G1 & ZGC & Shenandoah

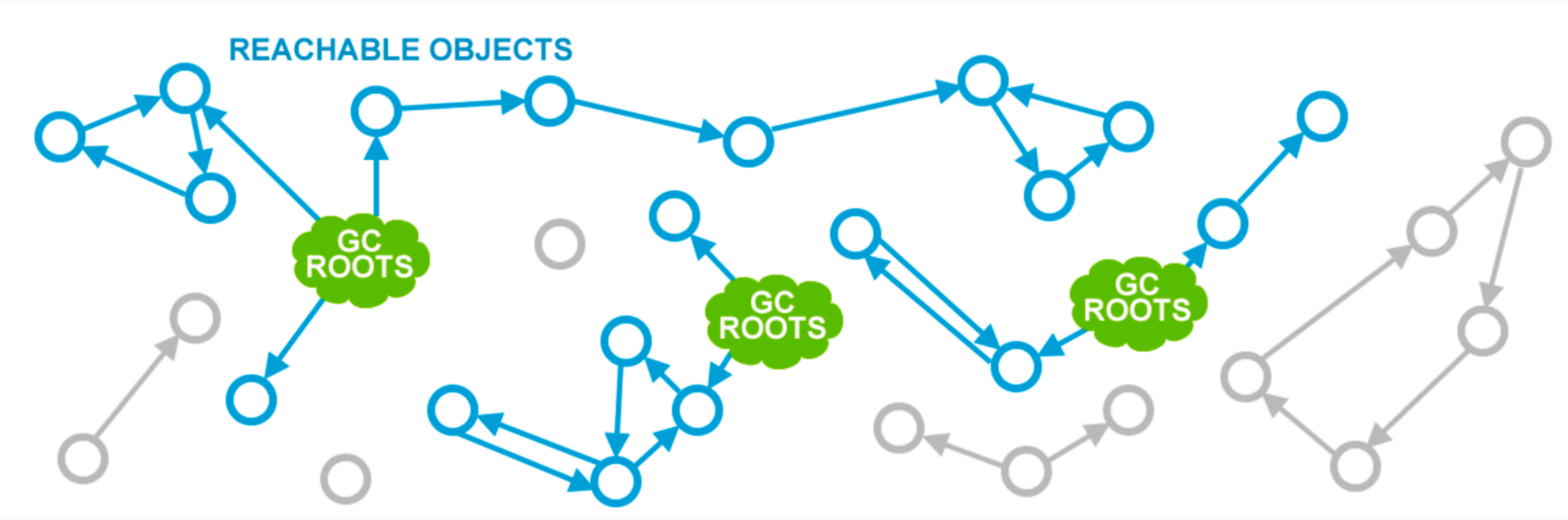
About me

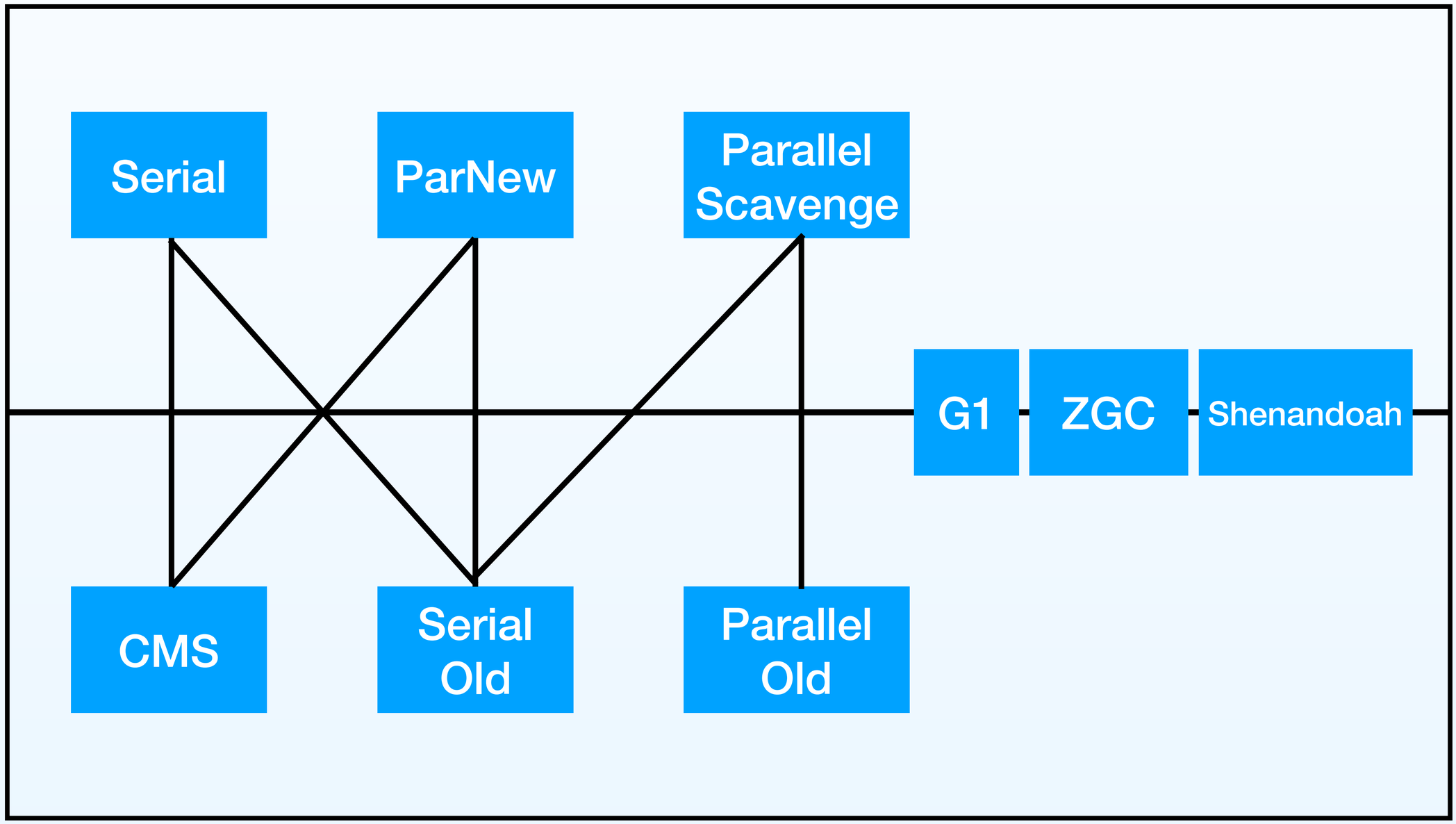
- 写Java的
- GitHub @blindpirate

目录

今天的课程有什么用?

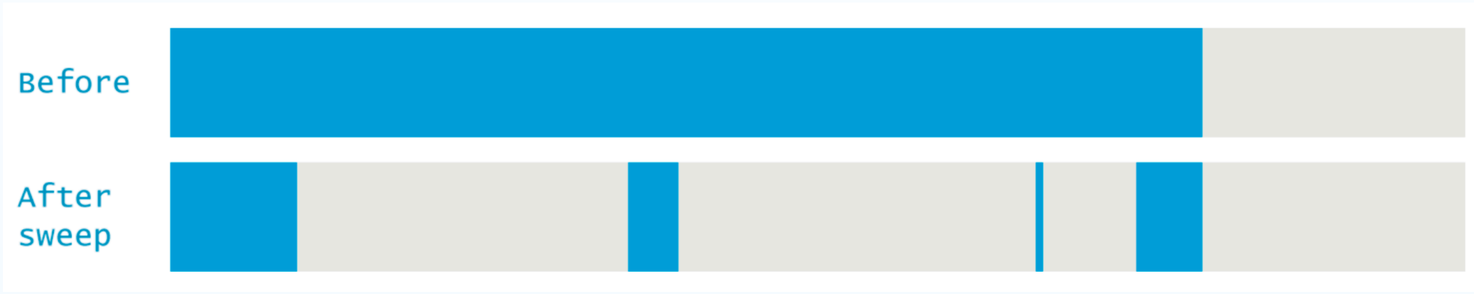
如果你还不知道GC是什么的话...



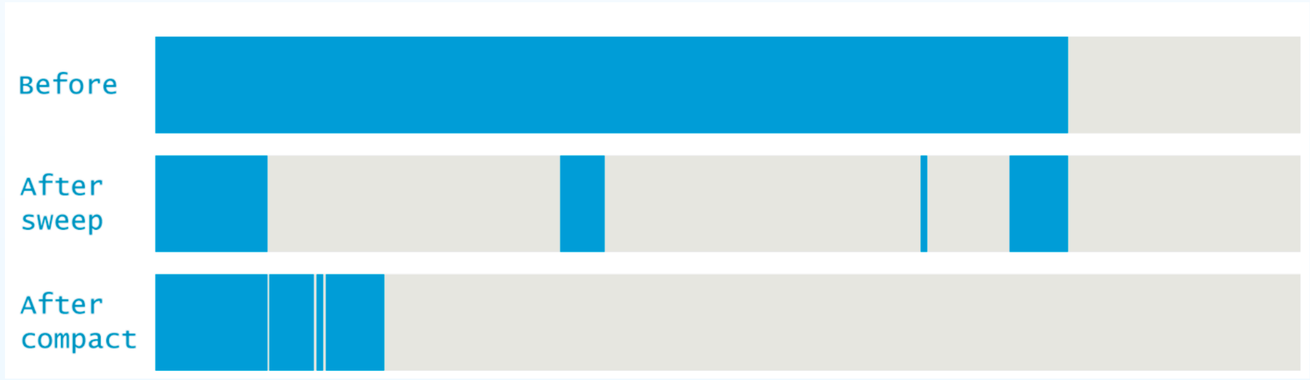

Serial
ParNew
Parallel Scavenge
G1
ZGC
Shenandoah
CMS
Serial Old
Parallel Old

如果让你来设计GC算法...

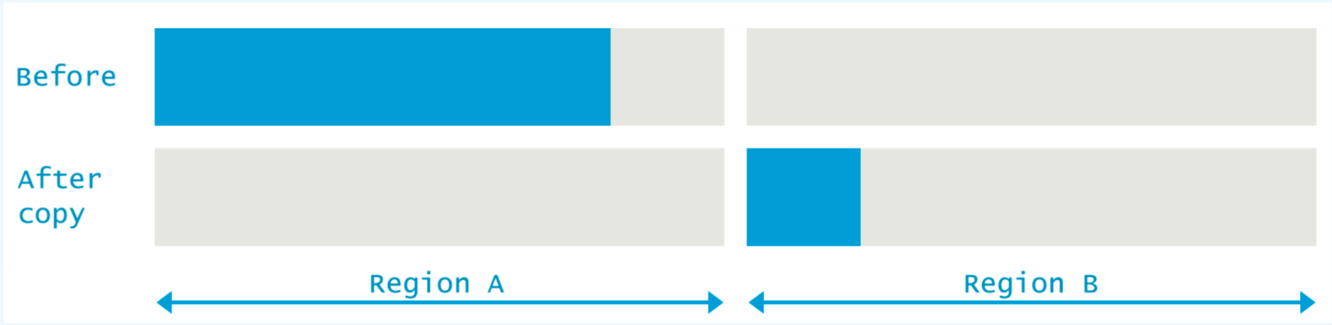
Mark-Sweep



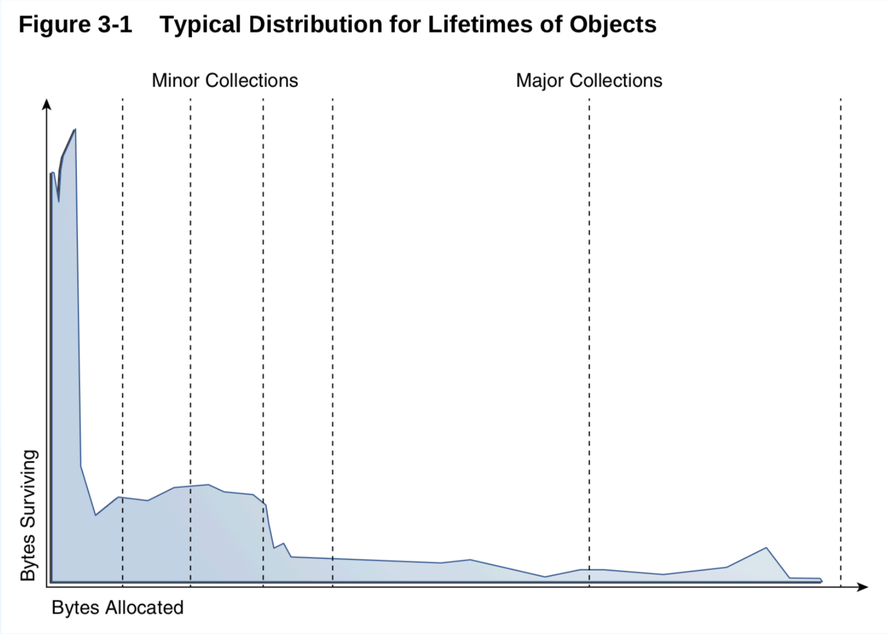
Mark-Sweep-Compact



Mark-Copy



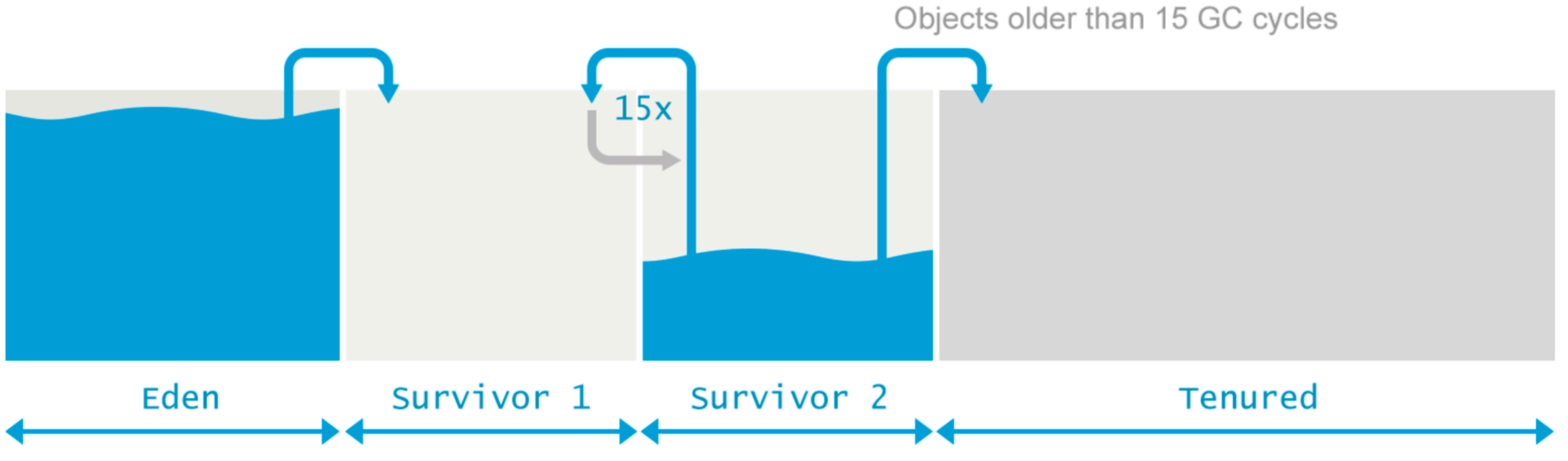
GC的分代假设



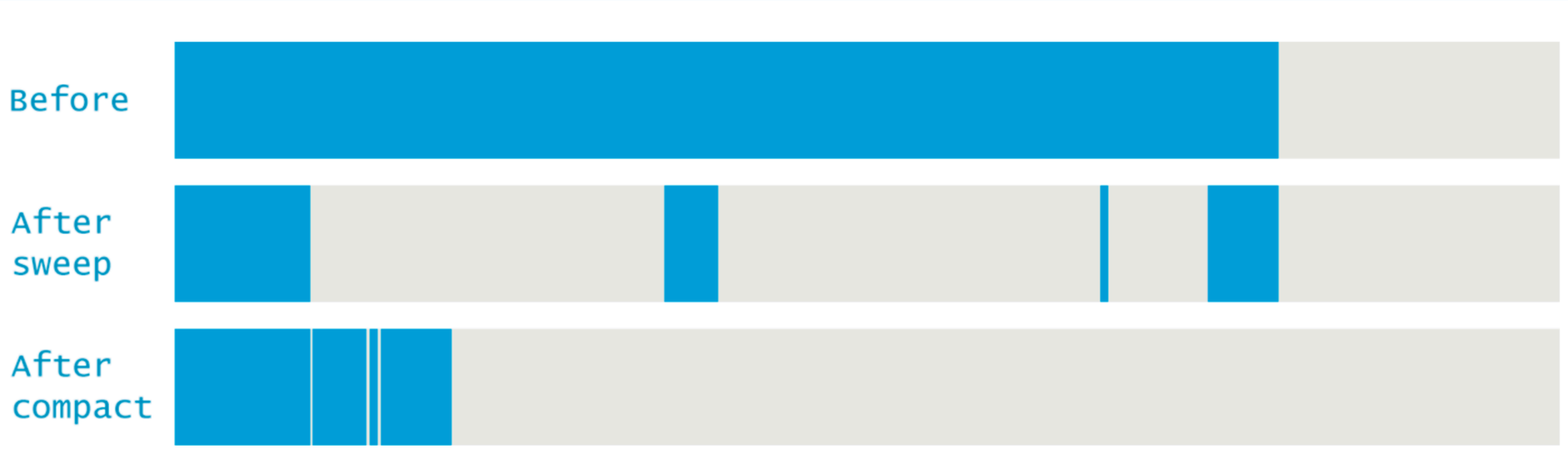
GC算法：古典时代

- Serial
  - 年轻代Serial
  - 老年代SerialOld
- Parallel
  - 年轻代Parallel Scavenge
  - 老年代Parallel Old

Mark-Copy

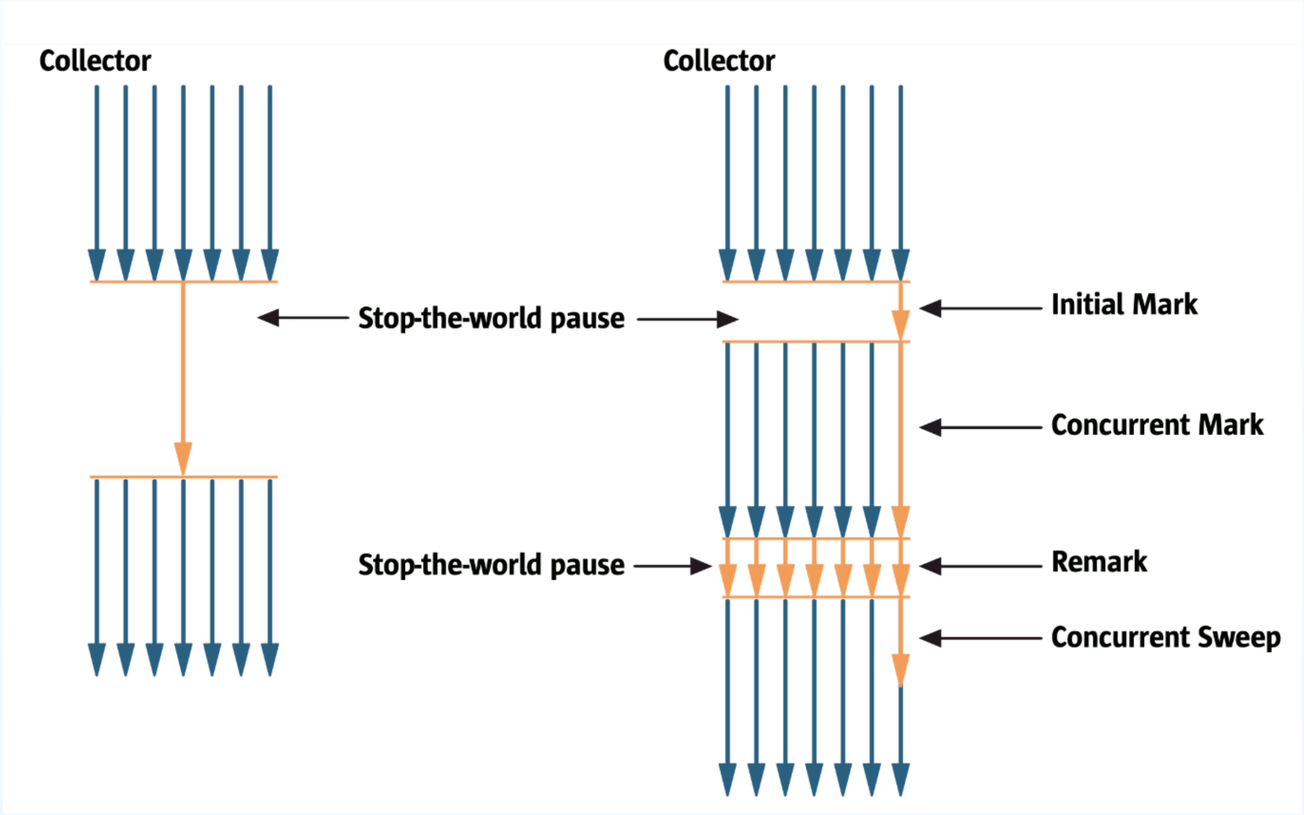


Mark-Sweep-Compact

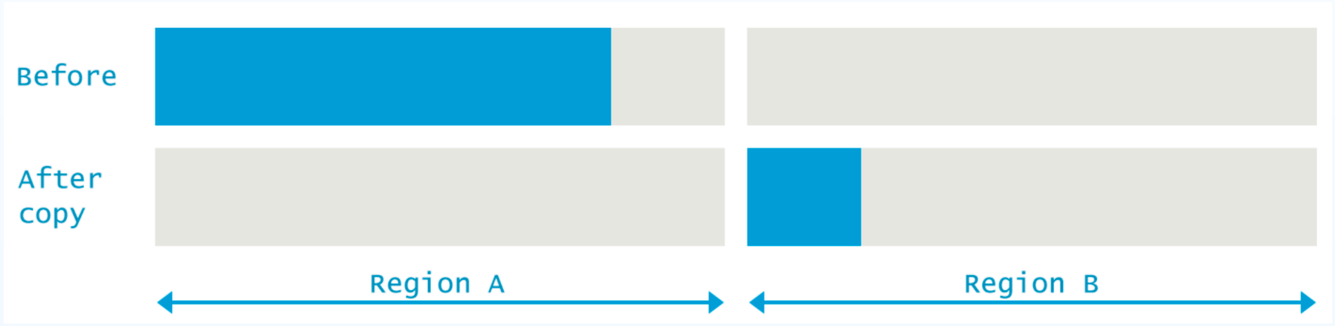


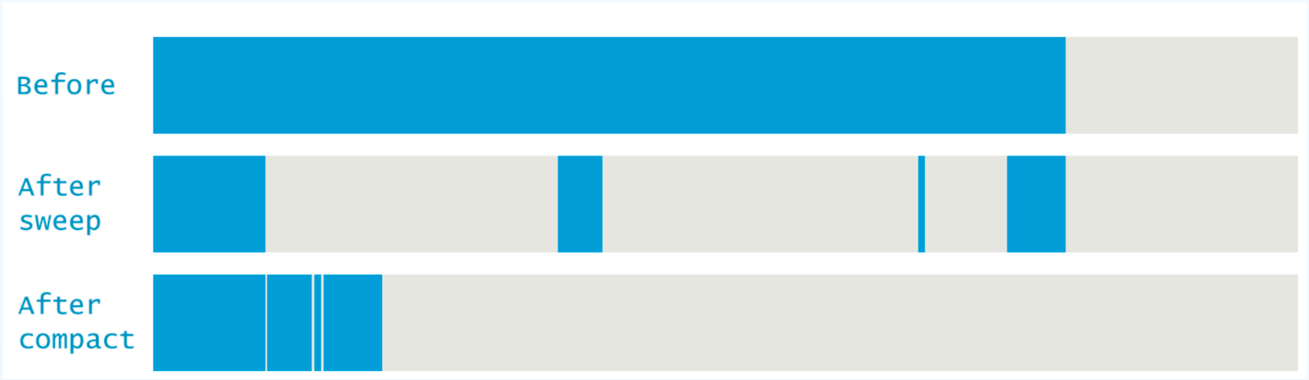
GC算法：中古时代

- CMS：Concurrent Mark Sweep
  - 低延时的系统
  - 不进行Compact
  - 用于老年代
  - 配合Serial/ParNew使用
  - Removed in JEP363

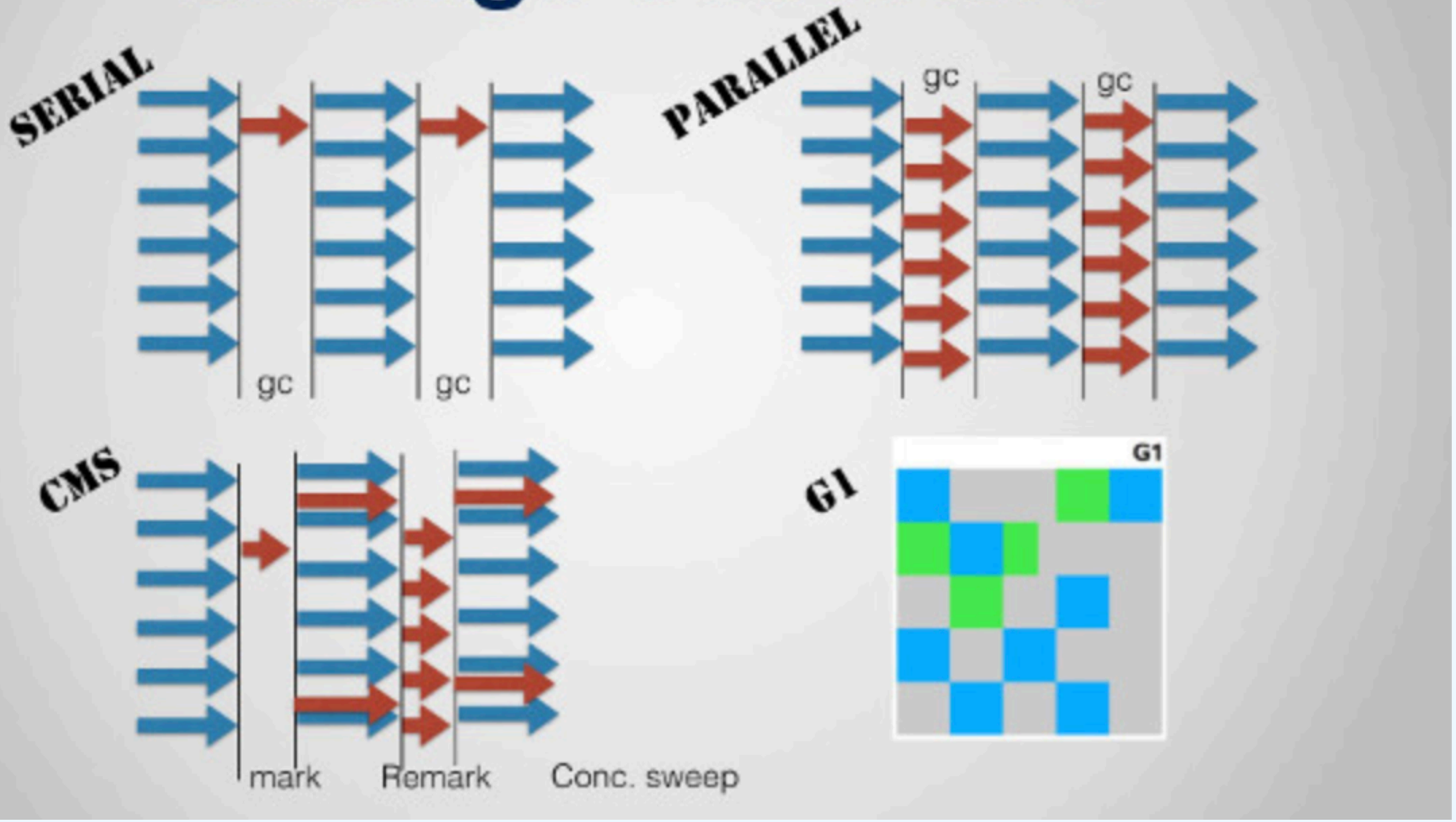


如果让你来设计GC算法...



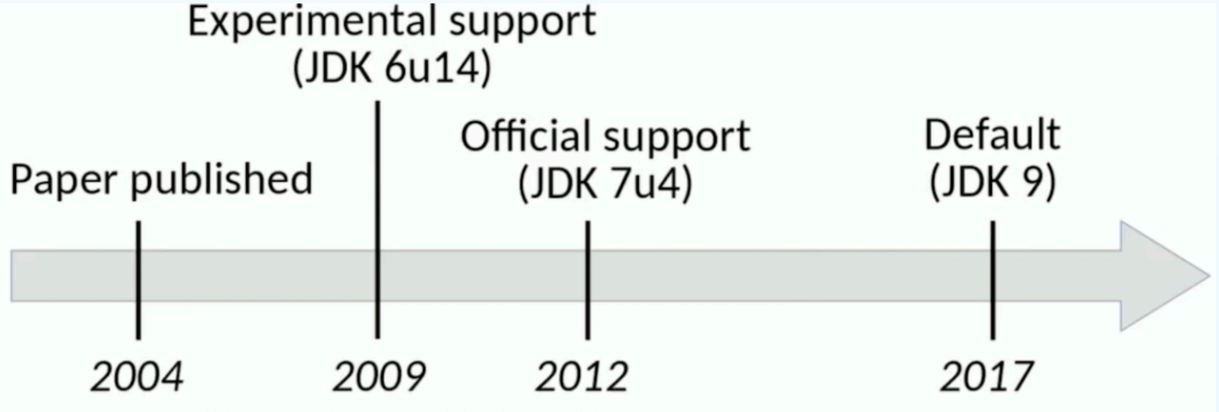


Garbage Collectors



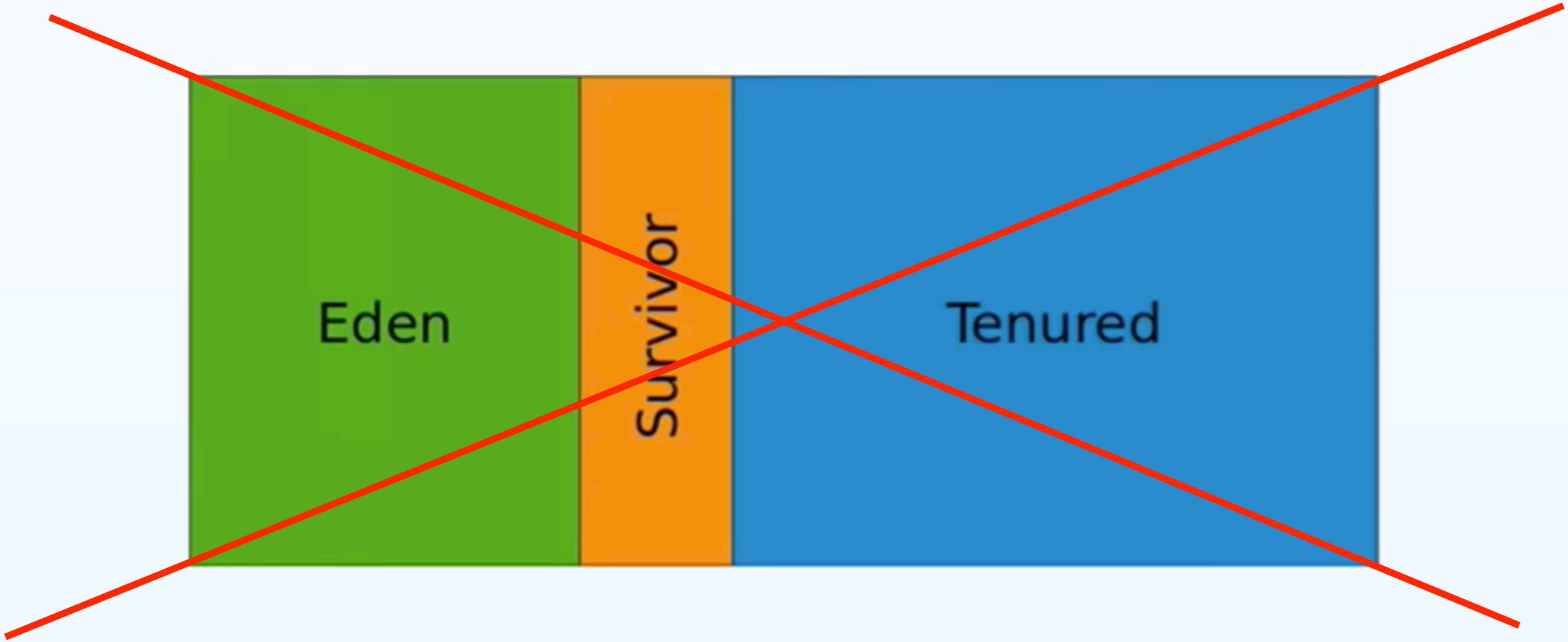
现代的GC算法：G1

- 软实时、低延时、可设定目标
- JDK9+的默认GC（JEP248）
- 适用于较大的堆（>4~6G）
- 用于替代CMS



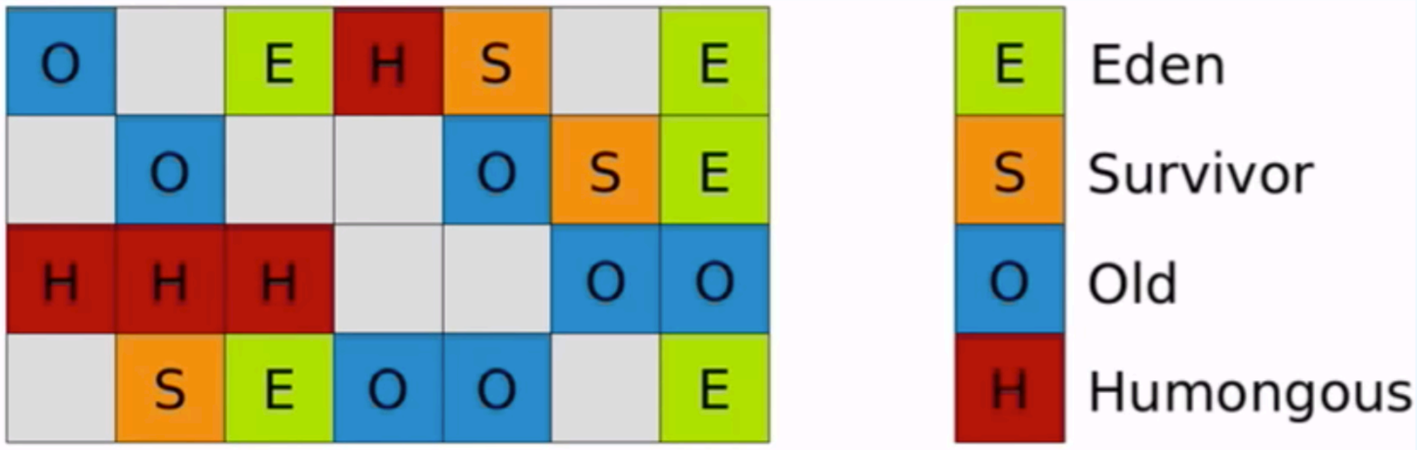
现代的GC算法：G1

- 可以设定最大STW停顿时间
  - –XX:MaxGCPauseMillis=N
  - 250 by default
- 年轻代GC算法
  - STW, Parallel, Copying
- 老年代GC算法
  - Mostly–concurrent marking (vs CMS)
  - Incremental compaction



G1的内存布局

- 将堆分成若干个等大的区域
- –XX:G1HeapRegionSize=N 2048 by default



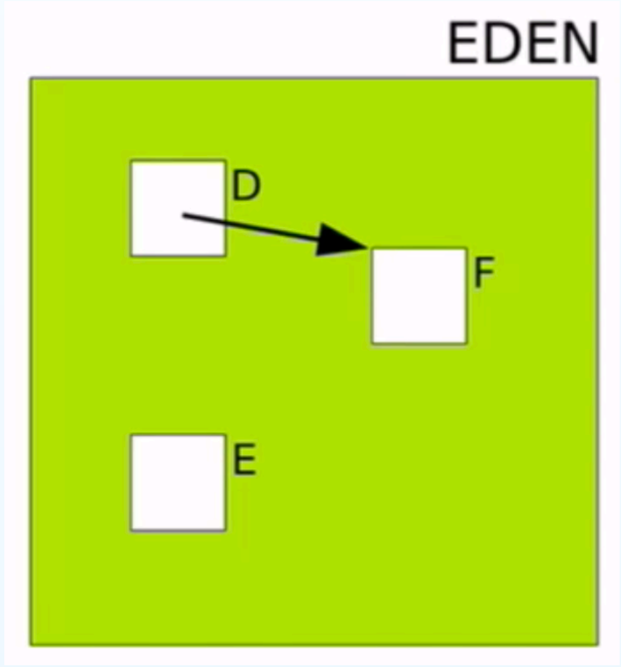
G1内部细节

- 无需回收整个堆，而是选择一个Collection Set (CS)
- 两种GC：
  - Fully young GC
  - Mixed GC
- 估计每个Region中的垃圾比例，优先回收垃圾多的Region
  - That’s why it’s called Garbage First

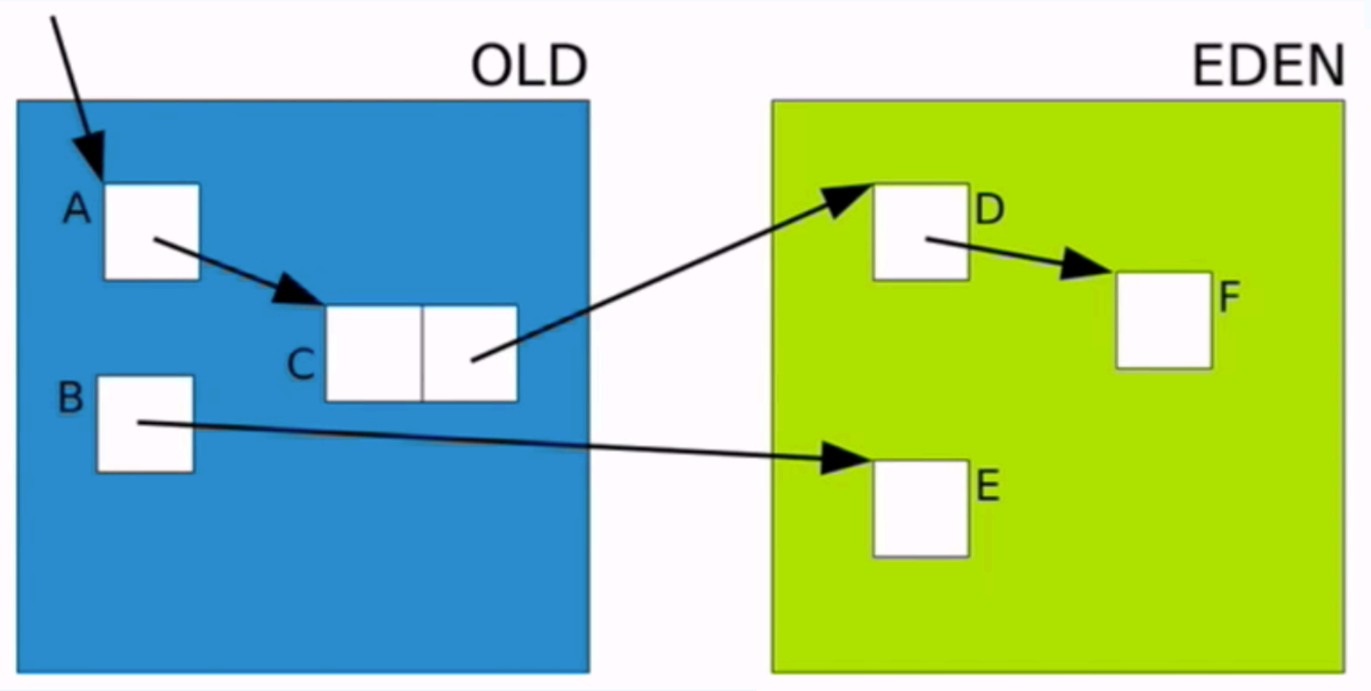
But how?

- 老年代对象可能持有年轻代的引用（跨代引用）
- 不同的Region间互相引用

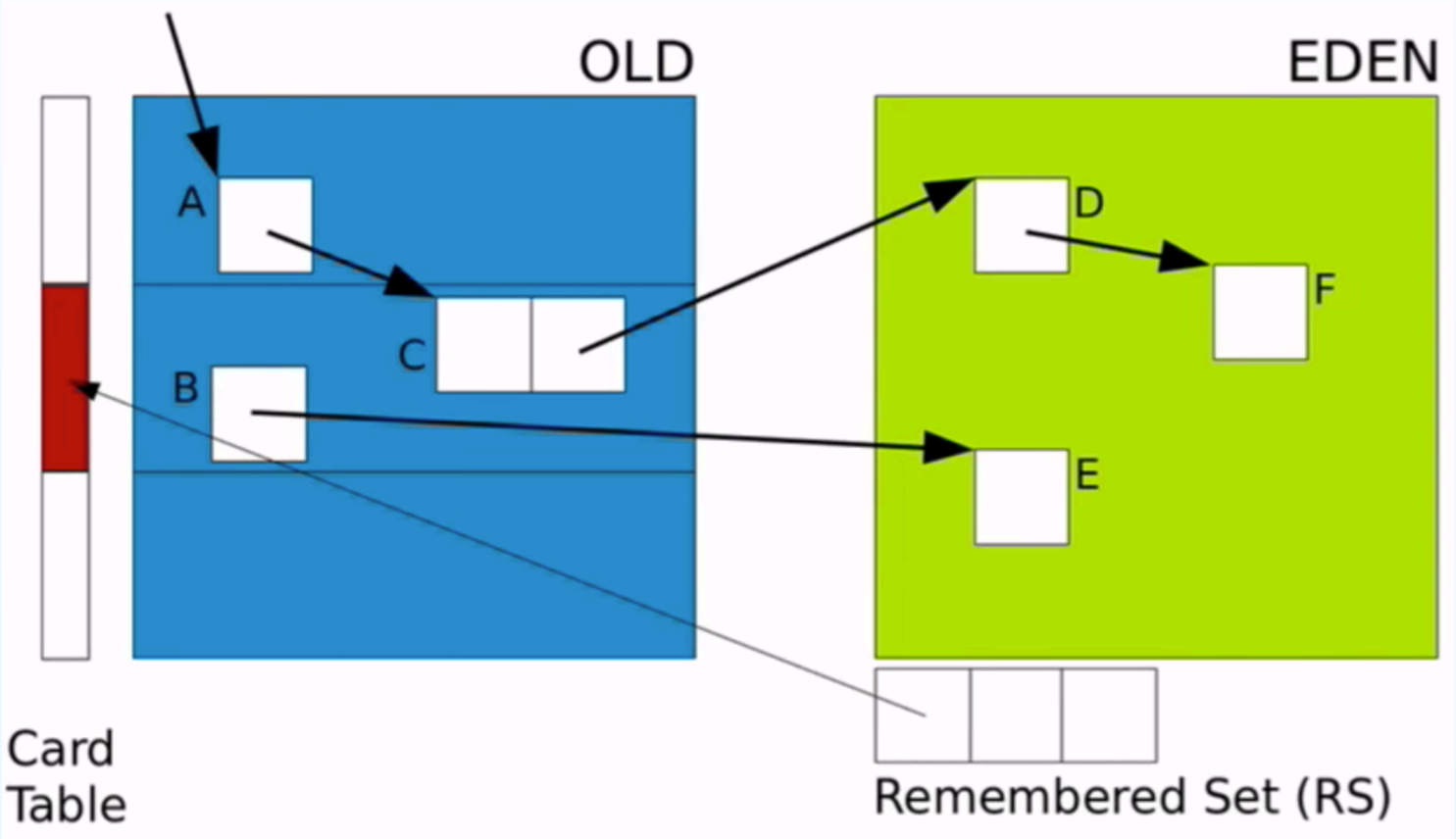
跨代/跨Region引用



跨代/跨Region引用

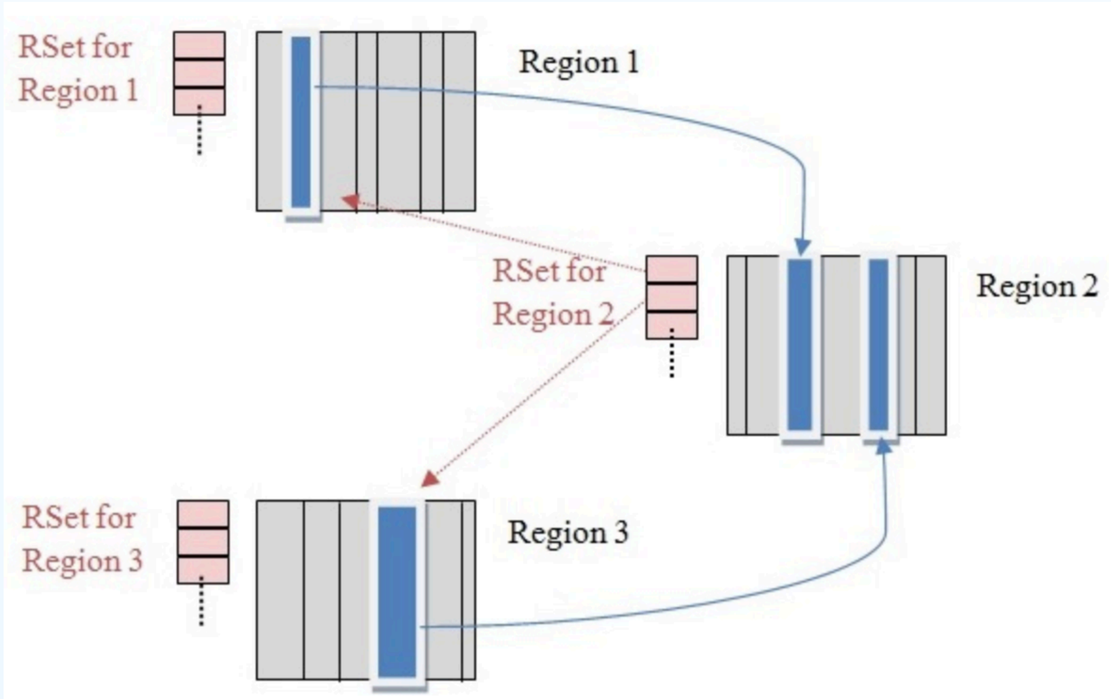


跨代/跨Region引用



Card Table & Remebered Set (RS)

- Card Table
  - 表中的每个entry覆盖512Byte的内存空间
  - 当对应的内存空间发生改变时，标记为dirty
- RememberedSet
  - 指向Card Table中的对应entry
  - 可找到具体内存区域
- 时间换空间
  - 用额外的空间维护引用信息
  - 5%~10% memory overhead



Write barrier

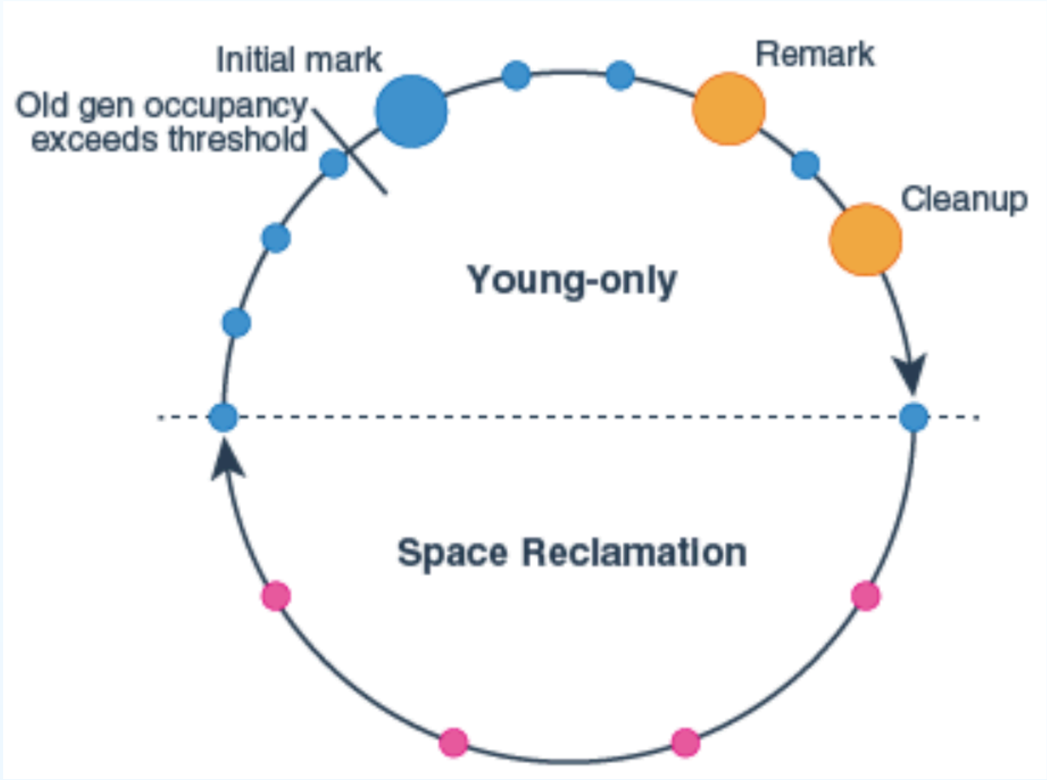
- JVM注入的一小段代码，用于记录指针变化
  - object.field = <reference> (putfield)
- 当更新指针时
  - 标记Card为Dirty
  - 将Card存入Dirty Card Queue
  - 白/绿/黄/红四个颜色

更新Remembered Set

- By concurrent refinement threads
- White
  - 天下太平，无事发生
- **Green** zone (–XX:G1ConcRefinementGreenZone=N)
  - Refinement线程开始被激活，开始更新RS
- **Yellow** zone (–XX:G1ConcRefinementYellowZone=N)
  - 全部Refinement线程开始激活
- **Red** zone (–XX:G1ConcRefinementRedZone=N)
  - 应用线程也参与排空队列的工作

G1的工作流程

- Young-only phase
  - Initial Mark
  - Remark
  - Cleanup
- Space-reclaimation phase



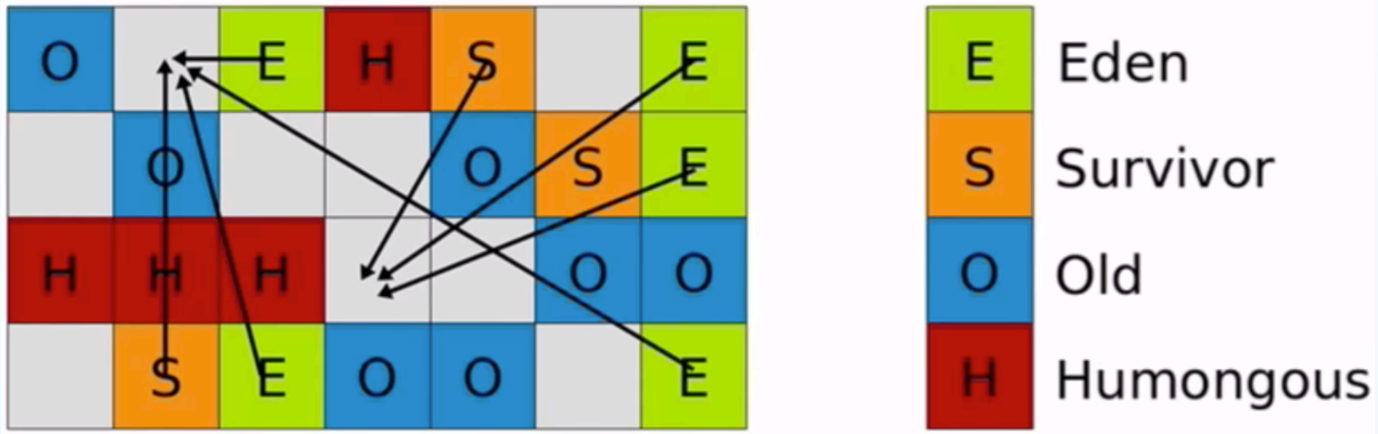
Fully young GC

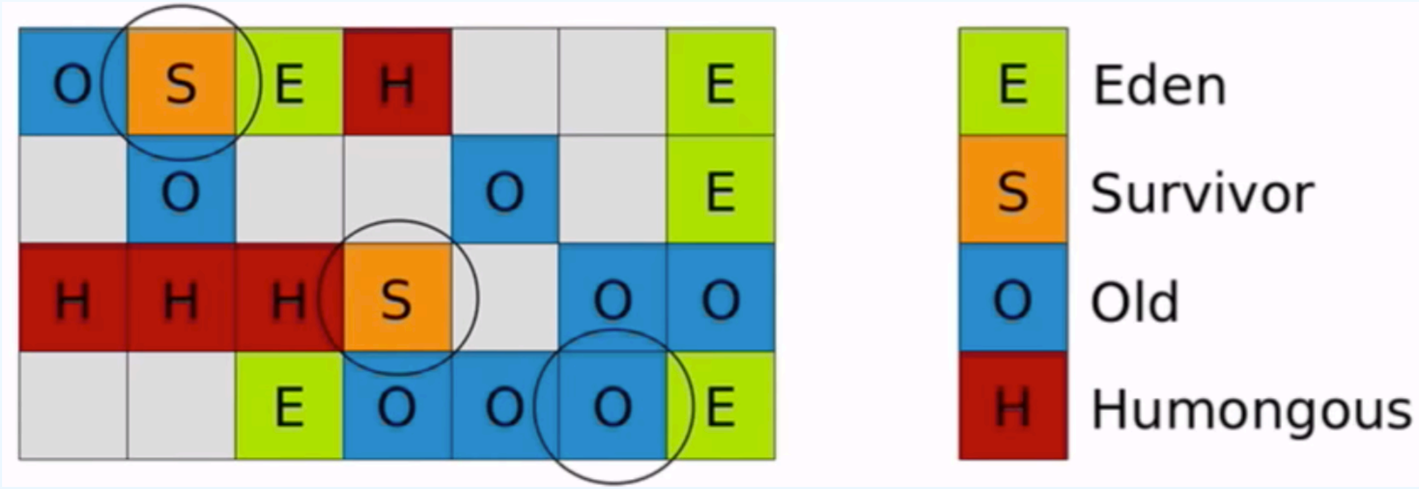
- STW (Evacuation Pause)
  - 构建CS (Eden+Survivor)
  - 扫描GC Roots
  - Update RS：排空Dirty Card Queue
  - Process RS：找到被哪些那些老年代对象所引用
  - Object Copy
  - Reference Processing

Fully young GC

- G1记录每个阶段的时间，用于自动调优
- 记录Eden/Survivor的数量和GC时间
  - 根据暂停目标自动调整Region的数量
  - 暂停目标越短，Eden数量越少
    - 吞吐量下降
  - –XX:+PrintAdaptiveSizePolicy
  - –XX:+PrintTenuringDistribution

Fully young GC





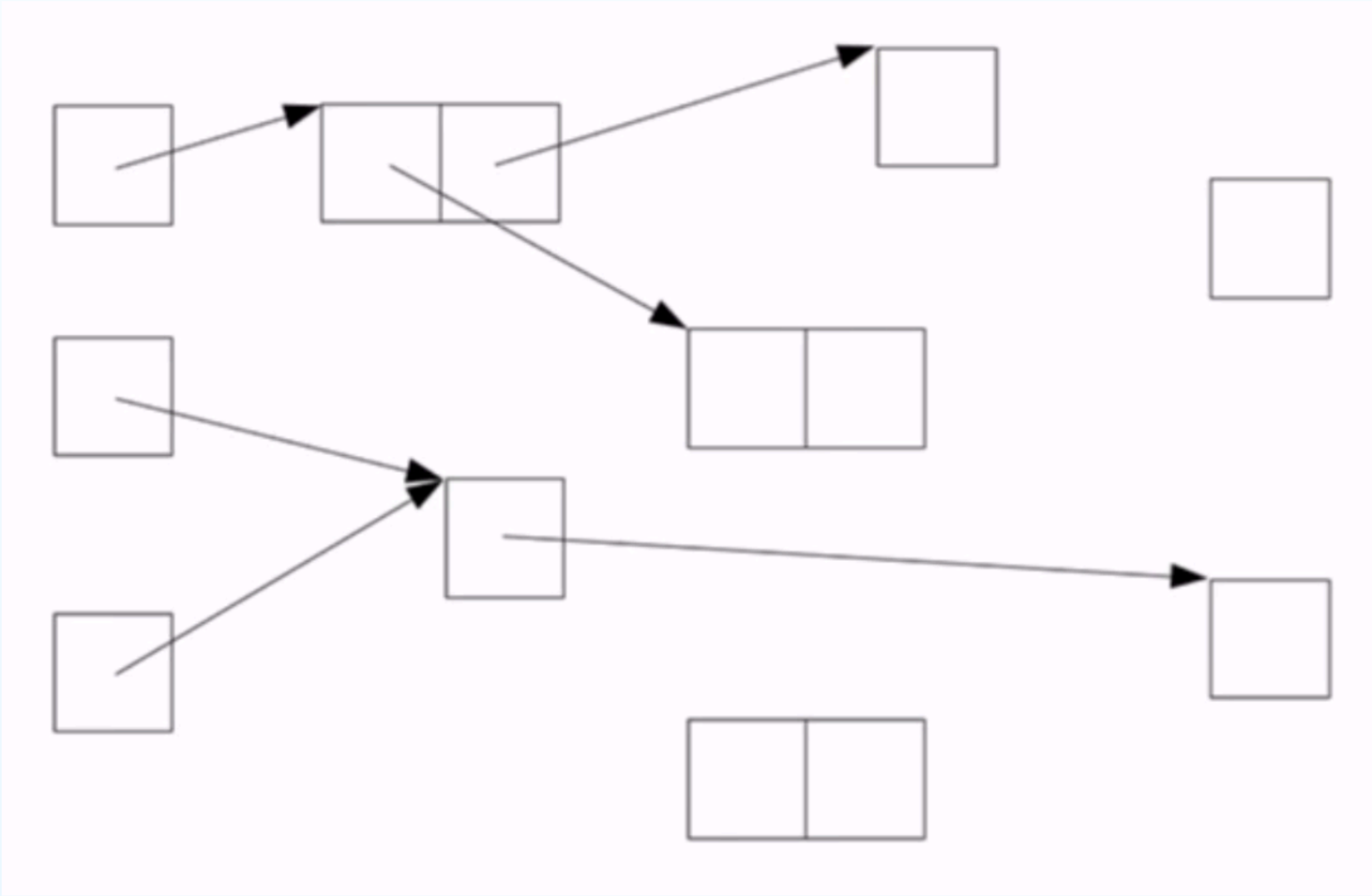
Old GC

- 当堆用量达到一定程度时触发
  - –XX:InitiatingHeapOccupancyPercent=N
  - 45 by default
- Old GC是并发(concurrent)进行的

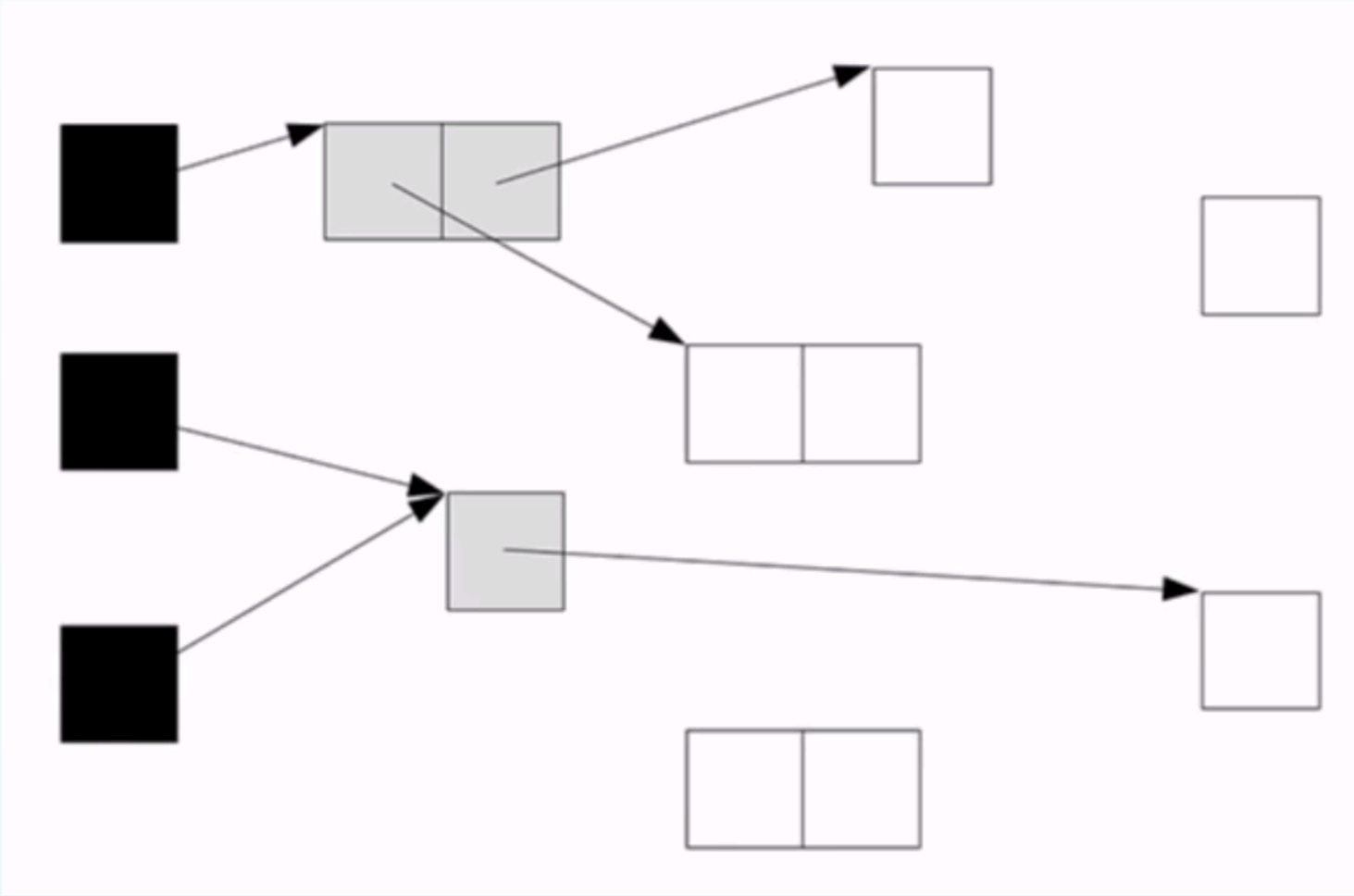
Mixed GC (Old GC)

- 当堆用量达到一定程度时触发
  - –XX:InitiatingHeapOccupancyPercent=N
  - 45 by default
- Old GC是并发(concurrent)进行的
  - 三色标记算法：不暂停应用线程的情况下进行标记

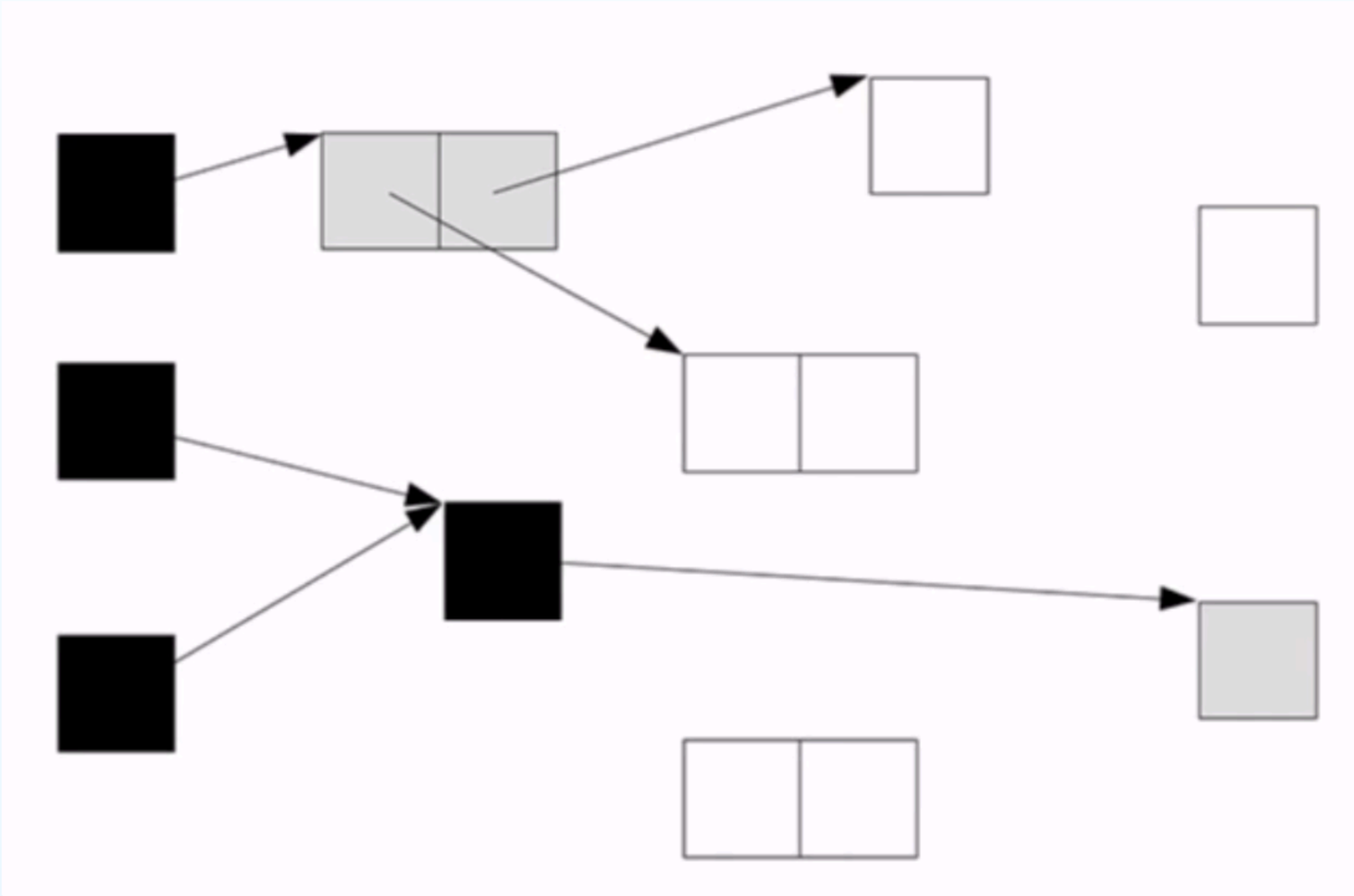
三色标记算法



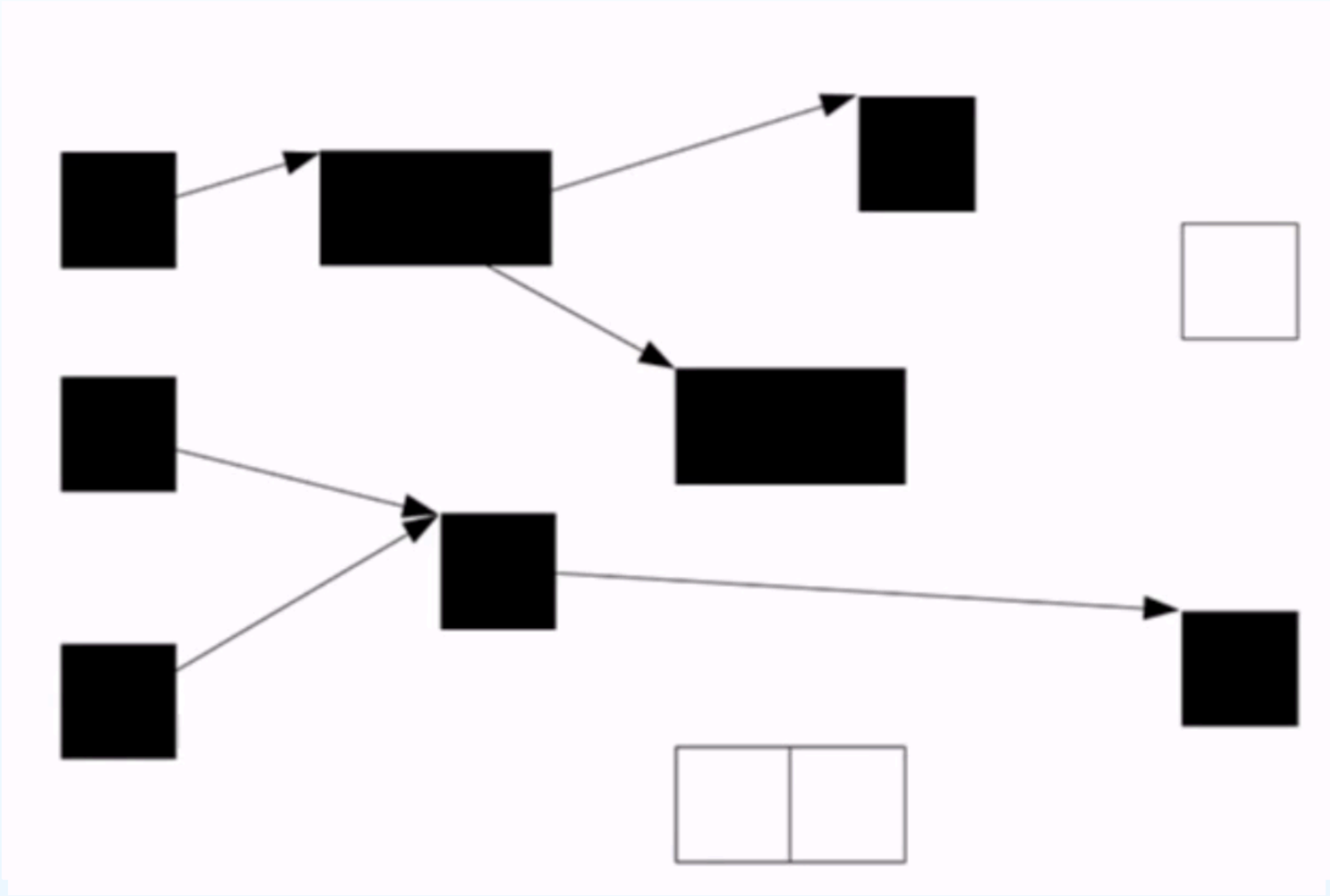
三色标记算法



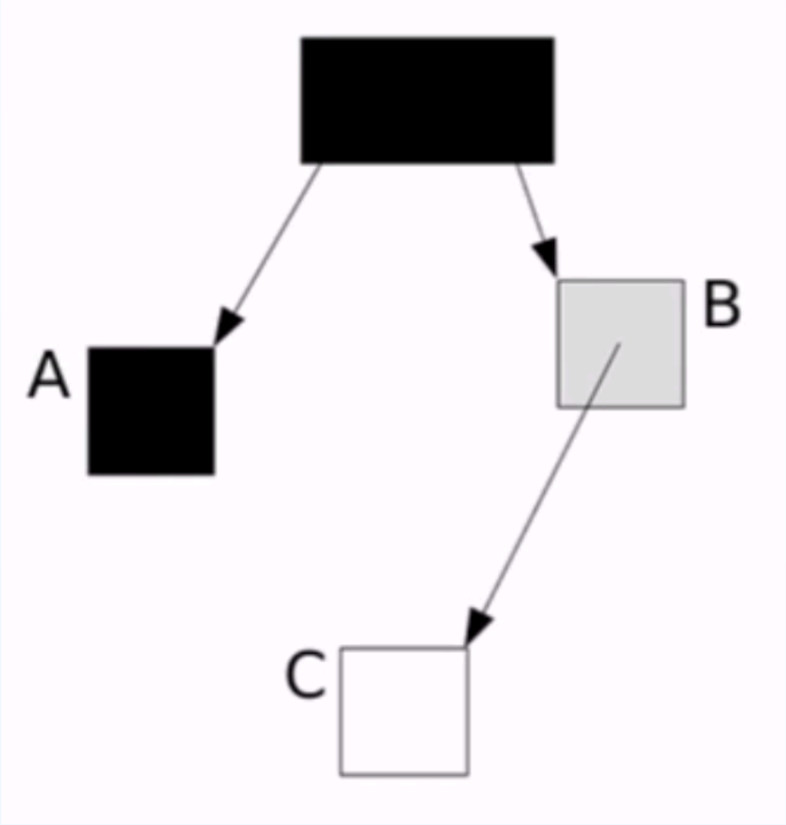
三色标记算法



三色标记算法

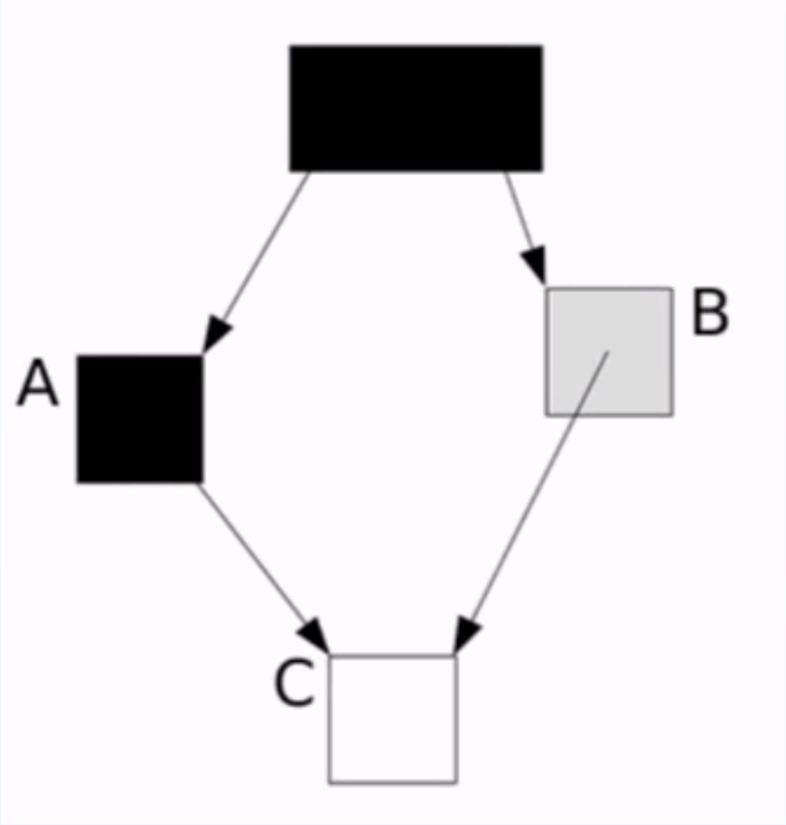


Lost Object Problem



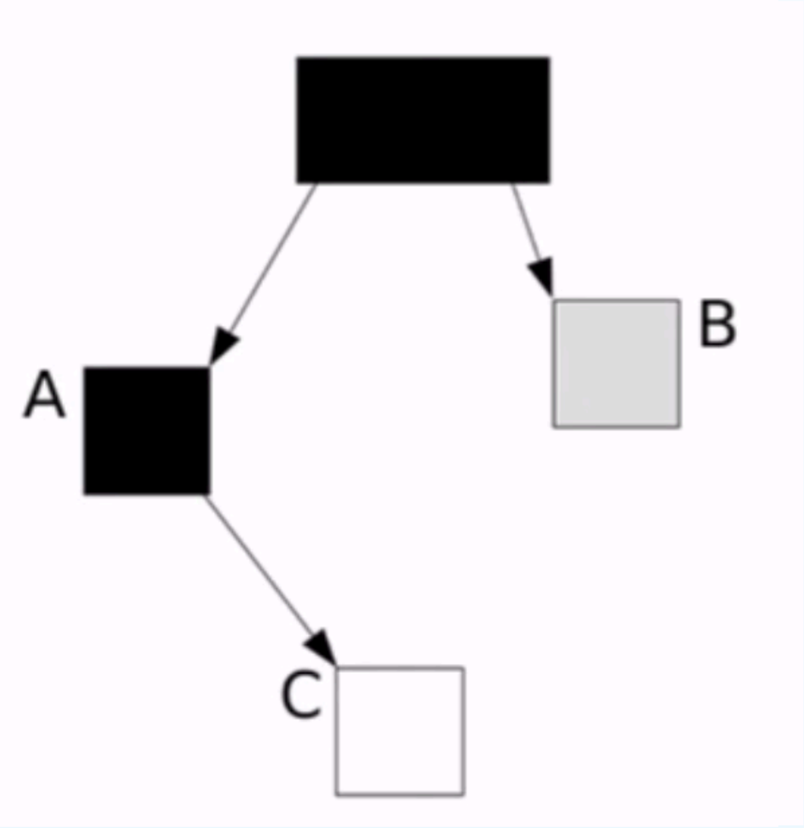
Lost Object Problem

Application thread
A.c = C



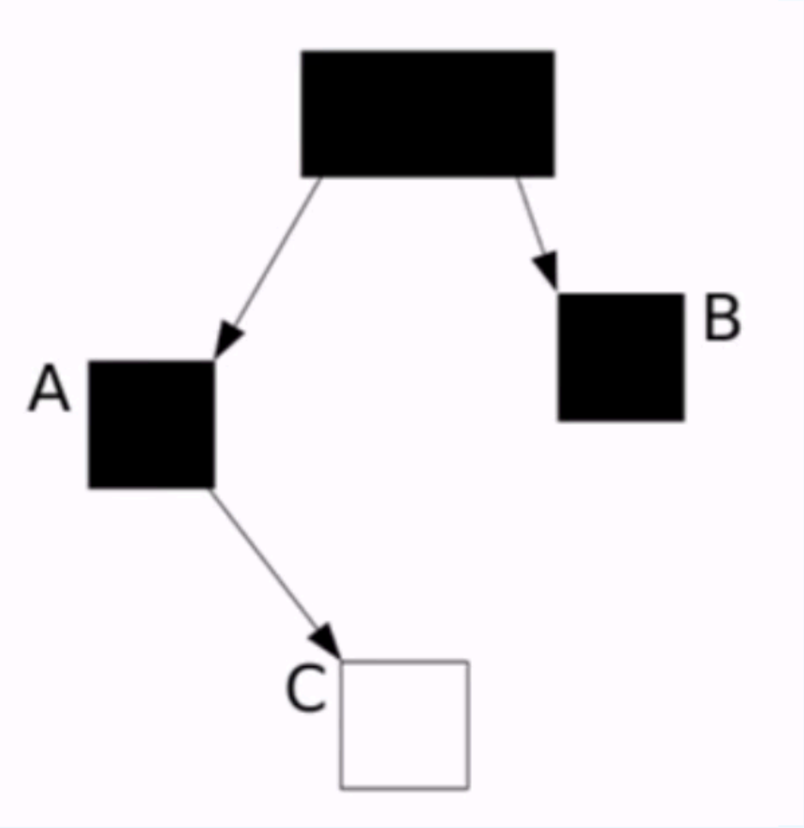
Lost Object Problem

Application thread
B.c = null



Lost Object Problem

G1: finished!



Write barrier

- B.c = null
  - 即当C指针被删除时
  - G1认为C仍然是活对象
- Snapshot-At-The-Beginning (SATB)
  - 保持在marking阶段开始的object graph
  - C仍被remark阶段处理
  - 可能产生浮动垃圾

G1 Old GC流程

- STW (Fully young GC)
  - piggy-backed on young GC
  - 利用Young GC的信息
- 恢复应用线程
- 并发初始标记 (Init marking)
- STW
  - Remark
    - SATB/Reference processing
  - Cleanup
    - 立刻回收全空的区
- 恢复应用线程

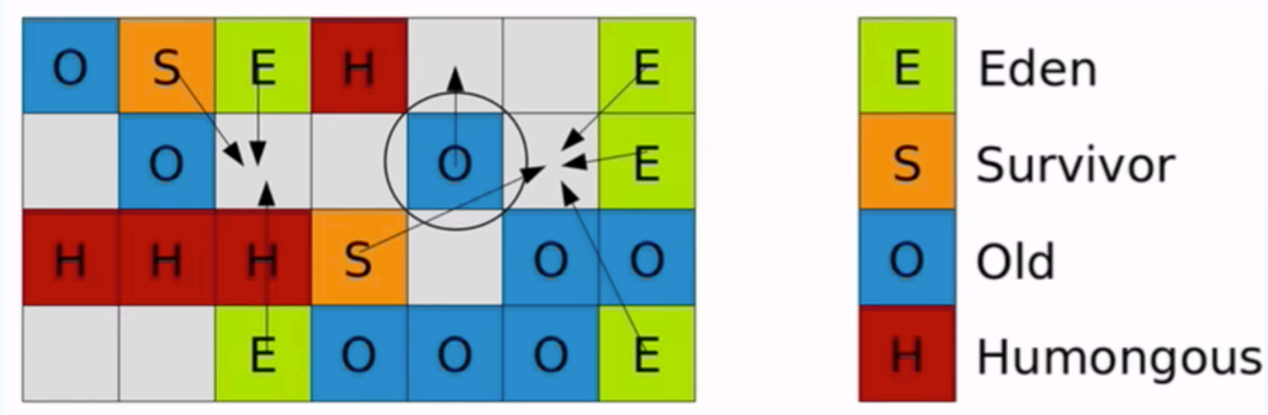
G1 Old GC流程

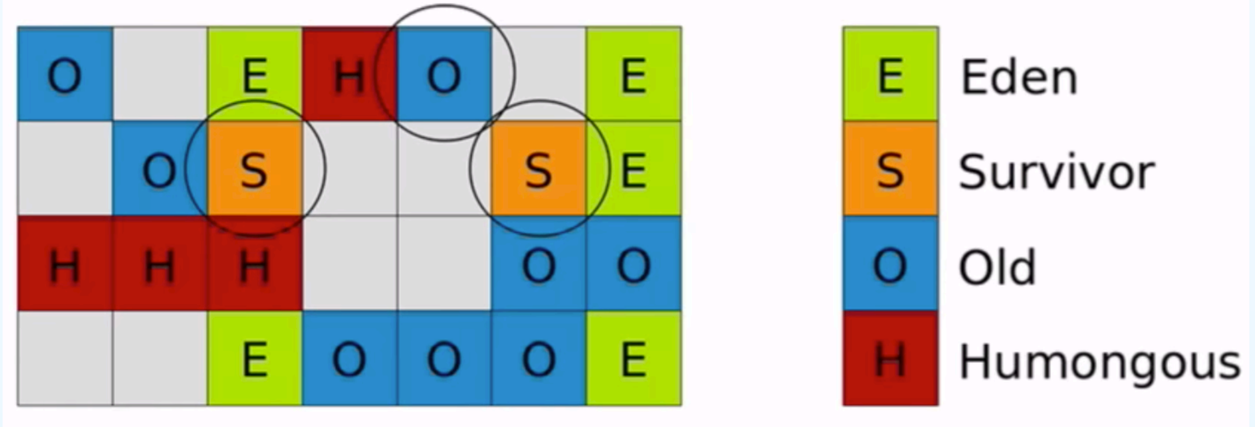
- [GC pause (G1 Evacuation Pause) (young) (initial-mark)
- [GC concurrent-root-region-scan-start]
- [GC concurrent-root-region-scan-end, 0.0566056 secs]
- [GC concurrent-mark-start]
- [GC concurrent-mark-end, 1.6776461 secs]
- [GC remark, 0.0488021 secs]
- [GC cleanup 16G->14G(32G), 0.0557430 secs]

Mixed GC

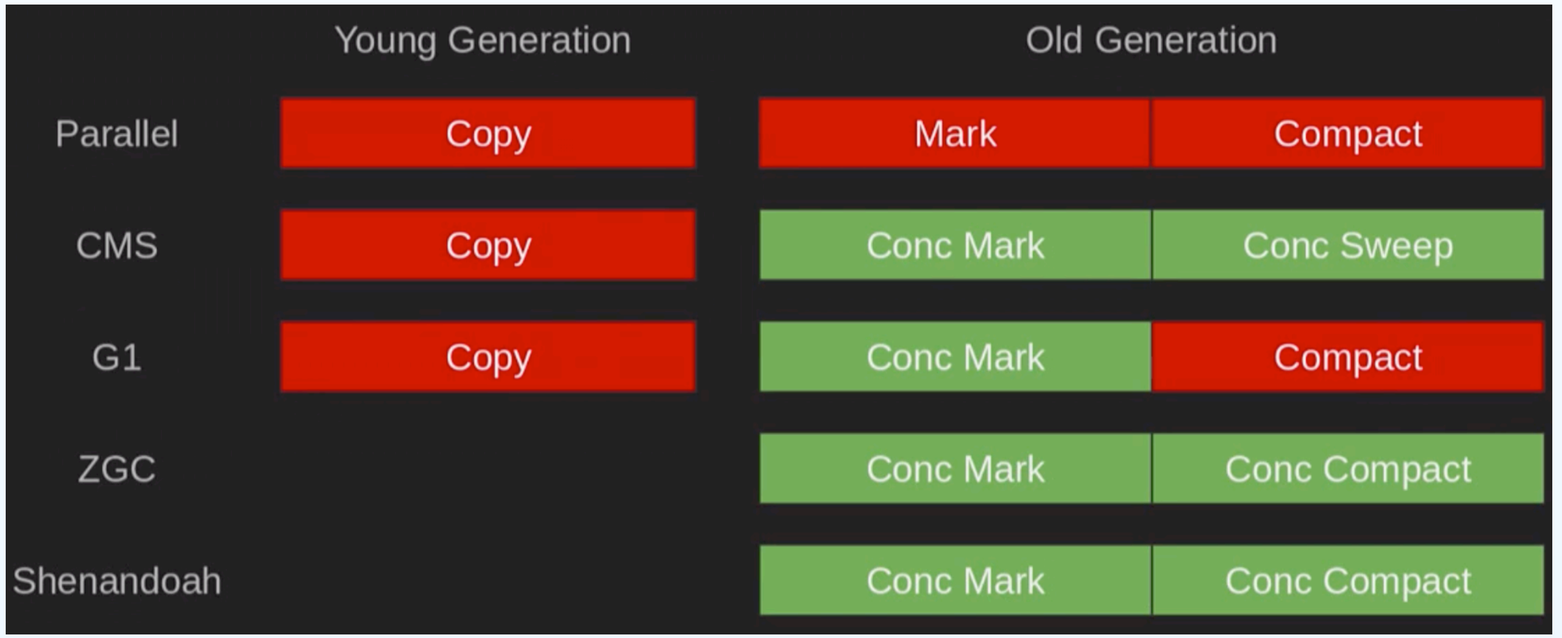
- 不一定立即发生
- 选择若干个Region进行
  - 默认1/8的Old Region
    - –XX:G1MixedGCCountTarget=N
  - Eden+Survivor Region
  - STW, Parallel, Copying
- 根据暂停目标，选择垃圾最多的Old Region优先进行
  - –XX:G1MixedGCLiveThresholdPercent=N (default 85)
  - –XX:G1HeapWastePercent=N

Mixed GC

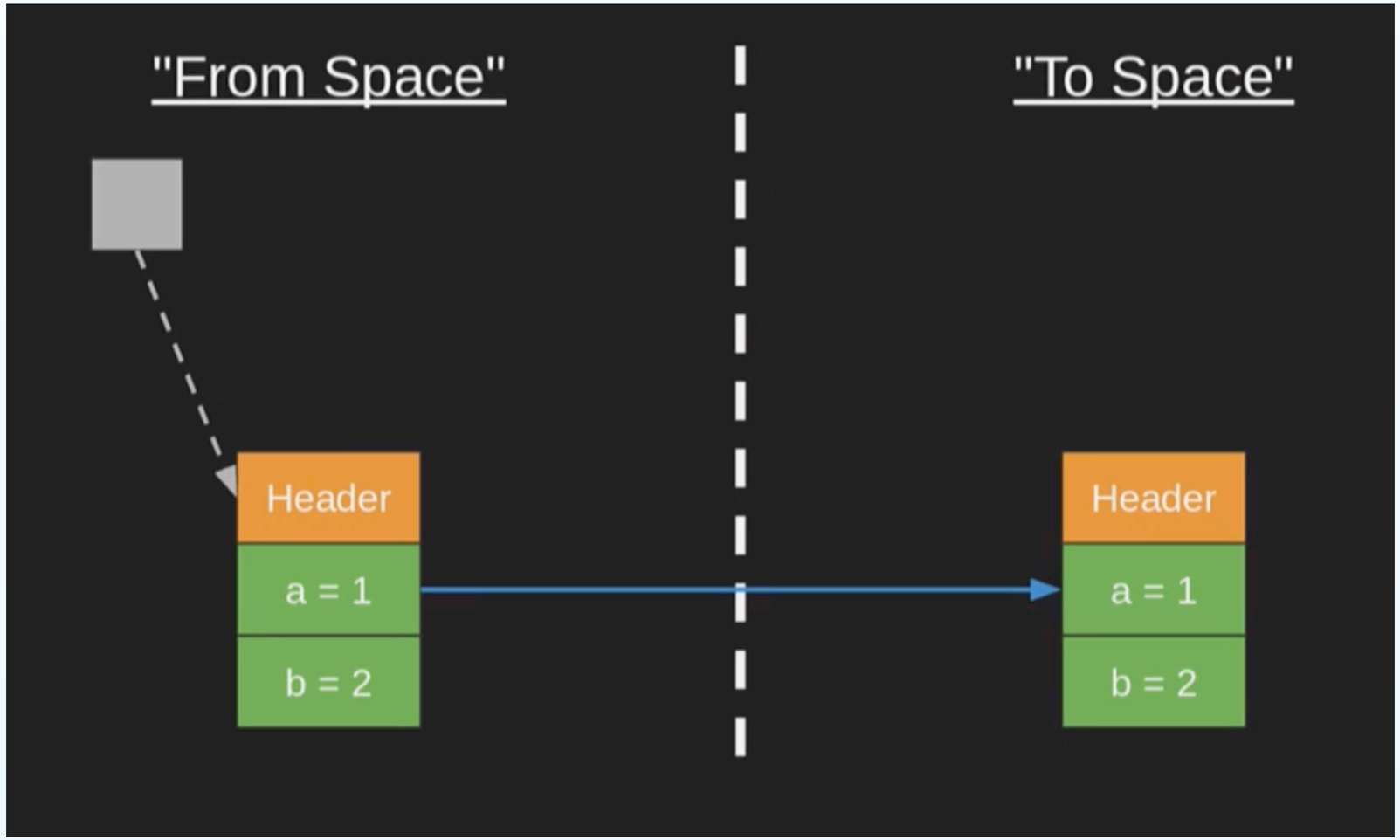




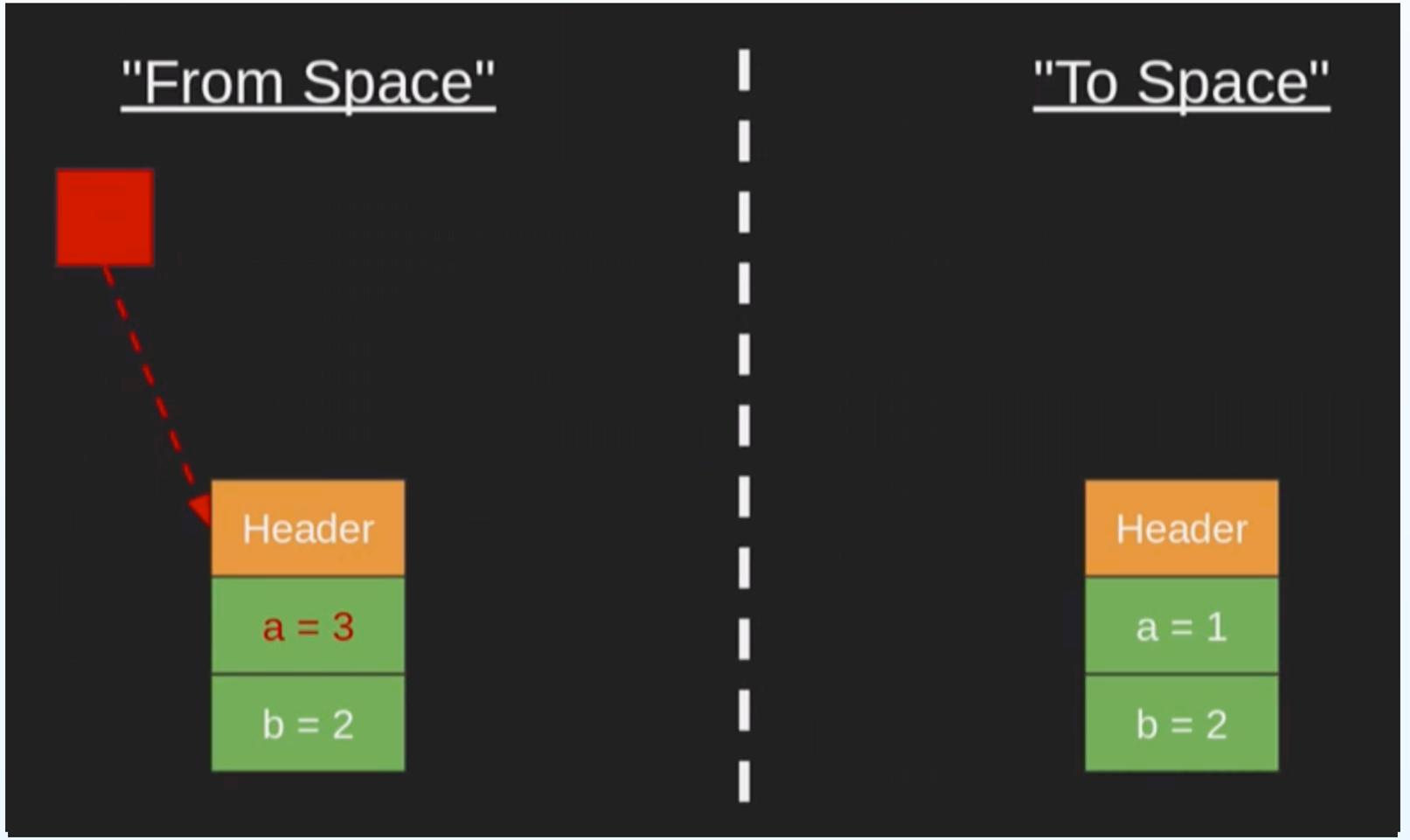
未来：ZGC/Shenandoah



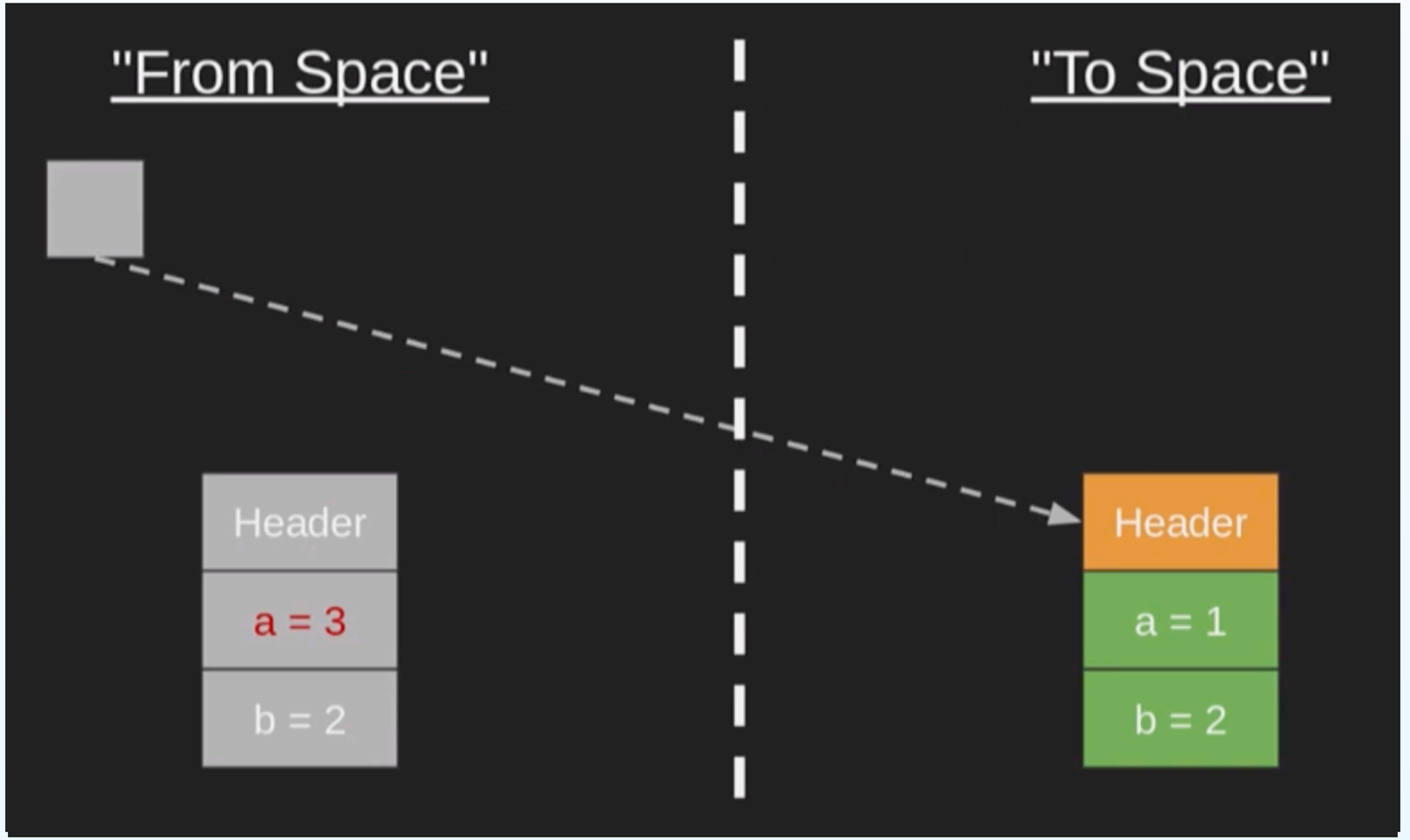
Problem: Concurrent Copying



Problem: Concurrent Copying



Problem: Concurrent Copying

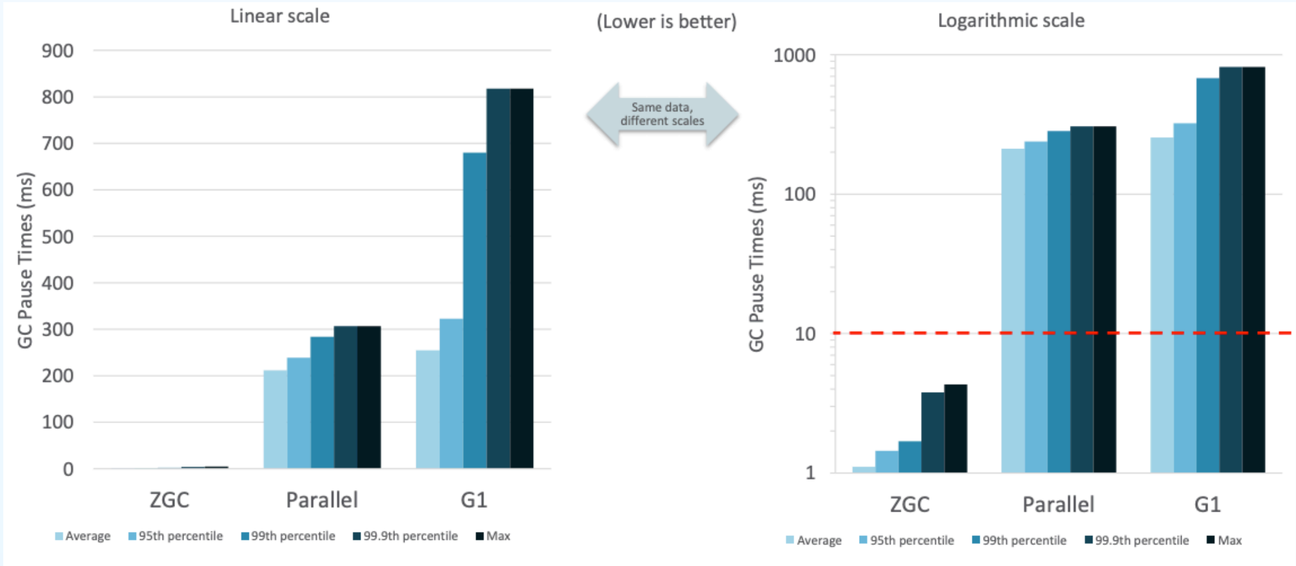


ZGC

- Since OpenJDK 11+
  - OpenJDK
    - OpenJDK (Oracle build)
    - AdoptOpenJDK build
- 令人恐怖的极低延时
- Concurrent compaction, 单一代
- 当前只在x86-64上可用
  - ARM的移植正在进行中
- 基于Region(ZPage)

ZGC

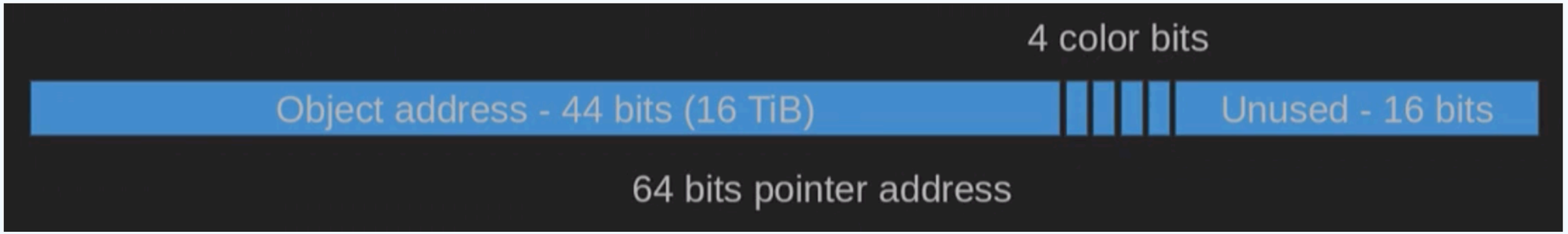
- Since OpenJDK 11+
- 令人恐怖的极低延时
- Concurrent compaction, 单一代
- 当前只在x86-64上可用
  - ARM的移植正在进行中
- 基于Region(ZPage)

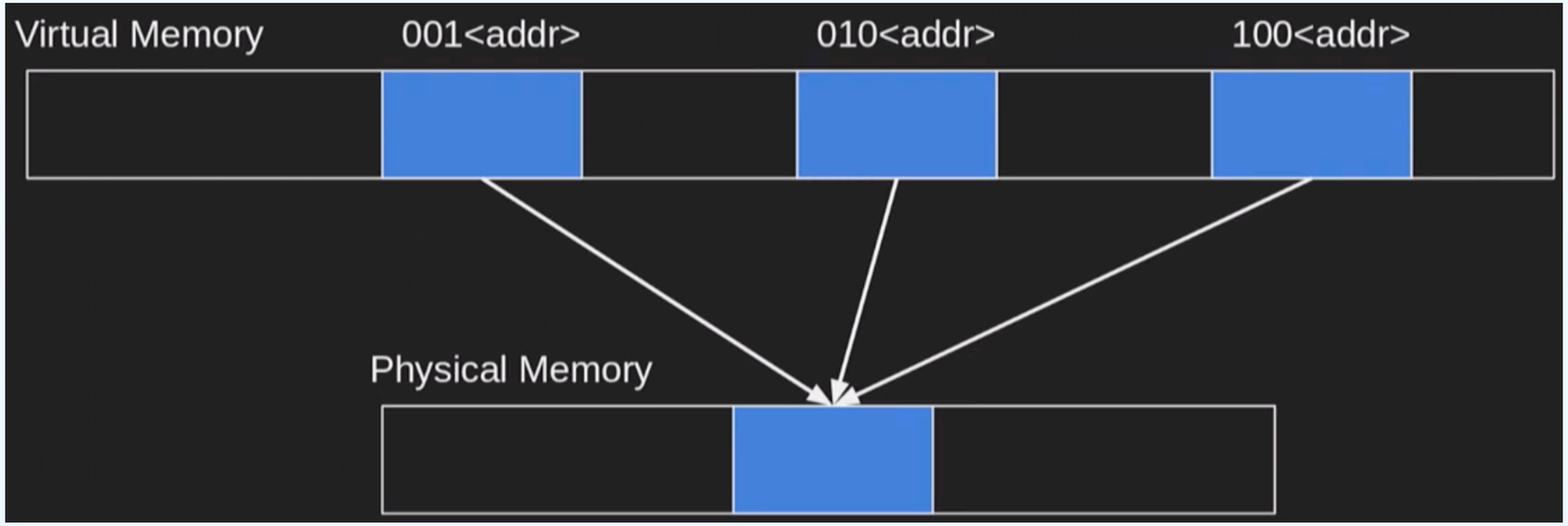


Size groups

- Small (2MB)
- Medium (32MB)
- Larage (N×2MB)

ZGC: 染色指针与多重映射





ZGC: problem

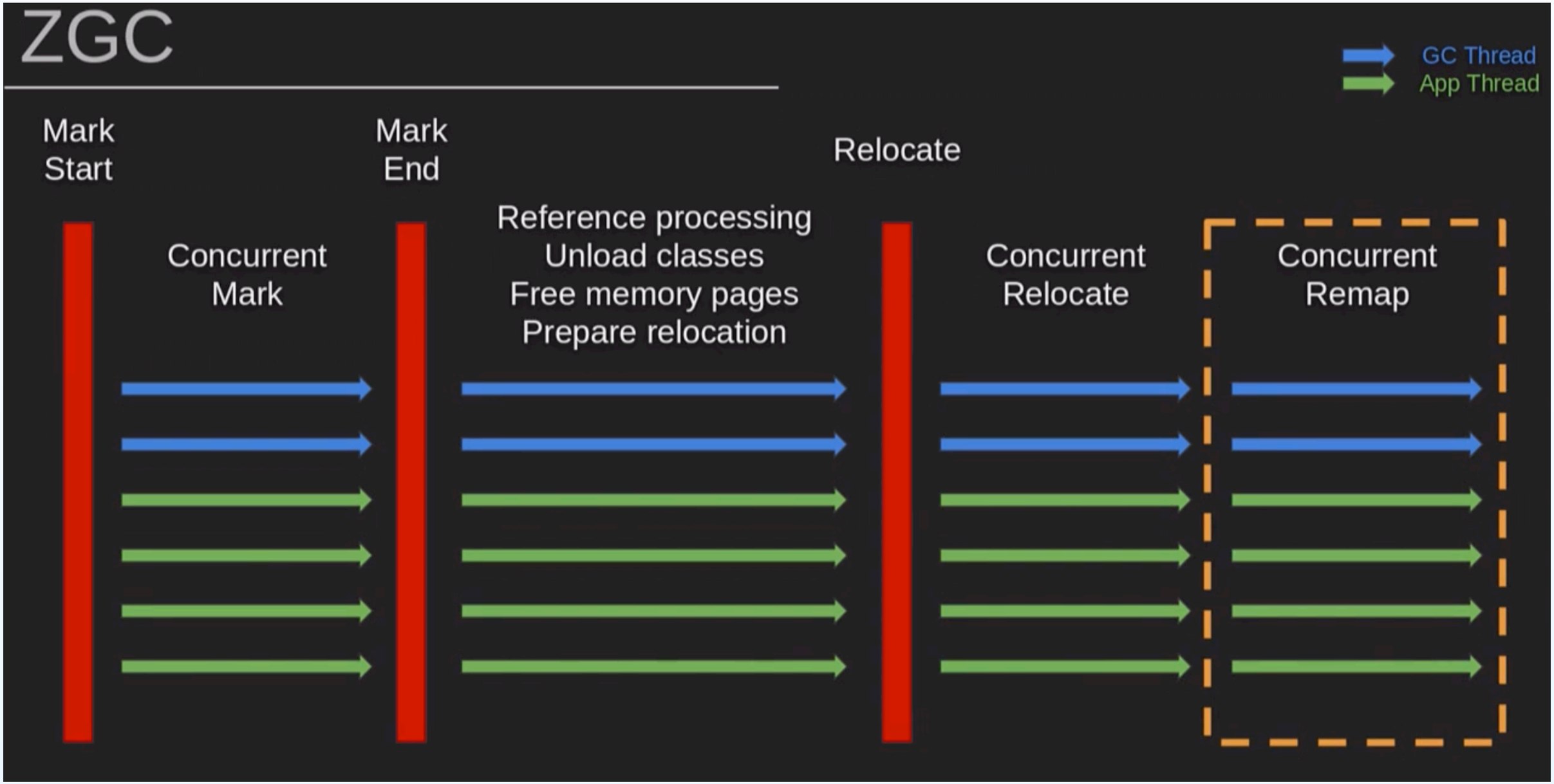
- 不能使用Oops
- RSS show 3x size
  - –Xmx16g –> RSS~=50G
  - OOM killer

ZGC: load barrier

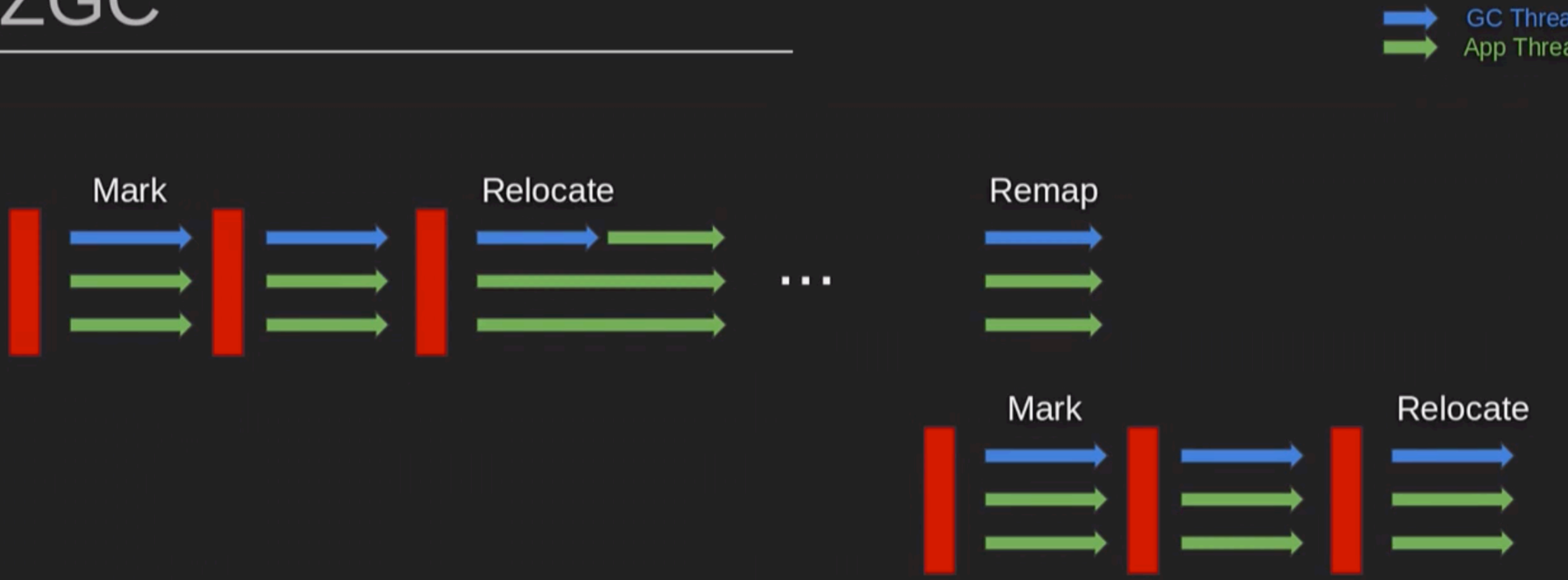
```
Object f = obj.field;
<load_barrier>

mov 0x10(%rdi), %rsi
test %rsi, 0x20(%r15)
jne slow_path
```

ZGC: load barrier

- 检查是否使用了正确的颜色
  - 即指针是否处于正确的状态
- 如果颜色错误 –> slow path
  - Fix color
  - 执行必要的行为
  - 具体行为取决于GC的阶段


ZGC
GC Thread
App Thread
Mark
Start
Mark
End
Relocate
Concurrent
Mark
Reference processing
Unload classes
Free memory pages
Prepare relocation
Concurrent
Relocate
Concurrent
Remap

ZGC

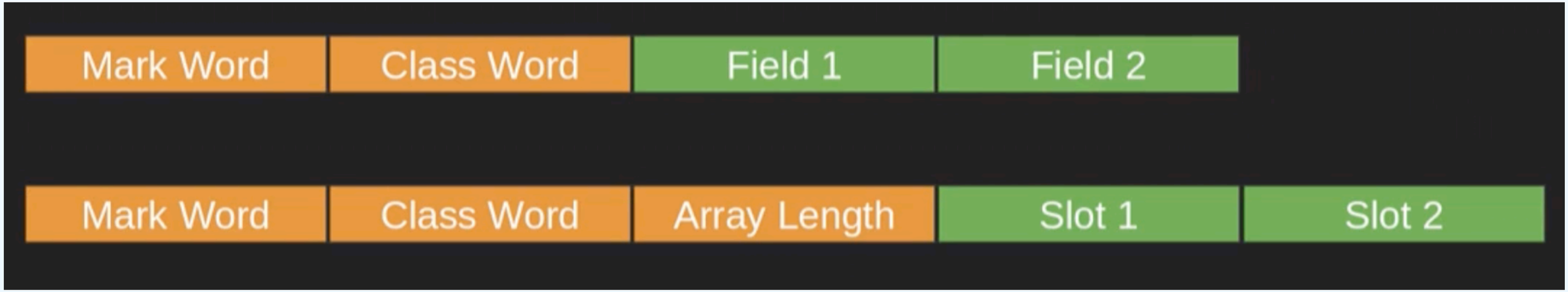


Shenandoah GC

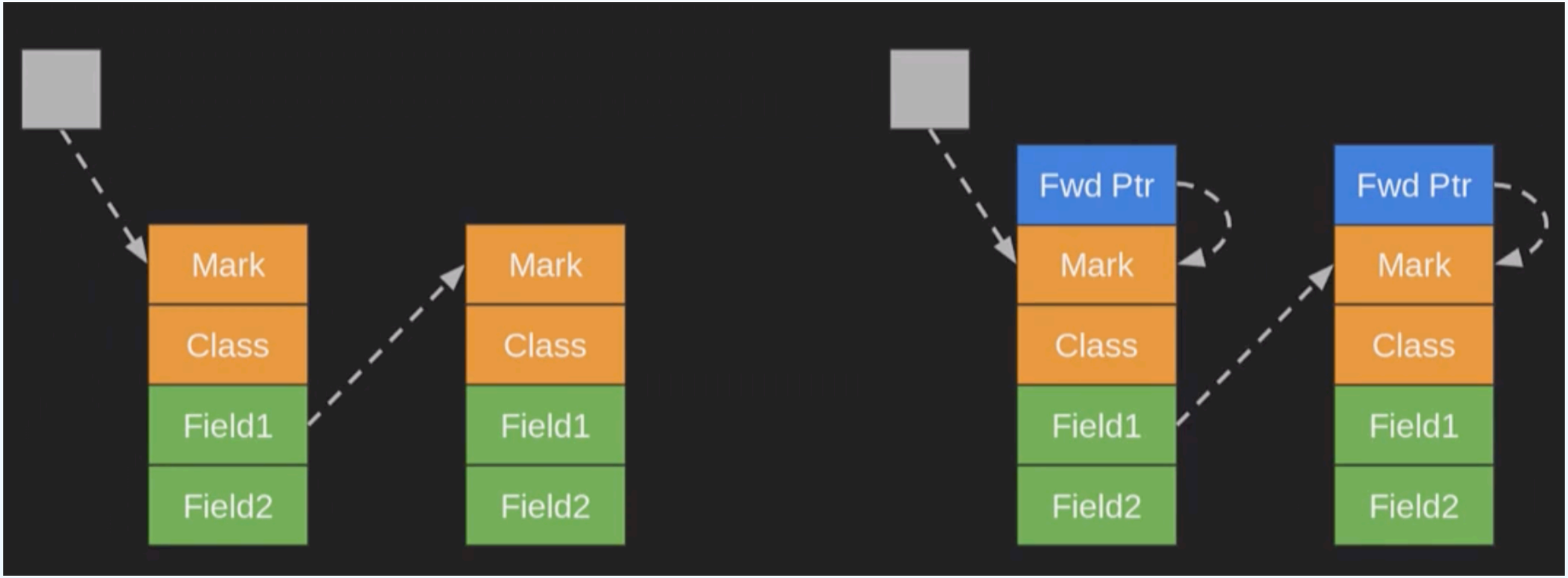
- RedHat开发的
- Only AdoptOpenJDK builds
- Not in Oracle OpenJDK builds
- 低延迟
- Concurrent Compaction
- Single Generation, Region Based
- 所有平台均可用

HotSpot Object Header

- Word = 32/64b



Shenandoah 1.0



Shenandoah 1.0

- Load GC barrier
  - 进行一次跳转

```
mov  -0x08(%r10), %r10
```

Shenandoah 1.0

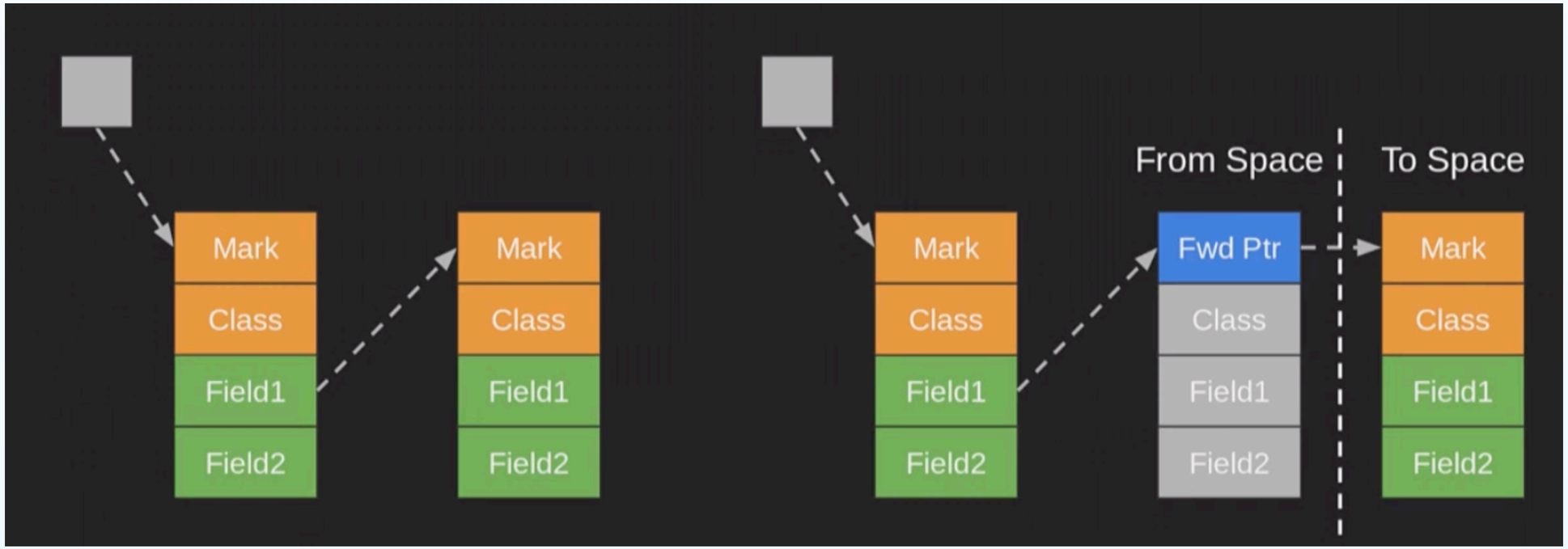
- Store GC barrier
  - 根据GC的阶段执行不同的操作

```
mov 0x3d8(%r15), %r11   # load GC phase
test %r11, %r11         # if (phase) ...
jne <slow_path>
```

Shenandoah 1.0

- Problem
  - 太多的额外的内存，最多50%

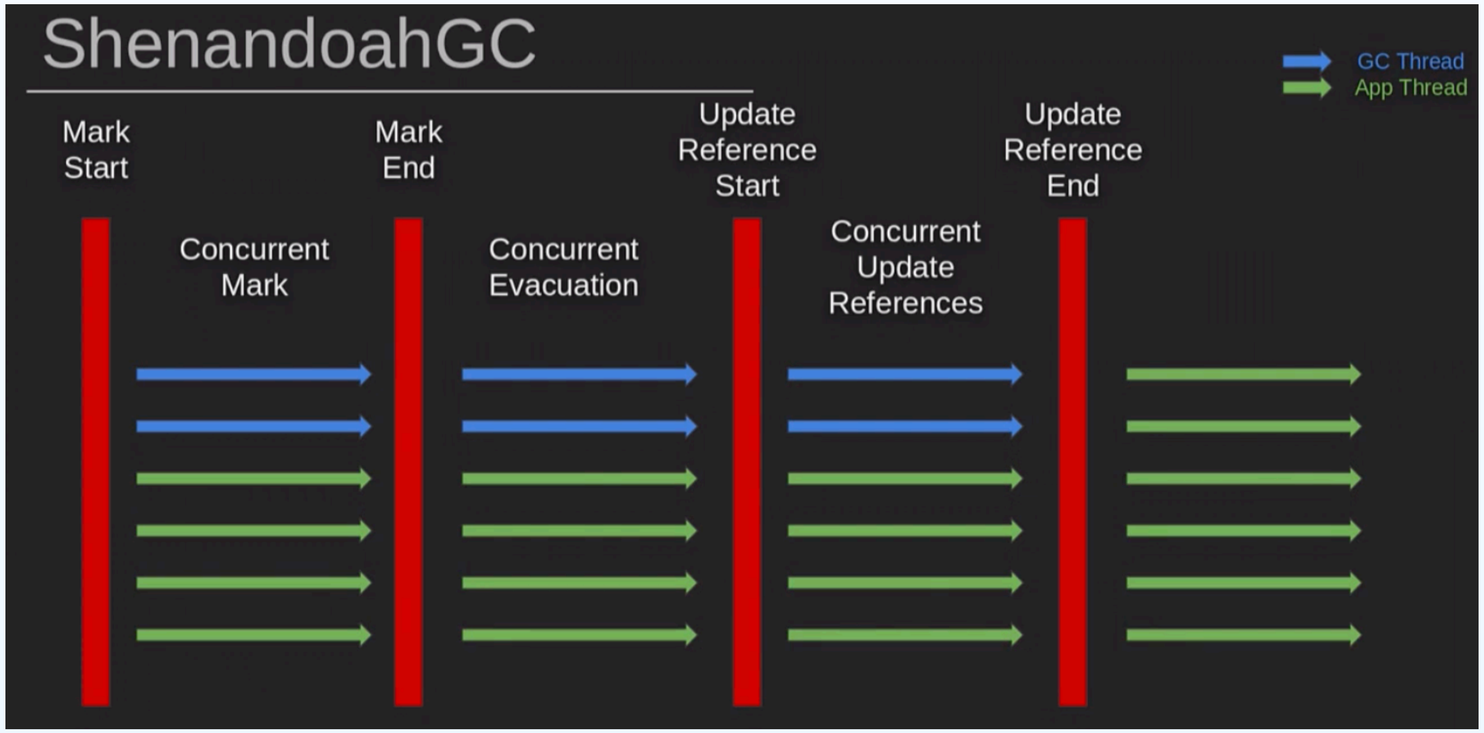
Shenandoah 2.0



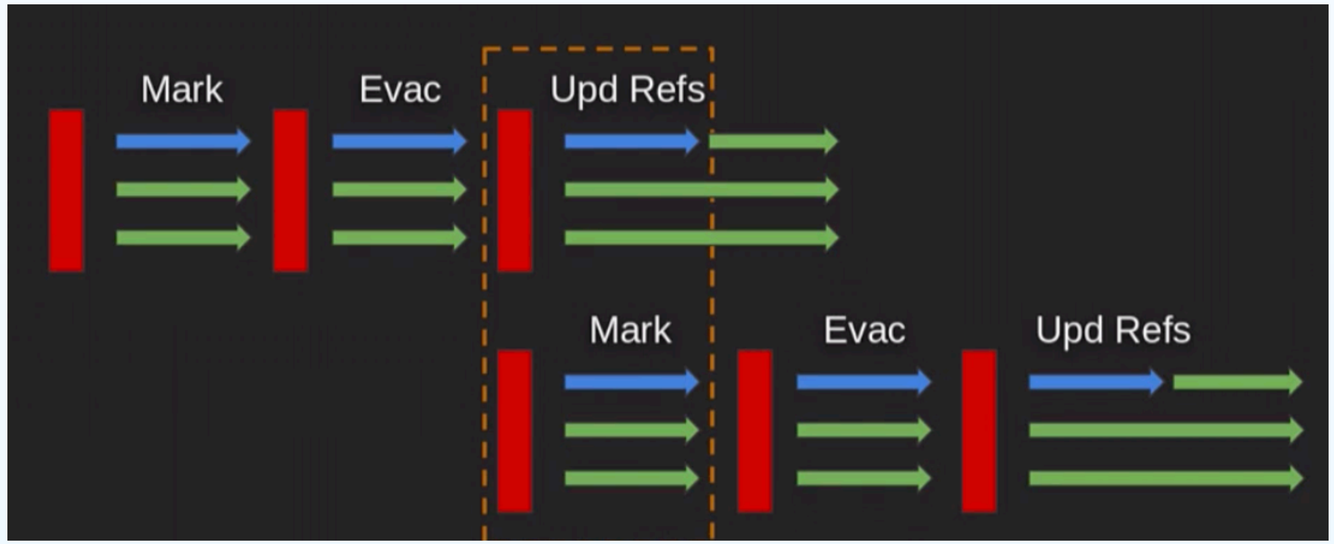
Shenandoah 2.0

- GC时候才将修改Mark Word
- 只在必要时候检查对象头即可

Shenandoah 2.0



Shenandoah 2.0



参考文献

Q & A