复旦大学计算机科学技术学院

编程方法与技术

B.7. 垃圾回收(GC)机制

周扬帆

2021-2022第一学期

垃圾回收机制的来由

- 回忆C/C++的内存管理
  - new/delete
  - malloc/free
  - 堆(heap)/栈(stack)
    - → 谁需要垃圾回收

垃圾回收机制的来由

- Java内存管理思想
  - 交由程序员进行内存管理容易出错
  - JVM接管内存管理
    - → JVM可实现内存管理的原因：无灵活的指针操作
  - 内存管理的要素
    - → 内存分配和回收

垃圾回收机制

- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

垃圾回收机制

- **时间停止 – Stop-the-world (STW)**
  - GC期间，很多环节下所有线程都需要暂停
  - 因此GC需要优化、节省STW时间
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代收集算法
  - 垃圾收集器

垃圾回收机制

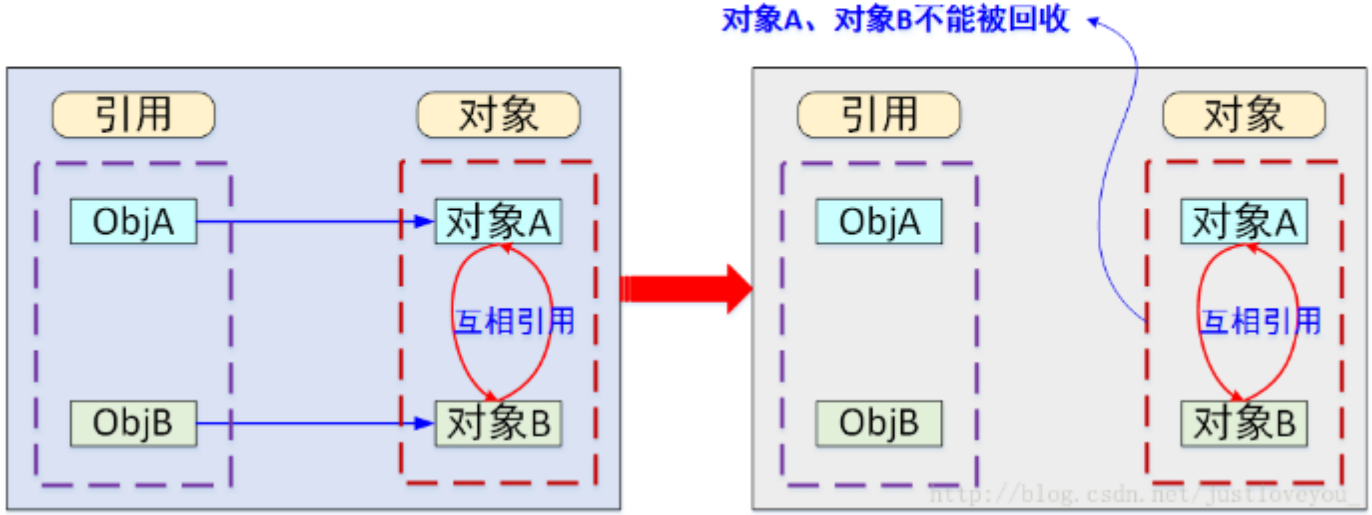
- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代收集算法
  - 垃圾收集器

引用计数器

- **引用计数器：** 记录对象的引用数量
- **规则**
  - 任何其它变量被赋值为这个对象的引用时
    - → 对象的引用计数器+1
  - 当一个对象实例的某引用超过生命周期
    - → 对象的引用计数器-1
  - 当一个对象实例的某引用被设置为新值
    - → 对象的引用计数器-1
  - 当一个对象实例被垃圾收集时
    - → 它引用的任何对象实例的引用计数器均-1

引用计数器

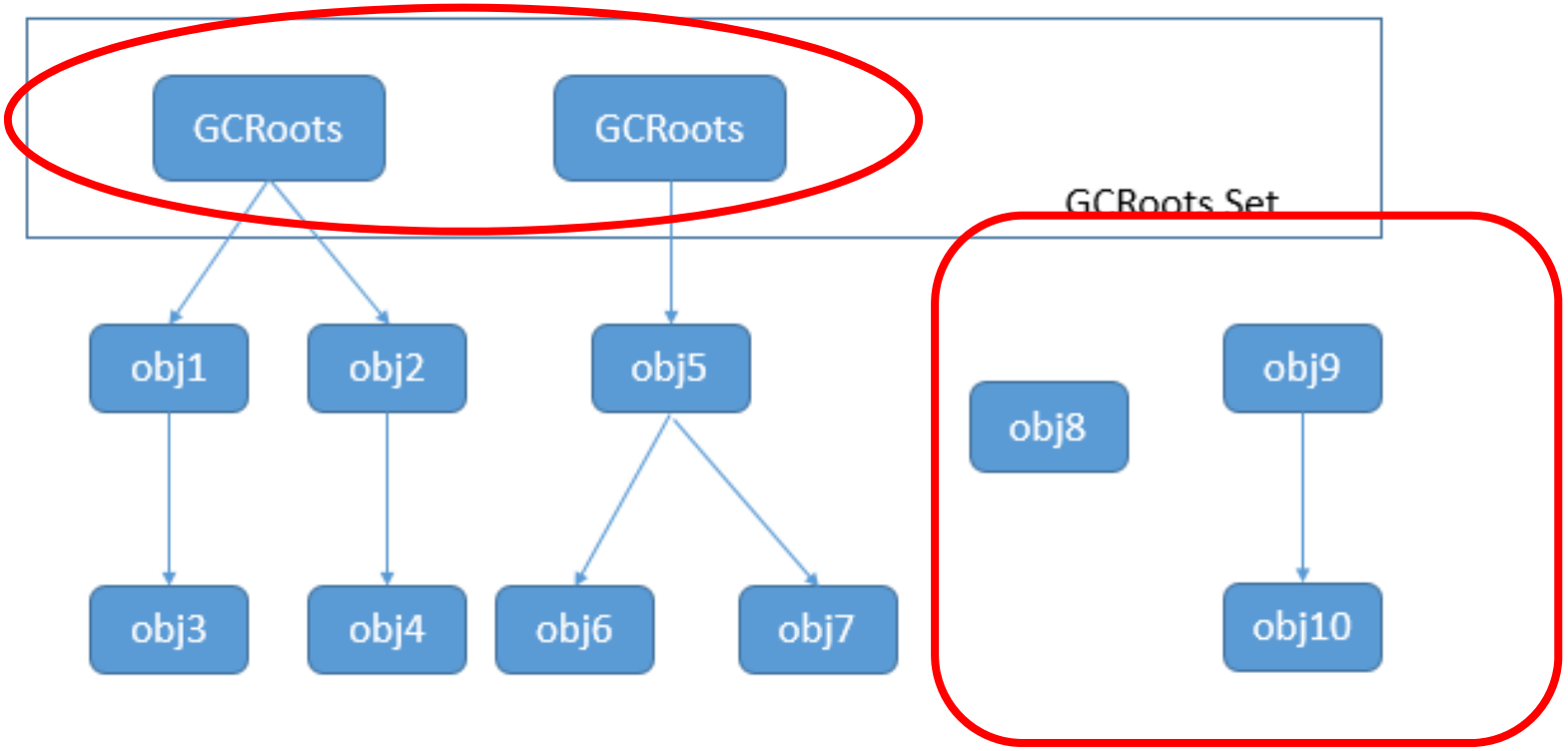
- **规则**
  - 计数器为0 → 没用了，垃圾回收的时候可回收
- **问题**
  - 难以解决循环依赖



  - Java现已不用!

可达性分析

- **规则**
  - 通过构建引用链，侦测没有依赖的对象
  - 当一个对象到GC Root没有路径时 → 可GC



可达性分析

- **GC Root: 正运行的程序可访问的引用变量**
  - 虚拟机栈(局部变量)中的对象引用
  - 本地方法栈中native方法的对象引用
  - 方法区中类的静态属性的对象引用
  - 方法区中常量的对象引用
- **从GC Root出发找它们引用的对象**
  - 递归 → 引用链
- **引用链上没有的对象**
  - 可GC

可达性分析

- GC Root: 正运行的程序可访问的引用变量
- 从GC Root出发找它们引用的对象
  - 递归 → 引用链
- 引用链上没有的对象
  - 可GC

可达性分析

- GC Root: 正运行的程序可访问的引用变量
- 从GC Root出发找它们引用的对象
  - 递归 → 引用链
- 引用链上没有的对象
  - 可GC
- 例子

```
Object aobj = new Object ( ) ;
Object bobj = new Object ( ) ;
Object cobj = new Object ( ) ;
aobj = bobj;
aobj = cobj;
cobj = null;
aobj = null;
```

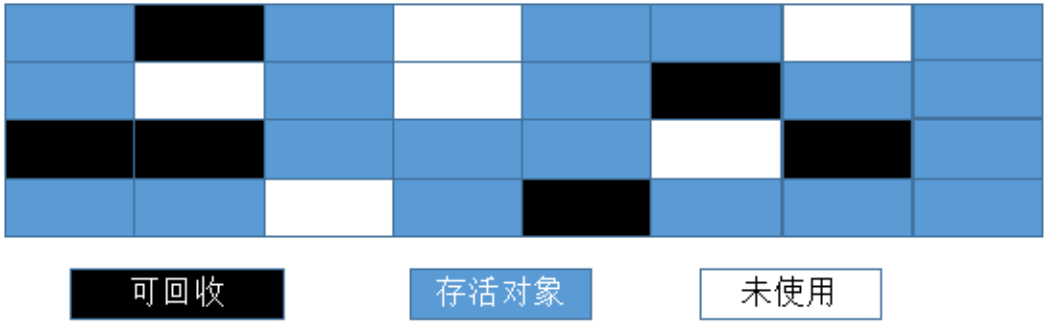
垃圾回收机制

- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

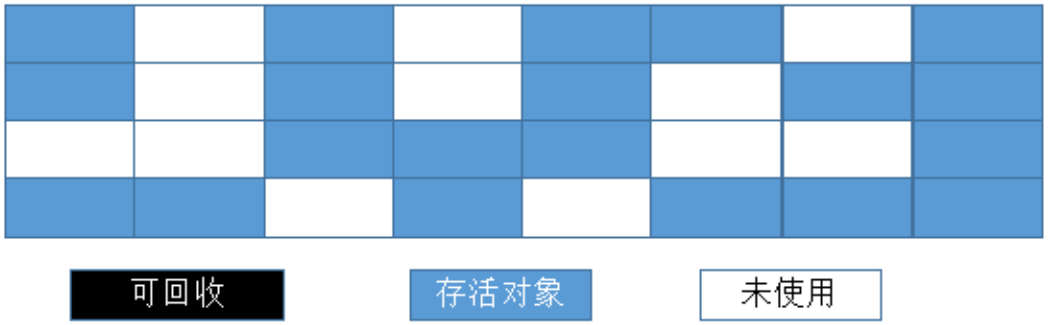
标记-清除算法

- **算法 (Mark-Sweep)**
  - 从GC Root出发，标记存活对象
  - 标记完毕后，对未标记对象进行清除
- **问题**
  - 内存空间不连续

回收前



回收后



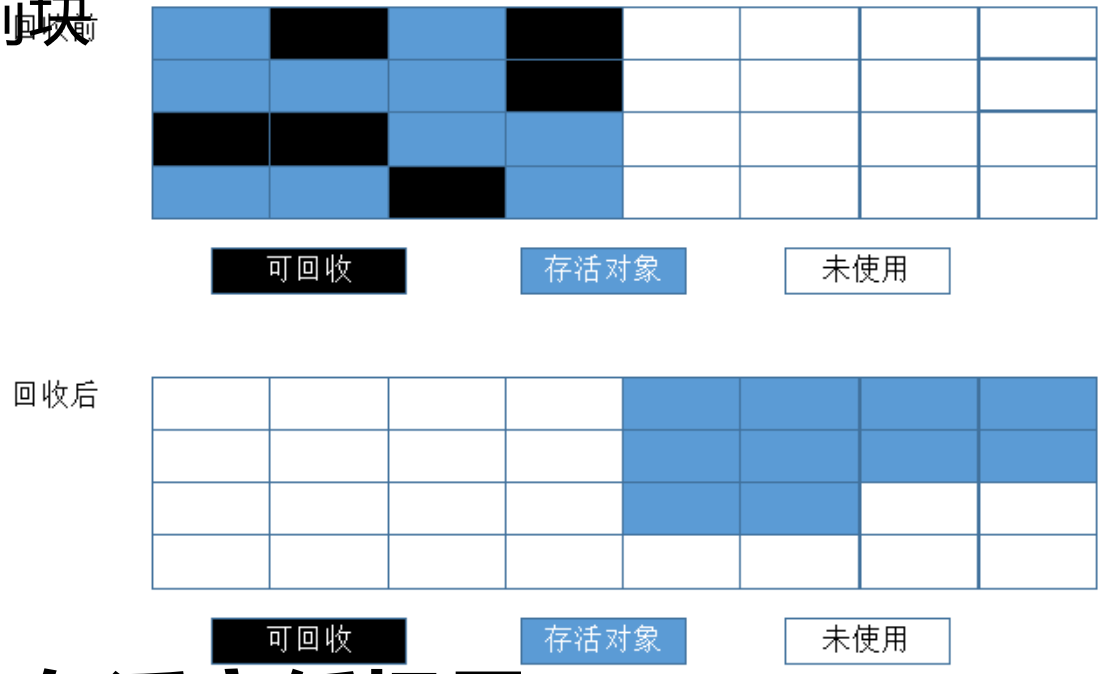
标记-整理算法

- **算法 (Mark-Compact)**
  - 从GC Root出发，标记存活对象
  - 标记完毕后，整理存活对象

- **特点**
  - 拷贝整理太慢
  - 整理之后内存连续

复制算法

- **算法 (Copying)**
  - 将空间分成等大小两块，每次只用一块
  - 当前块用完后，将这块存活对象移到另一块
  - 设置另一块为当前块
  - 重复上述过程



- **额外空间：适合存活率低场景**

垃圾回收机制

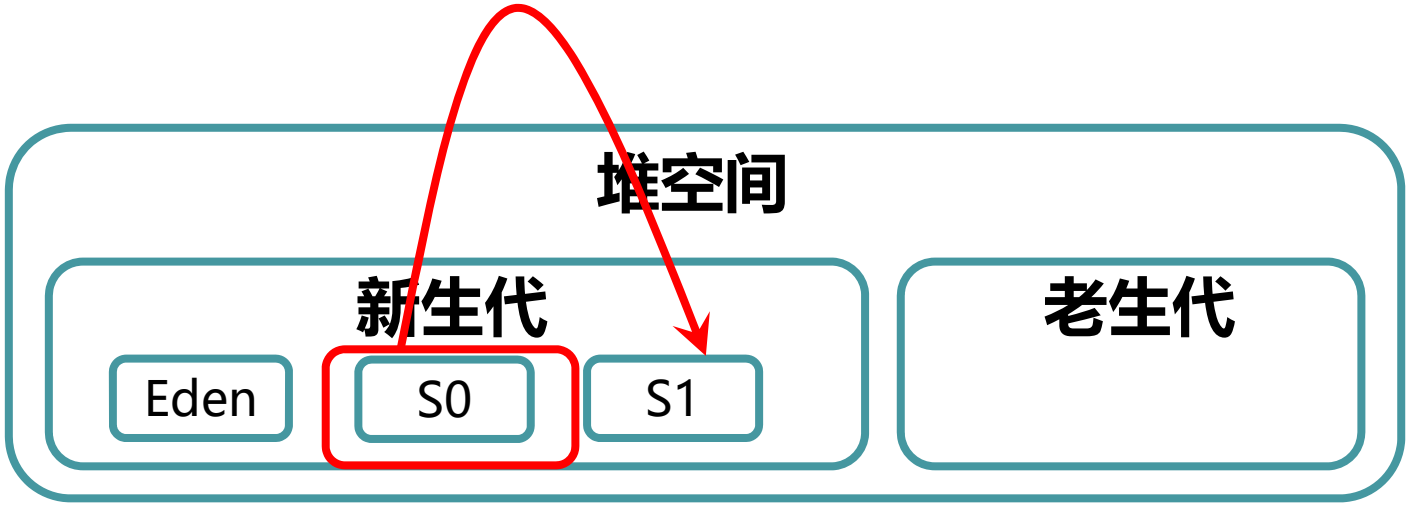
- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

JAVA内存管理+

- 栈：虚拟机栈/native方法栈
- 方法区 (静态区)
  - 方法代码
  - 常量池
- 堆

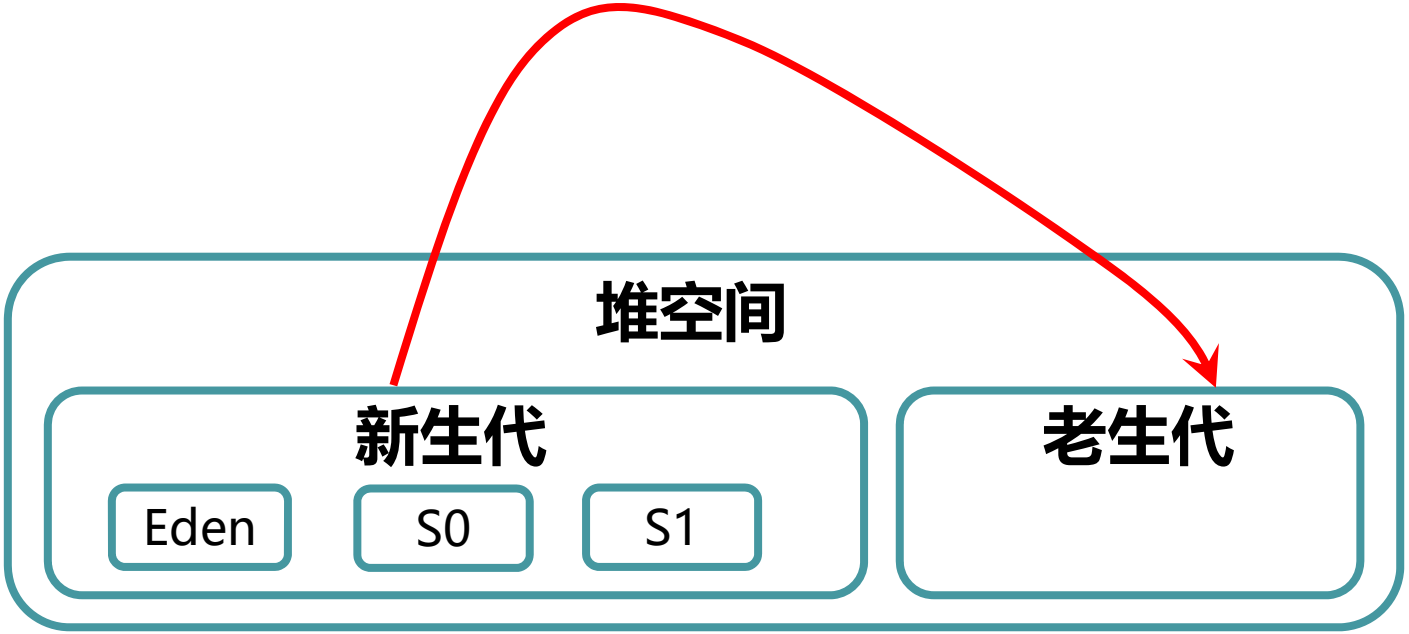
新生代GC: Minor GC

- 对象创建在Eden
- Eden回收，存活 → S0
- S0回收，存活 → S1, 交换S0/S1
- 采用复制算法原因?
- Eden: Survivor0: Survivor1 = 8:1:1



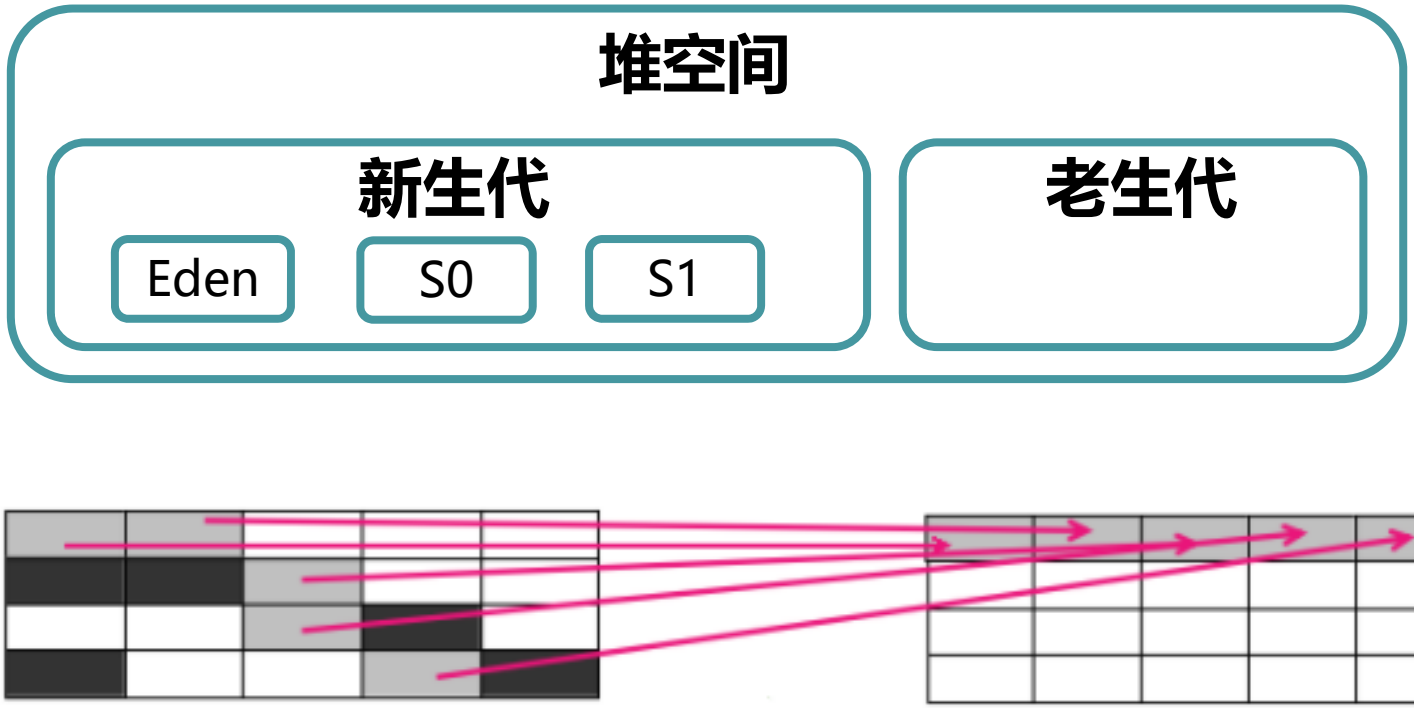
新生代GC

- 在Survivor GC中存每一个对象的counter
  - 如果对象存活，counter ++
- counter到一定大小，将对象移去老生代



老生代GC: Full GC

- 老生代一般采用标记整理算法

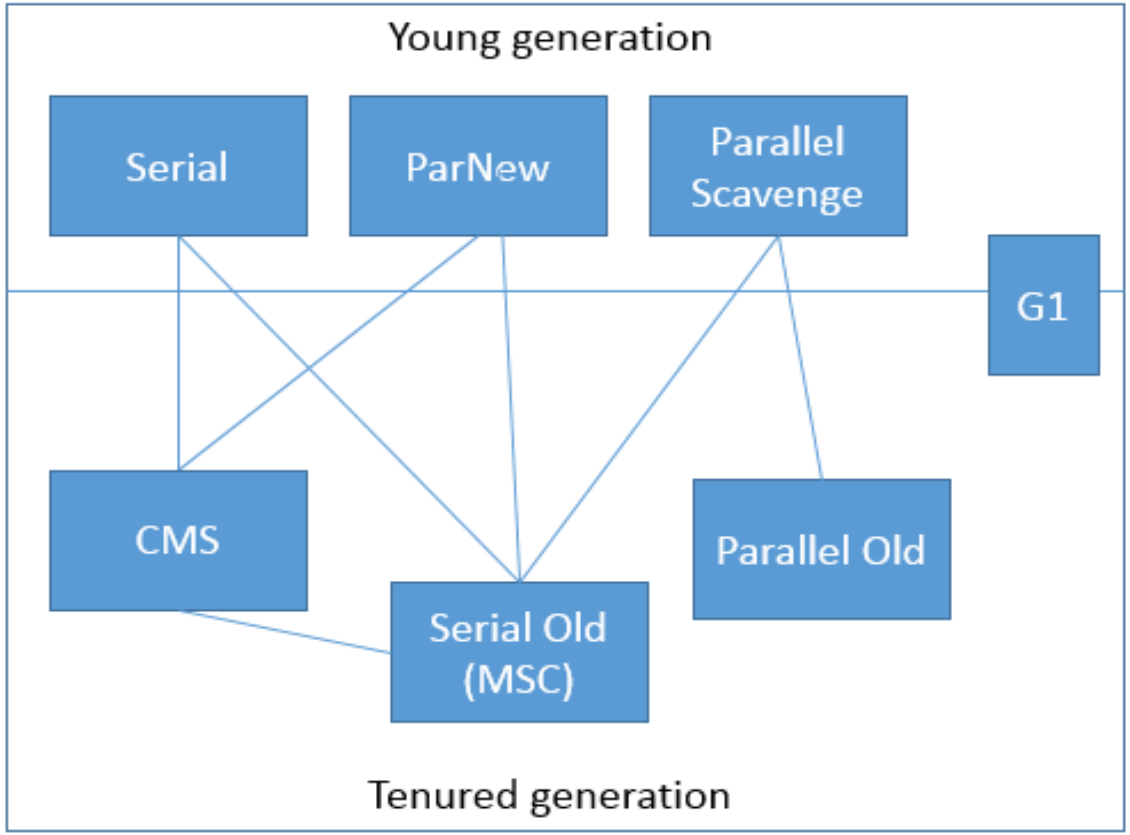


垃圾回收机制

- **哪些内存需要回收**
  - 错误回收 → 程序出错
  - 经典算法：引用计数法、可达性分析
- **如何进行内存回收**
  - 标记清除算法、复制算法、标记整理算法
  - 分代回收算法
  - 垃圾回收器

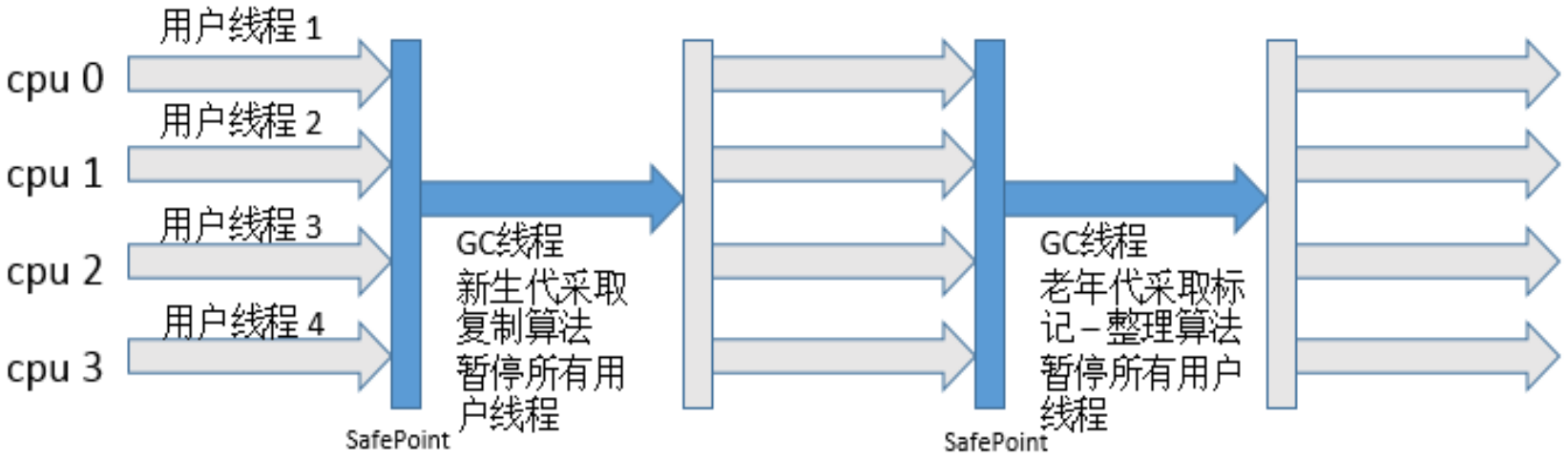
垃圾回收器概览

- 垃圾回收器
  - 虚拟机对垃圾回收算法的实现



Serial/Serial Old回收器

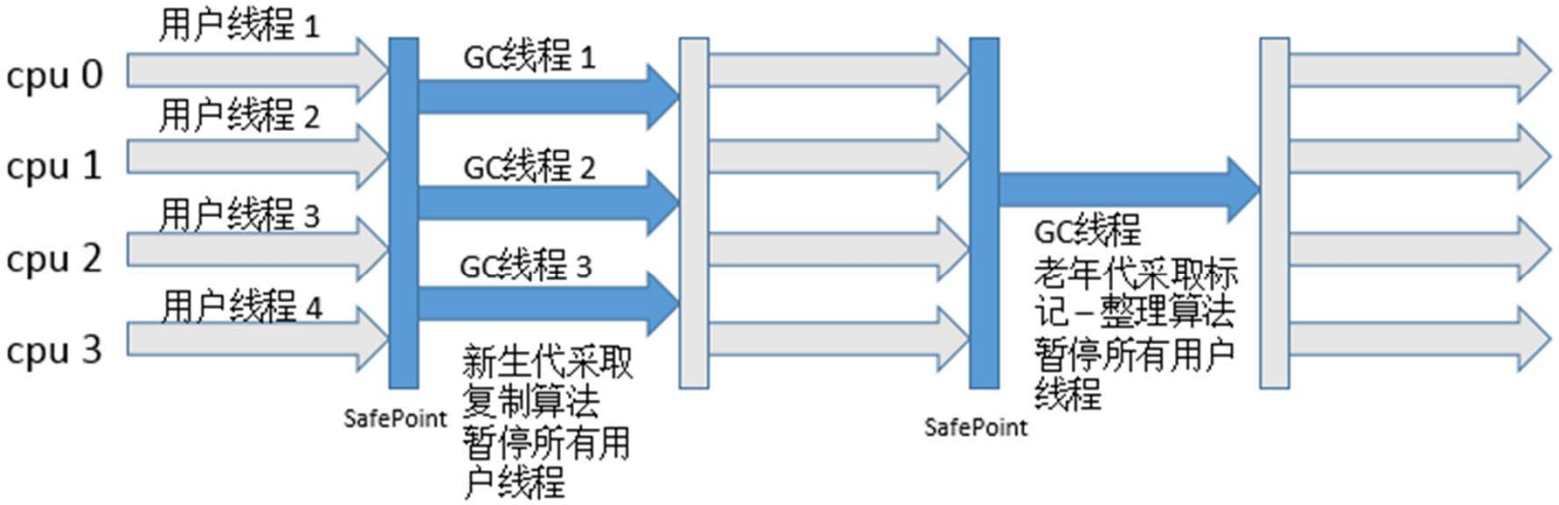
- 单线程回收器
- Serial回收器针对新生代采用Copying算法
- Serial Old回收器针对老生代采用Mark-Compact算法



Serial/Serial Old 收集器运行示意图

ParNew收集器

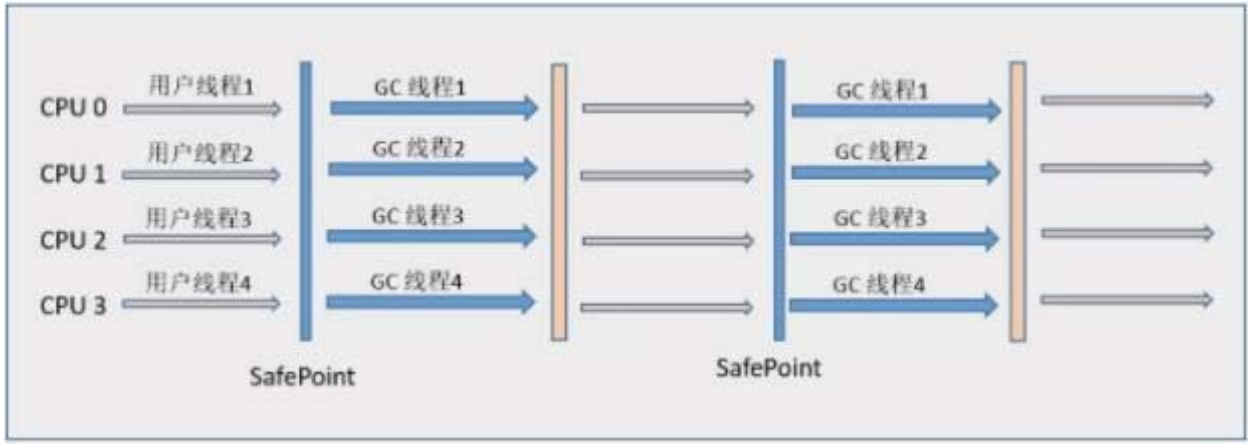
- Serial回收器的多线程版本



ParNew /Serial Old 收集器运行示意图

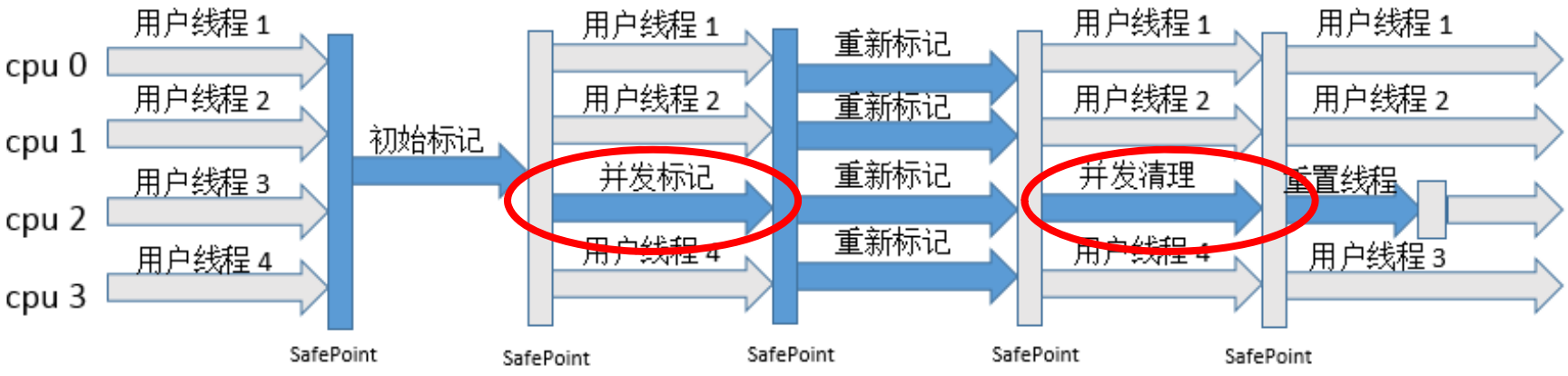
Parallel Scavenge/Parallel Old回收器

- **并行的多线程新生代回收器**
  - Parallel Scavenge采用copying算法
  - Parallel Old采用mark-compact算法
- **目标：追求高吞吐，也即高效利用CPU时间**
- **可自适应调节新生代大小，Eden和Survivor比例，新生代到老生代的年龄等参数**
- **适应后台对停顿时间相对不敏感的场合**
  - 服务器



CMS收集器

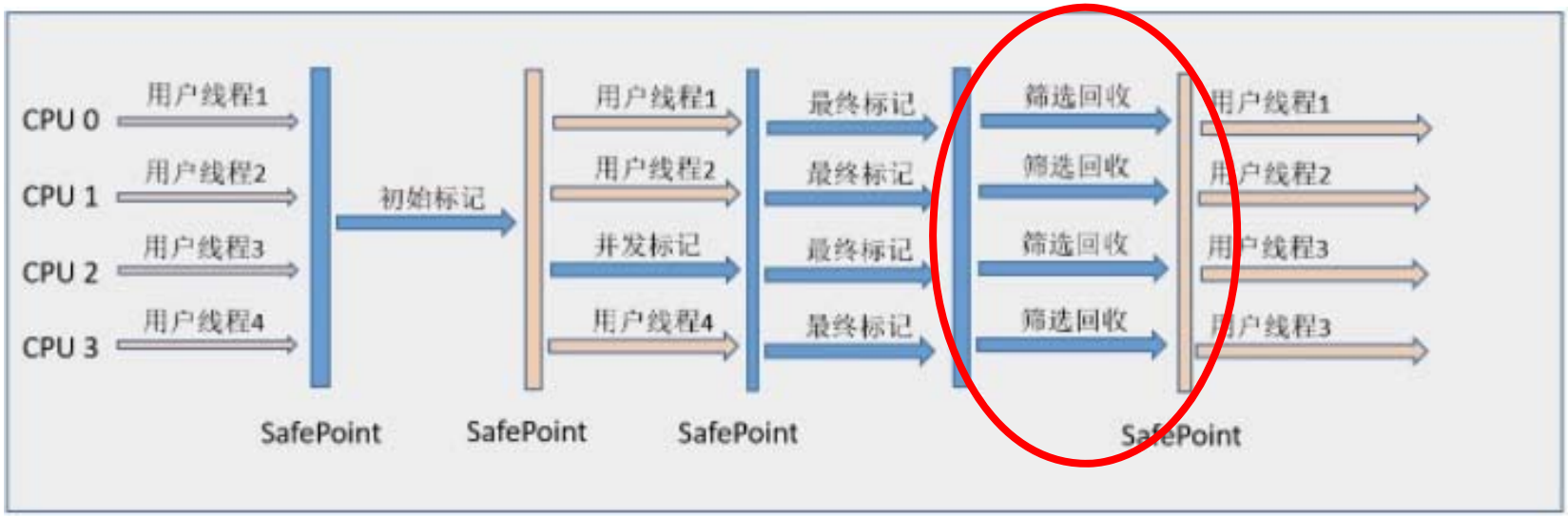
- 获取最短回收停顿时间为目标
- 采用mark-sweep算法
  - 初始标记: 标记GC Roots能直接关联到的对象，速度快
  - 并发标记: 进行GC Roots Tracing的过程，耗时长
  - 重新标记: 修正并发标记时因程序运行导致的标记变动
  - 并发清除: 回收



Concurrent Mark Sweep收集器运行示意图

G1回收器

- 以Region管理堆，新老生代对应Region集合
- 根据Region进行垃圾回收的价值维护优先级列表
  - 高优先级优先回收 （Garbage-First）



思考

- 有了GC，是否还需要考虑内存泄漏
- 查阅如何即时启动GC，思考你在编程中有没有即时启动GC的需求
- 比较Java的GC机制与你熟悉的语言（如Python）的GC机制，思考差异原因
  - Python用户大多数都在研究CV、NLP、统计、最优化…没时间去讨论垃圾回收等这些没用的东西