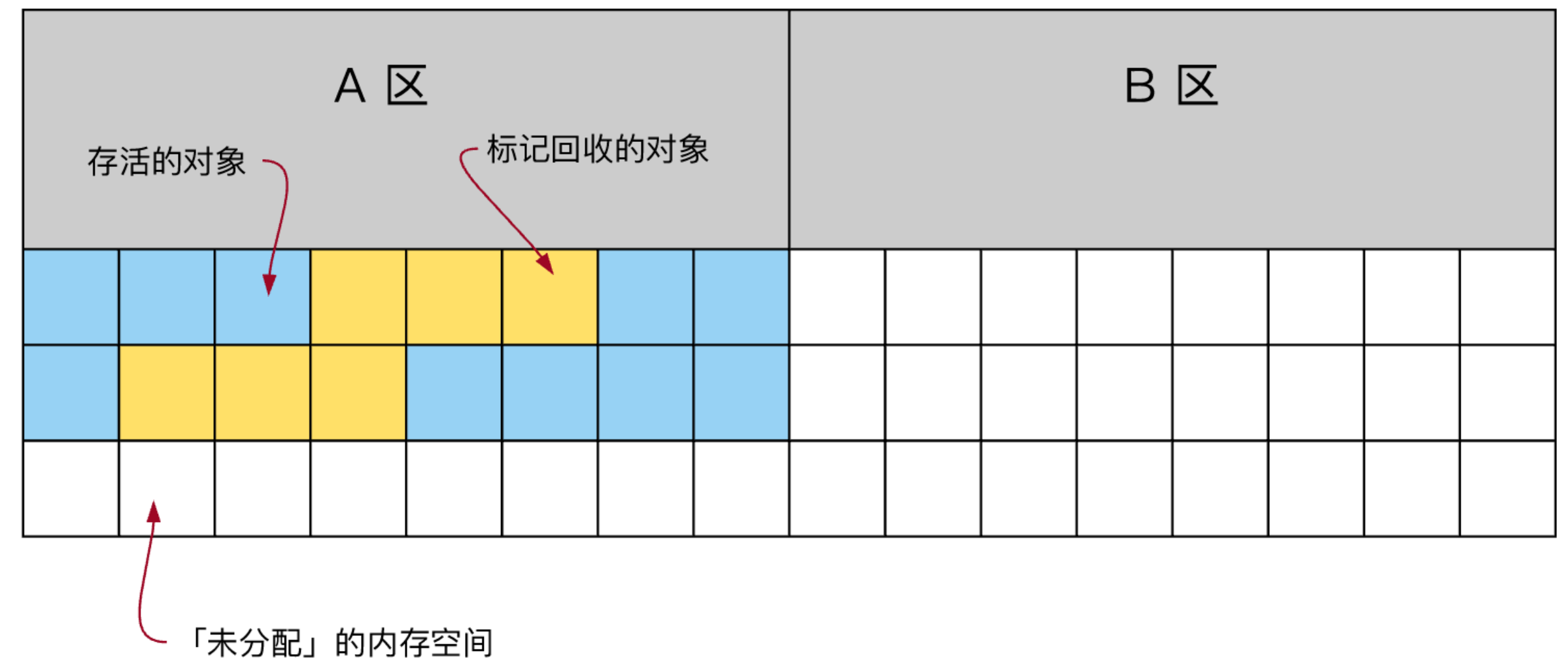
- 会影响 JVM 进程正常运行





# 核心问题二： 如何回收？

- 标记-复制-清除：（简称：**复制**）
  - 标记「不再使用的对象」
  - **复制**：将「存活的对象」复制走
  - **清除**：清理掉内存空间
- 优点：
  - 简单直接，速度快
  - 不会产生「**内存碎片**」
- 缺点：
  - 会影响 JVM 进程正常运行
  - **浪费一半的内存空间**

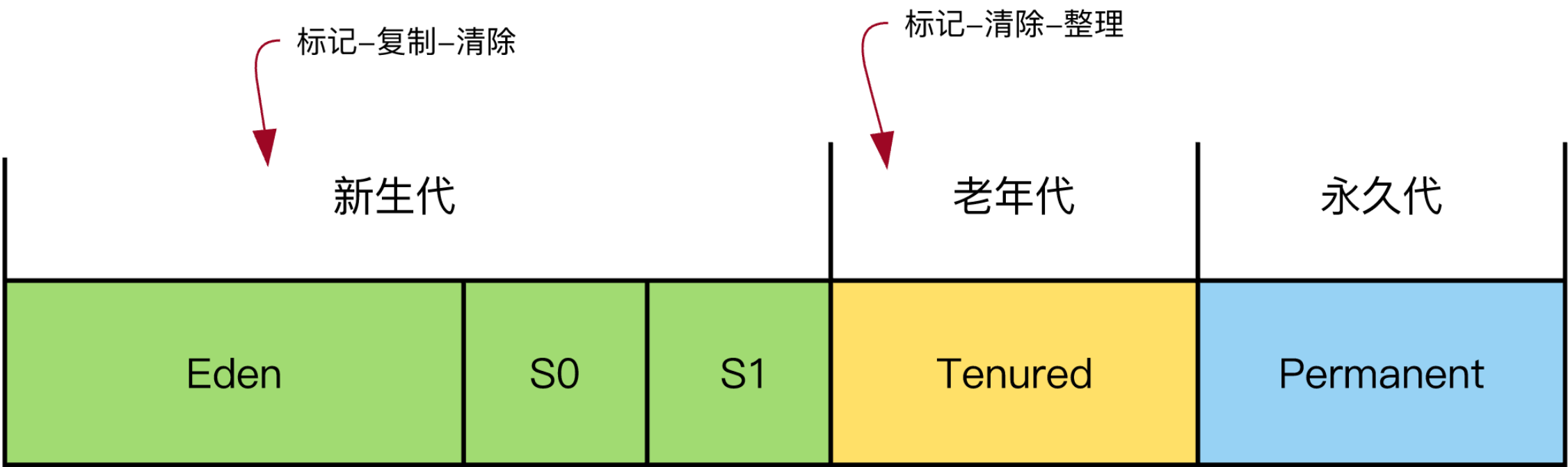


# 核心问题二： 如何回收？

- 分代回收策略： (Hotspot)
  - 根据对象「存活时间」， 分级管理
    - 新生代： 存储新建的对象， 存活时间短， 90%的对象用完就可以回收
    - 老年代： 新生代中， 存活时间较长的对象
    - 永久代： 类加载的信息， 存活时间特别长， 几乎不会被回收

- 优点：
  - 分级管理， 差异化管理
  - 减少重复劳动

- 缺点：
  - 高级别对象， 占用内存时间更长



# JVM 内存回收：小结

- Tips:
  1. 回收哪些内存？如何判断「对象已死」？
  2. 如何回收内存？常见策略？
  3. 回收内存时，是否需要暂停服务？

# 分享内容

- JVM 内存结构
- JVM 内存回收
- HotspotVM
  - 内存回收，具体实现
  - 参数调优
- JVM 常见问题与排查方法

怎么**存放**数据？

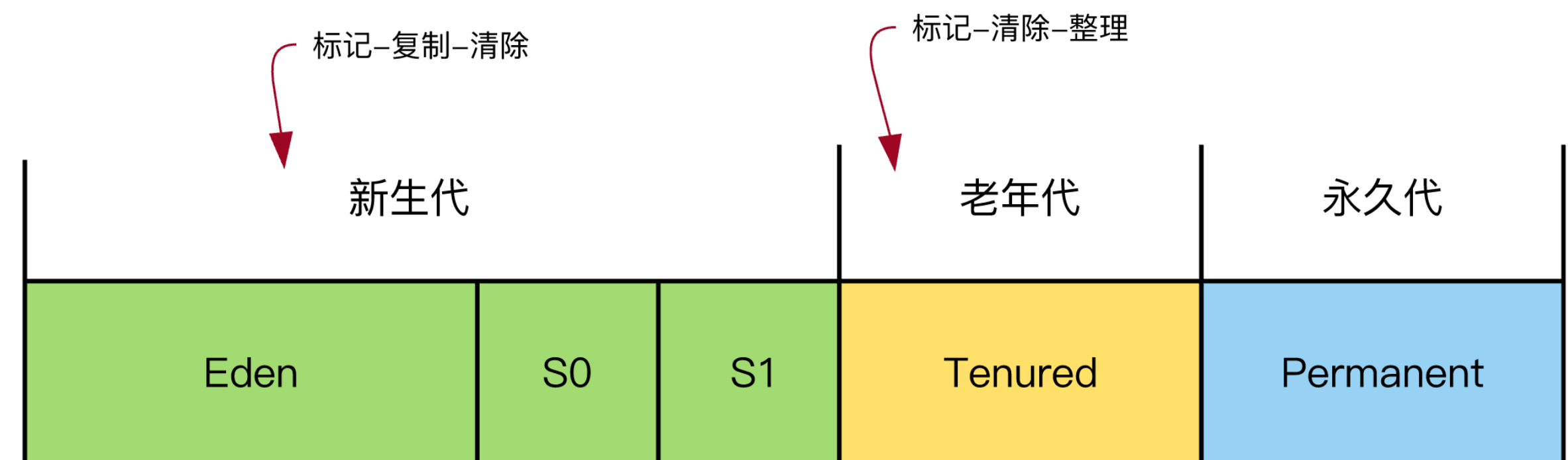
怎么**回收**数据？

具体**实现**，核心参数

# HotspotVM：垃圾收集器

- 背景：

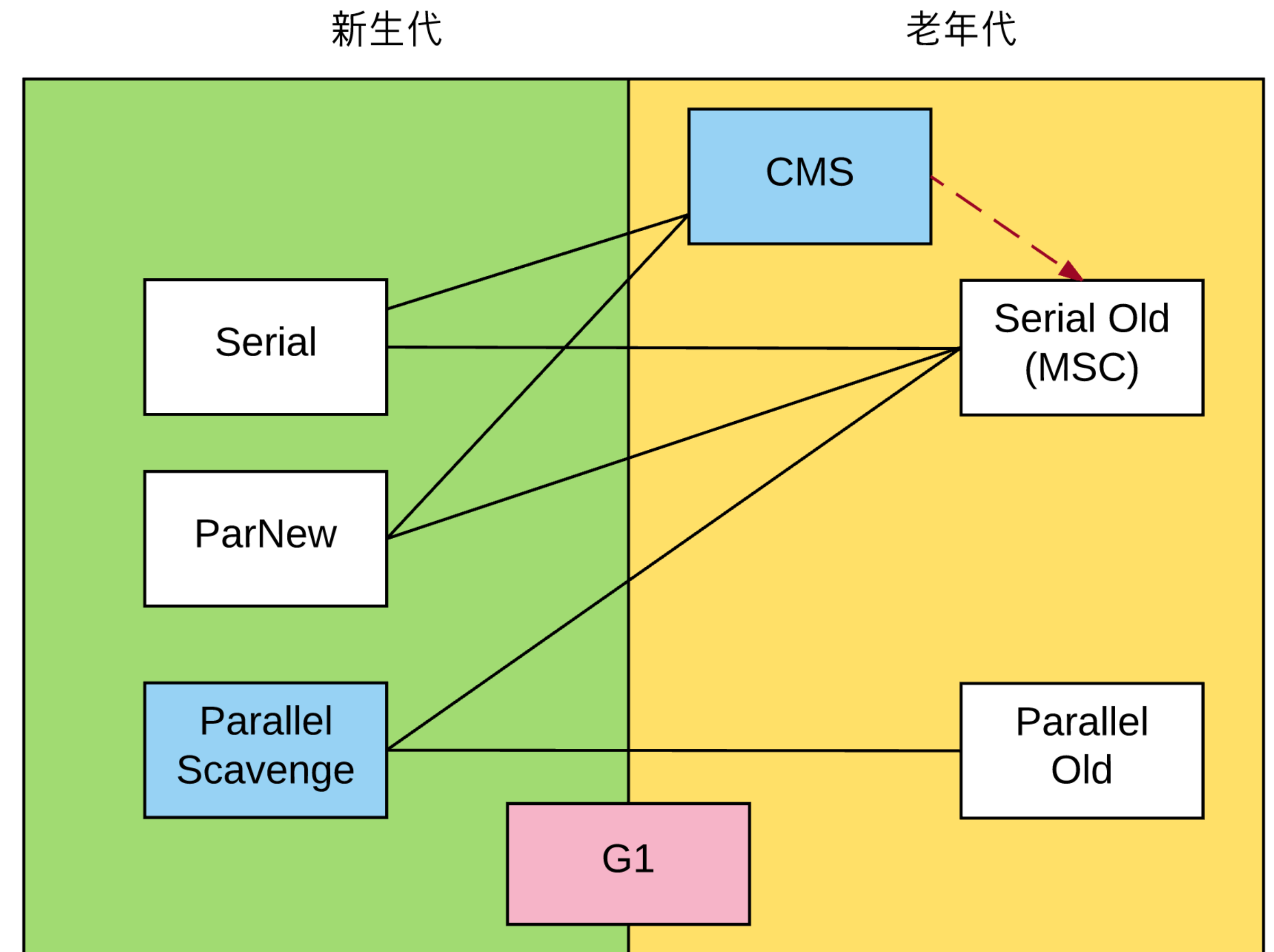
- 前面的「内存回收策略」是「方法论」，是核心思路
- 垃圾收集器，是内存回收的具体实现
- JVM 官方规范中，并没有规定垃圾收集器的实现细节
- 不同厂商、不同 JVM，垃圾收集器，存在差异较大
- HotspotVM 是最流行的 JVM 实现之一
- 后面针对 HotspotVM 内的具体实现进行介绍



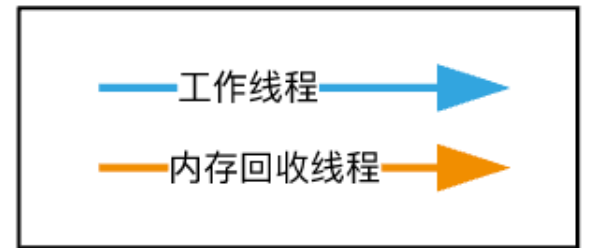
# HotspotVM：垃圾收集器

- 简介：

1. 判断是否回收对象：**根搜索**
2. **分代算法**：基于分代算法，采用不同策略
3. **关联关系**：存在连线的垃圾收集器，可以配合使用
4. **权衡场景**：没有万能的收集器，只有**适合场景**的收集器



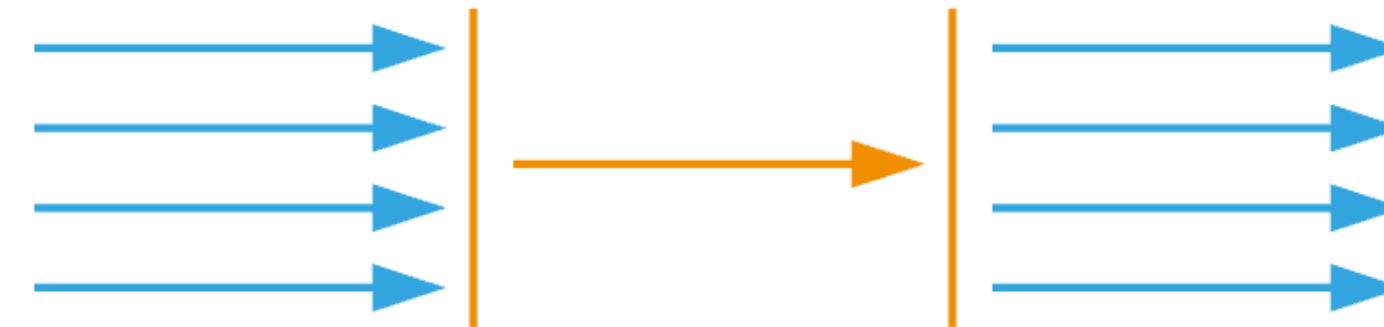
# HotspotVM: 串行、并行、并发



- **串行 (Serial) :**

- 单个 gc thread, 标记、回收内存
- work thread 挂起

串行  
Serial



- **并行 (Parallel) :**

- 多个 gc thread, 标记、回收内存
- work thread 挂起

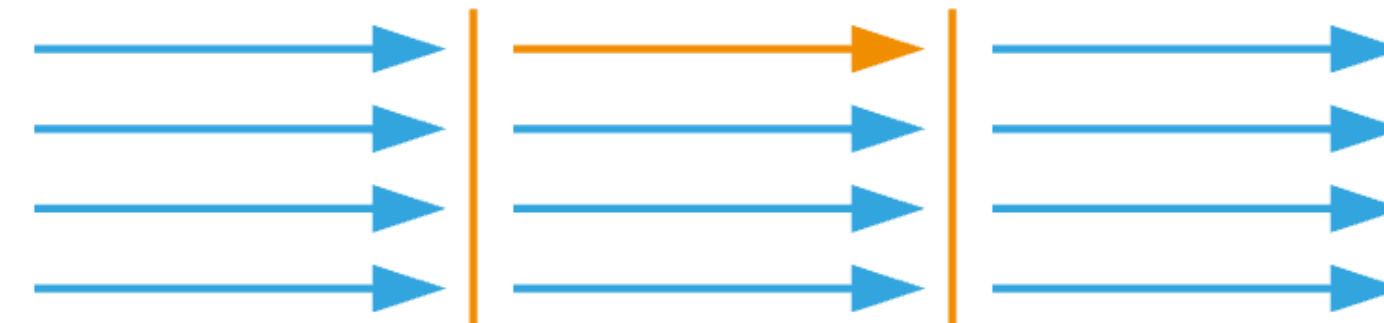
并行  
Parallel



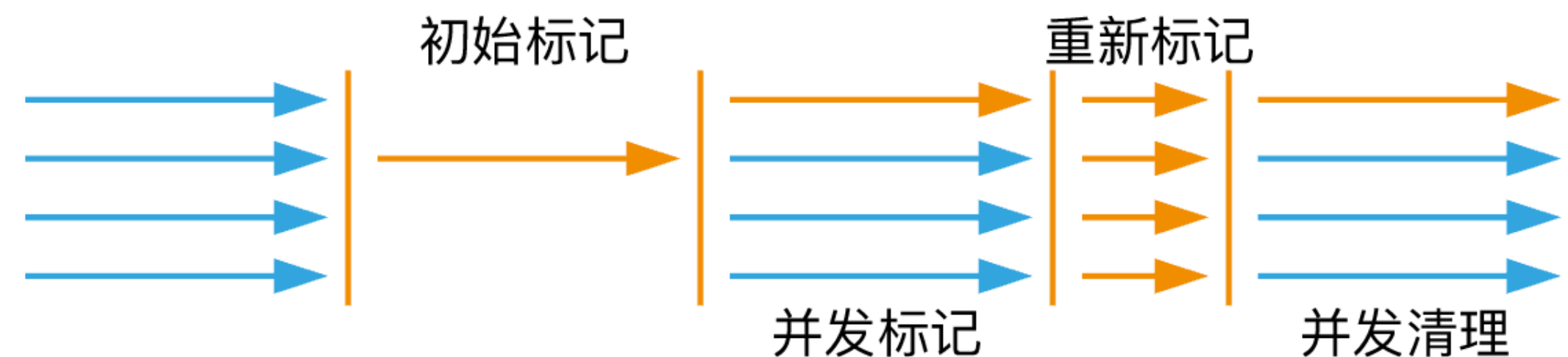
- **并发 (Concurrent) :**

- gc thread, 标记、回收内存
- work thread 正常执行

并发  
Concurrent  
Mark-Sweep

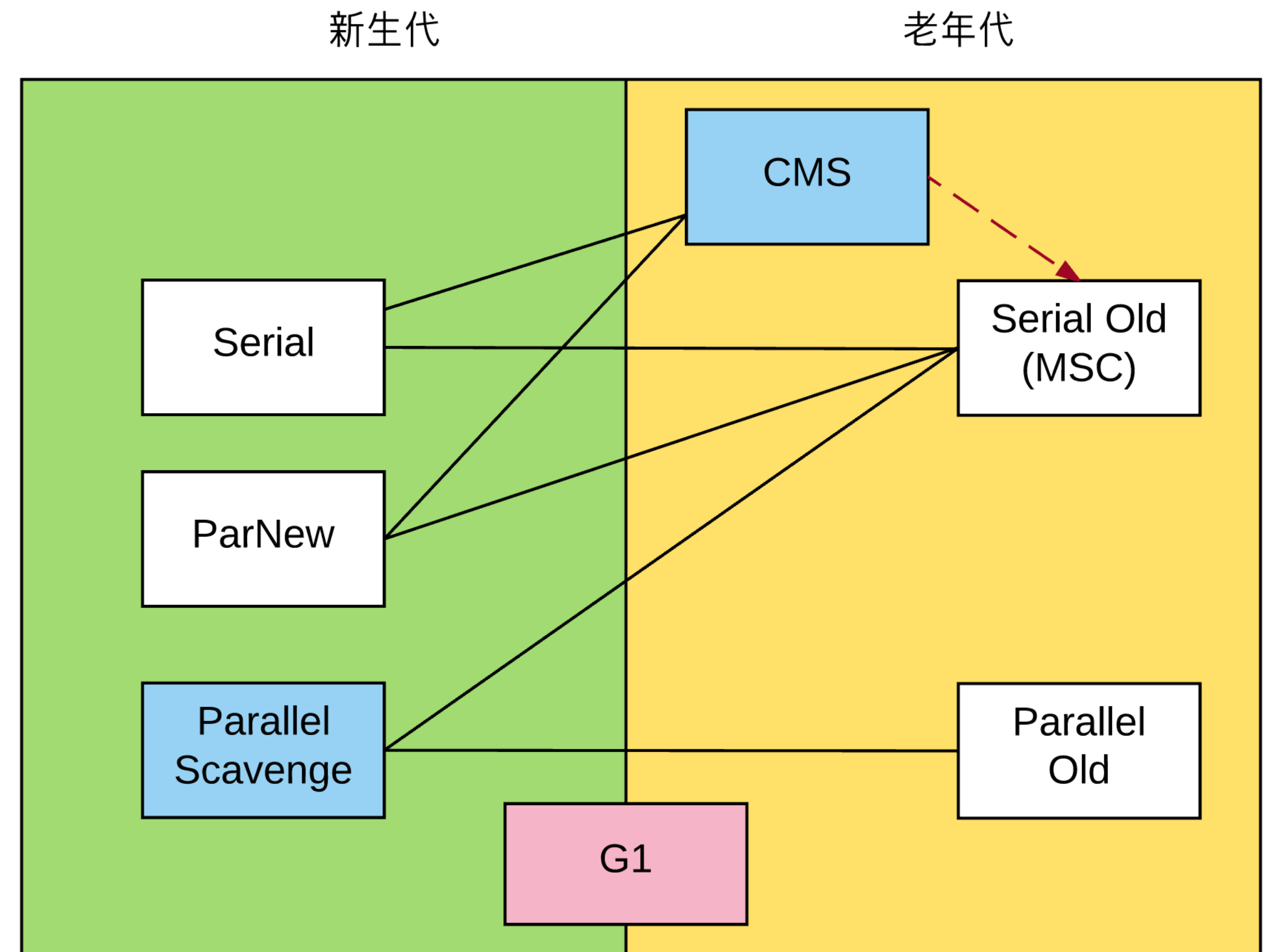


并发  
Concurrent  
Mark-Sweep  
(实际使用)



# HotspotVM：垃圾收集器（新生代）

- **Serial**（新生代-串行-收集器）：单线程，独占式，
  - 策略：标记-复制-清除
  - 优点：简单高效，适用 Client 模式的桌面应用（Eclipse）
  - 缺点：多核环境下，无法充分利用资源
- **ParNew**（新生代-并行-收集器）：多线程，独占式
  - 策略：标记-复制-清除（基于 Serial，多线程版本）
  - 优点：多核环境下，提高 CPU 利用率
  - 缺点：单核环境下，比 Serial 效率要低





# HotspotVM：垃圾收集器（新生代）

- **Parallel Scavenge**（新生代-并行-收集器）：多线程，独占式

- 策略：标记-复制-清除

- 优点：精准控制「吞吐量」、gc 时间

- 吞吐量 =  $\frac{\text{执行用户代码时间}}{\text{执行用户代码时间} + \text{内存回收时间}}$

- 配置参数：

- **MaxGCPauseMillis**：gc 时间的最大值

- **GCTimeRatio**：gc 时间占总时间的比例

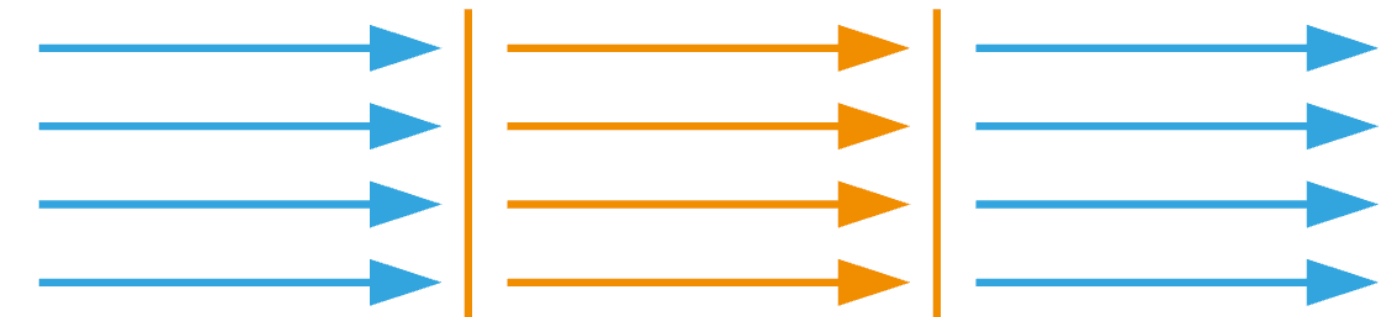
- **UseAdaptiveSizePolicy**：开启 GC 内存分配的「自适应调节策略」，自动调整：

- 新生代大小

- Eden与Survivor 的比列

- 晋升老年代的对象年龄

并行  
Parallel



并行  
Parallel Scavenge



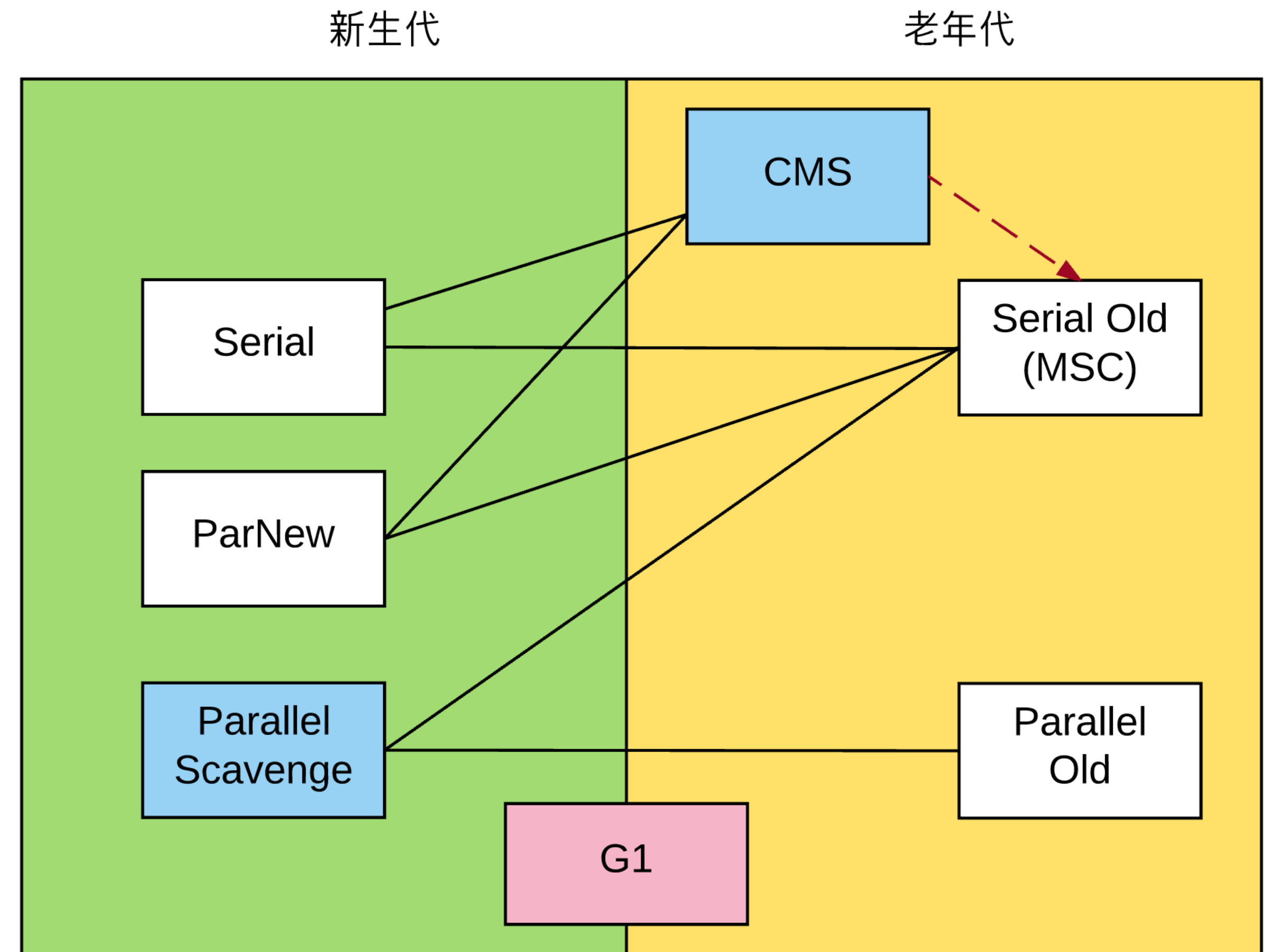
目标：精准控制 gc 时间 / 吞吐量

配置参数：

1. 单次 gc 的最大时间
2. gc 时间占总时间的百分比

# HotspotVM：垃圾收集器（老年代）

- **Serial Old**（老年代-串行-收集器）：单线程，独占卡  
  - 策略：标记-清除-整理
  - 优点：简单高效
  - 缺点：多核环境下，无法充分利用资源
- **Parallel Old**（老年代-并行-收集器）：多线程，独占卡  
  - 策略：标记-清理-整理
  - 优点：多核环境下，提高 CPU 利用率
  - 缺点：单核环境下，比 Serial Old 效率要低



# HotspotVM：垃圾收集器（老年代）

- **CMS, Concurrent Mark-Sweep**, （老年代-并发-收集器）：多线程，非独占式

- 策略：标记-清除

- 优点：「停顿时间」最短

- 缺点：内存碎片（有补偿策略）

- 适用场景：互联网 Web 应用的 Server 端，涉及用户交互、响应速度快。

- **CMS 具体过程**：

- 初始标记：仅标记「GC Roots」直接引用的对象

- 并发标记：从 GC Roots 出发，标记可达对象

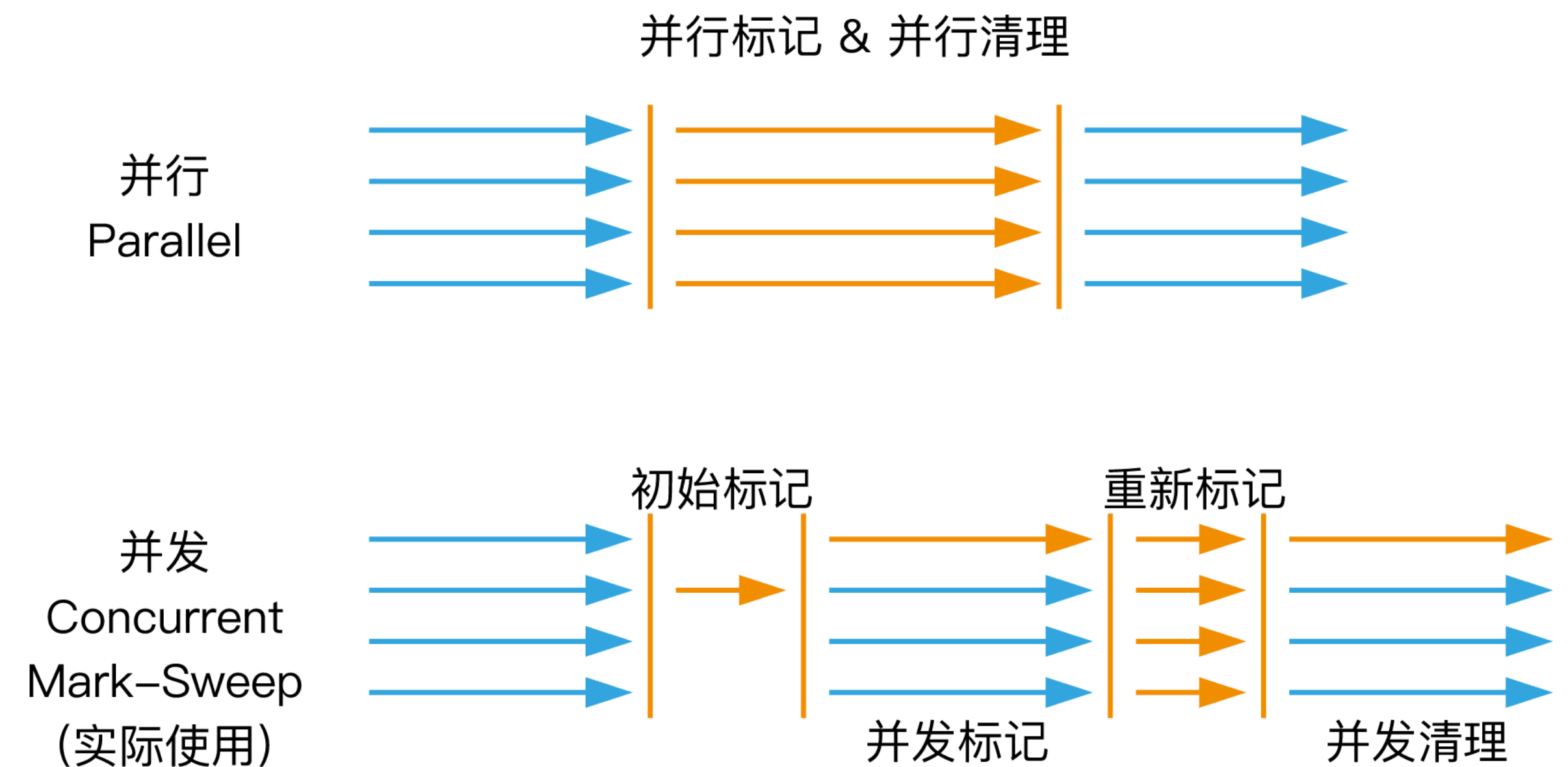
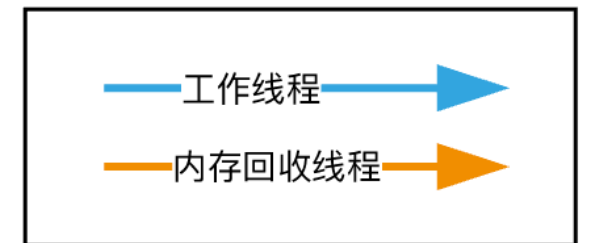
- 重新标记：标记「并发标记」过程中，变更的对象

- 并发清除：清除「无用对象」

- **CMS 降级**：Concurrent Mode Failure

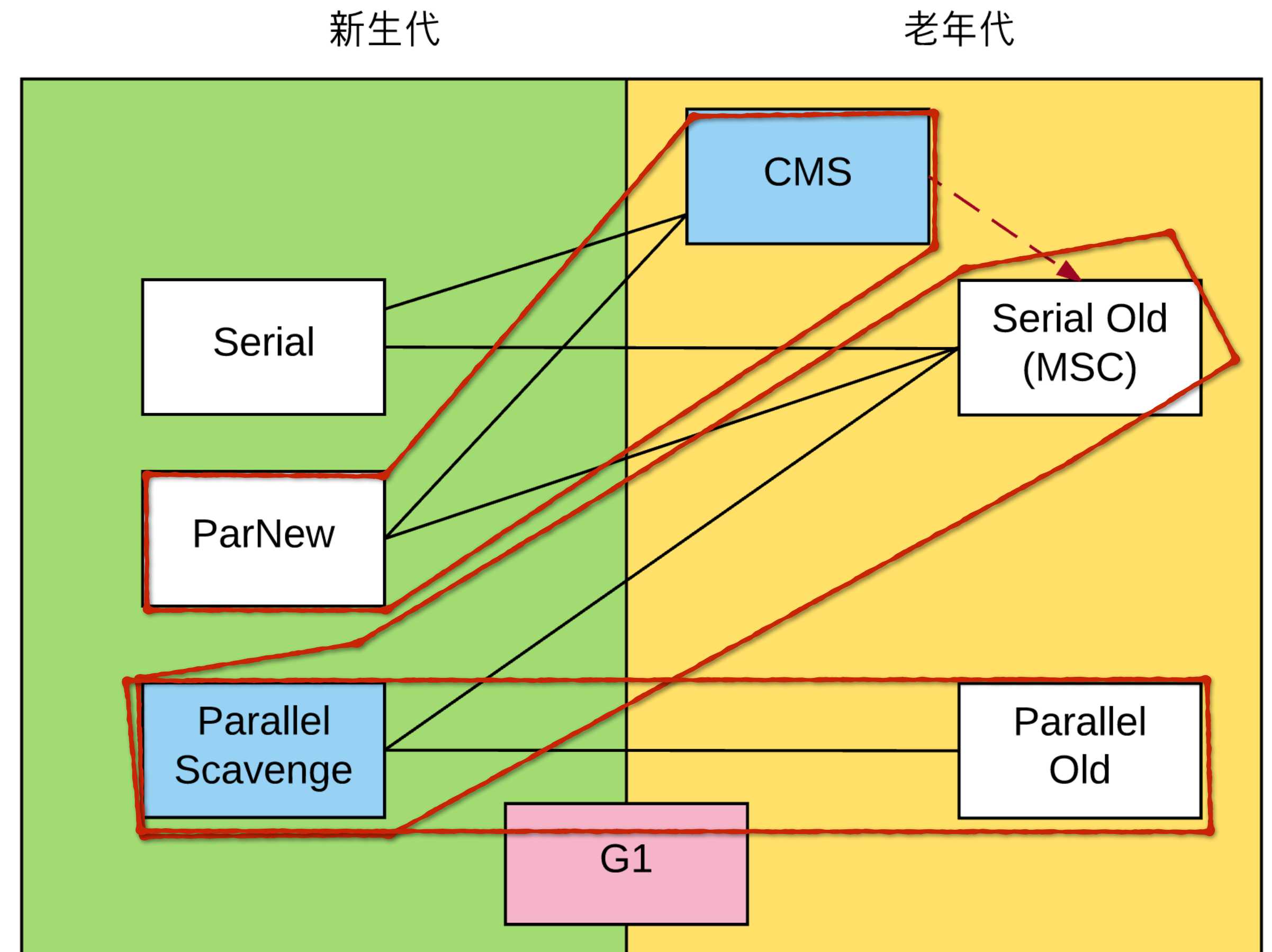
- 并发标记、清理过程，work thread 在运行，申请「老年代」空间可能失败

- 后备预案：临时启动 Serial Old 收集器



# HotspotVM：垃圾收集器（新生代 & 老年代）

- 关联关系：存在连线的垃圾收集器，可以**配合使用**
- 实现历史：
  - Parallel、G1 没有使用传统的 GC 代码框架，无法配合 CMS
  - JDK1.6+，引入 Parallel Old，用于配合 Parallel Scavenge 使用
  - JDK1.7+，引入 G1（Garbage First）
- **Server 模式，默认组合：**（已经过时）
  - 新生代：Parallel Scavenge
  - 老年代：Serial Old
- 吞吐量和 CPU 资源敏感的场景（**计算密集型**），推荐组合：
  - 新生代：Parallel Scavenge
  - 老年代：Parallel Old
- 响应时间敏感的场景（**交互型**），推荐组合：
  - 新生代：ParNew
  - 老年代：CMS



# HotspotVM 垃圾收集器：小结

- Tips:

- **分代管理**：新生代、老年代、永久代

- 新生代：标记-复制-清除

- 老年代：标记-清除-压缩

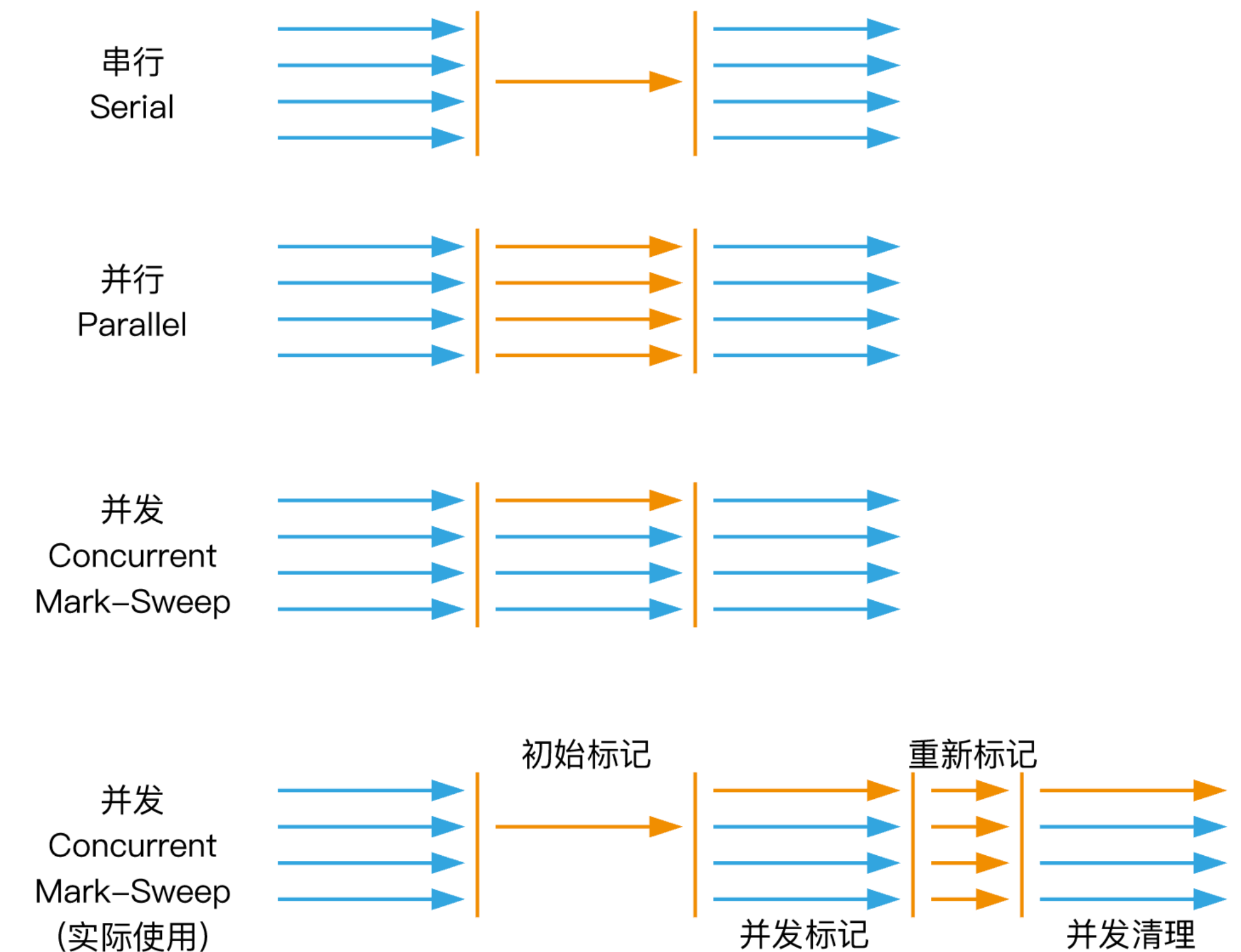
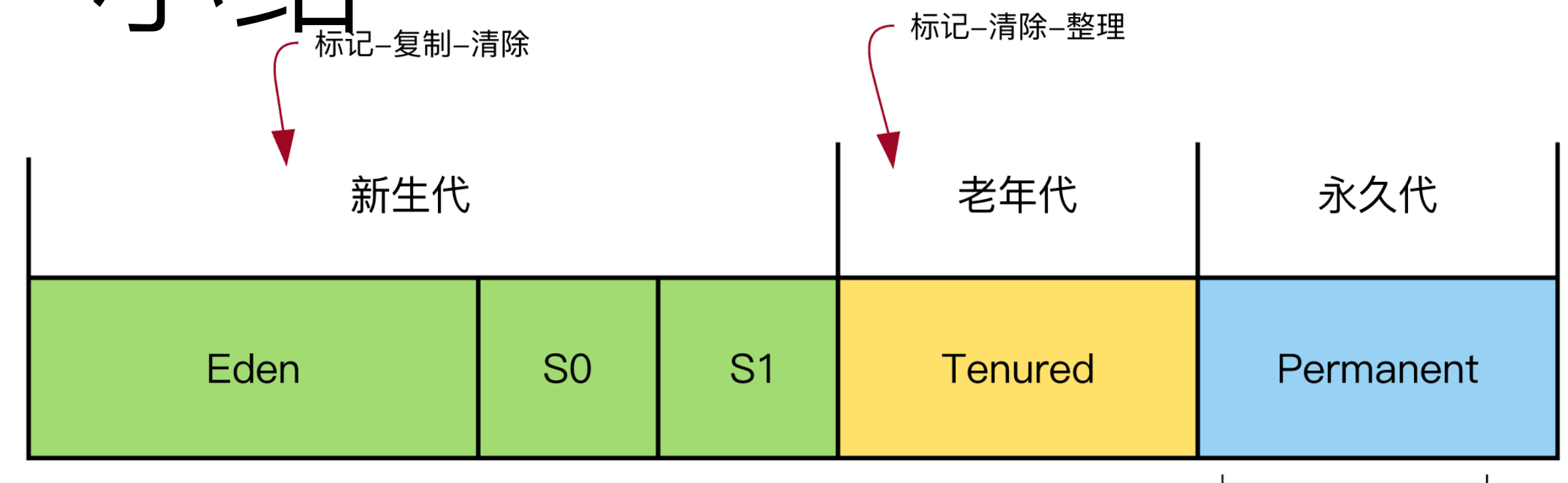
- **收集器**，具体实现：(串行\并行\并发)

- **新生代**：

- Serial
      - ParNew
      - Parallel Scavenge

- **老年代**：

- Serial Old
      - Parallel Old
      - CMS



# HotspotVM 垃圾收集器：补充

- **Young GC vs. Full GC：**

- **Young GC**，Minor GC，新生代 GC

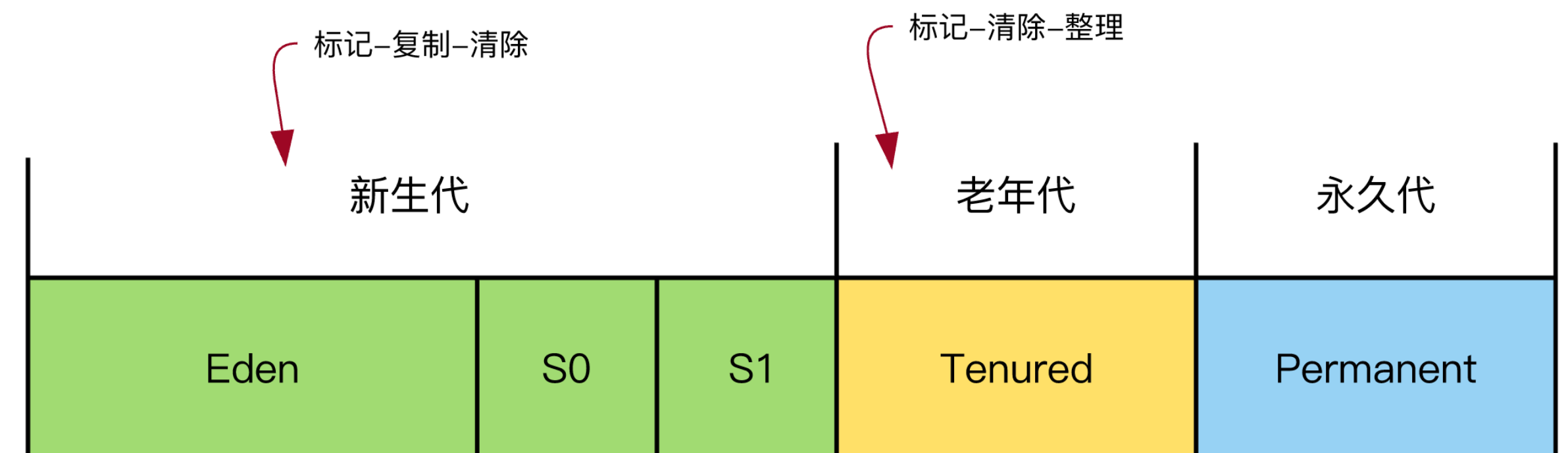
- 发生地点：**新生代**
    - 发生时间：在「新生代」创建对象时，连续存储空间不足，触发 Young GC
    - **特点：速度快、频次高**

- **Full GC**，Major GC，老年代/永久代 GC

- 发生地点：**老年代/永久代**
    - 发生时间：
      - Young GC 之前，预判「老年代」的空间是否充足；
      - 大对象直接进入「老年代」，但「老年代」空间不足；
    - **特点：速度慢**（比 Young GC 慢 10 倍+）、**需要控制频次**

- 补充：

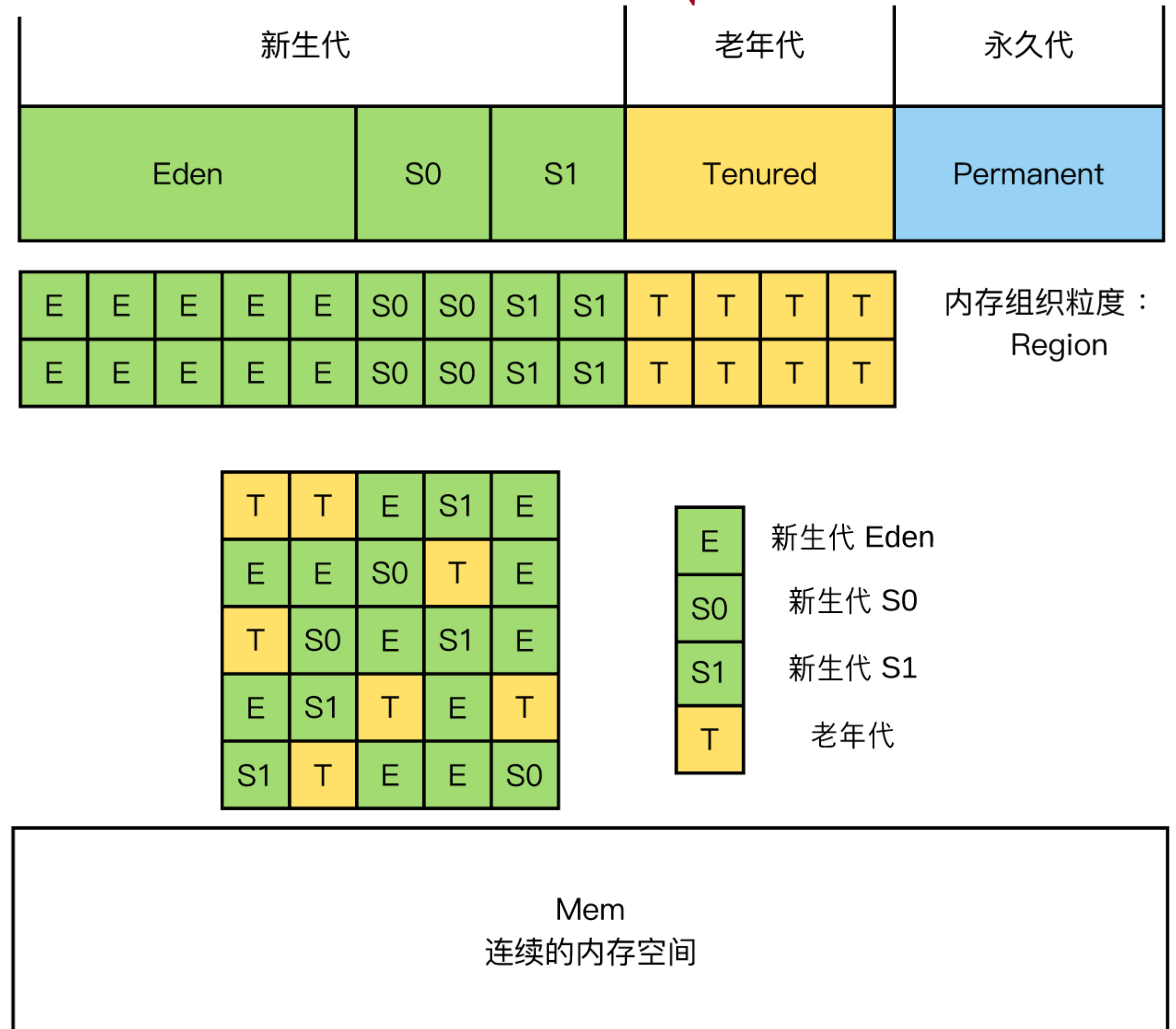
- **Full GC 并不包含 Young GC**；Full GC 一般伴随 Young GC （不绝对）
  - **Full GC，暂停时间比较长**，认为 **Stop-The-World（STW）**，参数配置时，重点考虑降低 Full GC 次数。



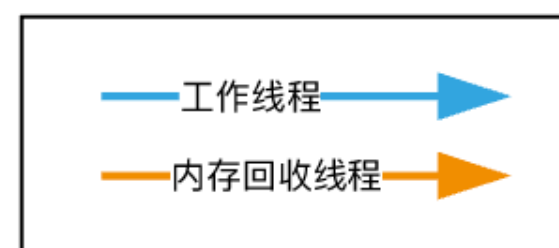
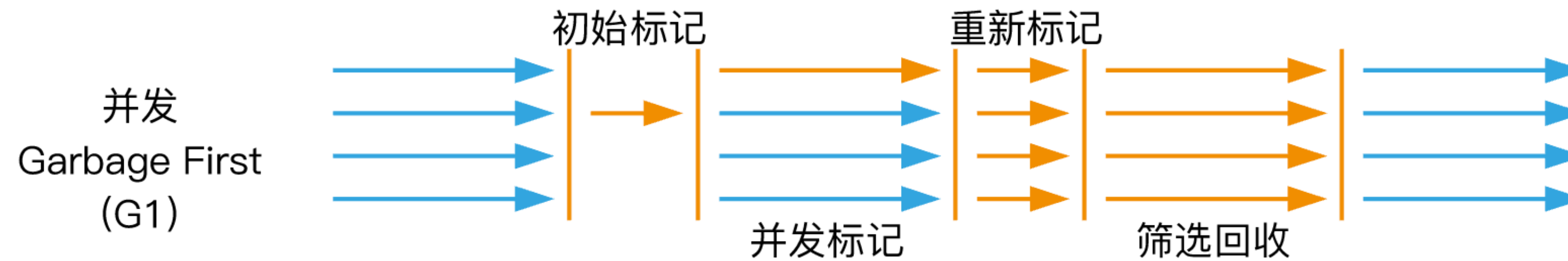
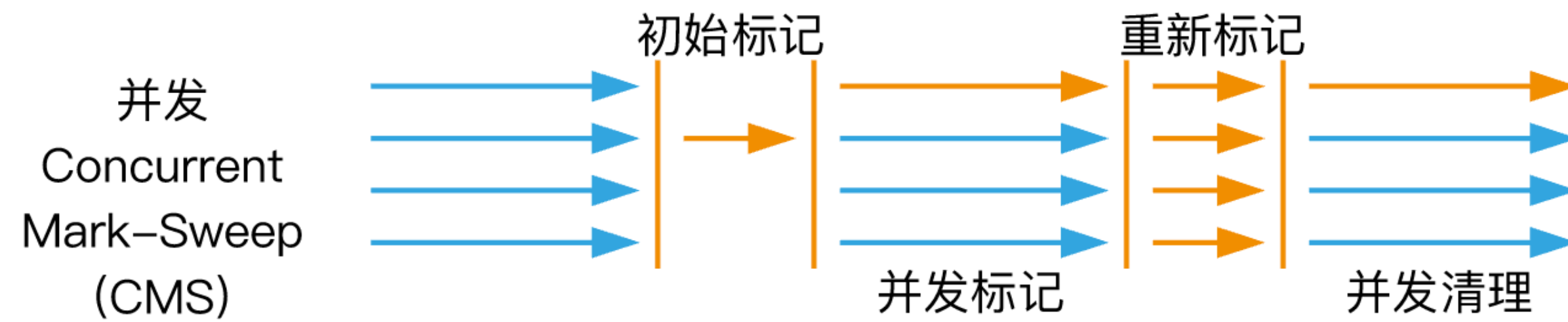


# HotspotVM 垃圾收集器： G1 (Part 1)

- G1, Garbage First:
  - **目标**: 替代 CMS
  - **内存布局**:
    - **内存组织粒度 Region**
    - 新生代/老年代不要求连续内存空间
  - **分代**: G1 独立管理, 新生代、老年代
  - **策略**: **标记-清除-整理**, 不会产生**内存碎片** (Region 间复制)
  - **并发**: 降低停顿时间, 减弱 STW 停顿
  - **可预测的停顿**: 精确控制 gc 停顿时间
    - 每个 Region 维护一个 「Garbage Value」, 优先队列
    - 优先回收 「Garbage Value」 最大, 回收价值最大的 Region
  - **Young GC 和 Full GC**: 跟前面概念完全一致
    - 尽可能减少 **Full GC**



# HotspotVM 垃圾收集器：G1 (Part 2)



细节：

1. 对 Region 的回收优先级排序
2. 根据「配置的gc停顿时间」，确定对哪些高优先级 Region 回收



# HotspotVM 垃圾收集器： G1 (Part 3)

- **G1 vs. CMS**

- **内存组织粒度**：G1 将内存划分为「Region」，避免内存碎片
- **内存灵活性**：Eden、Survivor、Tenured 不再固定，内存使用效率更高
- **适用范围**：G1 能够应用在「新生代」，CMS 只能应用在「老年代」
- **可控性**：可控的 STW 时间，根据预期的停顿时间，只回收部分 Region

- **G1 适用场景：**

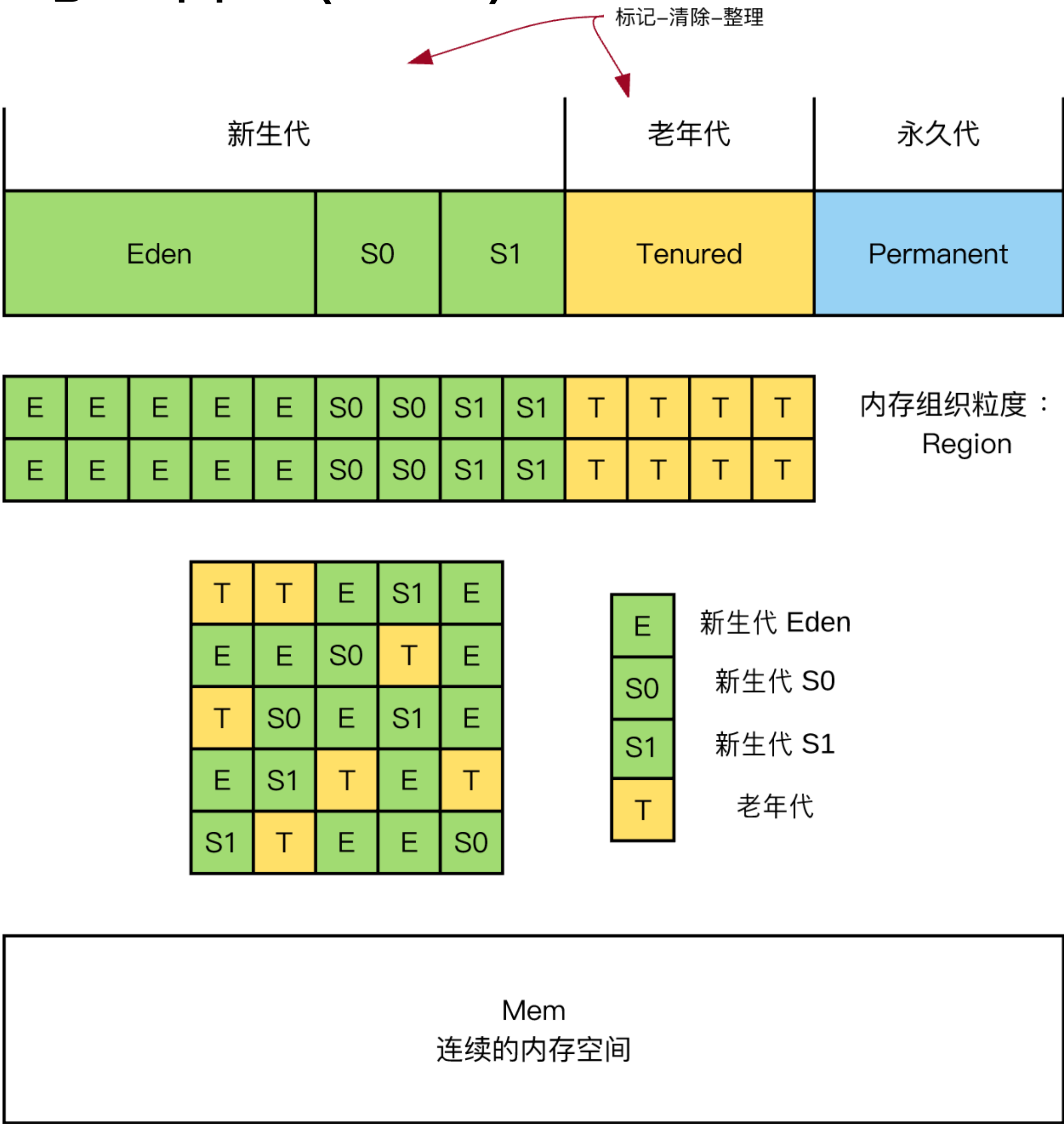
- **服务端，多核 CPU**，JVM 占用内存较大 (>4GB)
- 业务场景中，应用会产生大量**内存碎片**、需要经常压缩
- 可控、**可预期**的 **GC 停顿时间**，防止高并发下应用的血崩现象

- **是否升级到 G1：**

1. 现在采用的收集器**没有出现问题**，就暂时**没有理由**选择 G1，等待 G1 持续的优化即可
2. 服务器端，**交互型**应用，**追求快速响应**，现在就可以尝试一下 G1
3. **计算密集型**应用，**G1 并不会明显改善**吞吐量

# HotspotVM 垃圾收集器：小结（G1）

- Tips:
  - **G1:**
    - 目标：替代 CMS
    - 策略：分代 + 标记-清除-整理
    - 实现方式：内存组织粒度 Region
    - 收益：无「内存碎片」、可控的 STW 时间
    - 适用场景：服务器端，交互型应用，追求快速响应



# Hotspot 垃圾收集器：参数调优

- HotspotVM 参数调优，基本思路：
  1. 选择合适的「垃圾收集器」
  2. 新生代（Eden\Survivor）、老年代、方法区，分配合适的内存空间

# Hotspot 垃圾收集器： 参数调优

参数名称	含义	默认值	
-XX:+UseParallelGC	Server 默认值		选择垃圾收集器为并行收集器.此配置仅对年轻代有效. 打开此开关后，使用 Parallel Scavenge + Serial Old
-XX:+UseParNewGC	设置年轻代为并行收集		可与CMS收集同时使用 JDK5.0以上,JVM会根据系统配置自行设置,所以无需再设置此值
-XX:+UseParallelOldGC	年老代垃圾收集方式为并行收集(Parallel Compacting)		打开此开关后，使用 Parallel Scavenge + Parallel Old
-XX: +UseConcMarkSweepGC	使用CMS内存收集		
-XX +UseCMSCompactAtFullCollection	在FULL GC的时候， 对年老代的压缩		CMS是不会移动内存的， 因此， 这个非常容易产生碎片， 导致内存不够用， 因此， 内存的压缩这个时候就会被启用。 增加这个参数是个好习惯。 可能会影响性能,但是可以消除碎片

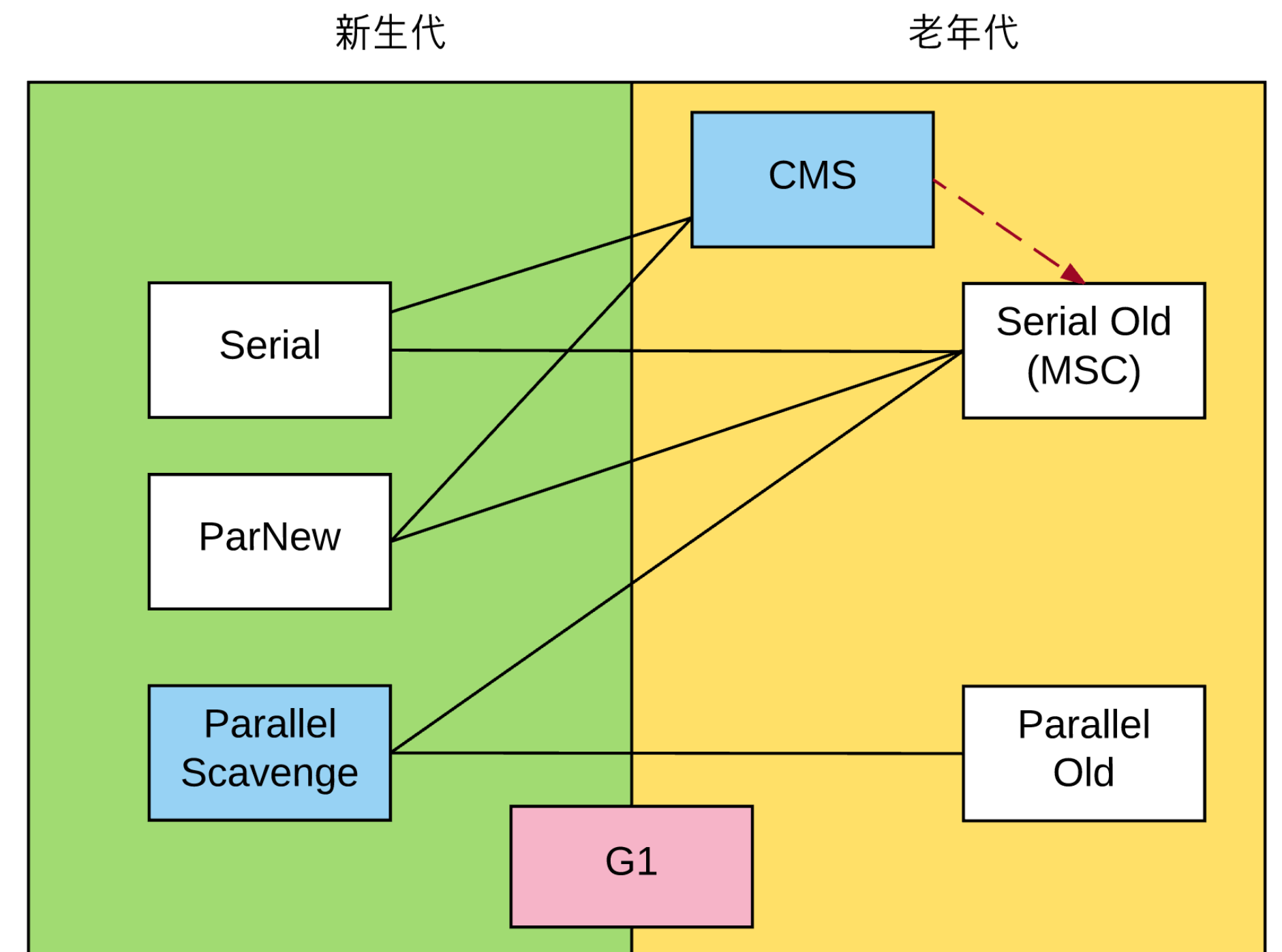
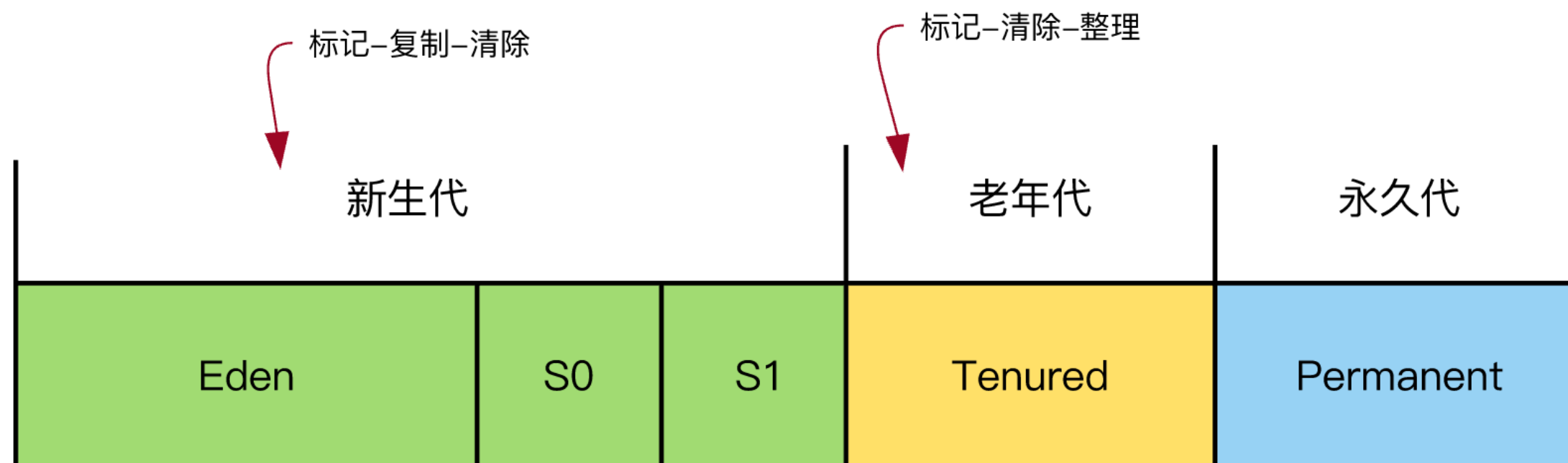
# Hotspot 垃圾收集器： 参数调优

参数名称	含义	默认值	
-Xms	初始堆大小	物理内存的1/64(<1GB)	默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制。
-Xmx	最大堆大小	物理内存的1/4(<1GB)	默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制
-Xmn	年轻代大小		<b>注意：</b> 此处的大小是 (eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8
-XX:PermSize	设置永久代初始值	物理内存的1/64	
-XX:MaxPermSize	设置持久代最大值	物理内存的1/4	
-Xss	每个线程的堆栈大小		JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。
-XX:NewRatio	年轻代(包括Eden和两个Survivor区)与年老代的比值(除去持久代)		-XX:NewRatio=4表示年轻代与老年代所占比值为1:4,年轻代占整个堆栈的1/5 Xms=Xmx并且设置了Xmn的情况下， 该参数不需要进行设置。
-XX:SurvivorRatio	Eden区与Survivor区的大小比值		设置为8,则两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10
-XX: +DisableExplicitGC	关闭System.gc()		
- XX:MaxTenuringThres hold	垃圾最大年龄		如果设置为0的话,则年轻代对象不经过Survivor区,直接进入老年代. 对于老年代比较多的应用,可以提高效率.如果将此值设置为一个较大值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象再年轻代的存活 时间,增加在年轻代即被回收的概率 该参数只有在串行GC时才有效。
- XX:PretenureSizeThre shold	对象超过多大是直接 在老年代分配		0 单位：字节，新生代采用Parallel Scavenge GC时无效 另一种直接在老年代分配的情况是大的数组对象,且数组中无外部引用对象。
-Xnoclassgc	永久代禁用垃圾回收		

# Hotspot 垃圾收集器：参数调优

```
boot.ini x
1 JVM_ARGS="-server -Dapp.port=${app_port} -Dapp.key=${app_key} -Dfile.encoding=UTF-8 -Dsun.jnu.encoding=UTF-8 -Djava.io.tmpdir=/tmp -Djava.net.preferIPv6Addresses
2 JVM_SIZE="-Xmx${jvm_xmx} -Xms${jvm_xms}"
3 JVM_GC="-XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintHeapAtGC -XX:+PrintTenuringDistribution -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:-OmitStackTraceInFast
4 JVM_GC=${JVM_GC} -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=0 -XX:+UseCMSCompactAtFullCollection -XX:+ExplicitGCInvokesConcurrent -XX:CMSInitiatingOccupancy
5 JVM_HEAP="-Xmn${jvm_xmn} -Xss${jvm_xss} -XX:SurvivorRatio=6 -XX:ReservedCodeCacheSize=64m -XX:InitialCodeCacheSize=64m -XX:+HeapDumpOnOutOfMemoryError"
6
```

```
#jvm
jvm_xmx=4g
jvm_xms=4g
jvm_xmn=1g
jvm_xss=1m
```





# 分享内容

- JVM 内存结构
- JVM 内存回收
- HotspotVM
  - 内存回收，具体实现
  - 参数调优
- JVM 常见问题与排查方法

怎么**存放**数据？

怎么**回收**数据？

具体**实现**，核心参数

**实践**

# JVM 常见问题

- 排查工具
- JVM CPU 资源占用过高
- JVM 内存资源占用过高：OOM, OutOfMemoryError



# 排查工具

- jps: 显示本地有几个 JVM 进程, 并获取 pid

- 命令格式: jps

- jinfo: 运行环境参数, Java System 属性、JVM 命令参数, class path 等

- 命令格式: jinfo [pid]

- jstat: JVM 运行状态信息, 类加载、内存回收等

- 命令格式: jstat [-class | -gc] [pid] [interval] [count]

- jstack: JVM 进程内部, 所有线程的运行情况

- 命令格式: jstack [pid]

- jmap: JVM 内, 堆中存储的对象「堆转存快照」, 使用 jhat 命令分析「堆转存快照」

- 命令格式: jmap [pid]

- **备注:** 可视化工具 JConsole、**VisualVM**

有哪些 JVM?

JVM 启动参数?

JVM 运行状态?

JVM 中, 栈?

JVM 中, 堆?

# CPU 占用过高

- **初步分析：**

- 线程「死循环」
- 线程「死锁」

- **具体定位步骤：**

1. top 命令：Linux 命令，查看实时的 CPU 使用情况，获得 pid
2. top 命令：Linux 命令，查看进程中线程使用 CPU 的情况，记录 tid
3. jstack 命令：jvm 命令，查看指定进程下，所有线程的调用栈和执行状态
  - 根据 top 命令获取的 tid（转换为 16 进制）
  - 找出目标线程的调用栈和执行状态
4. 分析「目标线程」调用栈和执行状态，对应到源代码，修复问题

# CPU 占用过高

## 查看进程

top命令 `top` - `P/M`, 结果:

1.	PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2.	4796	storm	20	0	9935m	264m	13m	S	134.5	0.8	3745:18	java

## 参看线程


top命令 `top -H -p 4796`, 结果:

1.	PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2.	4967	storm	20	0	3582m	256m	12m	S	0.7	0.8	72:37.82	java
3.	4915	storm	20	0	3582m	256m	12m	S	0.3	0.8	7:44.29	java

## 查看JVM中线程详细信息

4967是最耗CPU的线程，转换成16进制1367，再用 `jstack` 命令查看线程堆栈：

```
1. [storm@cib02166 temp]$ jstack -l 4796 | grep 1367 -A 20
2. "Thread-2" prio=10 tid=0x00007f7194445800 nid=0x1367 waiting on condition [0x00007f7170ccb000]
3.   java.lang.Thread.State: WAITING (parking)
4.       at sun.misc.Unsafe.park(Native Method)
5.       - parking to wait for <0x00000000f07f4178> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
6.       at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
7.       at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
8.       at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
9.       at backtype.storm.event$event_manager$fn__2467.invoke(event.clj:39)
10.      at clojure.lang.AFn.run(AFn.java:24)
11.      at java.lang.Thread.run(Thread.java:745)
12.
13.   Locked ownable synchronizers:
14.       - None
15.
16. "Thread-1" prio=10 tid=0x00007f7194507800 nid=0x1366 waiting on condition [0x00007f71710cf000]
17.   java.lang.Thread.State: WAITING (parking)
18.       at sun.misc.Unsafe.park(Native Method)
19.       - parking to wait for <0x00000000f07e2e90> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
20.       at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
21.       at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
22.       at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)  __
```



# 内存占用过高：OOM, OutOfMemoryError

- **现象**：java.lang.OutOfMemoryError: **Java heap space**
- **原因**：JVM 堆内存不足，2 类原因：
  - 堆内存**设置不够**，参数-Xms、-Xmx来调整
  - **内存泄漏**：jmap 输出「堆转储快照」，通过工具查看 GC Roots 的引用链，定位出泄漏代码的位置
- 常见：
  - 业务代码中创建了大量「**大对象**」，并且长时间不能被垃圾收集器收集（存在被引用），此时，通过 jmap 命令输出「堆转存快照」，分析后，优化业务逻辑（老年代空间不足）
  - 不限长度的**队列**，**一直生产**对象，**没有消费**对象，导致内存溢出



# 内存占用过高：OOM, OutOfMemoryError

- **现象**：java.lang.OutOfMemoryError: PermGen space
- **原因**：JVM 「永久代」不足，3 类原因：
  - 「永久代」 **设置不够**，参数 -XX:MaxPermSize 来调整
    - 32位机器默认 64M
    - 64位的机器默认 85M（指针膨胀）
  - 程序启动时，加载**大量的第三方jar包**
  - ASM（编译期）、CGLib（运行时），等**动态代理**技术，生成大量的 Class

# 内存占用过高：OOM, OutOfMemoryError

- **现象：** java.lang.OutOfMemoryError: null
- **原因：** 「直接内存」空间不足
- 分析过程：
  - 检查一下是否使用 Java NIO
- **解决办法：** 设置「直接内存」大小
  - 默认，跟 Java 堆一样大小，即，参数 -Xmx 的取值
  - MaxDirectMemorySize 设置「直接内存」
  - 注意：约束「直接内存」+「Java 堆」 < 「OS 物理内存」

# 参考资料

- 《深入理解 Java 虚拟机— JVM高级特性与最佳实践（第2版）》
- 《实战 Java 虚拟机—JVM故障诊断与性能优化》
- [Java Garbage Collection Basics](#) (Oracle)
- [The Java Language Specification](#) Java SE 6\7\8
- [The Java Virtual Machine Specification](#) Java SE 6\7\8
- <http://www.infoq.com/cn/articles/Java-PERMGGEN-Removed/> Java 8 移除永久代
- <http://ifeve.com/useful-jvm-flags-part-4-heap-tuning/> JVM 参数调优



# 补充：直接内存

- **JVM 中，「直接内存」、Java 堆，之间的关系**
  - 直接内存：不属于运行时数据区，不受 Java 堆大小的限制，即，不受-Xmx 限制；
- **如何设置「直接内存」大小？**
  - 默认，跟 Java 堆一样大小，即，参数 -Xmx 的取值
  - MaxDirectMemorySize 设置「直接内存」
- **什么时候会使用「直接内存」？**
  - JDK 1.4 的 NIO，有一个缓冲区（Buffer），调用 Native 方法，使用的直接内存，减少数据的复制次数
  - JDK 1.7，「方法区」内部的「常量池」，迁移到「直接内存」中
- **「直接内存」的垃圾回收：什么时候进行？如何进行？**
  - JVM 会回收「直接内存」
  - 「直接内存」回收，跟 Java 堆的新生代、老年代不同，无法在发现「直接内存」空间不足时，通知垃圾回收器，来回收。
  - 老年代进行 Full GC 时，会**顺便清理一下「直接内存」**的废弃对象。

# 补充：永久代

- 永久代，是否会回收？

- 永久代，存储「方法区」的内容，主要是「类」和常量、静态变量
- 其中，「类」占用空间很大，判断「无用的类」，需要同时满足：
  1. 类所有的实例都已被回收
  2. 加载该类的 ClassLoader 也被回收
  3. 类对应的 Class 对象，没有被任何地方引用，主要是无法通过反射获取

- 永久代，什么时候会回收？

- 参数 -Xnoclassgc 控制是否开启永久代回收
- 永久代满了，会触发 Full GC，如果 GC 后空间仍然不足，会抛出 OOM: PermGen space
- 永久代和老年代捆绑在一起，无论谁满了，都会触发永久代和老年代的垃圾收集

# 补充：JDK8 的差异

- 有哪些差异？
  - 方法区：放在「直接内存」中，永久代的参数-XX:PermSize和-XX:MaxPermSize也被移除
  - 永久代：去除「永久代」概念，避免 OOM 问题（因为参数 MaxPermSize 约束了永久代大小）
- 带来的收益？不同？
  - JDK7 中，永久代的最大空间一定得有个指定值，而如果MaxPermSize指定不当，就会OOM
    - 「永久代」参数 -XX:MaxPermSize 来调整
      - 32位机器默认 64M
      - 64位的机器默认 85M（指针膨胀）
  - JDK8 中，-XX:MetaspaceSize 和-XX:MaxMetaspaceSize设定「方法区」大小，但如果不指定，Metaspace的大小仅受限于「直接内存」大小，上限是物理内存大小

Q & A