垃圾回收基础知识

杨勇勇 yangyongyong@huawei.com

课程目标

- 建立起对垃圾回收领域的自顶向下的全局认识。
  - 了解垃圾回收的各个设计方面、各种算法，及其对性能的影响。了解目前垃圾回收研究的前沿和发展方向。
  - 了解编译器需要对垃圾回收系统提供的基本支撑能力。

课程内容

- 垃圾回收的原理和算法
  - 垃圾回收的概念和意义
  - 垃圾回收算法：对象分配、垃圾识别、对象回收
  - 分代垃圾回收
  - 并行、并发垃圾回收
- 编译器、虚拟机为了高效支持上述垃圾回收原理所需要提供的编译器机制
  - GC barriers
  - 内存布局、
    - ✓ Object Map
  - Stack map
  - safepoint

目录

CONTENTS

什么是垃圾回收（Garbage collection）？

基本内存操作

- 分配内存
- 写内存
- 读内存
- 释放内存
  - free (C/C++)
  - delete (C++)

```
class C0 {
    ...
}
class C1 {
    v : Int64 // 值成员
    r : C0  // 引用成员，在自动内存管理中具有特殊意义
}
var obj1 : C1 = C1() // 分配内存，写引用变量
obj1.v = 99 // 写
obj1.r = C0() //分配内存，写引用成员
let v1 = obj1.v // 读
let r1 = obj1.r // 读引用成员，写引用变量
```

- 内存如何释放是managed language(Cangjie/Java/Golang/Swift等) 和 unmanaged language(C/C++/Rust等)的主要差别之一。
- 垃圾回收解决的问题是“内存如何释放”。
- 为了做到高效正确地自动释放内存，也必须适当地处理分配分配、写、读等其它基本操作。

为什么使用垃圾回收?

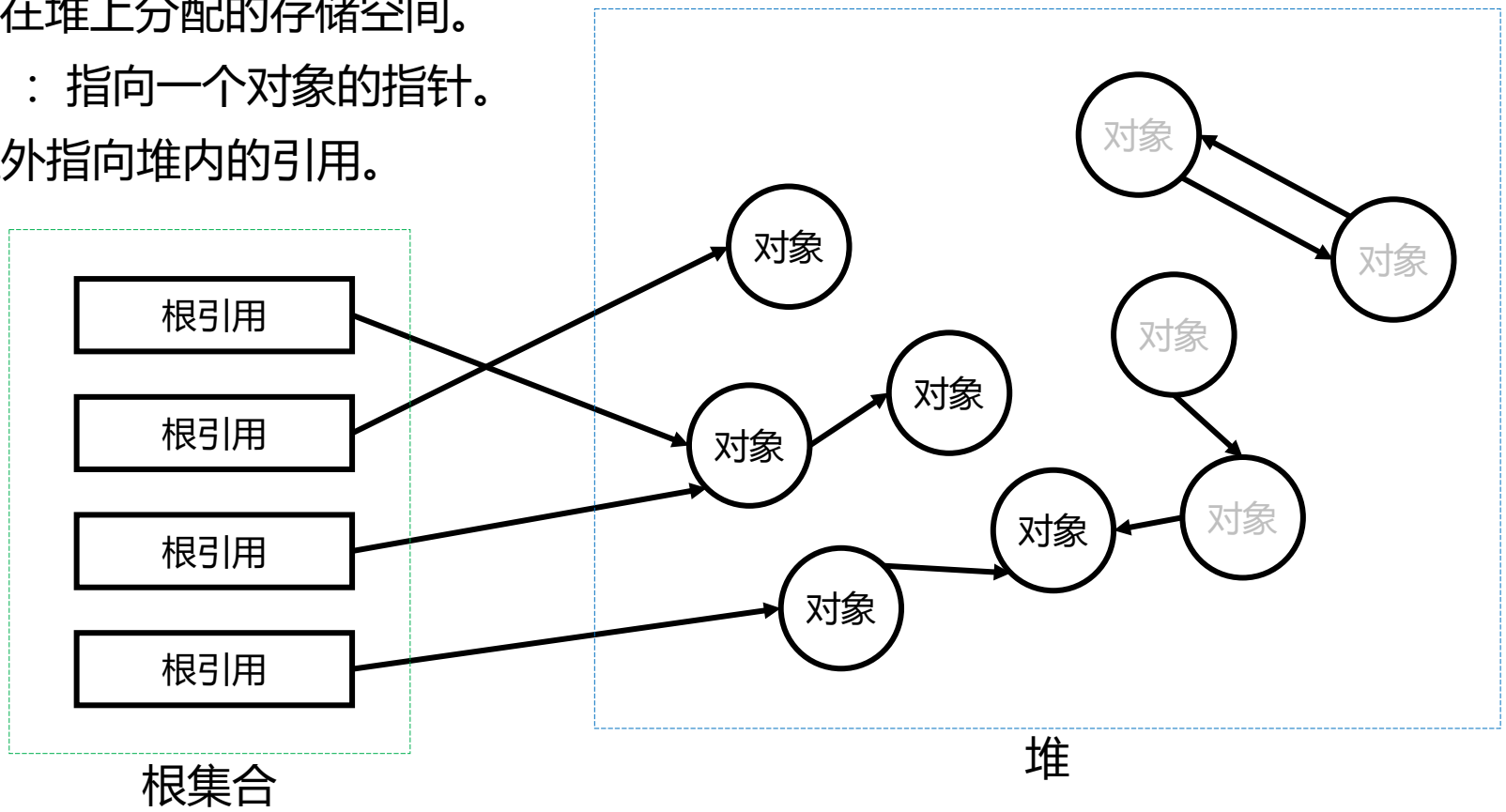
为什么使用垃圾回收?

- 避免常见的内存管理错误
  - 无用单元：不能被访问，但还没有释放。将永远无法被释放（内存泄露）。
  - 悬垂引用：还能被访问，却已经被回收。程序崩溃、未定义行为。
- 责任分离
  - 程序员集中注意力于业务逻辑，而不是内存管理。
- 提高性能
  - 基于bump-pointer的分配回收算法比C语言的基于free-list的malloc和free更快。
  - 有编译器的配合，生成快速路径
- 对并发、多核的硬件资源有利
  - 即使应用程序是单线程的，也可以有多个GC线程帮助它管理内存。
  - 利用异构CPU，将GC线程放在小核上，可以减少能耗。
  - 一些并发无锁数据结构在有垃圾回收的情况下实现更容易。
    - Michael-Scott lock-free queue
    - Read-copy-update (RCU)

垃圾回收基本原理

基本概念

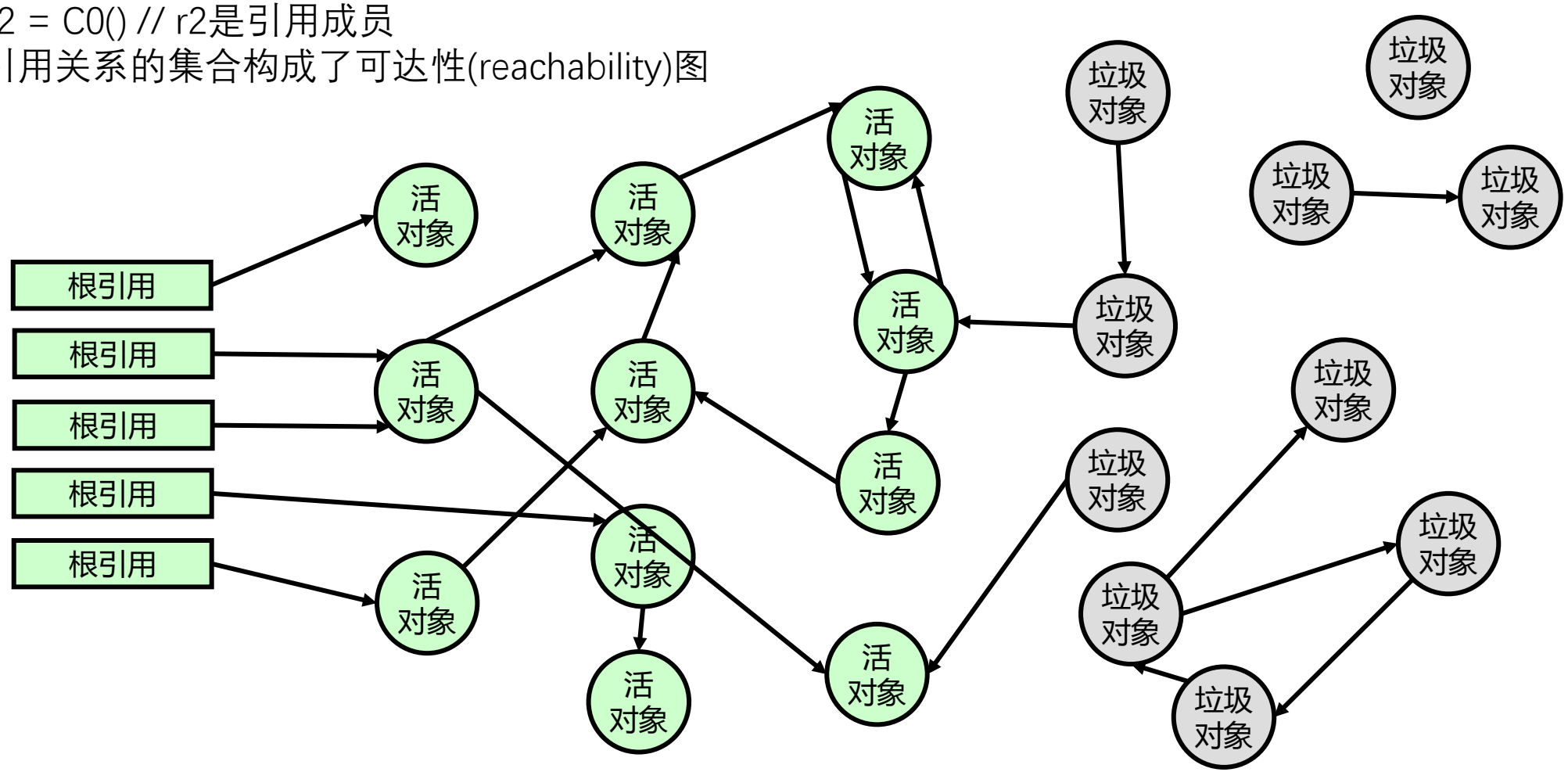
- 堆（heap）：堆里可以分配对象，堆由GC管理。
- 对象（object）：在堆上分配的存储空间。
- 引用（reference）：指向一个对象的指针。
- 根（root）：从堆外指向堆内的引用。



引用关系和可达性

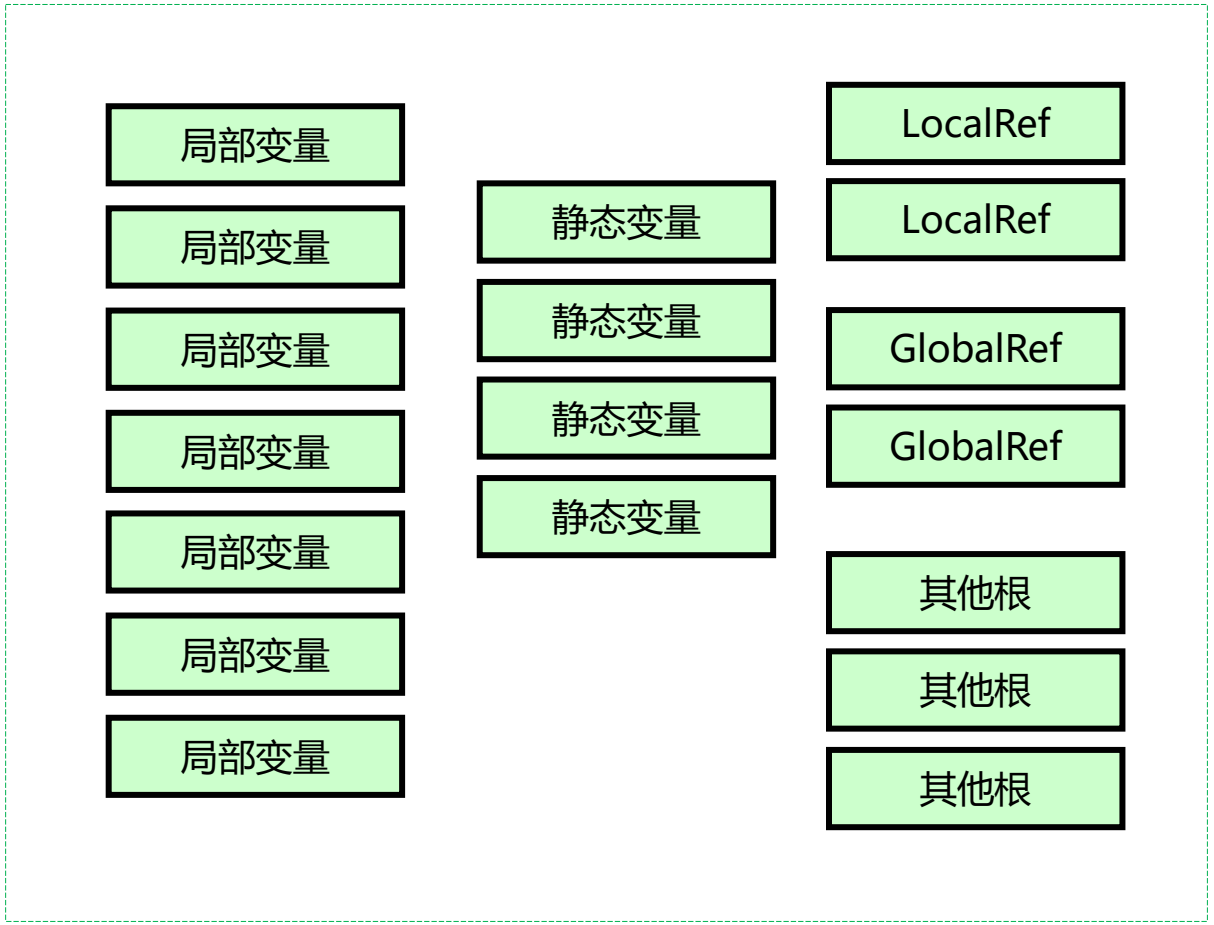
- 对引用变量和引用成员的写操作定义了引用关系：
  var r1 = C0() // r1是全局或者局部引用变量
  obj1.r2 = C0() // r2是引用成员
- 所有引用关系的集合构成了可达性(reachability)图

- 如果一个对象**不可从“根”到达**，就是垃圾。
  - 注：根是应用程序可以直接访问的变量（如局部变量）
  - 可以直接访问 -> 近似认为将来会被用到



根集合（root set）

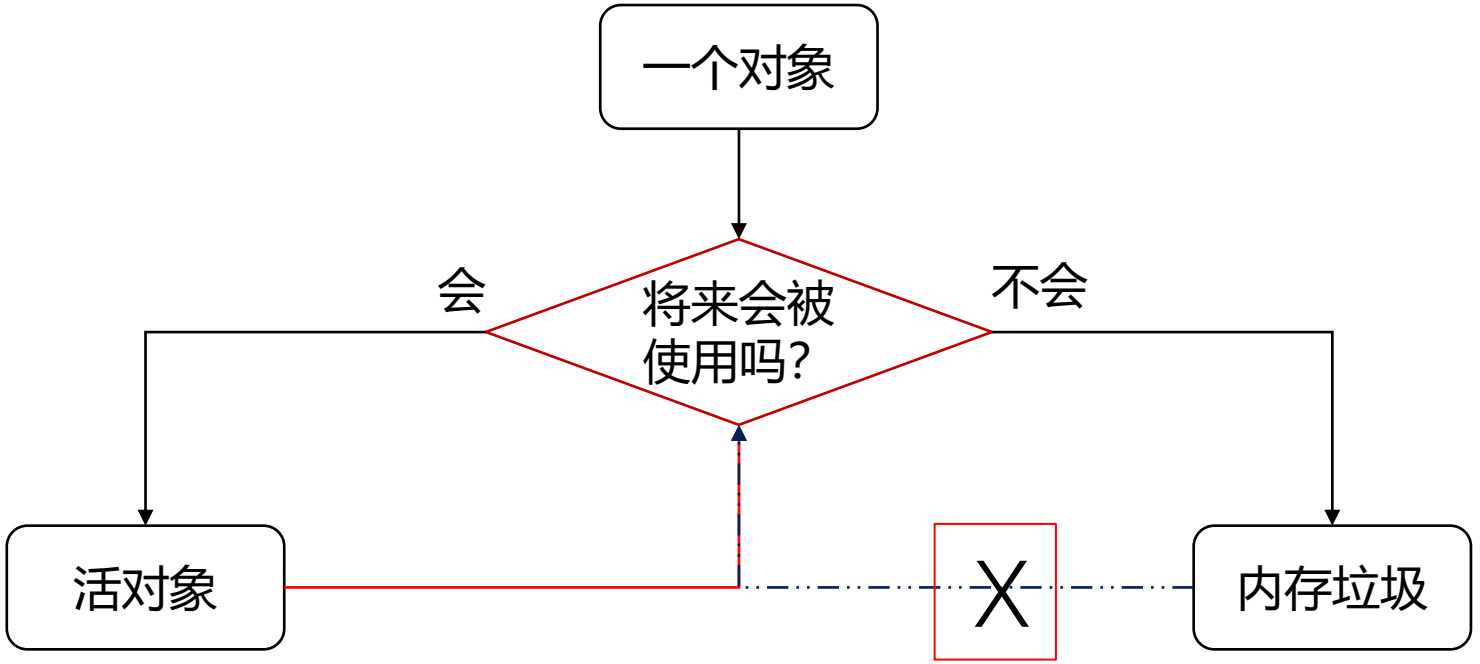
- 根集合中的引用
  - 可以被应用程序直接访问
  - 因此根指向的对象都是活的
- 具体包括
  - **局部变量**
  - **静态（全局）变量**
  - 被外部接口保留的
    - 例如JNI的LocalRef等
  - 其他根
    - 由语言、虚拟机、运行环境定义



根集合

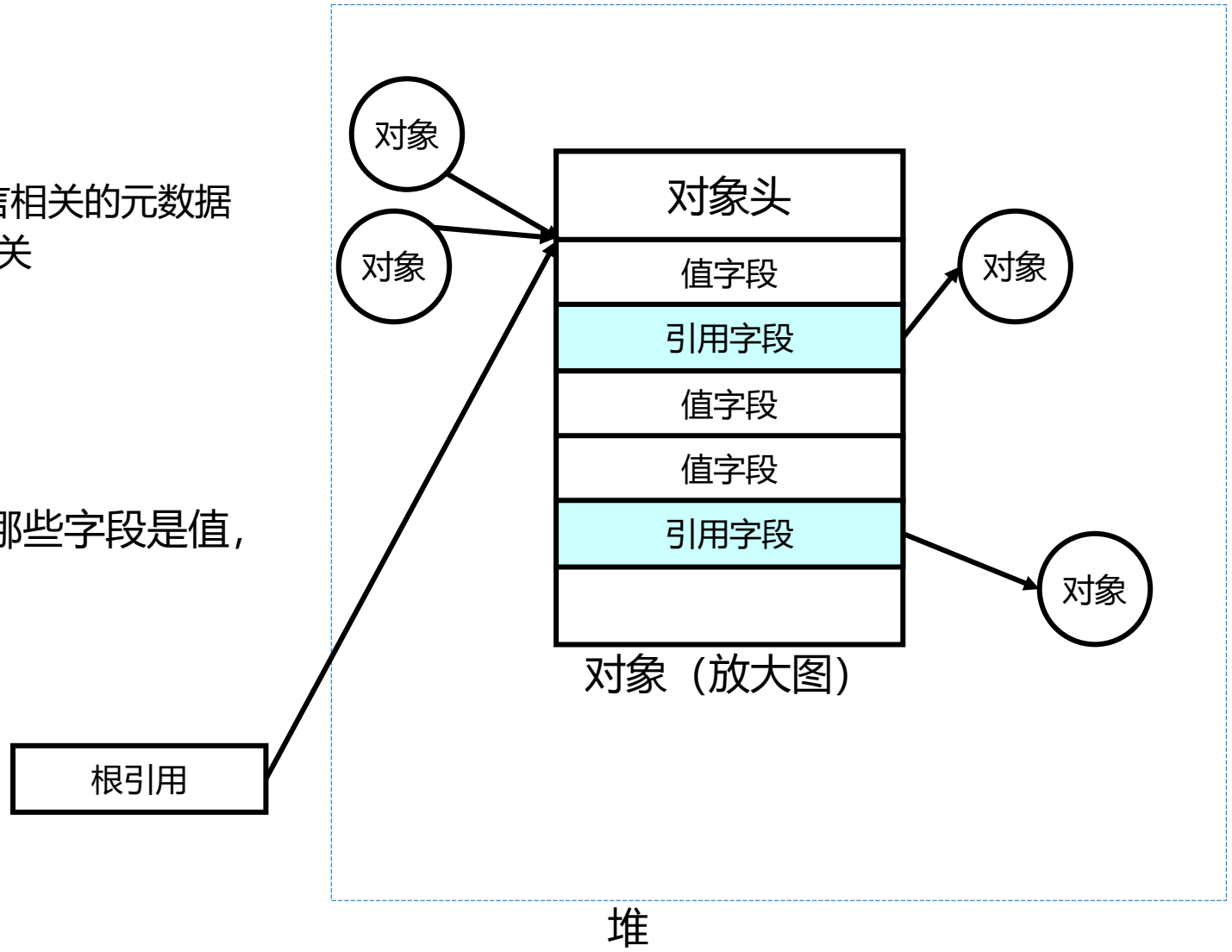
内存垃圾

- 如果一个对象**将来永远不会被使用到**，它就是垃圾。
  - 被判定为内存垃圾的对象不会再被程序访问到



对象布局（object layout）

- 对象头（header）
  - 有和GC相关的元数据，也有和语言相关的元数据
  - 可有可无，和具体语言、虚拟机有关
    - 元数据可以集中放在一块特定区域
- 字段（field，也叫“域”）
  - 值字段
  - 引用字段
- 运行时（虚拟机）有能力识别对象哪些字段是值，哪些字段是引用。



目录

垃圾回收的三个组成部分

垃圾回收算法的三个组成部分

- 内存分配：给新建的对象分配空间
- 垃圾识别：识别哪些对象是垃圾
- 内存回收：将垃圾占用的空间回收，以便将来继续分配

具体的垃圾回收算法（如mark-sweep, mark-compact等）是以上三者的组合。

内存分配

内存分配的两种基本方法

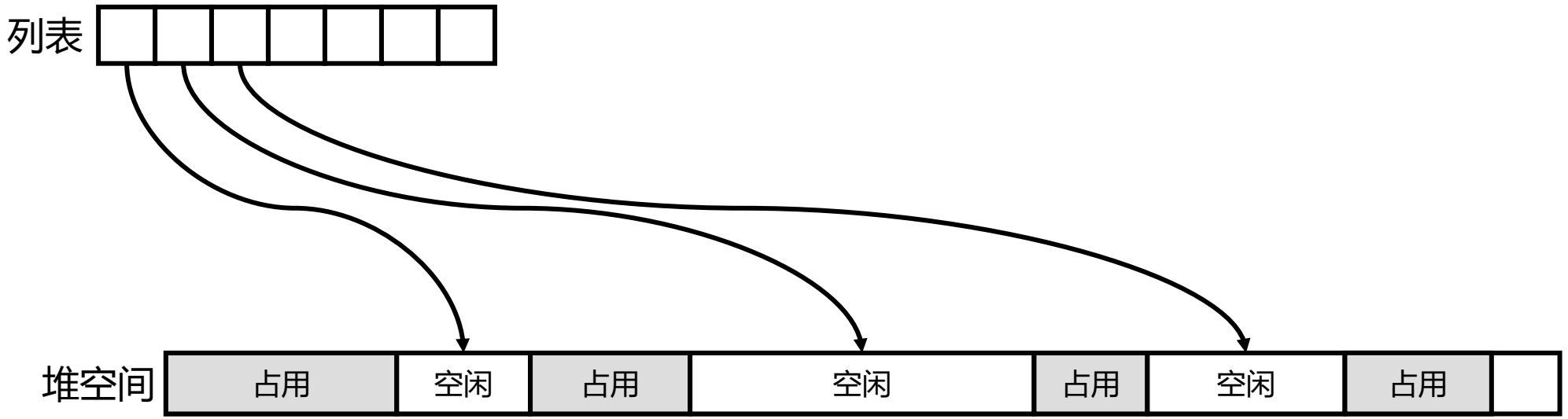
Free-list
(空闲列表)

Bump-pointer
(顺序分配)

Region-based （基于区块的分配器）
(属于综合型的方法)

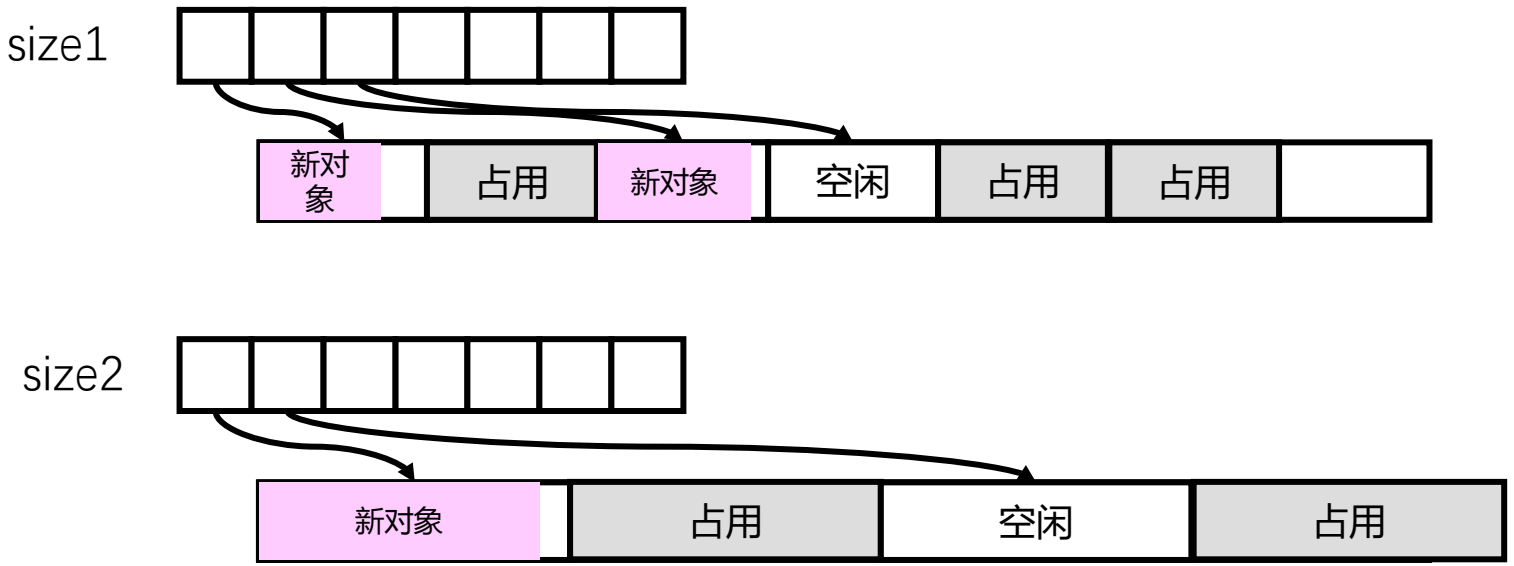
Free-list
空闲列表

- 用一个列表记录哪些区域是空闲的。



Segregated Free-list

- 实践中，空间一般按照对象大小分割管理。
  - 提高速度
  - 一定程度上缓解内存碎片（不能彻底缓解）



Bump-pointer
顺序分配

- （有时译作“阶跃指针”）
- （也叫bump-a-pointer、contiguous allocation（连续分配））
- 空闲空间是连续的，从空闲空间的开头连续分配即可。

内存分配方式比较

- Bump-pointer分配方式比free-list更快
  - 操作简单，Cache局部性好

分配方式	内存分配速度	Cache局部性（影响访问速度）
Free-list	慢	差
Bump-pointer	快	好

垃圾识别

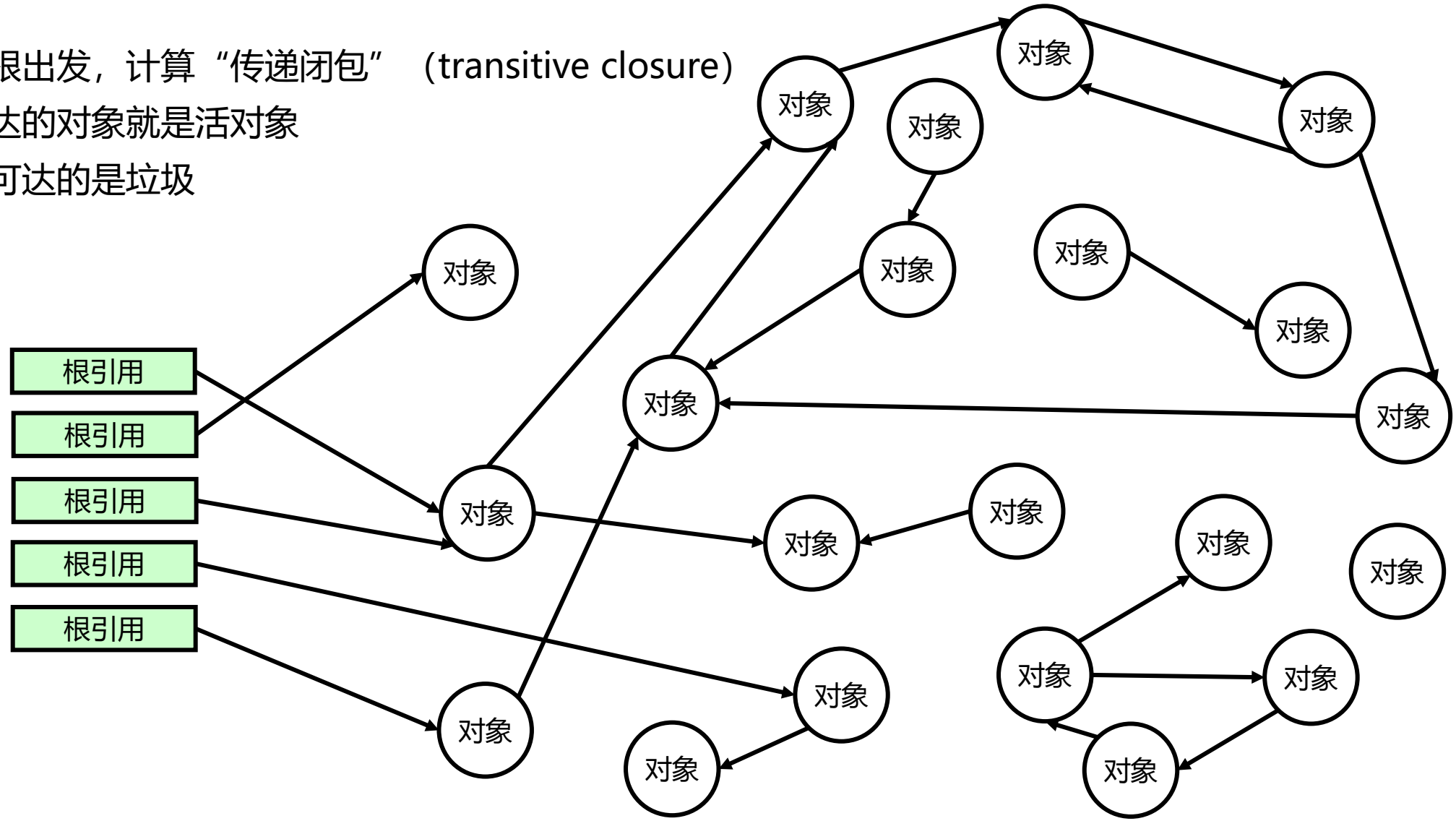
垃圾识别的两种基本方法

Tracing
(跟踪)

Reference counting
(引用计数、RC)

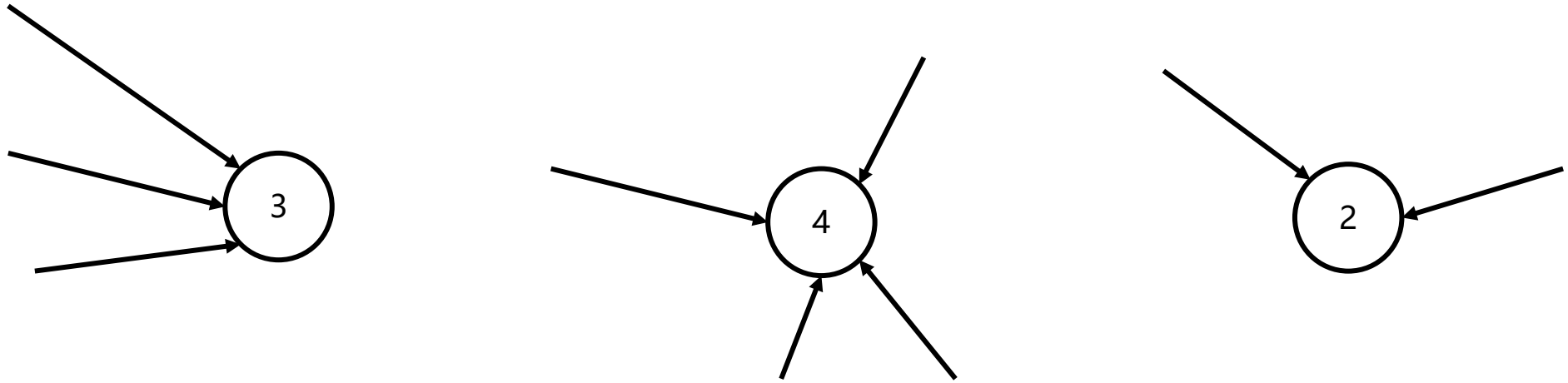
Tracing（跟踪）

- 从根出发，计算“传递闭包” （transitive closure）
- 可达的对象就是活对象
- 不可达的是垃圾



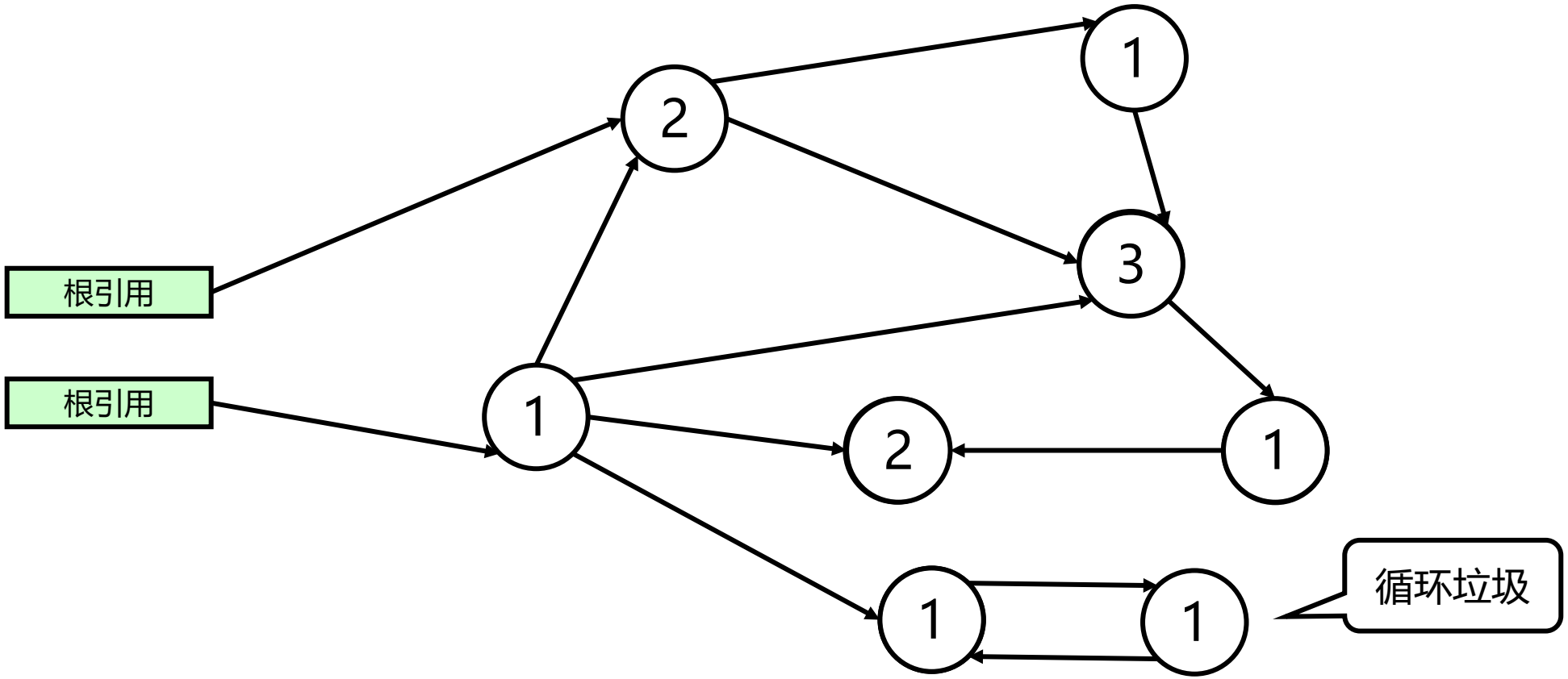
Reference counting（引用计数）

- 为每一个对象记录：有多少个引用指向它，称为这个对象的 “引用计数” （reference count，RC）
- 当一个对象的引用计数降到0时，这个对象必然不能从根抵达，必然是垃圾。



演示：Naive reference counting（naive RC，朴素引用计数）

- 又称immediate RC（立即引用计数）
- 每次有引用增删，立即更新引用计数



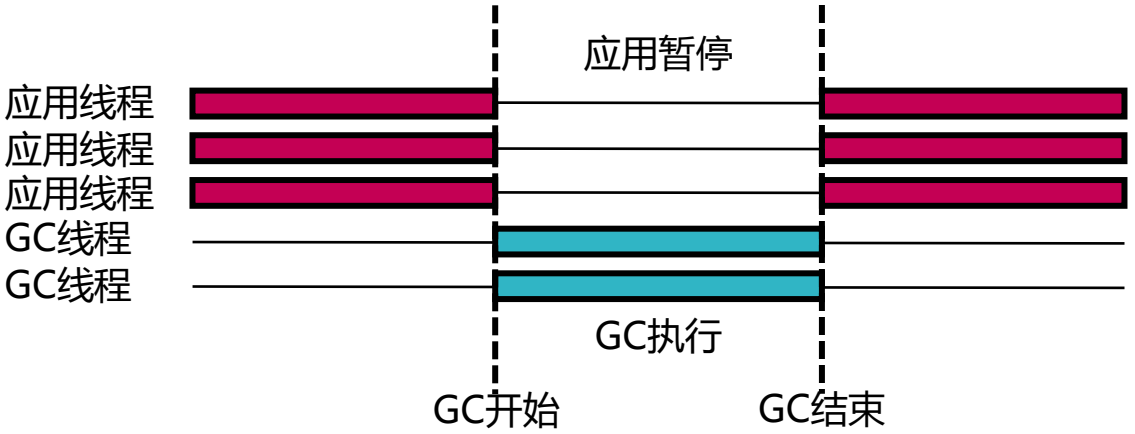
垃圾识别方式比较

- Tracing是完整的，环形垃圾不影响可达性。
- Reference counting不能识别环形垃圾。
  - 可以用tracing补充reference counting，称为“backup tracing”。
    - 注：有的RC算法使用trial deletion，但trial deletion速度很慢，不建议使用。
- Reference counting可以及时回收内存以供重用。
- Tracing需要扫描整个堆才能发现垃圾。
  - 可以用分代（generational）的GC来缓解。

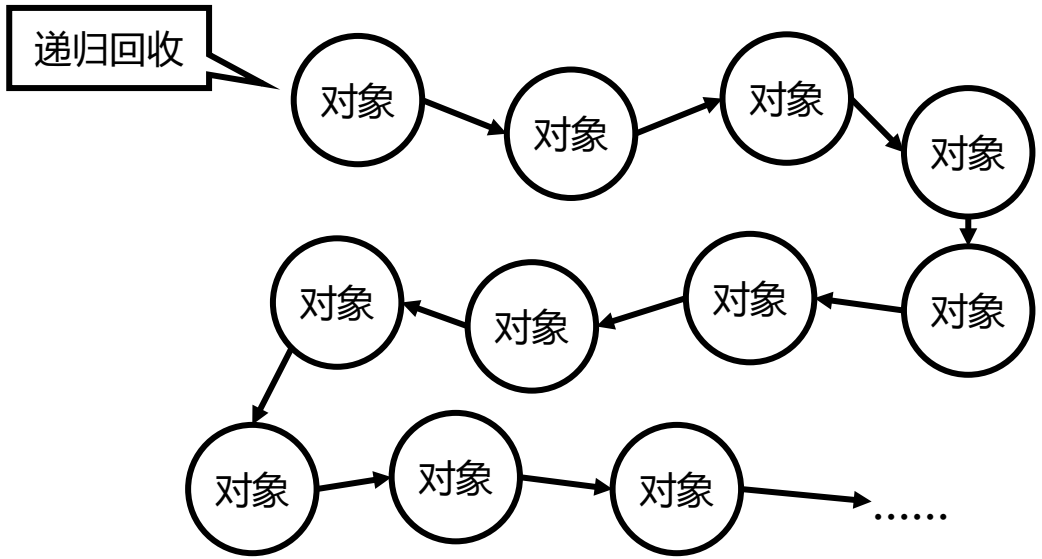
垃圾识别方式	环形垃圾处理	回收的及时性
Tracing	可以处理	不及时
Reference counting	不能处理	及时

注意：“及时”不等于“不卡顿”

- **“及时”**的意思是可以尽早发现垃圾，释放它们的内存，以供继续分配，**不容易耗尽内存。**
- **“卡顿”**的意思是GC造成应用程序线程暂停执行（即stop-the-world），无法响应业务请求。
  - 触发stop-the-world GC，以及用RC同步地回收一个极大的单向链表，都会使应用程序线程停顿。
  - 可以用并发（concurrent）GC来及解决



触发stop-the-world GC造成卡顿



RC同步回收大数据结构造成卡顿

内存回收

内存回收的三种基本方式

适用于free-list分配器

Sweep to free-list
清扫到空闲列表

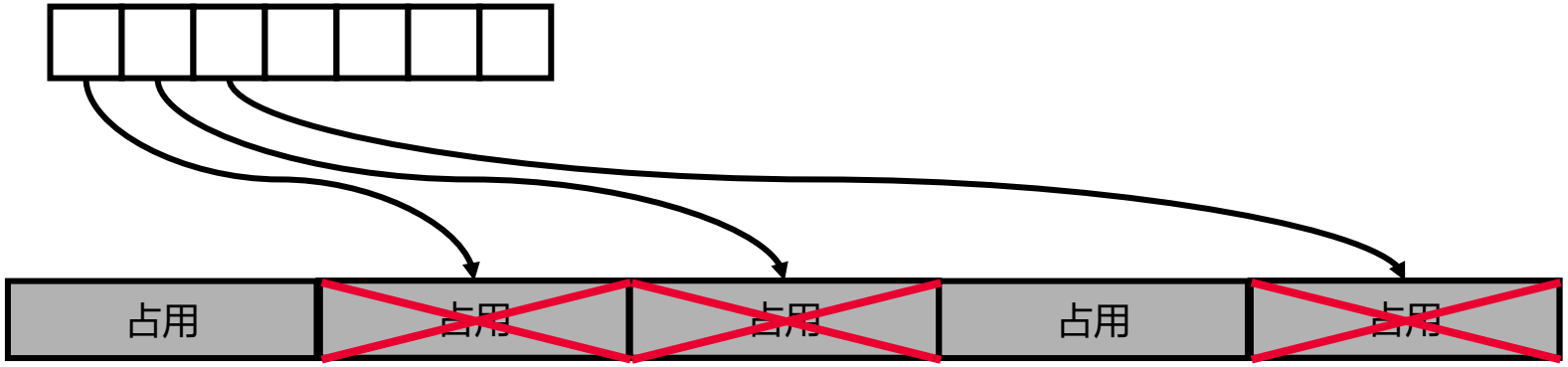
适用于bump-pointer分配器

Compaction
压缩

Evacuation
撤离

Sweeping (to free-list)
清扫（到空闲列表）

- 垃圾占用的空间直接加回free-list

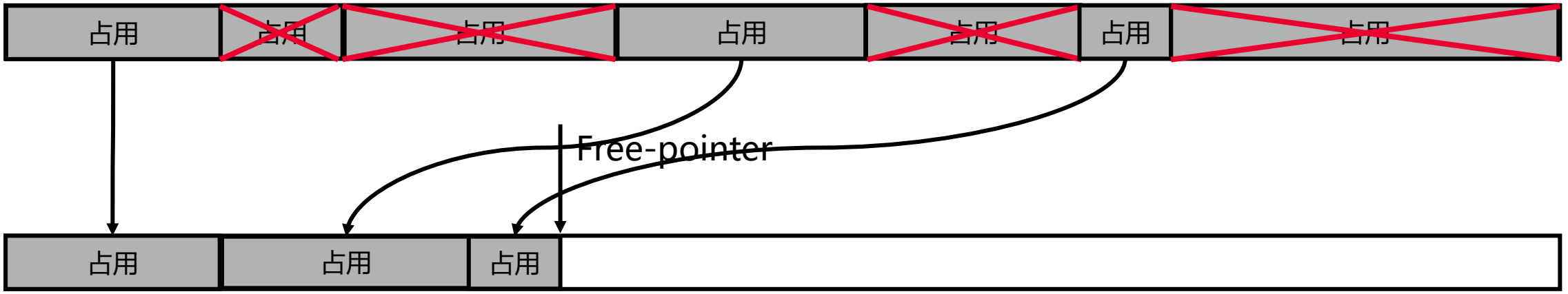


Compacting
压缩（也叫“整理”）

- 将活对象推向空间的一侧

Evacuation
撤离

- 将活对象拷贝到另一块连续空间
- 原空间全部释放



内存碎片问题（fragmentation）

- 大量零散的小空闲空间
- 大对象无法找到合适的位置分配
  - 即使总空间足够也无法分配
- 内存利用率降低
- 通过移动对象（压缩、撤离）来解决

无处可分配

内存回收方式比较

- Sweep to free-list回收速度快，因为无需拷贝对象，只需要加入列表即可。
- Compaction和evacuation都需要拷贝对象，速度慢。
  - Compaction节省空间，但由于原地址和目标地址重叠，速度很慢。
  - Evacuation相对较快，但需要额外的空间。

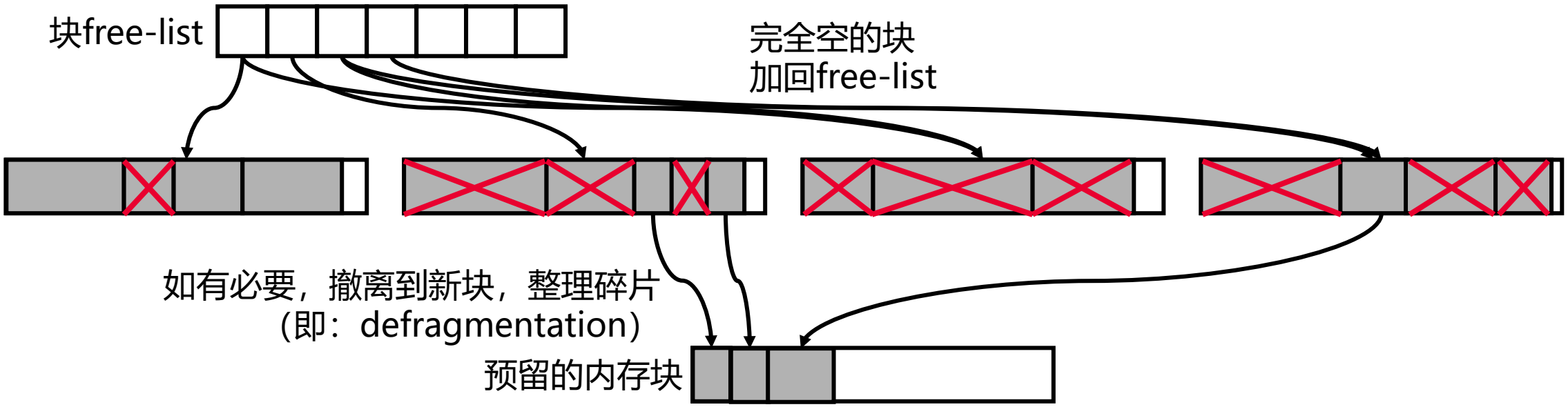
回收方式	移动对象	回收速度	内存碎片
Sweep to free-list	不移动	快	有碎片问题
Compaction	移动	很慢	无碎片问题
Evacuation	移动	慢	无碎片问题

具体的垃圾回收算法是
内存分配、垃圾识别、内存回收
的有机组合

目录

分块内存分配、回收器（region-based、regional）

- 内存分成等大的大块（chunk，一般超过一个页，如32KiB）
- 分配时，在块内顺序分配，对象不可以跨块。
- 回收时，完全空闲的块可以整块放入free-list。
- 如果有需要*，可以通过拷贝，整理内存碎片。
  - *注：有的GC可以往不全空的块里继续分配，有的GC则要求必须先清空再分配。



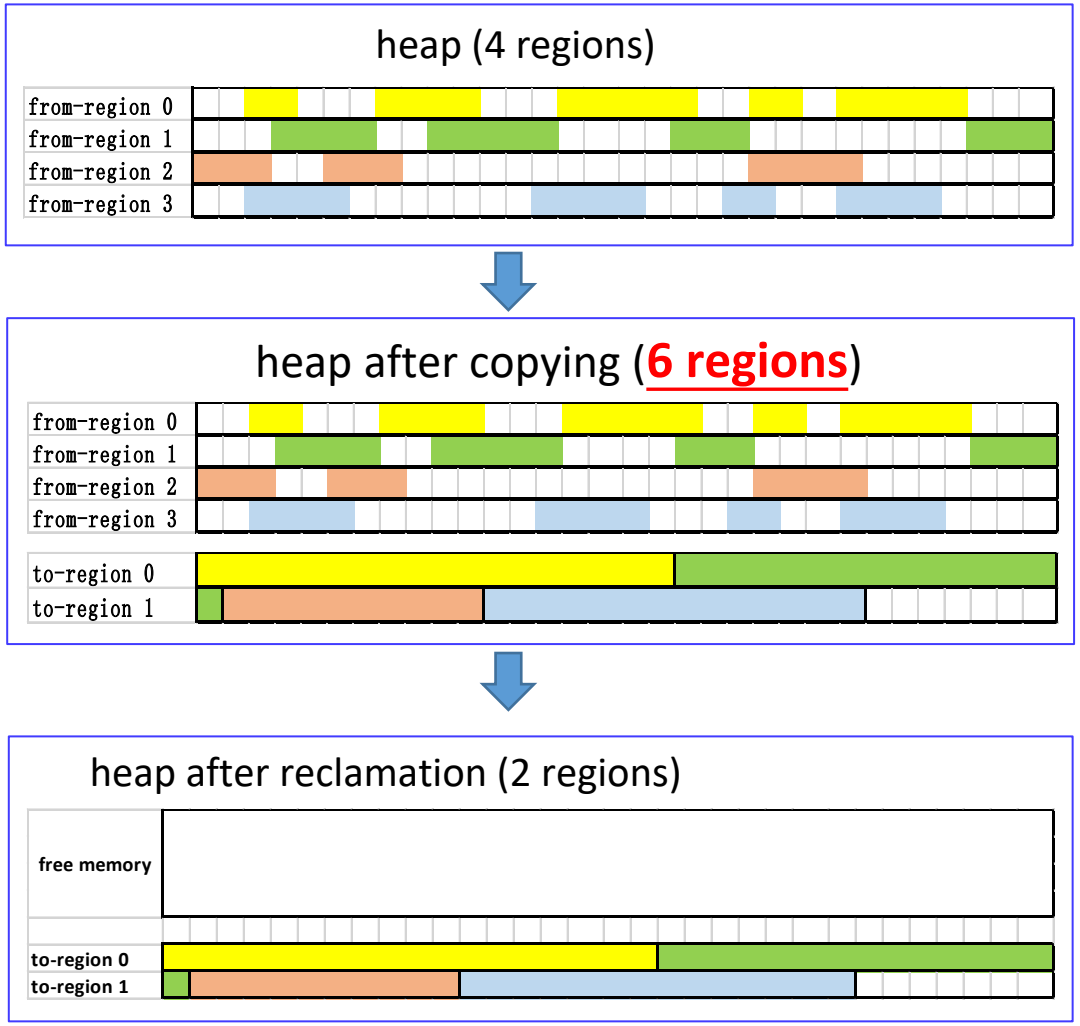
分块垃圾回收器举例

2000年以来出现了很多分块的垃圾回收算法

GC或虚拟机	诞生时间	描述
Lang & Dupont	1985	分块用于整理碎片，但仍然使用free-list分配内存。
Mature Object Space	1992	分块、分代，采用remembered set。
Garbage First（G1）	2004	并发GC算法。OpenJDK 9开始作为缺省GC算法。
Pauseless	2005	Azul开发的并发GC，实现在Azul的Zing JVM上。
JRockit	不详	可能是最早的mark-region GC，已停止维护。
Immix	2008	一种Mark-region GC。可以重用部分分配的块，有必要时才整理内存。
RC-Immix	2012	Immix的变种，使用deferred RC。
Shenandoah	2016	RedHat开发的并发GC算法，包含于OpenJDK 12
ART (Android RunTime)	2017	自8.0以来，ART的GC也是分块的并发GC算法
ZGC	2018	Oracle开发的并发GC算法，包含于OpenJDK11

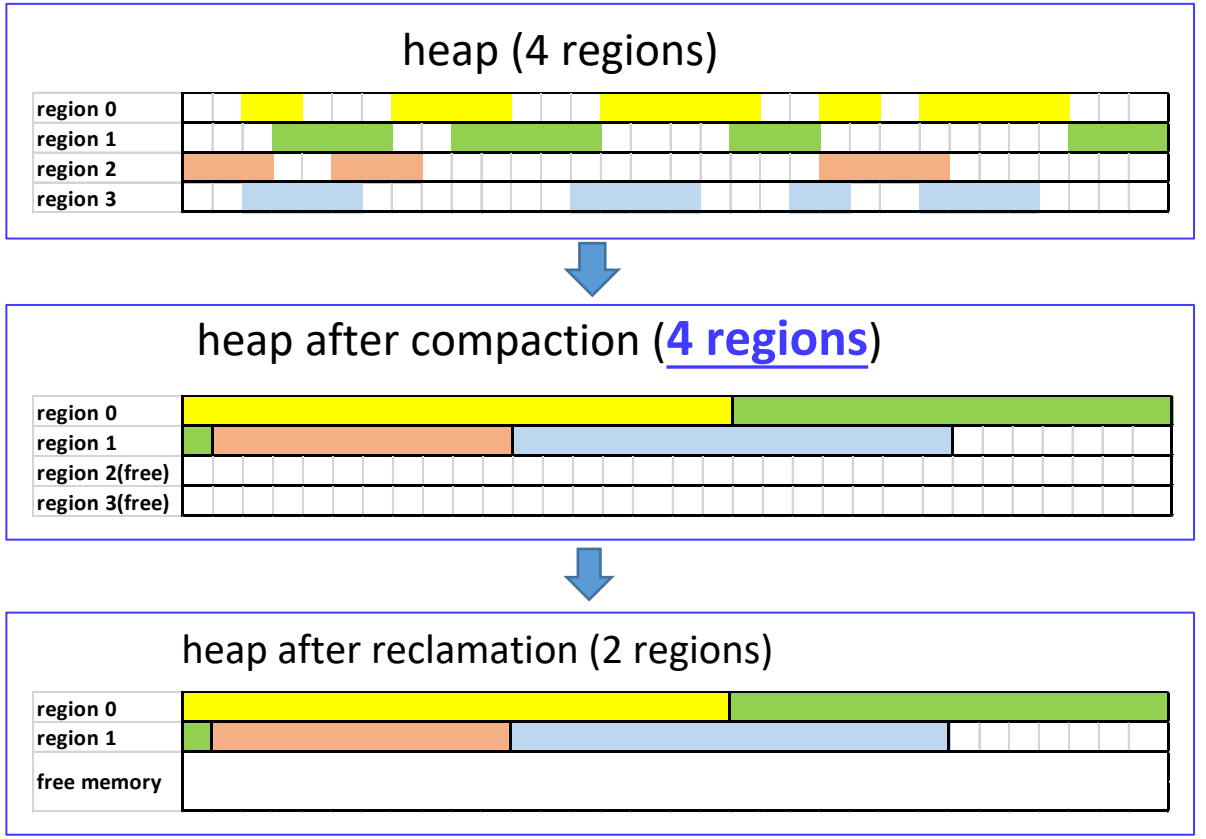
内存搬移的策略比较

内存复制



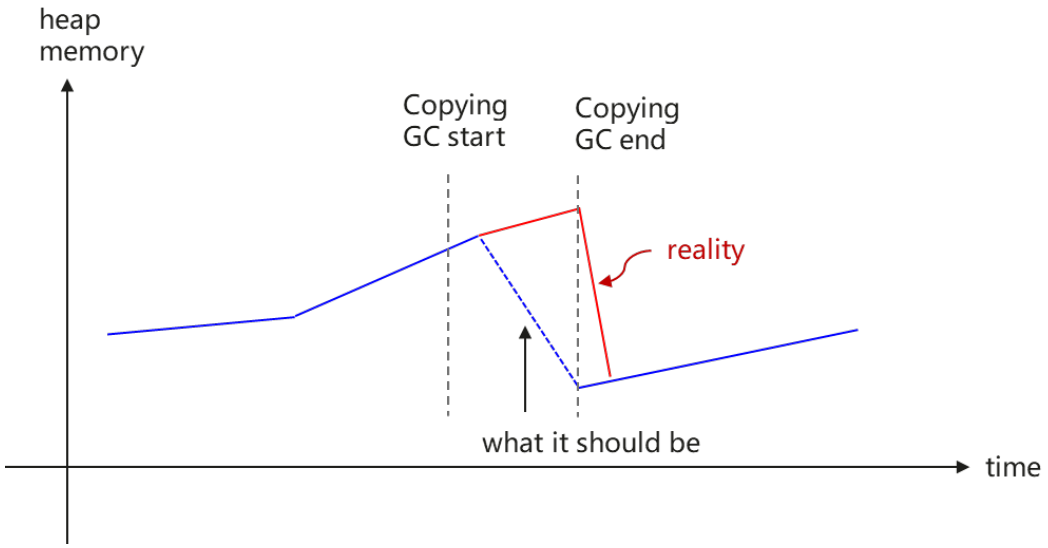
对比

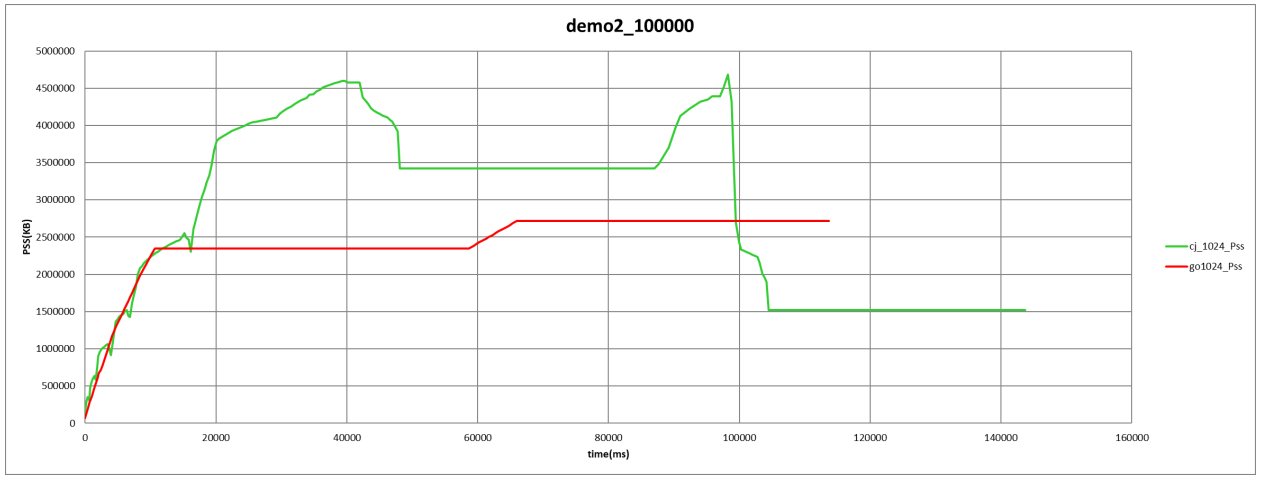
内存整理



在内存整理算法里，from-region里的活对象被搬移后可立即用于内存分配，从而降低内存峰值

复制算法的内存峰值问题





早期基于并发复制GC某仓颉应用执行过程的内存曲线

- semi-space copying算法导致GC过程中的存活对象占用两份内存，GC完成后才能释放旧数据占用的内存

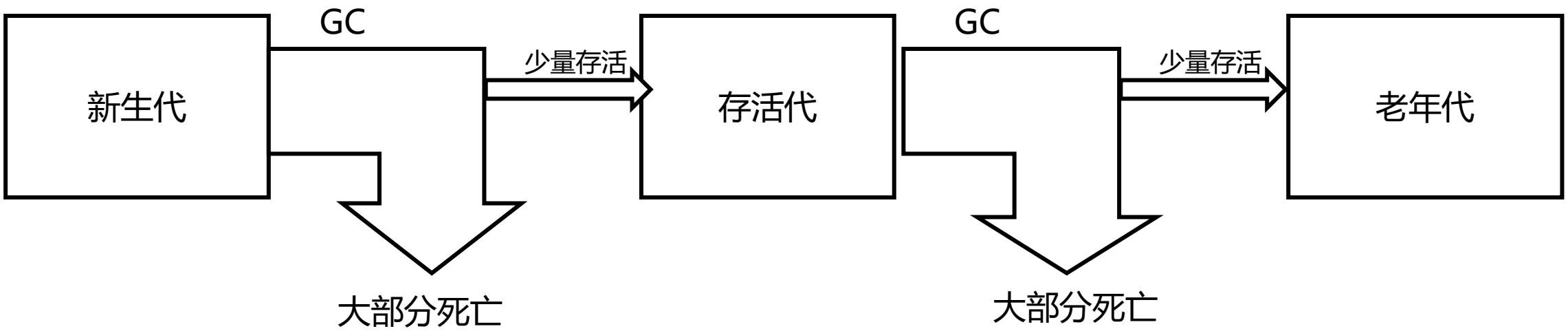
- 次生问题：尾延迟劣化
  > 内存峰值导致堆内存提前耗尽
  > 应用线程需等待GC完成才能分配内存： Allocation Stall
  > Out-Of-Memory GC需要在全局暂停(Stop-The-World)状态执行

分代垃圾回收

提问：对象的“新生儿死亡率”高不高?

分代垃圾回收
Generational GC

- 大多数对象都“早死”。
- 少数对象可以活下来，而且活得很久。

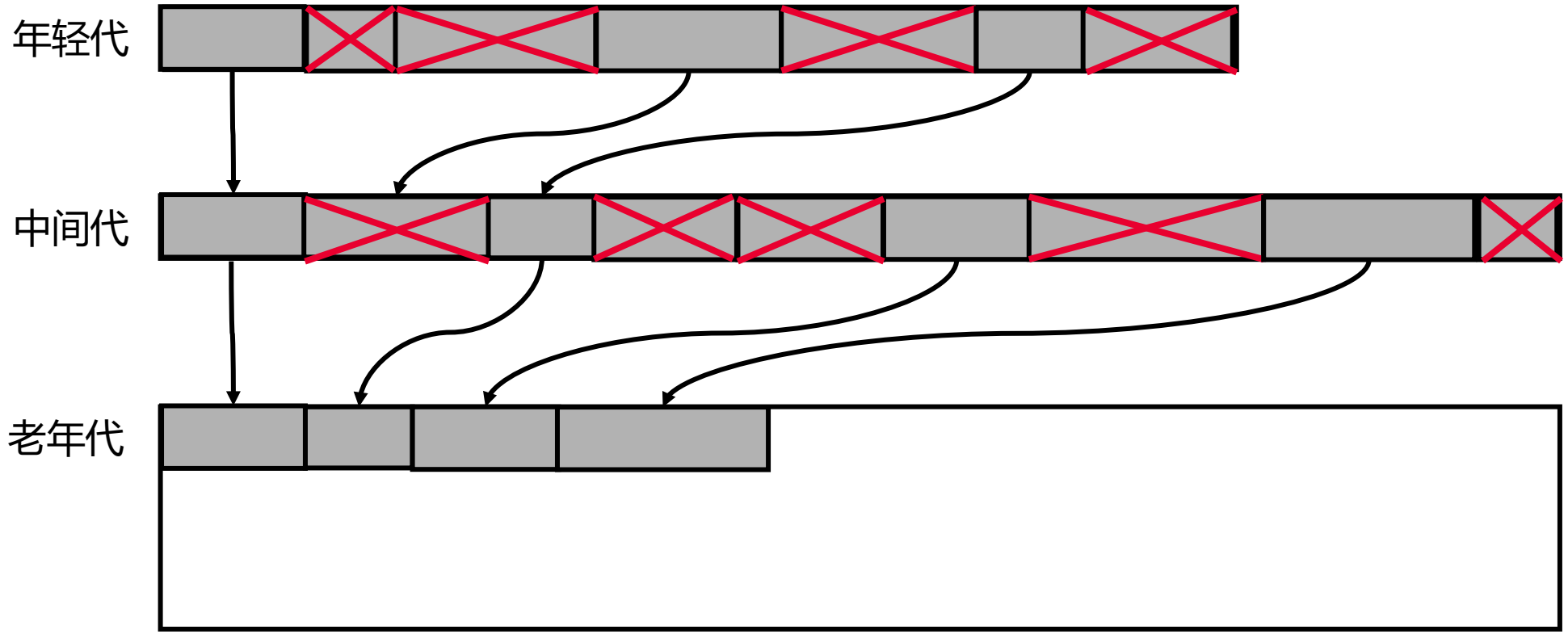


所以，经常在新对象中找垃圾

尽量少去扫描老对象（不太可能变成垃圾）

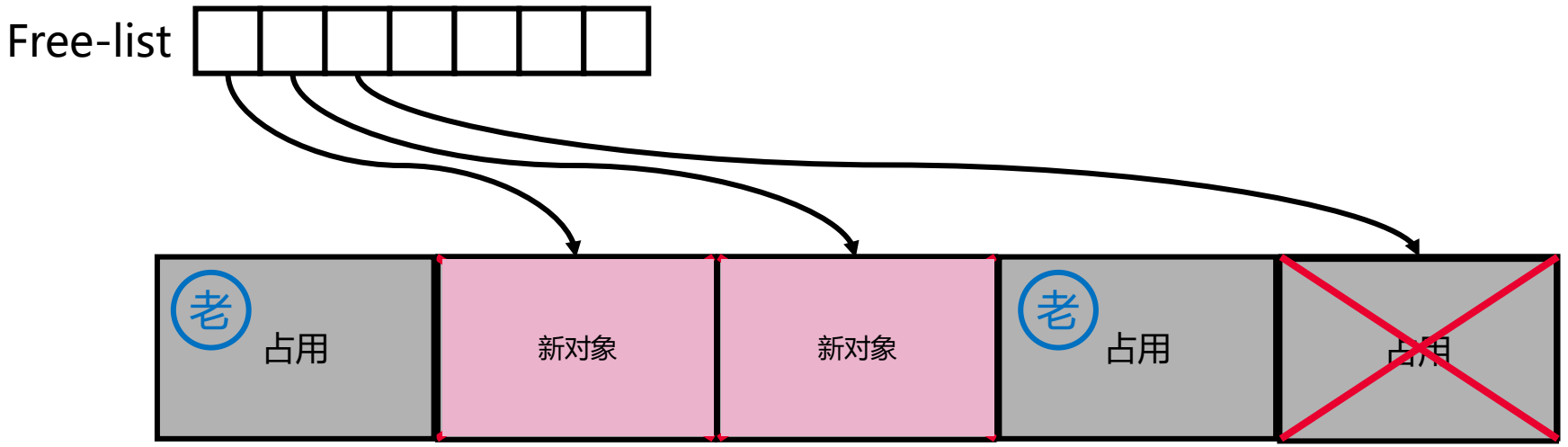
分代垃圾回收演示：法1：按代分割区域

- 类似semi-space，但大小不同



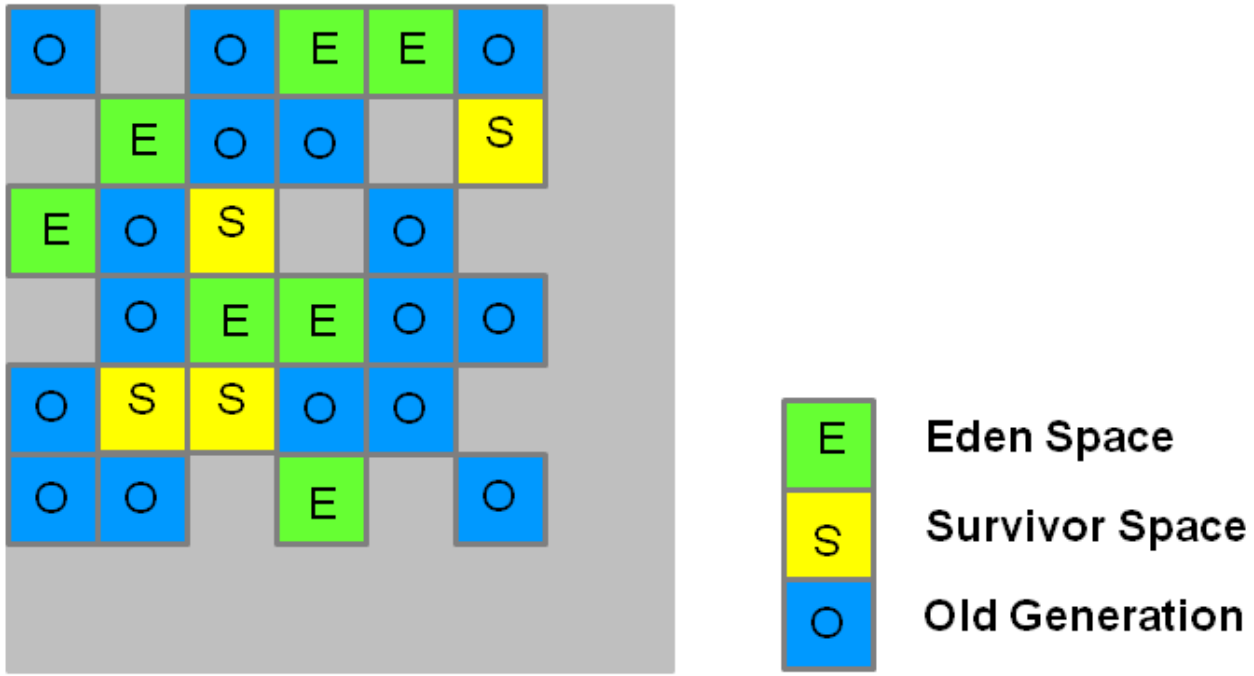
分代垃圾回收演示：法2：用位（sticky bit）标记对象

- 不用移动对象，适用于free-list



分代垃圾回收演示：法3：不同区块担任不同角色

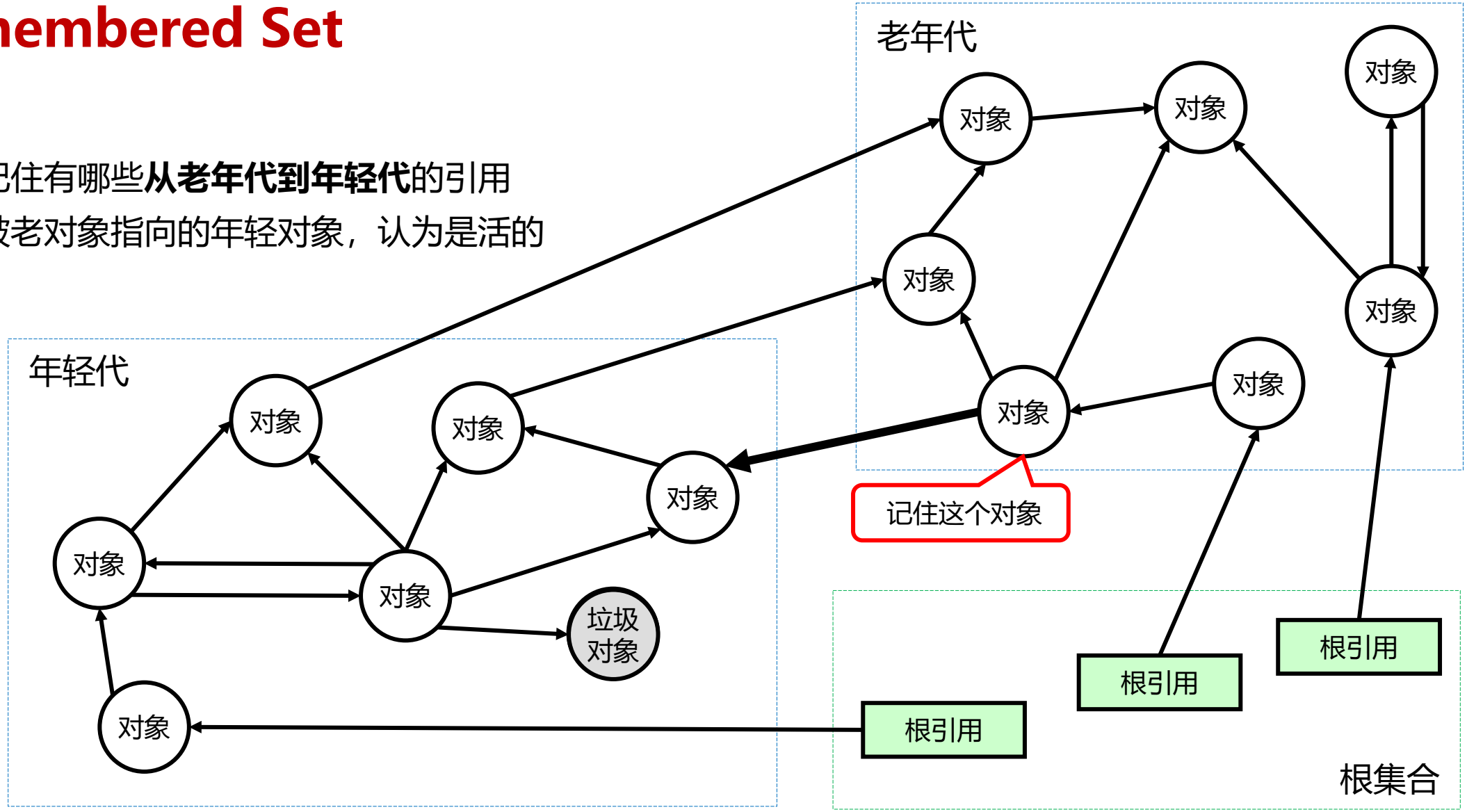
- 每个区块做标记，不要求同一代的对象都位于连续区域。
- 适合于分块的GC算法，如Garbage First （G1）



来源：Oracle. **Getting Started with the G1 Garbage Collector**. https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

Remembered Set

- 记住有哪些**从老年代到年轻代**的引用
- 被老对象指向的年轻对象，认为是活的



分代GC的算法选择

年轻代常采用
copying

年轻代
evacuation

Eden代
evacuation

Survivor代
evacuation

老年代可采用
sweep/compact

老年代
mark-sweep

Tenured代
Mark-compact

Generational mark-sweep

OpenJDK 1.8及以前的“Parallel”
Generational mark-compact

目录

并行与并发垃圾回收

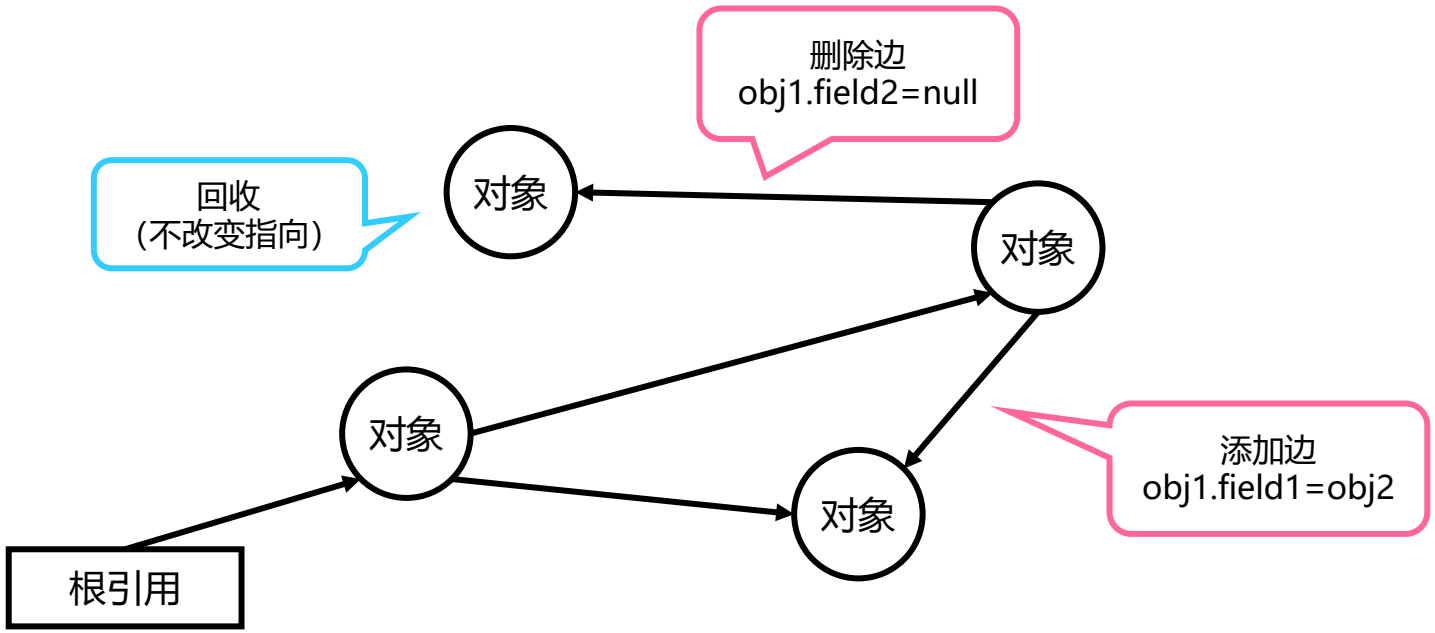
概念：mutator和collector

Mutator

- 译作“赋值器”
- 一般指应用程序线程
  - Mutator可以改变对象图的结构
  - 即：增加引用、删除引用、修改引用等

Collector

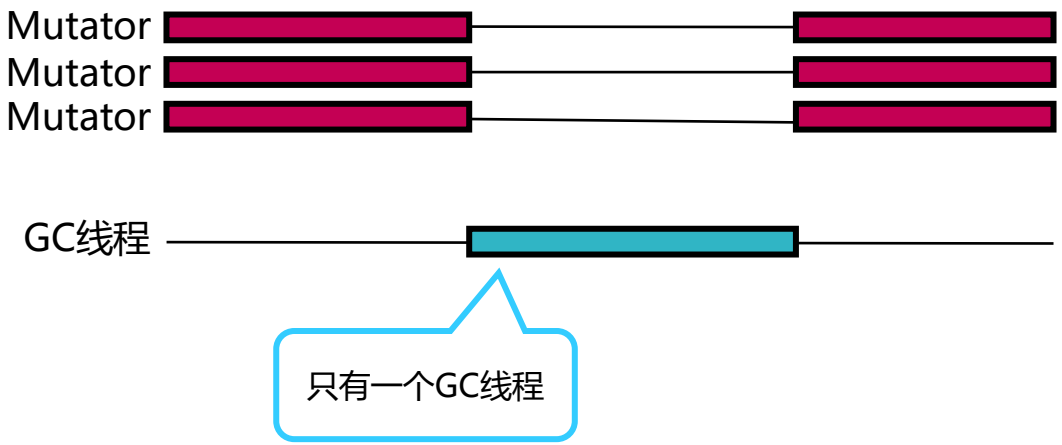
- 译作“回收器”
- 指GC线程
  - GC线程从来不会改变对象图的结构



并行GC　(Parallel GC)

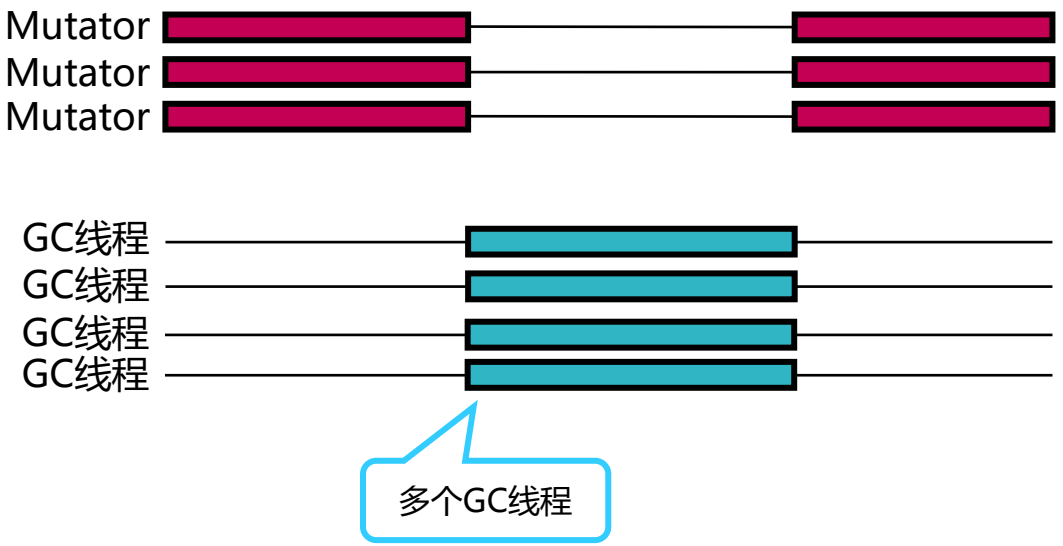
串行GC

- 只有一个GC线程



并行GC

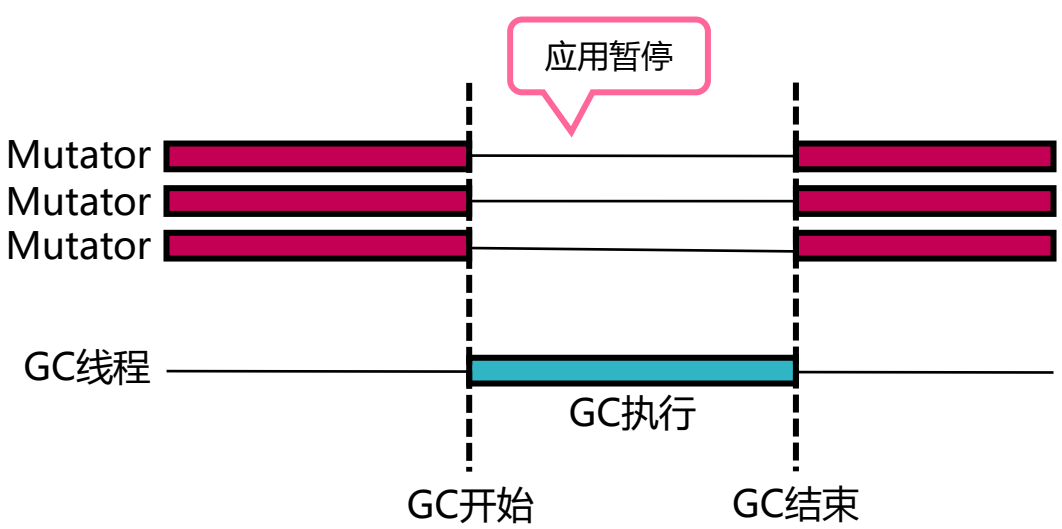
- 有多个GC线程



并发GC （Concurrent GC）

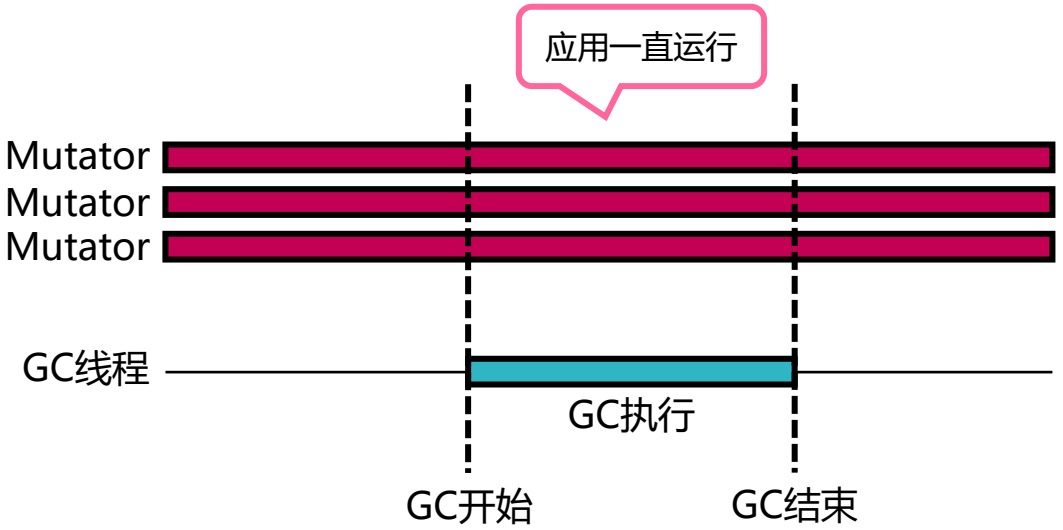
Stop-the-world (STW) GC

- GC线程不能和mutator线程同时运行

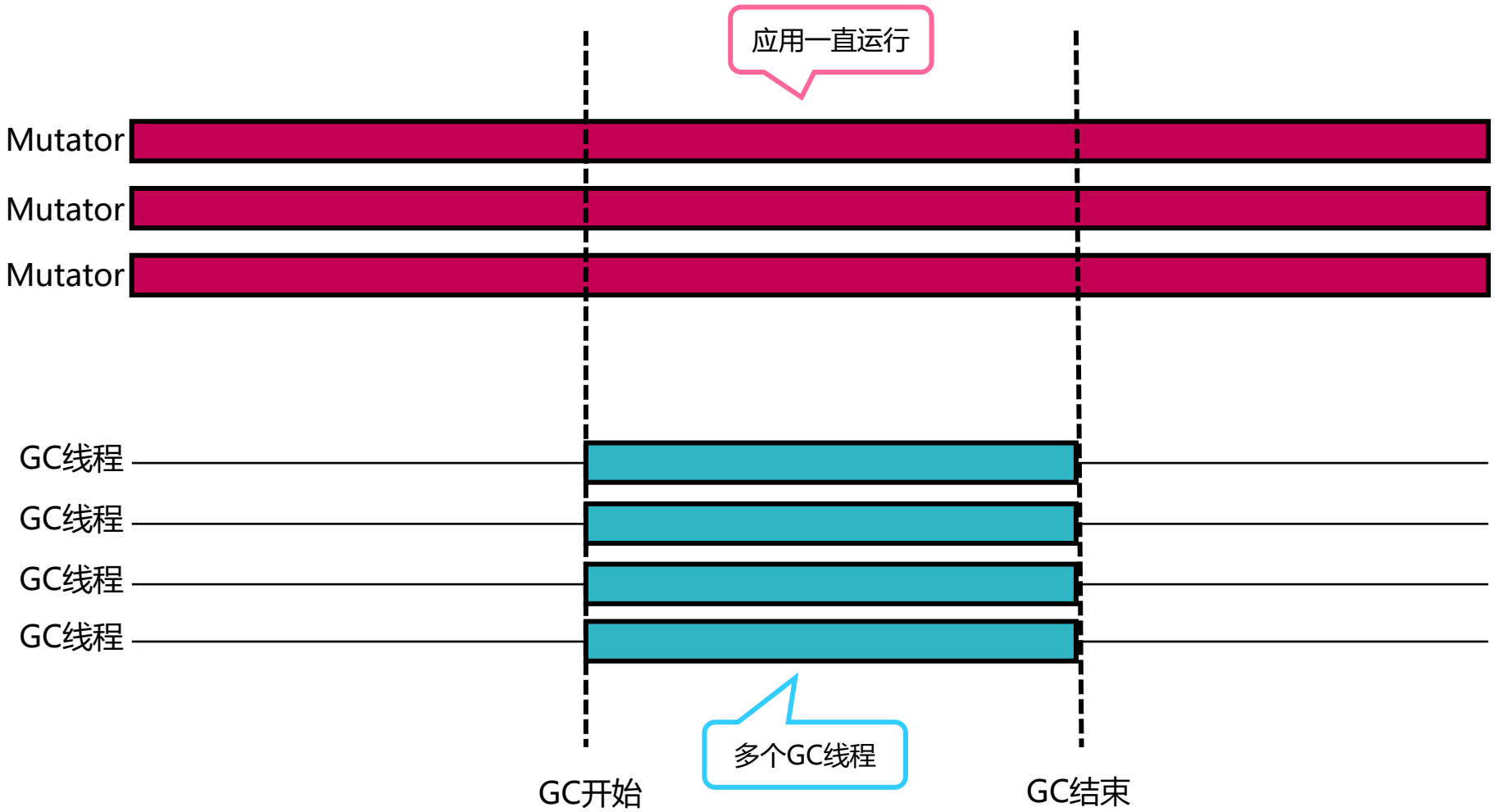


并发GC

- 有多个GC线程



又并发又并行也是可以的



并行和并发GC解决不同的问题

并行（parallel）

- 提高GC运行速度
- 提高吞吐率

- 对**数据处理**来说，吞吐率很重要

并发（concurrent）

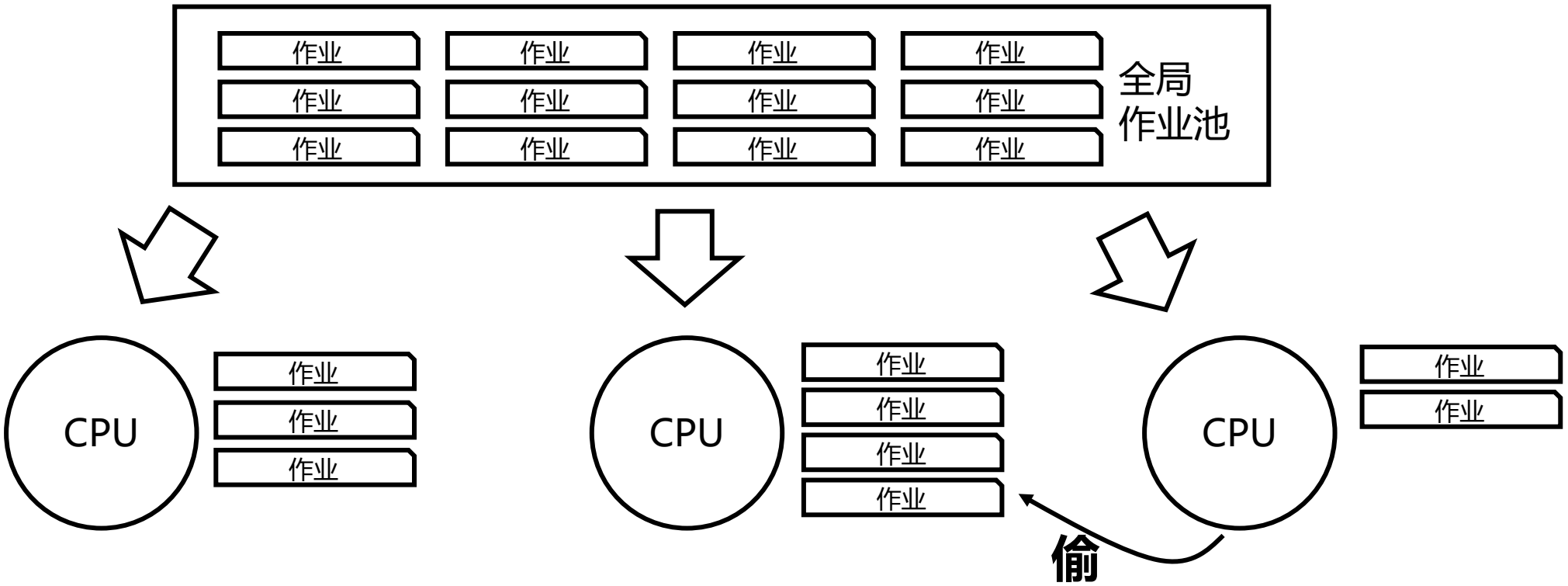
- 不停止应用程序（mutator）线程
- 降低GC延迟，防止“卡顿”

- 对**交互应用**（桌面、手机），响应时间很重要
- 对**服务器**，响应时间也很重要

我们不做选择题：人们追求吞吐率高，同时延迟率低

并行（parallel） GC的难点

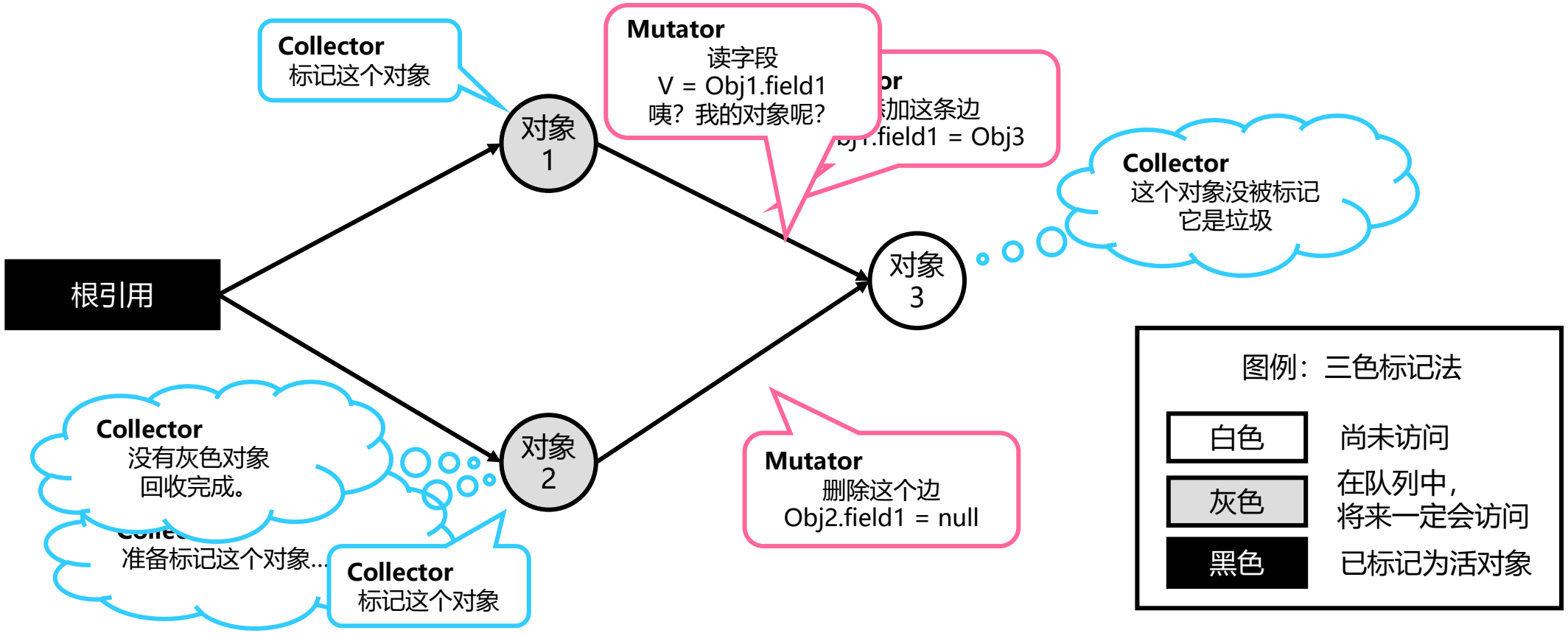
- 如何将大量的GC作业（对象扫描）均匀地分布在多个CPU上
- 典型的多线程负载均衡问题
- **work-stealing**



并发（concurrent）GC的难点

Mutator会在Collector工作的时候，改变对象的指向

- 这会影响垃圾识别（即marking）的过程
  - 活对象被当成垃圾

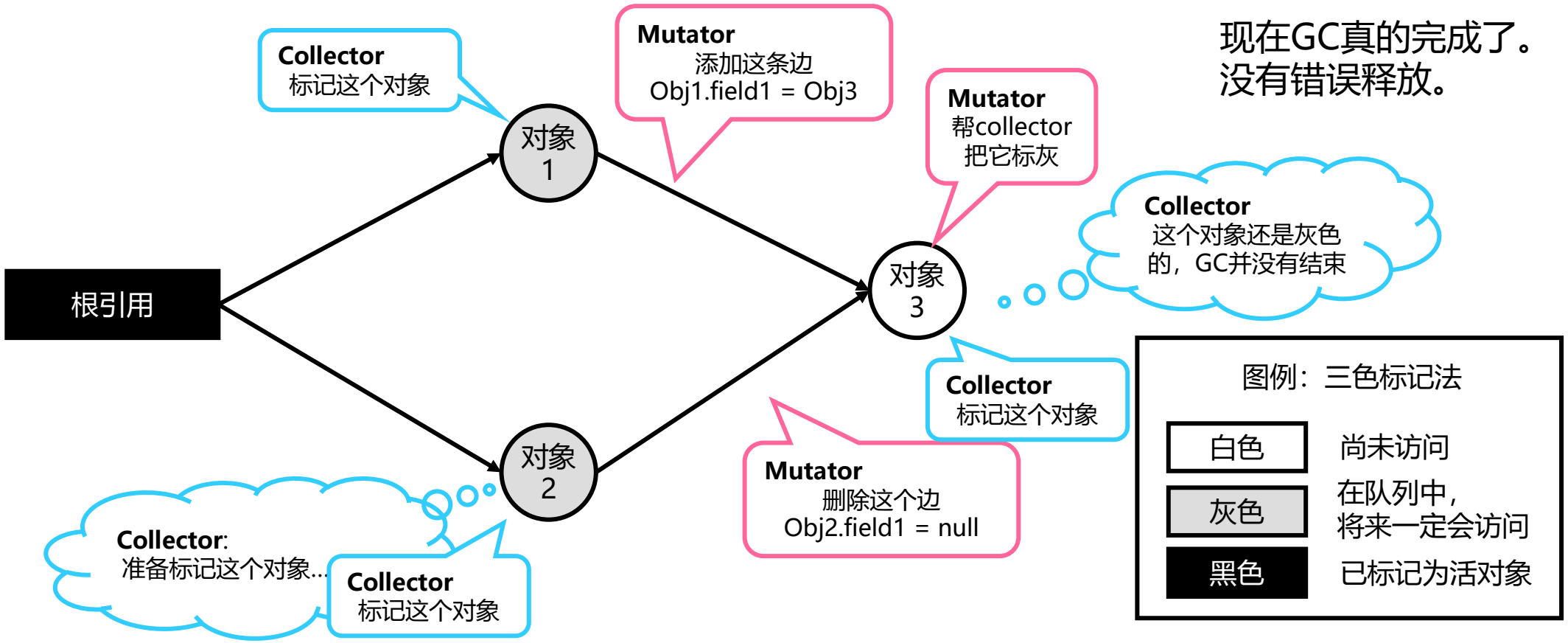


重点：

Mutator必须帮助Collector识别垃圾

Mutator帮助Collector标记对象

- 举例：在添加边（写引用字段）的时候，将目标对象加入队列（标灰）

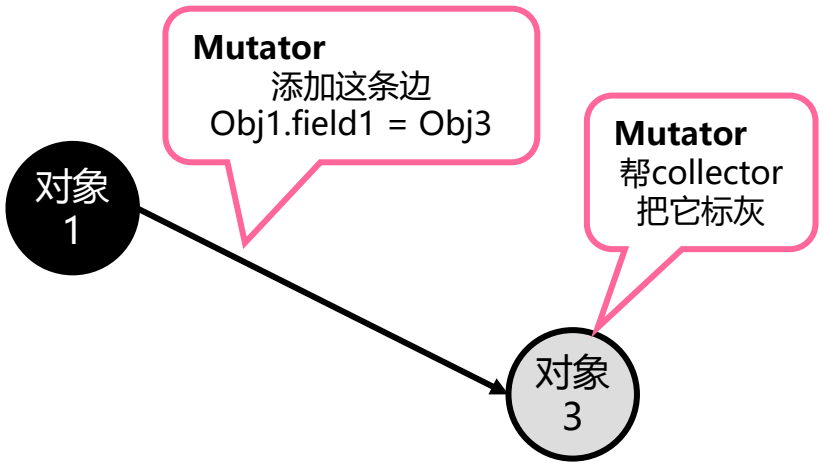


要点：

barriers for access heap

Barriers

- Mutator伴随着读、写操作而执行的，辅助GC的操作。




```
Address storeObjectField(
      Address object,
      Offset field,
      Address target) {
  object[field] = target;
  if (object.color == BLACK &&
      target.color == WHITE) {
    target.color = GREY;
  }
}
```

并发GC的barrier以及算法举例

GC算法或barrier设计	诞生时间	Barrier	移动对象	描述
Dijkstra's barrier	1976	写	否	增加边时，将目标对象标黑，推进标记过程。用于Go语言。（上一页演示的算法）
Steele's barrier	1976	写	否	增加边时，将源对象标白，后撤标记过程。
Baker's barrier	1978	读	是	读引用时将目标对象标灰。
Brook's barrier	1984	读	是	对象头有个指向拷贝后的对象的指针，读写任何字段之前，都要更新为新版指针。
SATB barrier	1990	写	否	修改边时，将原来的目标标灰。即Snapshot At The Beginning。
Recycler	2001	写	否	基于Deferred RC的并发非拷贝GC。
Sapphire	2001	写	是	分阶段拷贝。可从任意space读，但写时要同时向from space和to space写。
Garbage First（G1）	2004	写	是	基于SATB barrier和zone barrier的分块拷贝GC。并发marking，stop-the-world copying
Pauseless	2005	读	是	都是基于Baker或Brook风格的read barrier的分块、并发、拷贝式GC。 其中，ZGC利用OpenJDK 12的机制，使其可以单独和每个线程握手
Shenandoah	2016	读	是	
ART虚拟机的GC	2017	读	是	
ZGC	2018	读	是	

其他Barriers举例

- Mutator伴随着读、写操作而执行的，辅助GC的操作。
- 对性能非常关键。
  - 编译器（JIT或AoT）有能力将快速路径（fast-path）内联

名称	操作	作用	实现举例
Card	写	用于分代GC记录remembered set	`cards[obj >> 9] = 1`

Object	写	记录变化了的对象	`if (isLogged(obj.header)) {`
			`  setLogged(obj);`
			`  modbuf.insert(obj);`
			`}`

Boundary	写	判断对象是否位于区间内，如“年轻代”空间	`if (!inNursery(src) &&`
			`  inNursery(tgt))`
			`  remset.insert(src)`

Zone	写	判断两个对象是否位于同一个区块内，用于G1的remembered set记录	`if ((src ^ tgt) > ZONE_SIZE)` `  remset.insert(src)`
Read	读	清除指针的最后几位，用于指针加了tag的情况	`obj = obj & (~3)`

参考文献： Yang, Blackburn, Frampton, Hosking, **Barriers reconsidered, friendlier still!**, in *Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012.* https://doi.org/10.1145/2258996.2259004

GC算法总结

GC算法总结

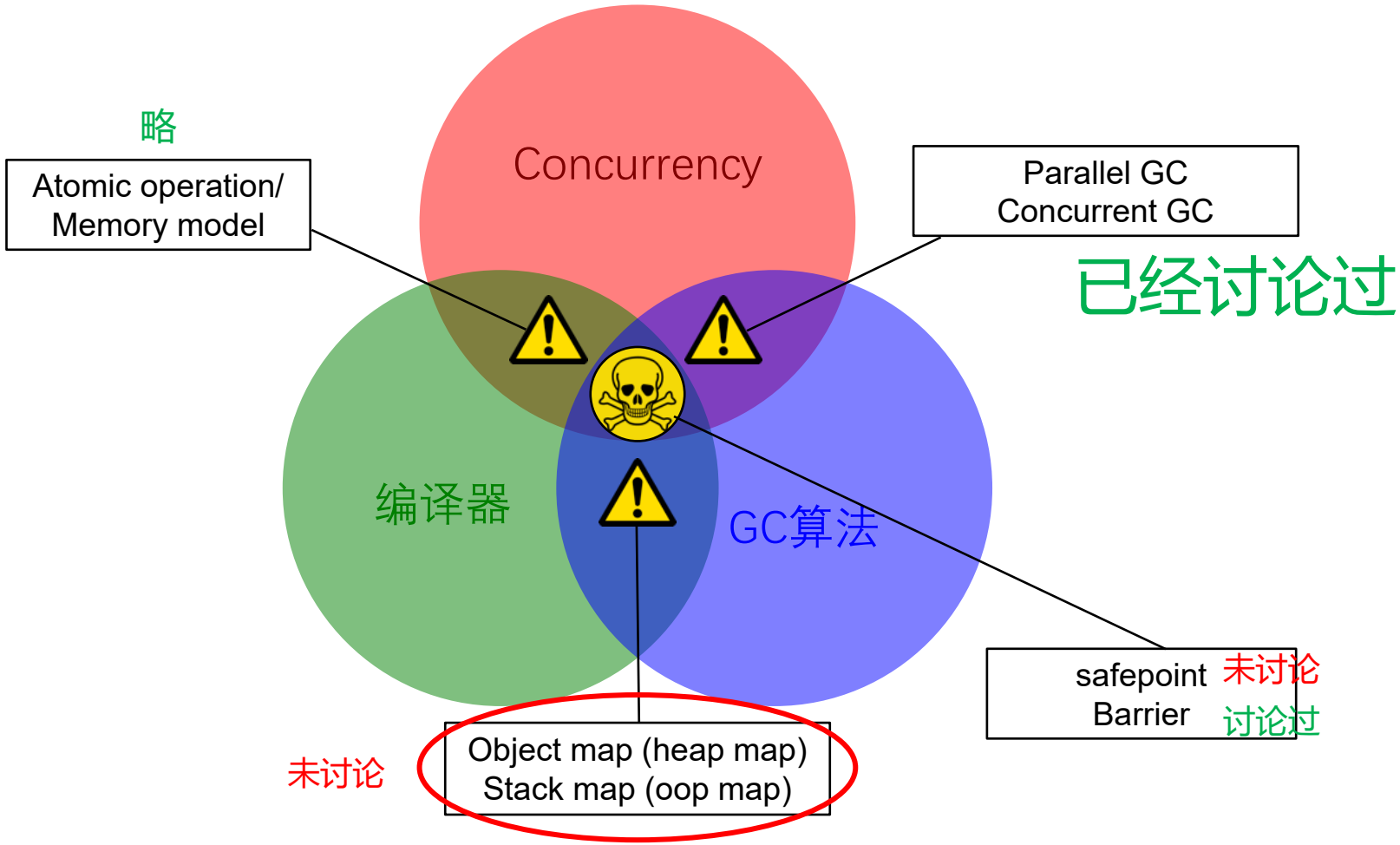
- GC算法是内存分配、垃圾识别、内存回收的有机组合
  - 内存分配：free-list, bump-pointer
  - 垃圾识别：tracing, reference counting
  - 内存回收：sweeping, compaction, evacuation
  - 分块的分配器：free-list和bump-pointer的组合
- 分代垃圾回收
  - 避免每次都做全堆扫描
- 并行、并发垃圾回收
  - 并行：多个GC线程，提高吞吐率
  - 并发：GC和Mutator同时运行，减少卡顿

目录

CONTENTS

编译器需要对垃圾回收提供什么样的机制支持?

编程语言的实现中，编译器与垃圾回收是紧耦合的



来源：Steve Blackburn. Micro Virtual Machines. In PLISS’17. https://youtu.be/T2WViuD5GrQ

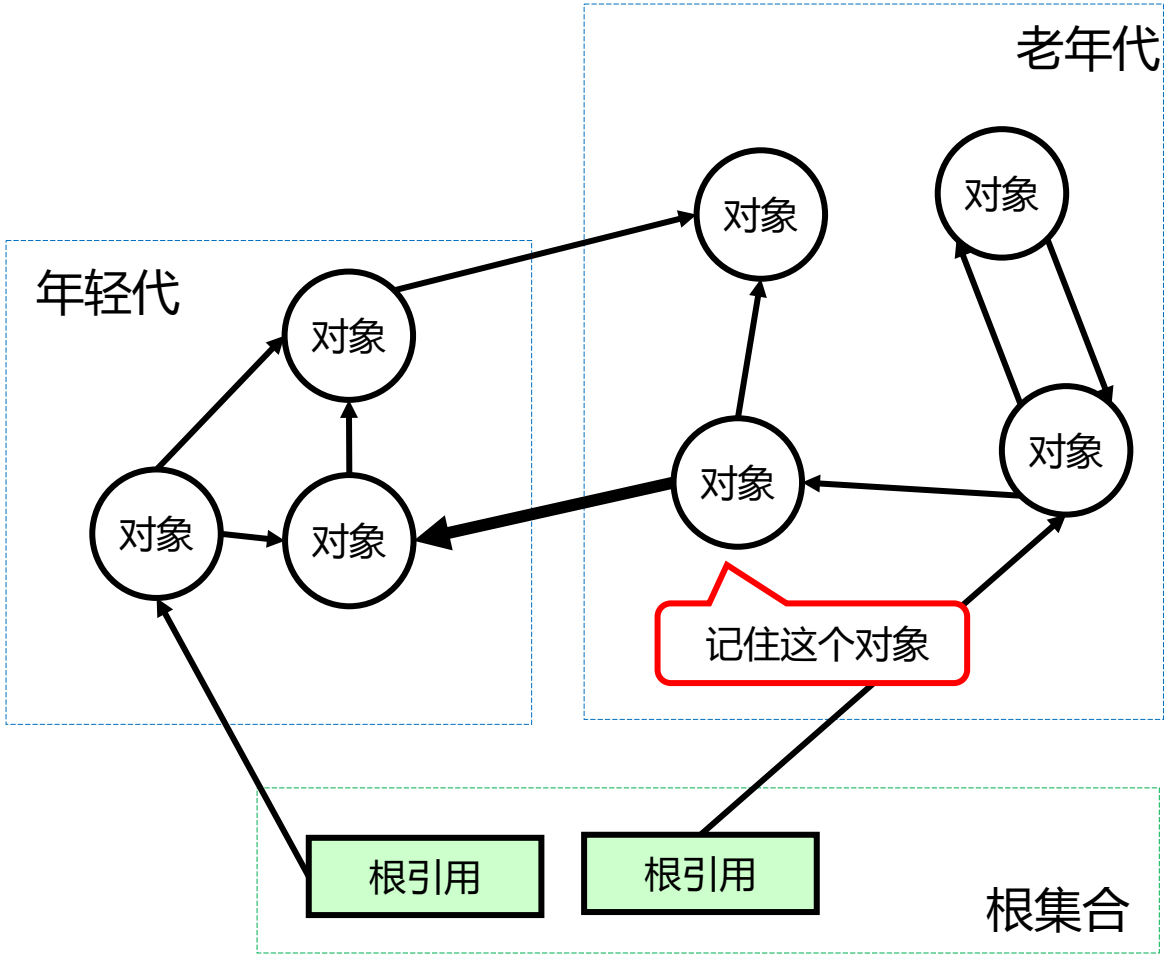
Barrier：

Mutator为了帮助GC，而在分配对象、访问对象时执行的操作

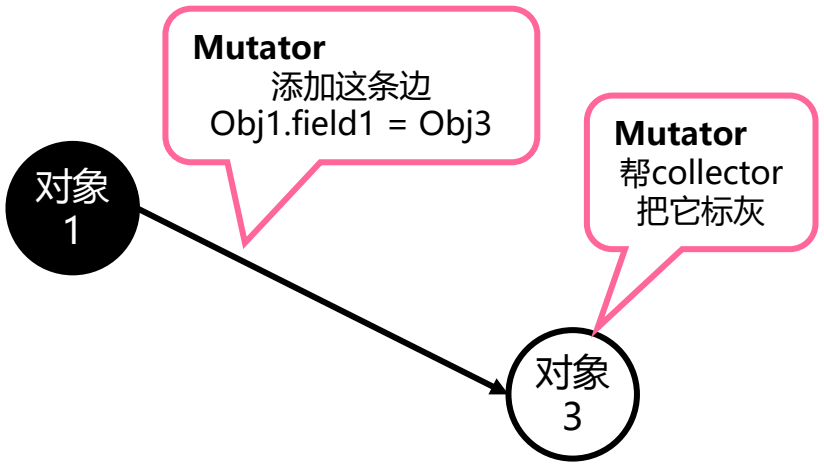
Barrier复习1：分代GC

- 记住有哪些**从老年代到年轻代**的引用
- 被老对象指向的年轻对象，认为是活的

```
Address storeObjectField(
        Address object,
        Offset field,
        Address target) {
  object[field] = target;
  if (isOld(object) && isYoung(target)) {
    remSet.add(object);
  }
}
```

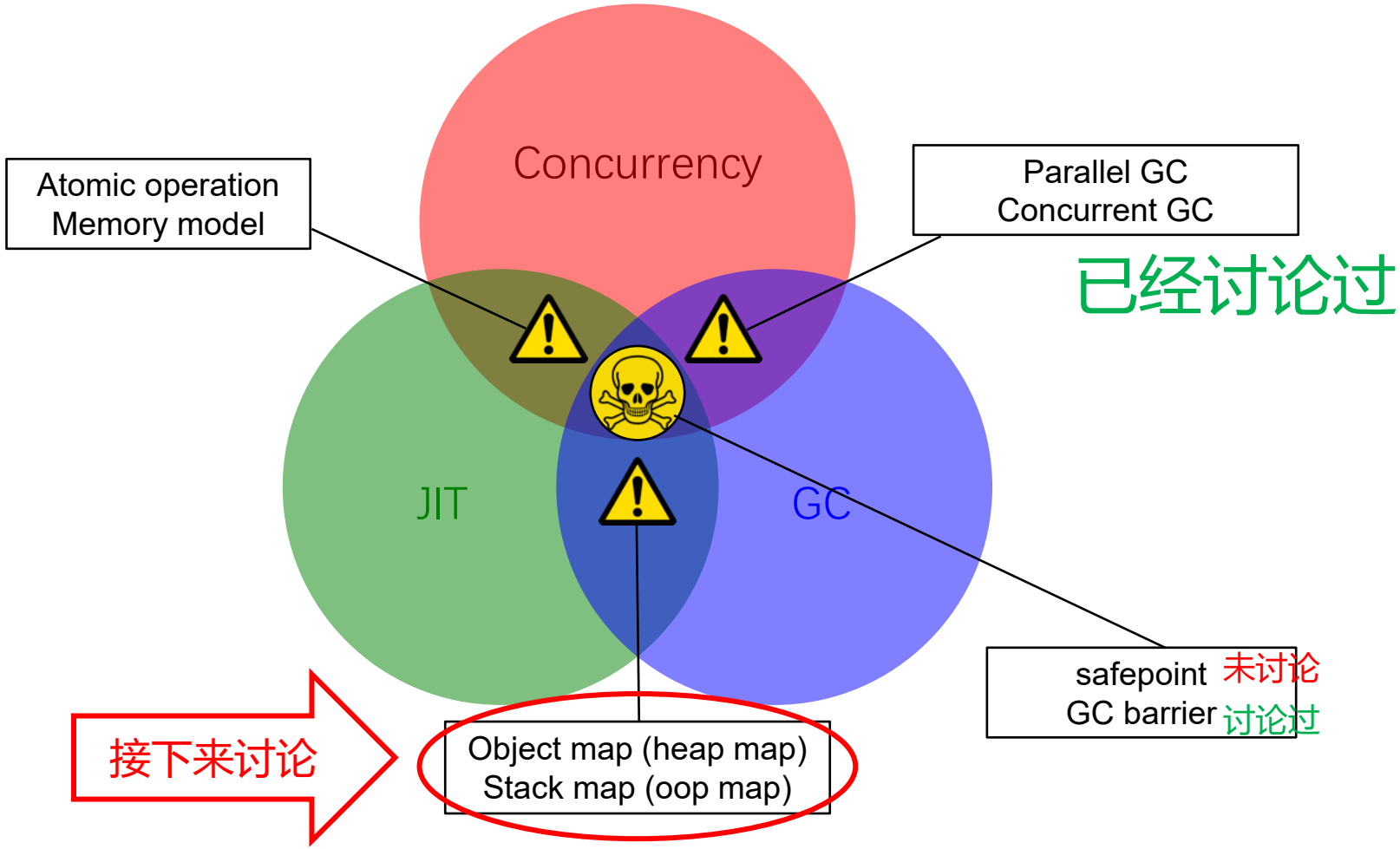


Barrier复习2：帮助GC标记对象




```
Address storeObjectField(
      Address object,
      Offset field,
      Address target) {
  object[field] = target;
  if (object.color == BLACK &&
      target.color == WRITE) {
    target.color = GREY;
  }
}
```

编程语言的实现中，编译器与垃圾回收是紧耦合的



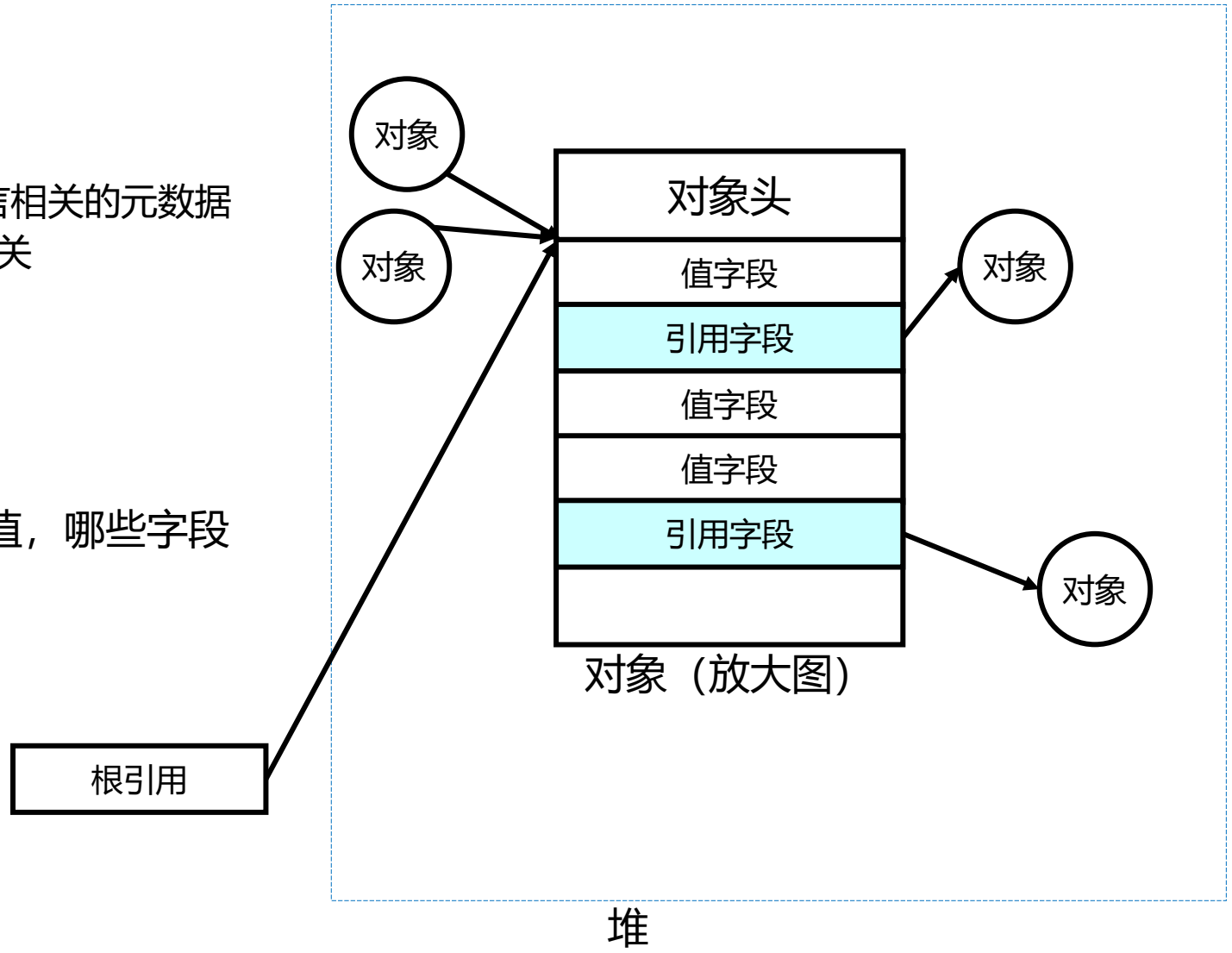
来源：Steve Blackburn. Micro Virtual Machines. In PLISS’17. https://youtu.be/T2WViuD5GrQ

对象布局

制定对象布局是编译器的责任

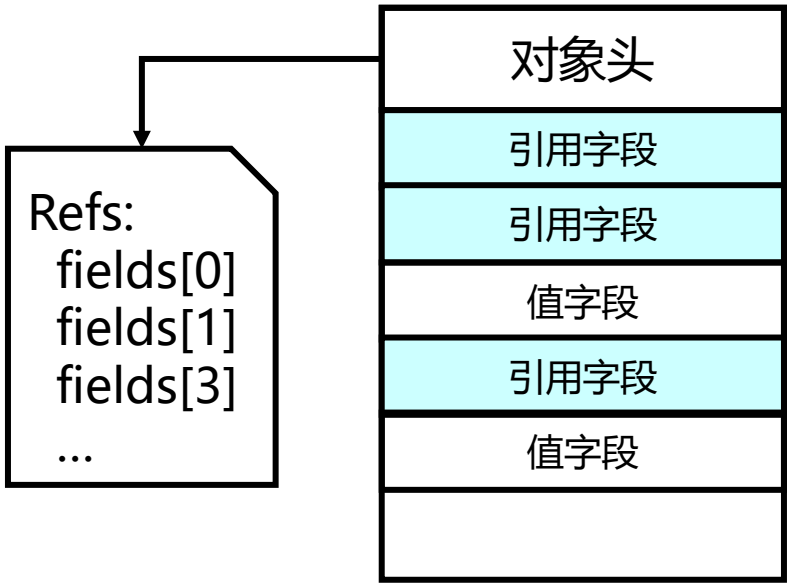
对象布局（object layout)

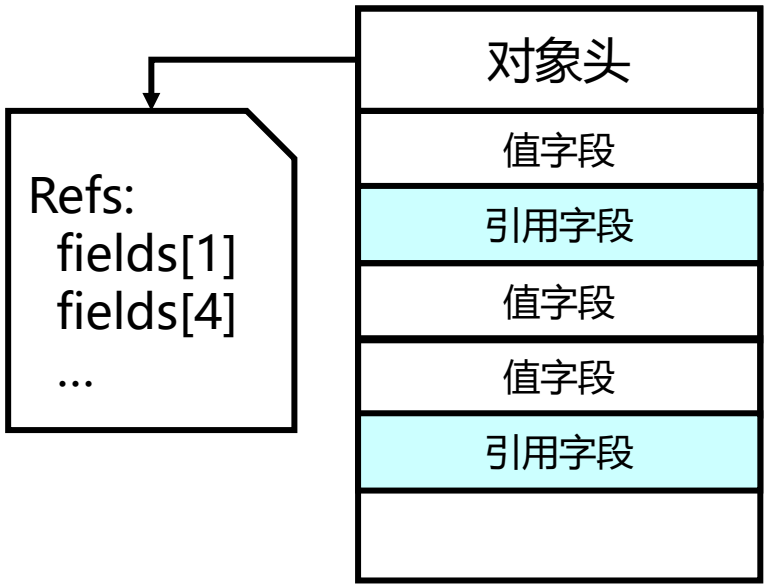
- 对象头（header)
  - 有和GC相关的元数据，也有和语言相关的元数据
  - 可有可无，和具体语言、虚拟机有关
    - 元数据可以集中放在一块特定区域
- 字段（field，也叫“域”）
  - 值字段
  - 引用字段
- 运行时有能力识别对象哪些字段是值，哪些字段是引用。



Object map

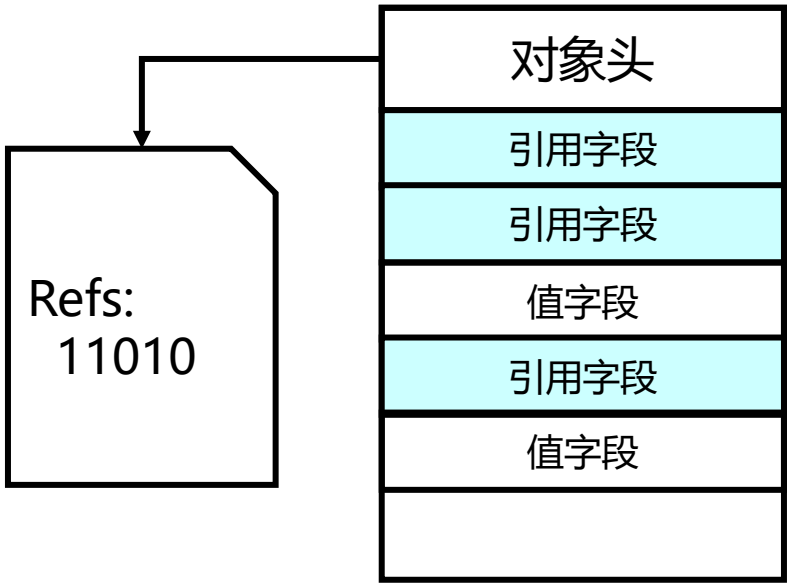
- 描述对象的哪些部分是引用的数据结构
- 可以放在类型信息里
  - 即Type Information Block（TIB）

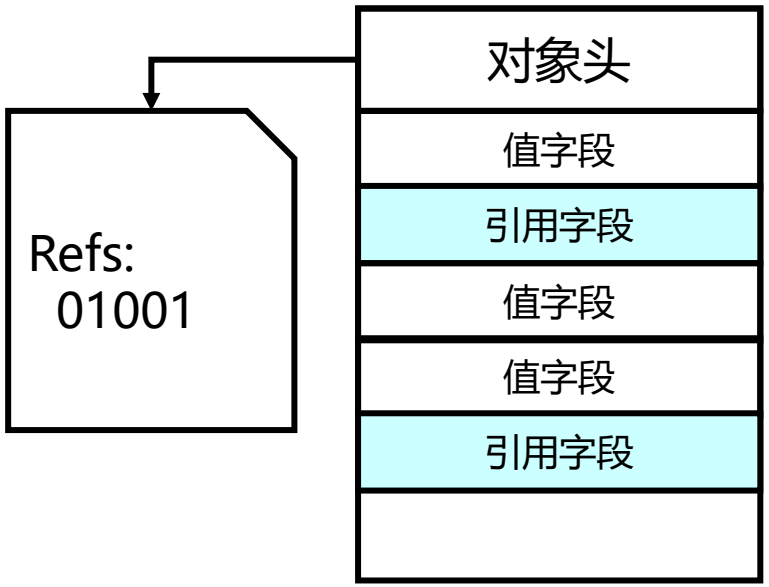




Object map

- 描述对象的哪些部分是引用的数据结构
- 可以在TIB里用bitmap表示



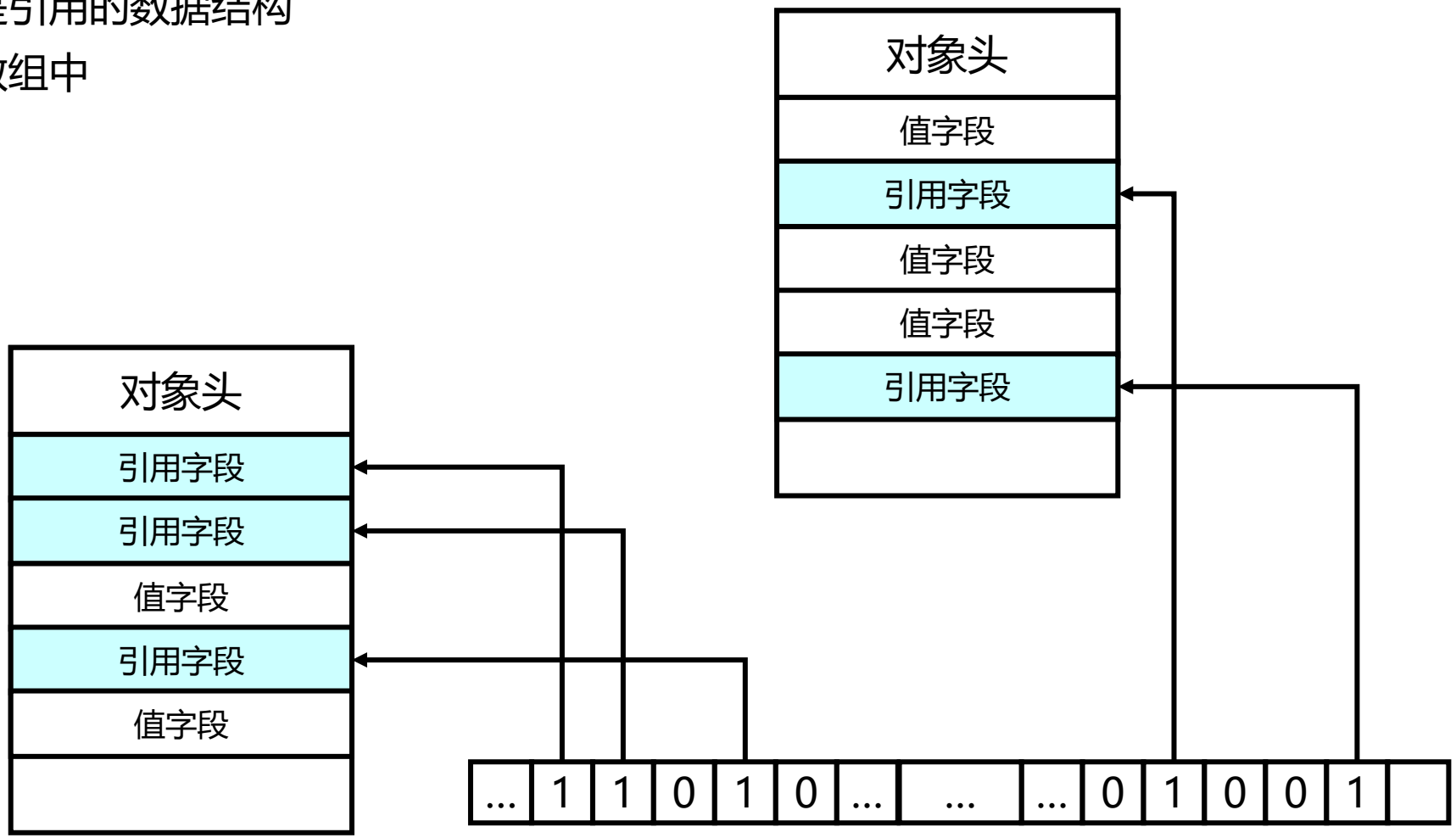


Object map

- 描述对象的哪些部分是引用的数据结构
- 可以用bitmap放在对象头里表示

Object map

- 描述对象的哪些部分是引用的数据结构
- 可以放在一个全局的数组中

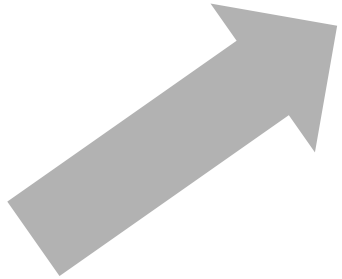


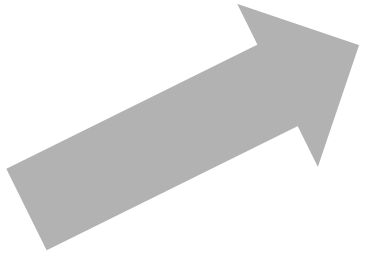
生成object map是编译器的责任

- 因为编译器、虚拟机：
  - 理解编程语言（或字节码）
  - 负责生成机器码
  - 负责执行程序
  - 负责访问对象
- 必须了解对象布局

```
class Foo {
    int a;
    Object b;
    char c;
    long d;
    Object e;
}
```

```
class Bar {
    Object a;
    Object b;
    double c;
    Object d;
    short e;
}
```

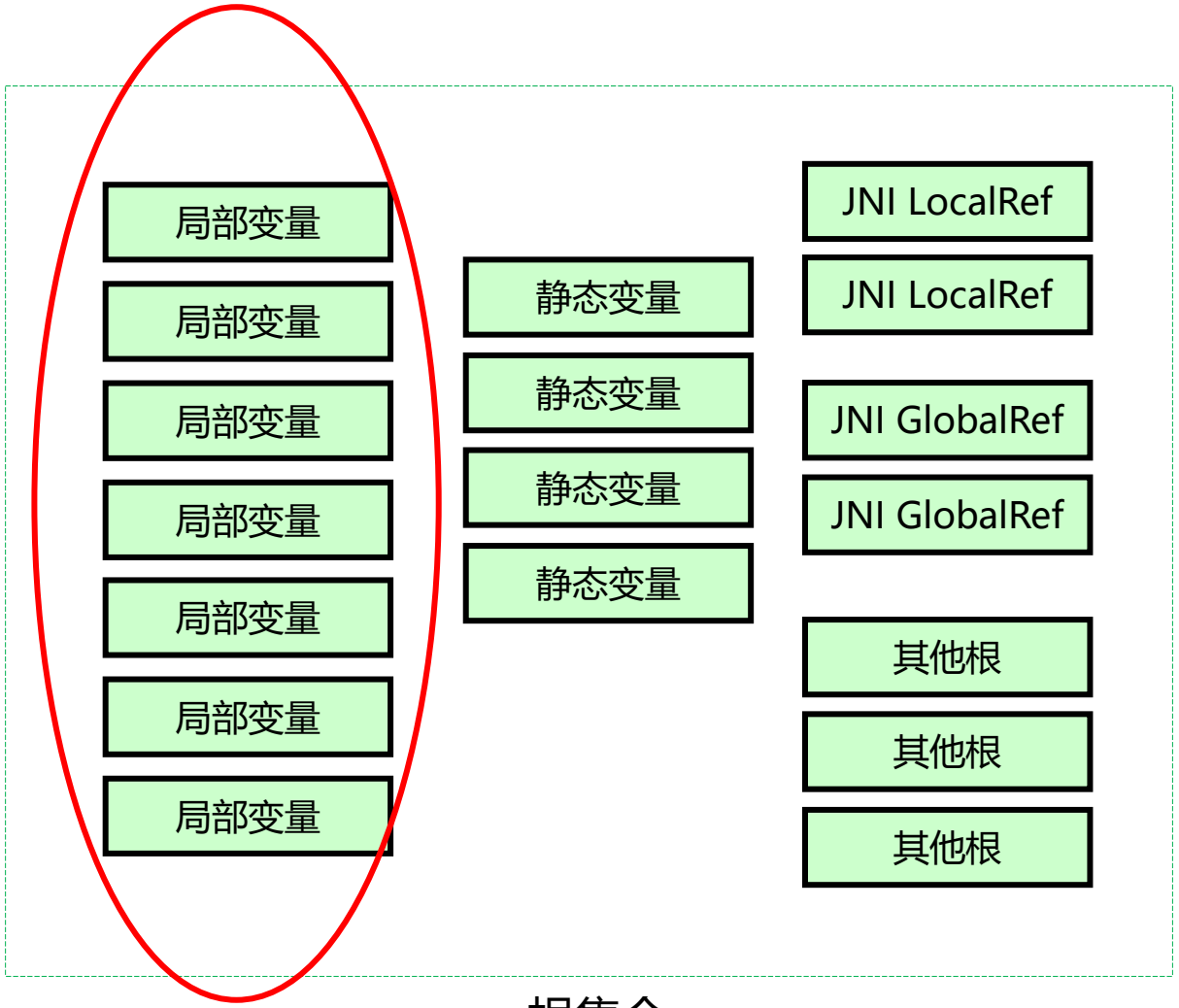




Stack Map

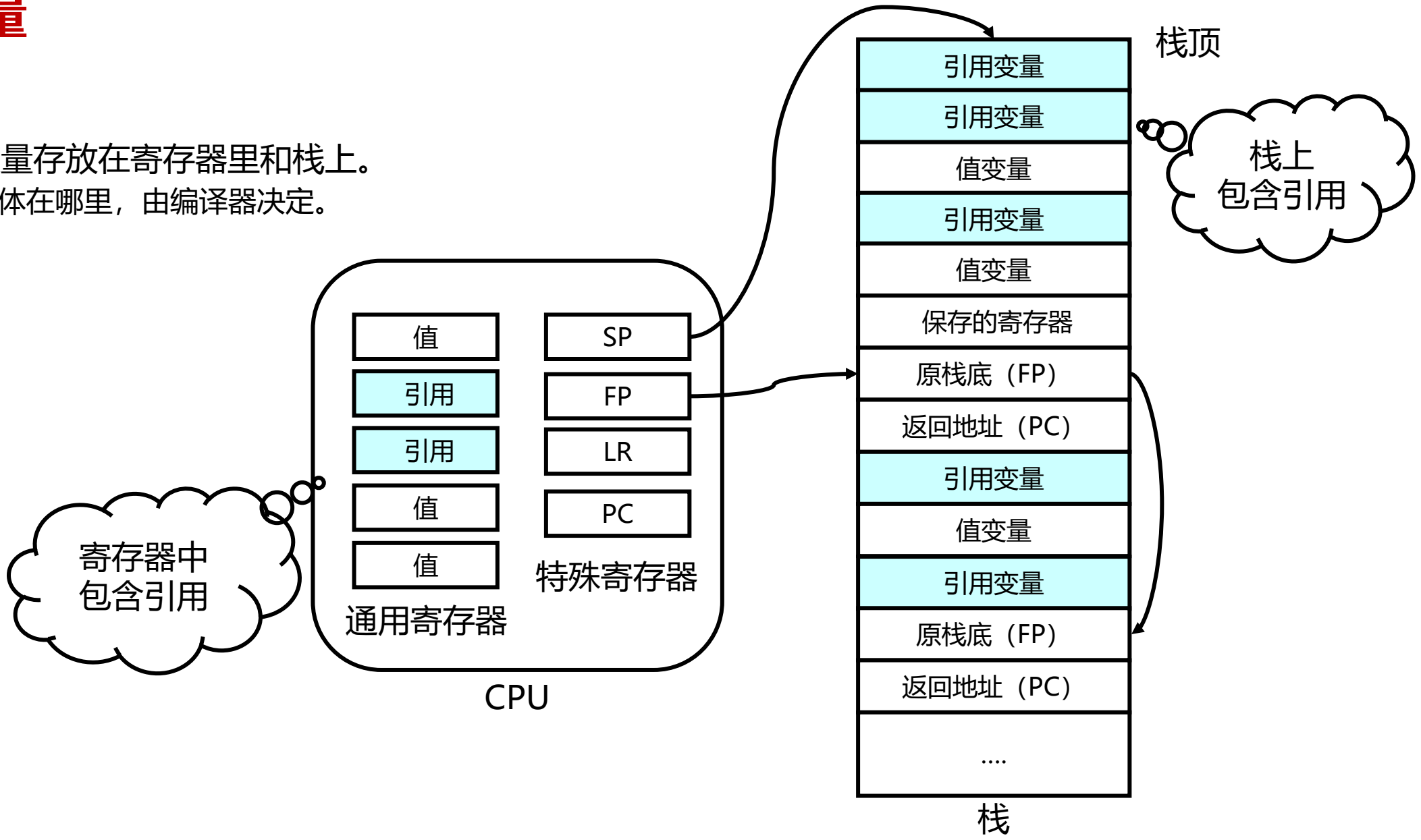
根集合（root set）

- 根集合中的引用
  - 可以被应用程序直接访问
  - 因此根指向的对象都是活的
- 具体包括
  - **局部变量**
  - **静态（全局）变量**
  - 被外部接口保留的
    - 例如JNI的LocalRef等
  - 其他根
    - 由语言、虚拟机、运行环境定义



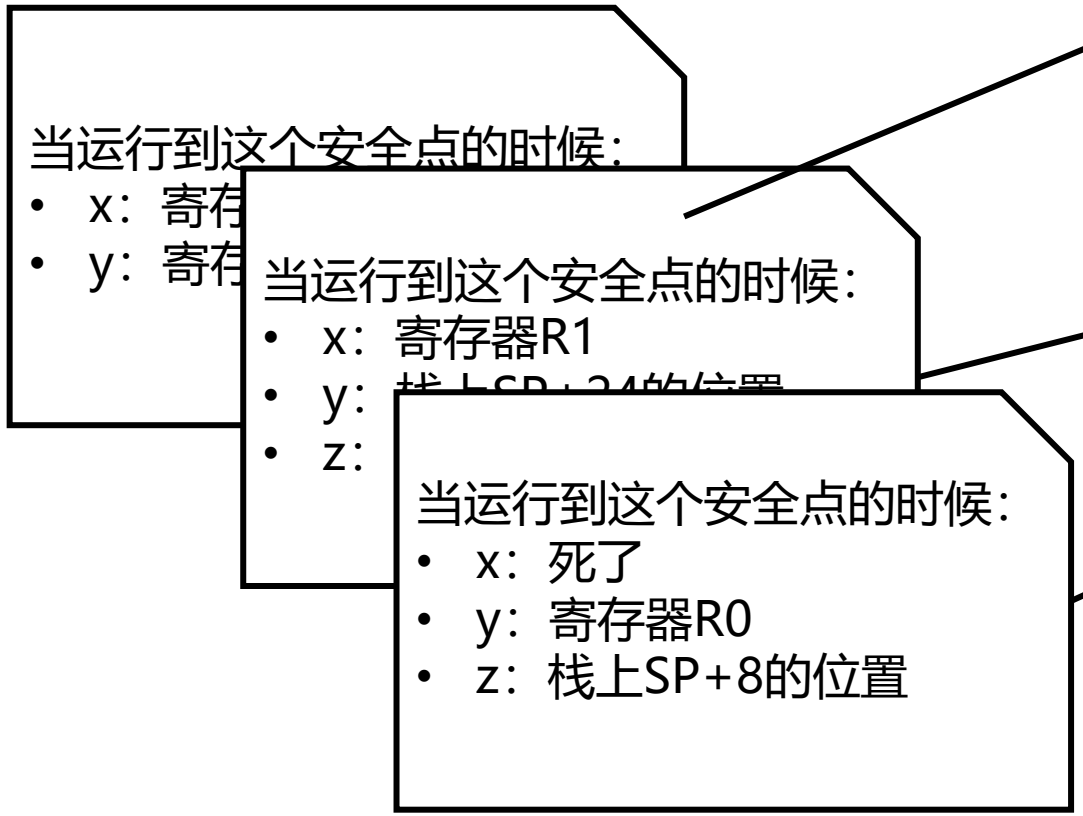
局部变量

- 局部变量存放在寄存器里和栈上。
  - 具体在哪里，由编译器决定。



编译器生成stack map

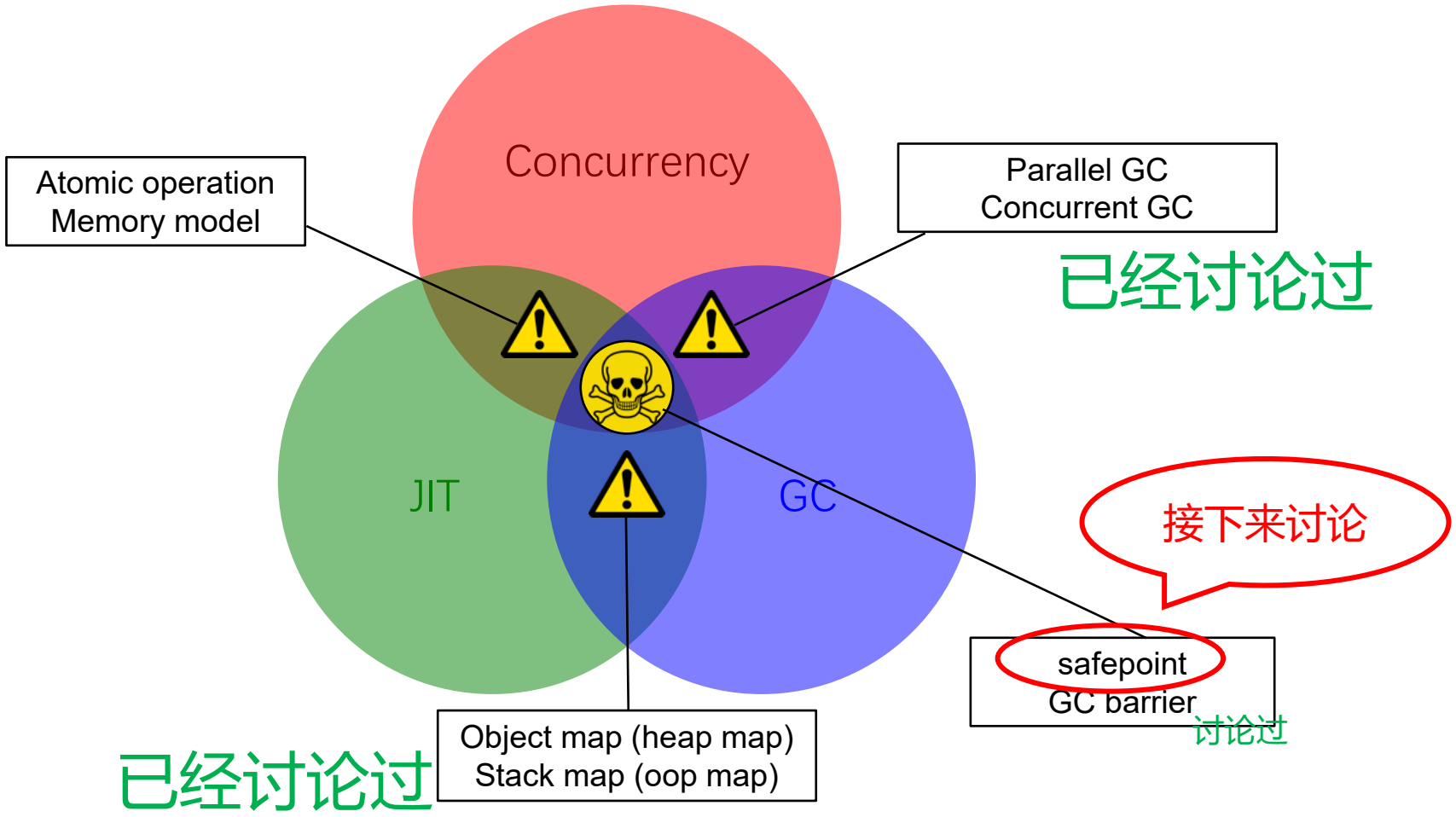
- Stack map（OpenJDK中叫OOP map）
  - 由编译器生成（编译器知道每个寄存器的用途）
  - 告诉运行时：当代码运行到**某位置**时，**哪些寄存器**里有引用
- 举例：




```
public static void foo() {
    Object x = …;
    Object y = …;

    while (…) {
        x = …;
        <自动生成的GC安全点>;
    }

    Object z = …;
    …

    bar(); // <函数调用点（安全）>

    …

    <自动生成的GC安全点>;
    return;
}
```

safepoint

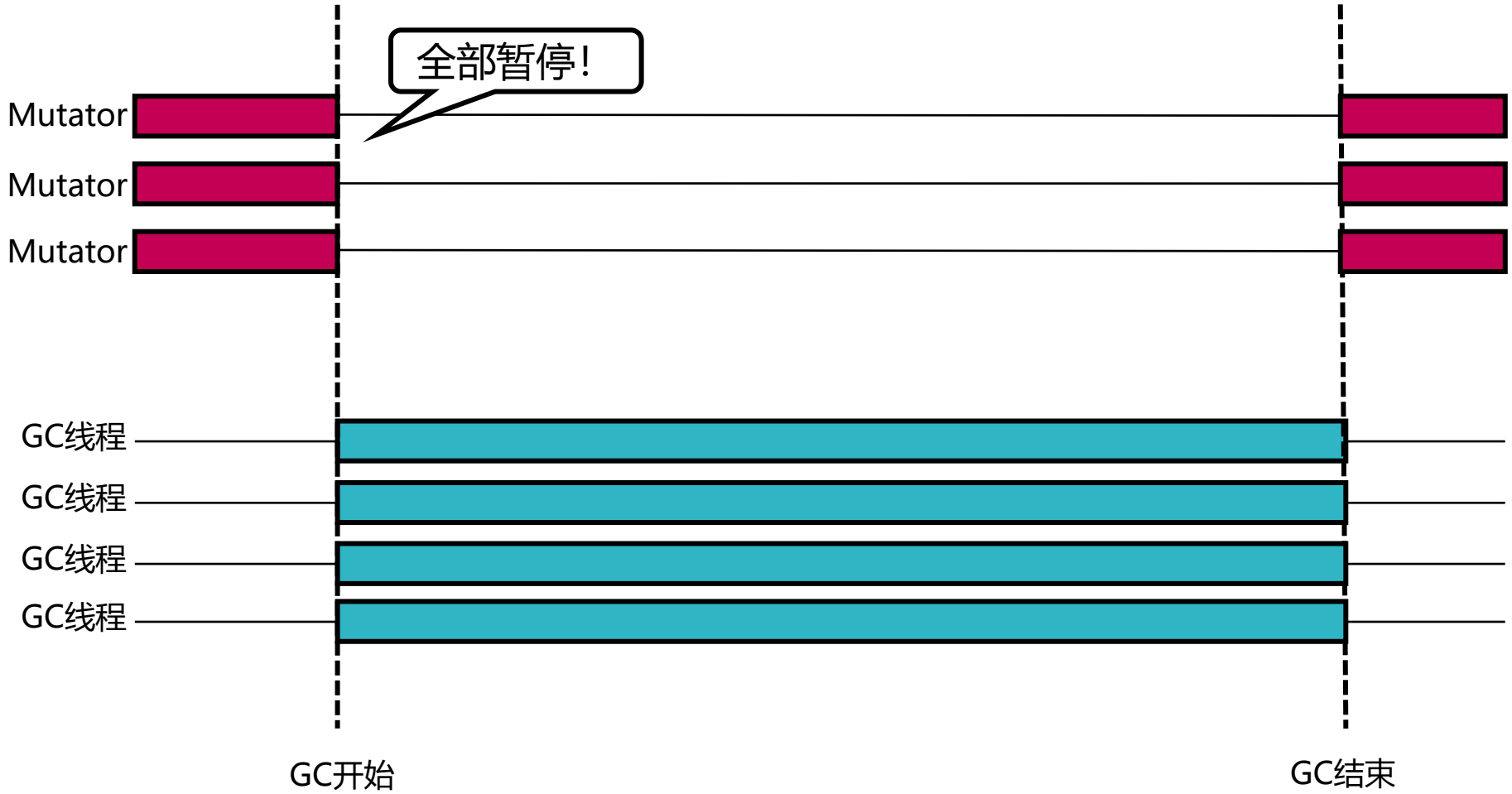
saftpoint



来源：Steve Blackburn. Micro Virtual Machines. In PLISS’ 17. https://youtu.be/T2WViuD5GrQ

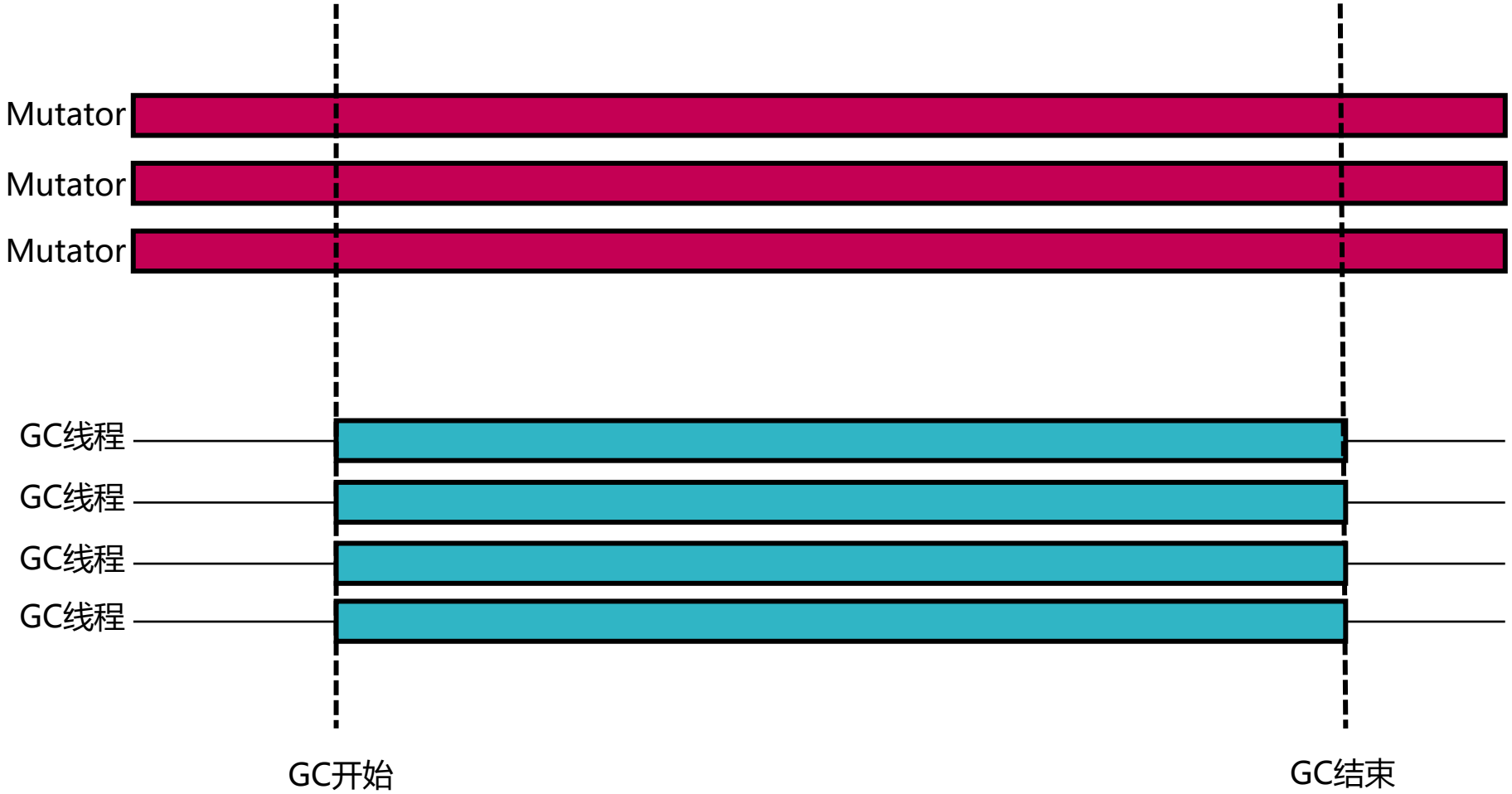
GC线程与Mutator线程的握手

- Stop-the-world GC是这样的



GC线程与Mutator线程的握手

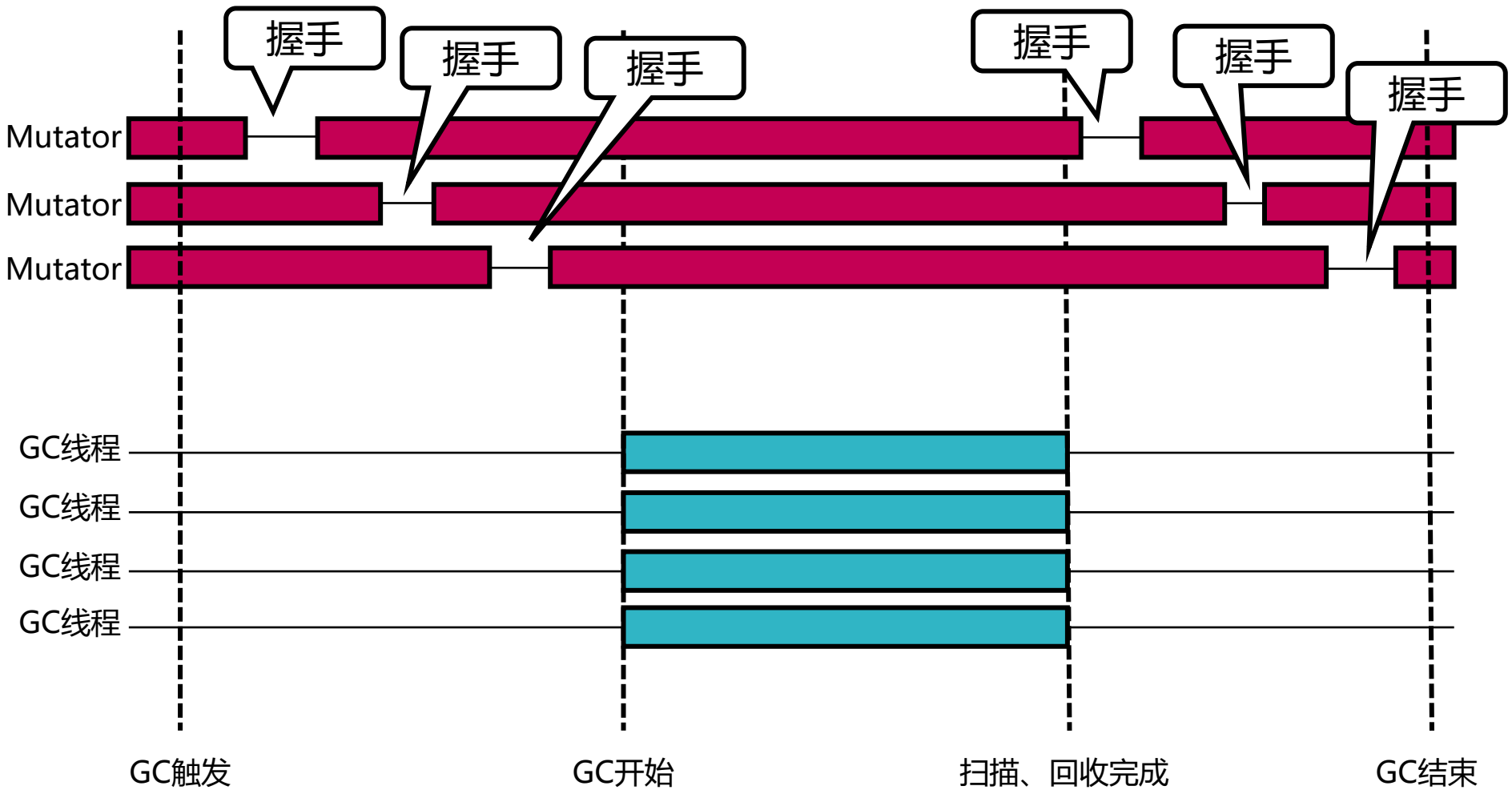
- 我们以为concurrent GC是这样的



GC线程与Mutator线程的握手

注：握手时间很短暂，只需要1ms以下

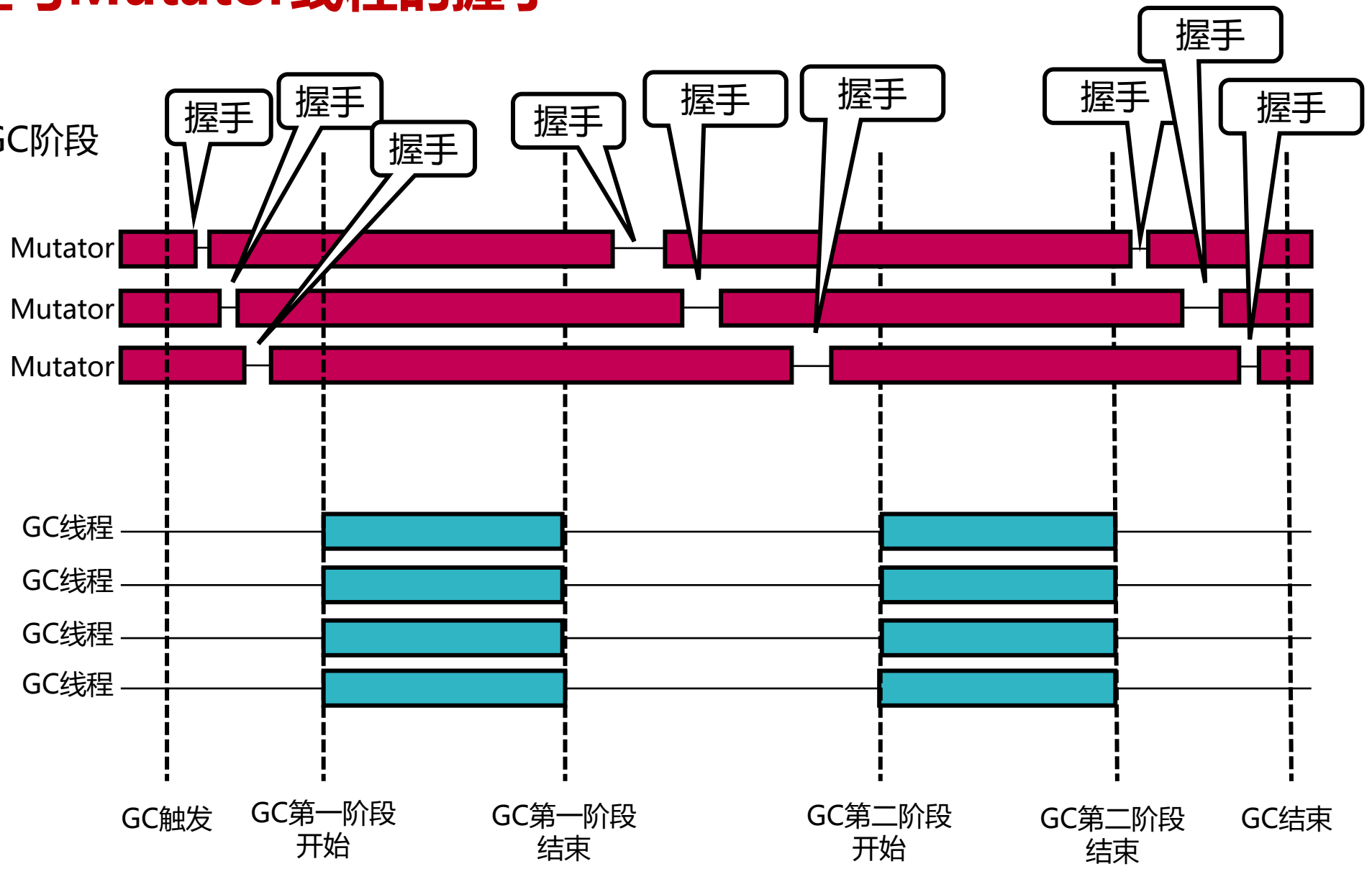
- 实际上是这样的



GC线程与Mutator线程的握手

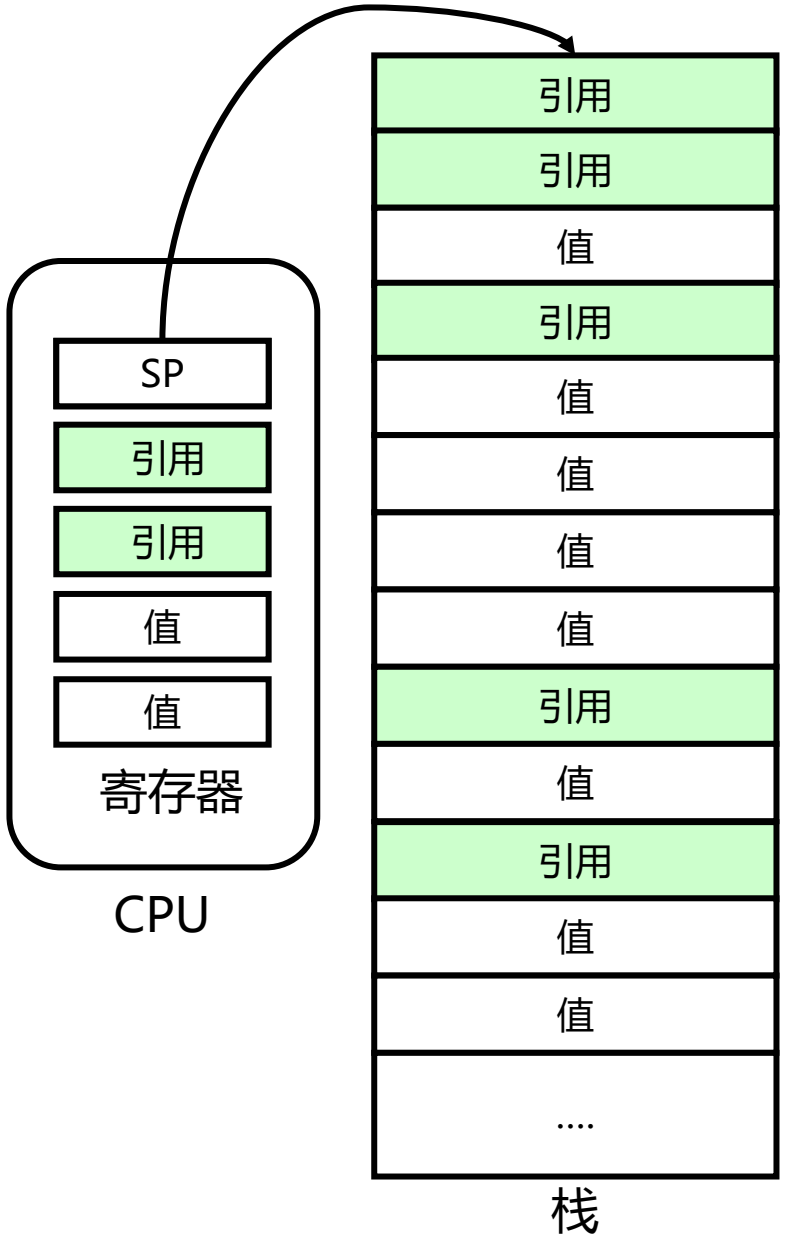
注：握手相比STW耗时更短

- 多个GC阶段



握手做什么?

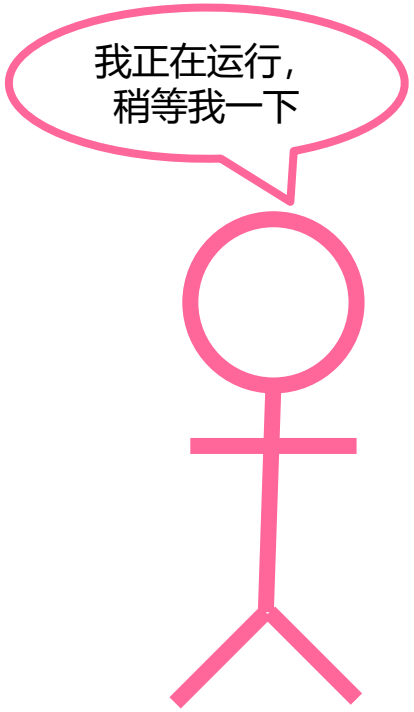
- 扫描栈（根集合）
- 更换barrier（但大多数算法barrier无法更换）
- Mutator的本地buffer传递给GC线程。
- 仅仅是让mutator与collector达成共识。
- ……



形象地说，就是

GC算法请求Mutator执行到安全点时暂停一下，
做一些必要的事情

GC线程



mutator线程

应用线程好比汽车，
应用代码好比行车道，
安全点就是道路上的红绿灯

应用线程不能在任意位置停下来

线程可能处于不安全的状态

- 线程可能握着锁
  - 不能用UNIX signal。
    - 安全红线：不能在信号处理函数里阻塞
    - UNIX signal可能让线程在任何位置停下来
    - 甚至在printf执行了一半的地方
- 线程可能在做“对于GC来说原子的操作”
  - atomic with respect to GC
  - 无需加锁，但不能让GC看到做了一半
  - 例如：正在分配对象
    - 但对象的头还没有写好
  - 例如：在执行write barrier
    - 例如：写了field，却没有给目标对象标记成灰色

```
printf(“hello\n”); // 拿着stdout的锁
```

```
atomic new_object() {
    object = free_pointer
    free_pointer += OBJ_SIZE
    <☠被中断☠>
    object.header = TypeInfo
    object.color = BLACK
}
```

```
atomic write_barrier() {
    source.field = target
    <☠被中断☠>
    SetGrey(target)
}
```

应用线程只能在安全点停下来才是安全的

safepoint(yieldpoint)

- 到达safepoint时，Mutator***主动查询GC是否被触发***

- Mutator必须能够**及时**响应
  - 插入在循环的开头或回边
    - 确保大循环中能及时执行到一个safepoint
  - 插入在函数的开头或结尾
    - 确保大量递归调用时能及时执行到一个safepoint
  - （直线型代码一般不是问题）
    - （CPU一毫秒能执行几十万条指令）

循环回边

函数末尾

```
public static void foo() {
    Object x = …;
    Object y = …;

    while (…) {
        x = …;
        <自动插入的safepoint>;
    }

    Object z = …;
    …

    bar();

    …

    <自动插入的safepoint>;
    return;
}
```

safepoint如何高效地实现

safepoint的特点

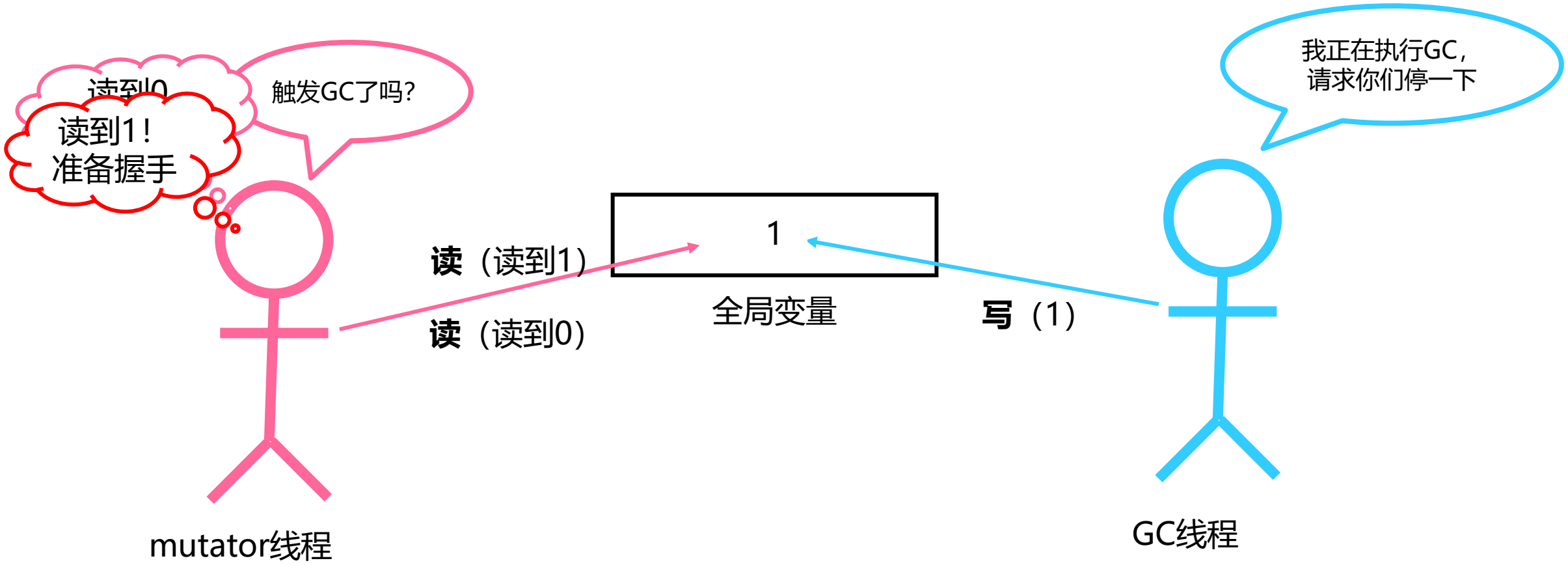
- 经常被执行，却很少真正被触发
- 一个Java程序每秒钟执行safepoint多达1亿次
  - 每10ns（约40个指令周期）执行一次
  - 每20000次执行（每20ms）中只有一次被触发
  - 约每1000000次执行中只有一次因GC而触发
    - 注：统计来自JikesRVM。JikesRVM使用safepoint做profiling和biased locking，触发频率远大于其他VM

```
for (long i = 0; i < 1000000000; i++)
{
    s = s + a;
    a = a + i;

    if (blahblah) {
       blahblah;
    }

    <自动插入的safepoint>;
}
```

参考文献： Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. **Stop and go: understanding yieldpoint behavior**. *In Proceedings of the 2015 International Symposium on Memory Management (ISMM '15). Association for Computing Machinery, New York, NY, USA, 70–80.* https://doi.org/10.1145/2754169.2754187

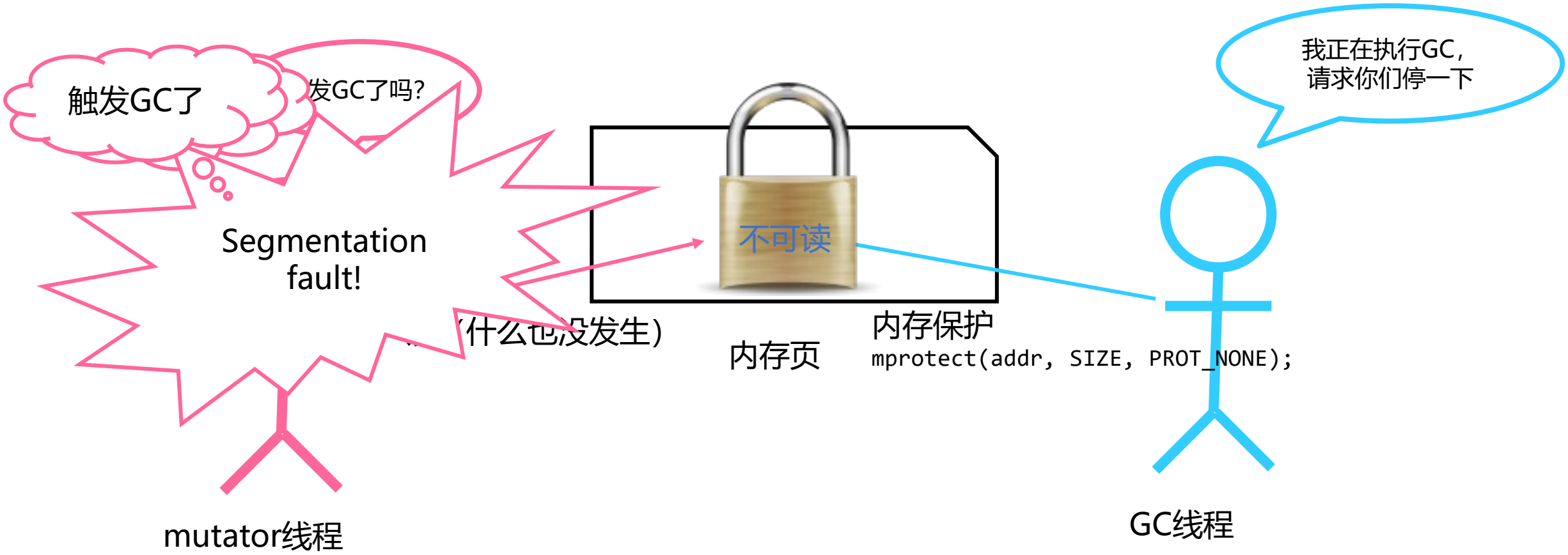
safepoint的实现（checking）

- 用一个全局变量表示是否已经触发GC



safepoint的实现（page protection，也叫trapping）

- 用内存保护机制造成segmentation fault



实现为高效的机器指令

Checking

```
// x86
.safepoint :
        cmp 0, [TLS_REG + offset]
        jne call_safepoint
.normal_code:
        ...
```

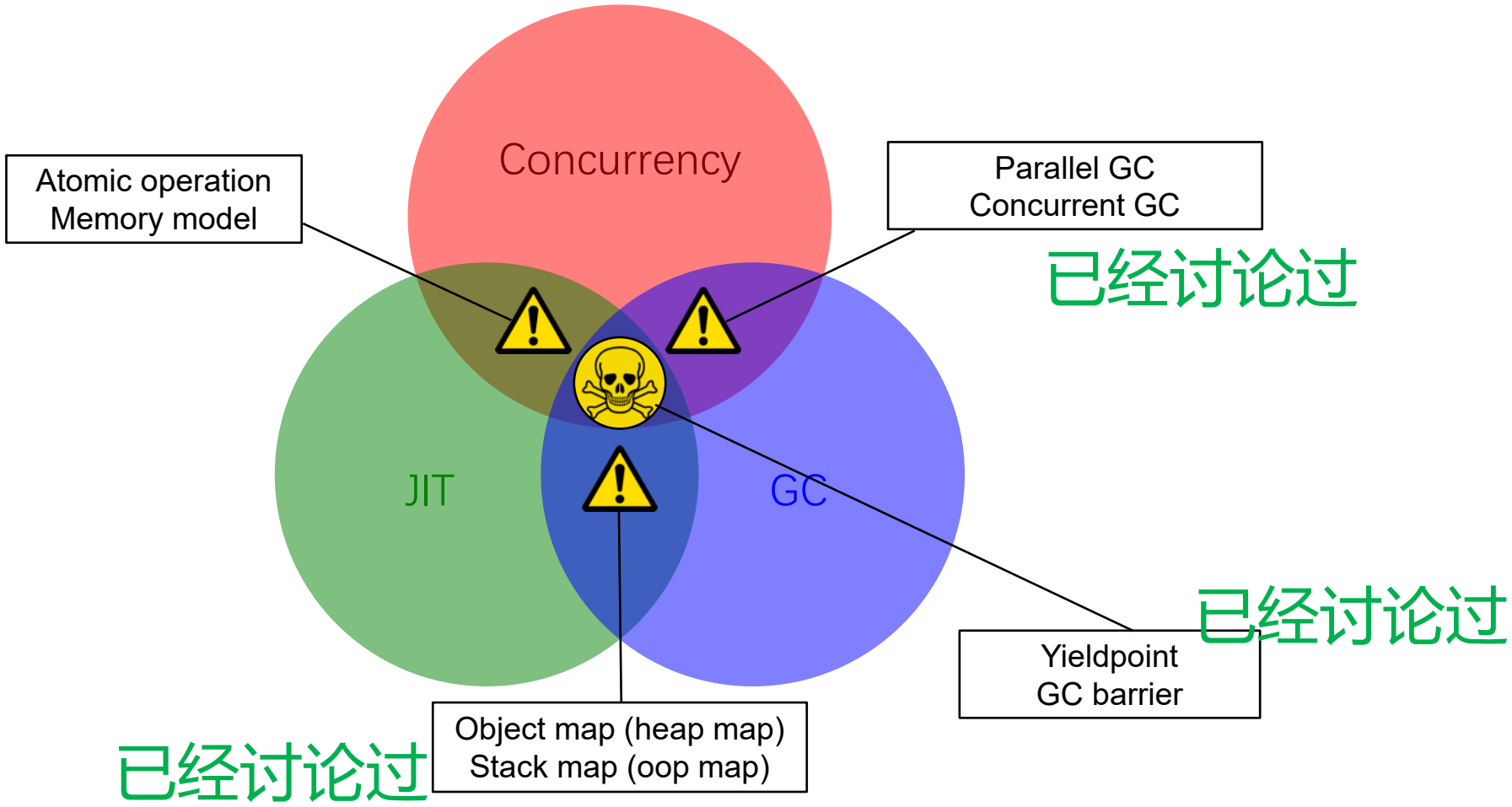
```
// aarch64
.safepoint :
        ldr w0, [TLS_REG, #offset]
        cbnz w0, call_safepoint
.normal_code:
        ...
```

Trapping

```
// x86
.safepoint :
        test 0, [TLS_REG + offset]
.normal_code:
        ...
```

```
// aarch64
.safepoint :
        ldr wzr, [TLS_REG, #offset]
.normal_code:
        ...
```

That’s all for basics



来源：Steve Blackburn. Micro Virtual Machines. In PLISS’ 17. https://youtu.be/T2WViuD5GrQ

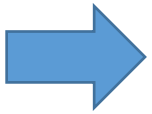
编译器提供的GC机制总结

- 高性能的GC实现，是靠运行时和编译器共同努力达成的。
- 编译器要为GC提供：
  - 在应用代码中插入高效的Barrier代码，用于mutator正确访问堆内存
  - 生成Object Map，以找到堆上的对象里的引用
  - 生成Stack Map，以找到栈上局部变量里的引用
  - 在应用代码中插入高效的safepoint，用于同步mutator与GC线程的状态

仓颉语言的全并发GC

仓颉并发GC的主要设计目标和技术特征

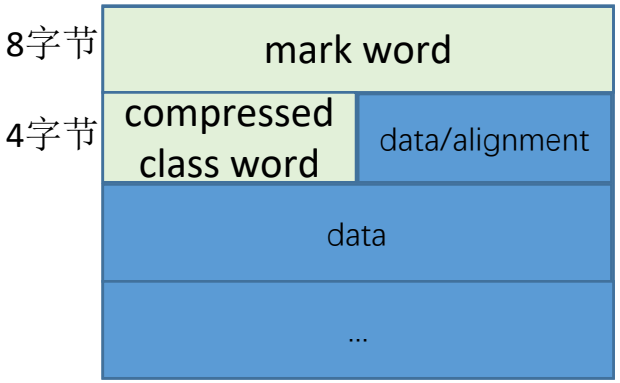
- 仓颉应用的高性能
- 极低时延
- 内存占用少



- 内存分配快/屏障开销低
- 完全并发
- 仓颉对象基础开销低/内存整理

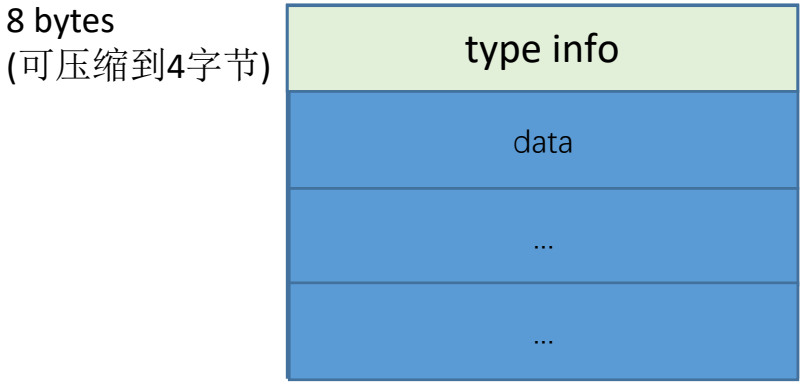
仓颉对象布局

Java对象布局
JDK21 hotspot 64位平台

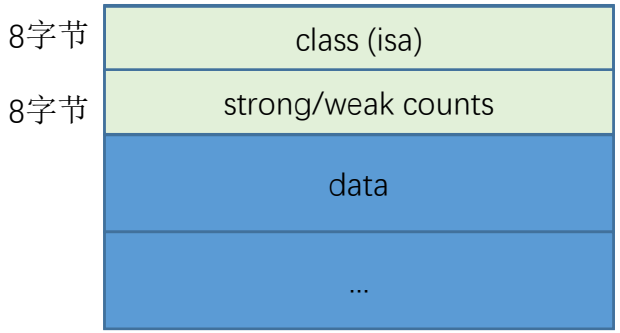


Compact Object Headers https://openjdk.org/jeps/450
8-byte header + separate forwarding table

仓颉对象布局
64位平台



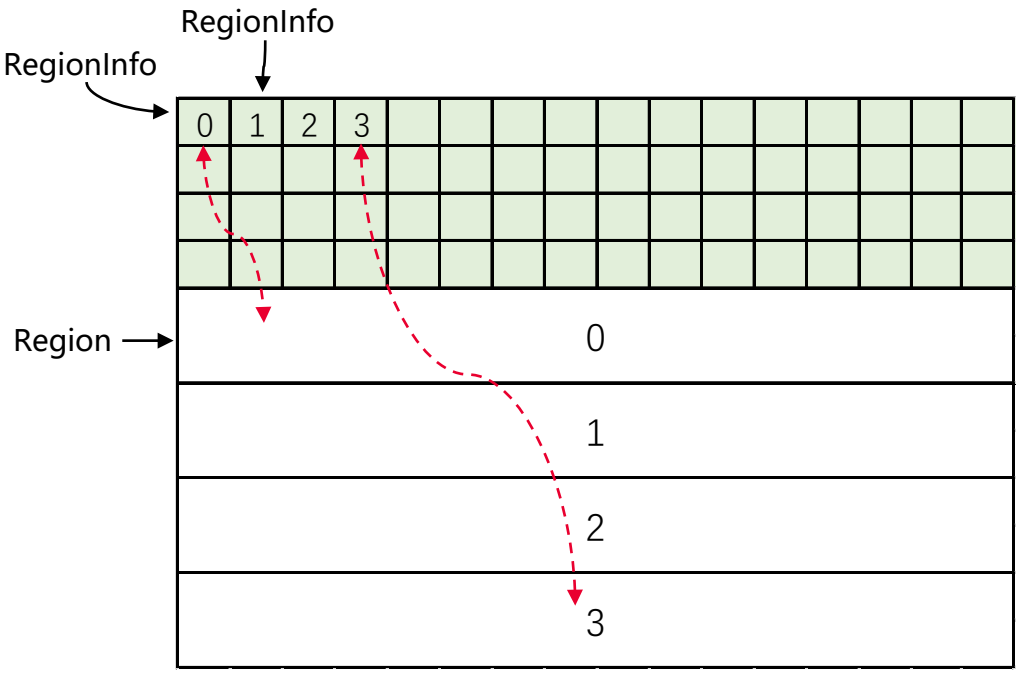
Swift/Objective C

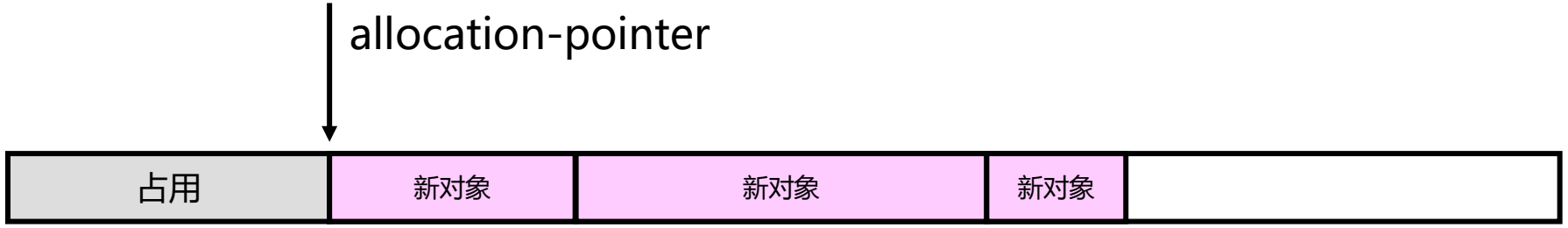


仓颉对象的基础内存开销减少近10%

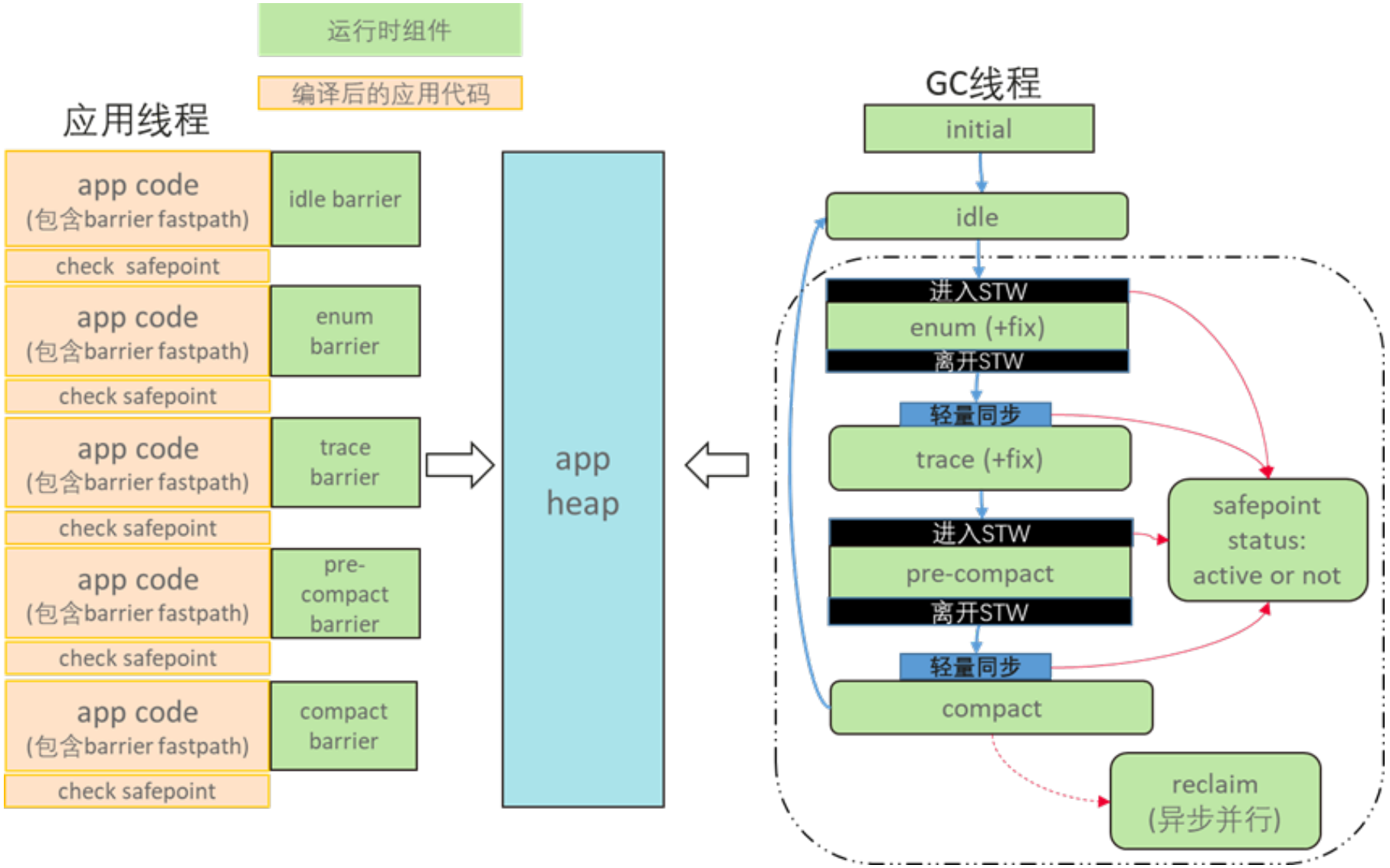
仓颉堆内存布局

- region元数据地址和region内存地址通过线性关系对应
- 对象分配速度非常快 **(约10条机器指令)**
  - ✓ 在region内通过指针跳跃分配小对象





典型的内存整理GC



ennum: 收集GC根集合

trace: 根据GC根集合及对象引用关系
递归地标记活对象

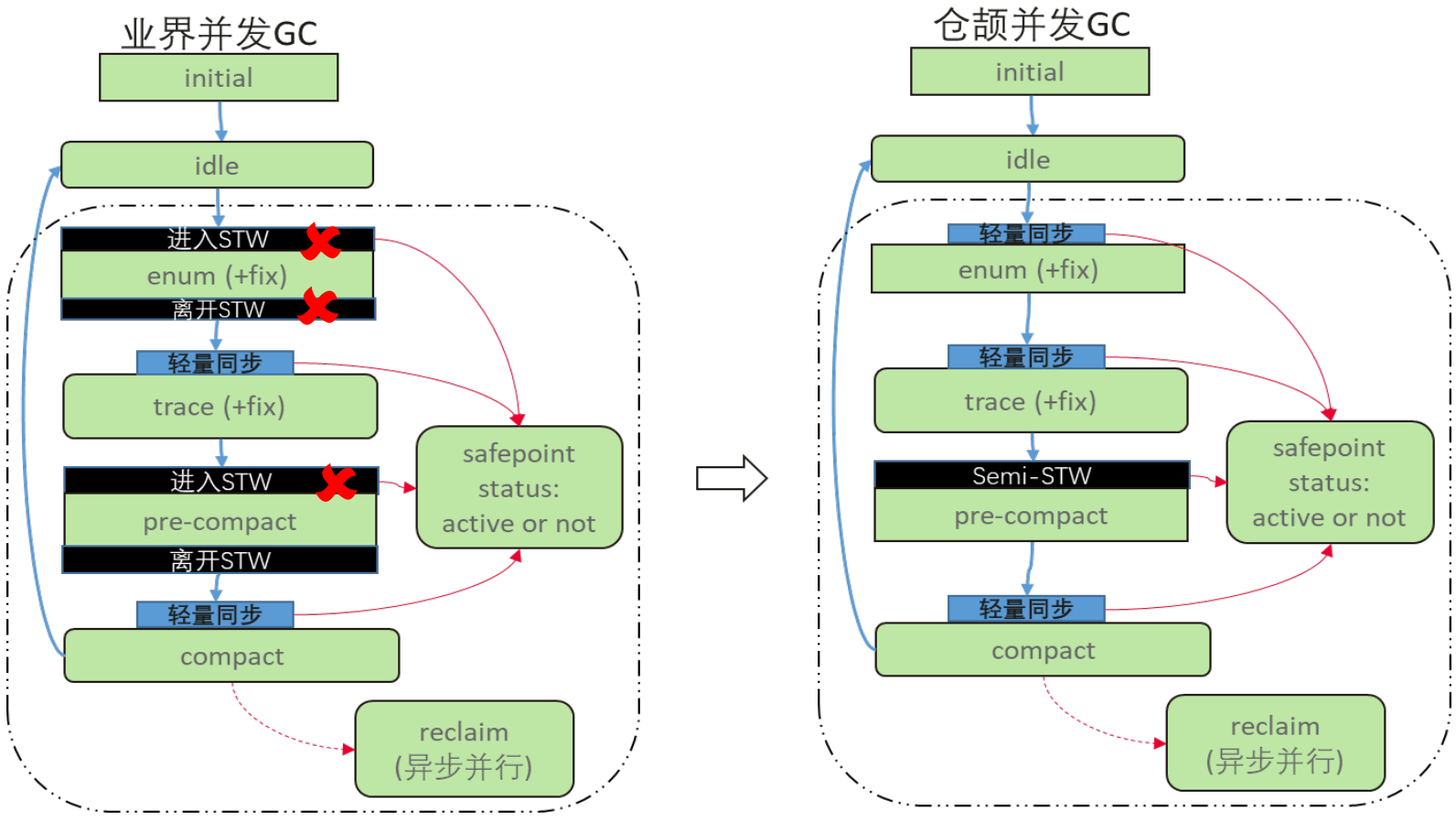
pre-compact: 搬移栈上根引用指向的仓颉对象，并把该引用更新为新的内存地址

compact: 搬移region里的活对象，
完成后可回收该region

fix: 修复堆中的旧引用

idle: GC未开始/已完成

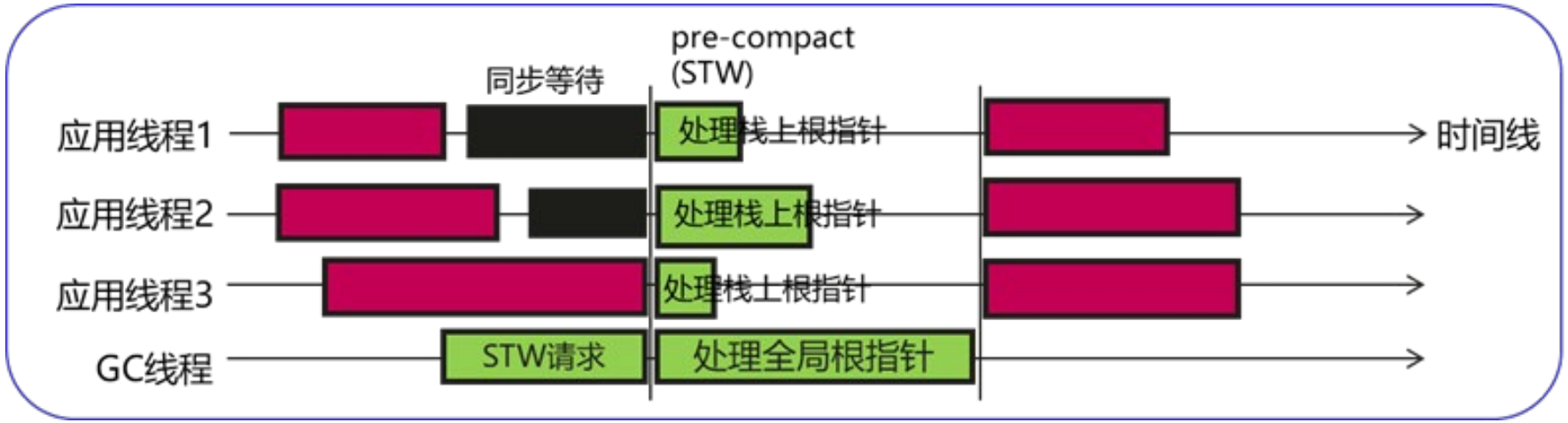
仓颉全并发内存整理GC



通过安全点和读屏障
实现轻量GC同步，避免在
STW中执行GC任务，
可达成更低时延。

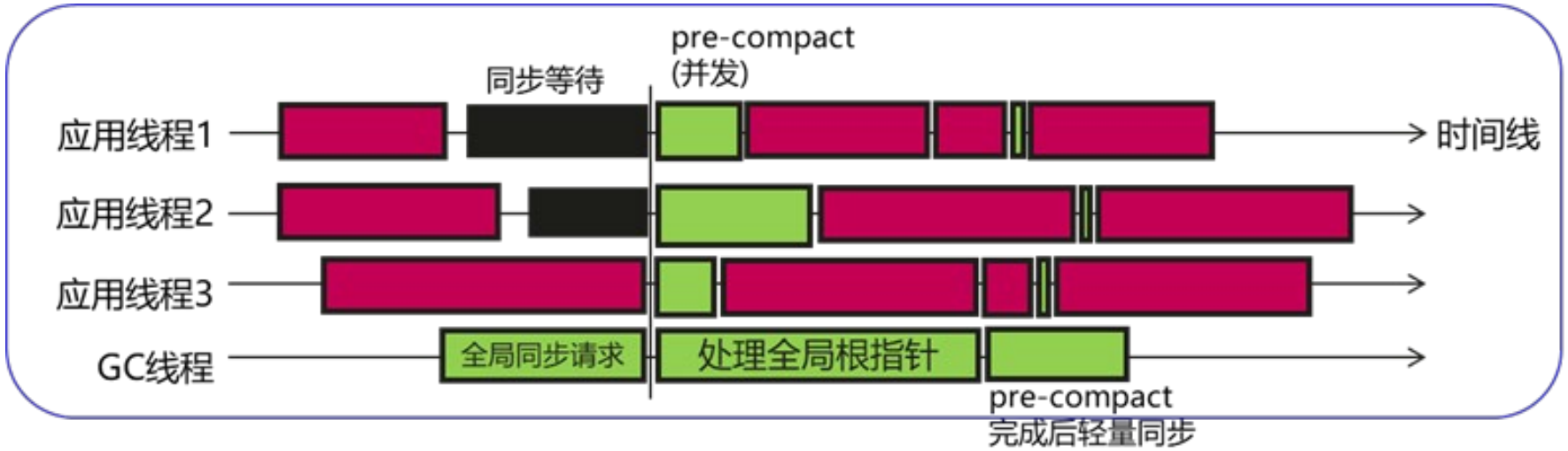
消除STW

应用线程的暂停时间 = 同步等待时间 + GC任务时间



STW方案：

1. GC任务在STW状态下进行
2. 应用的暂停时间**取决于同步任务耗时最长的线程**。

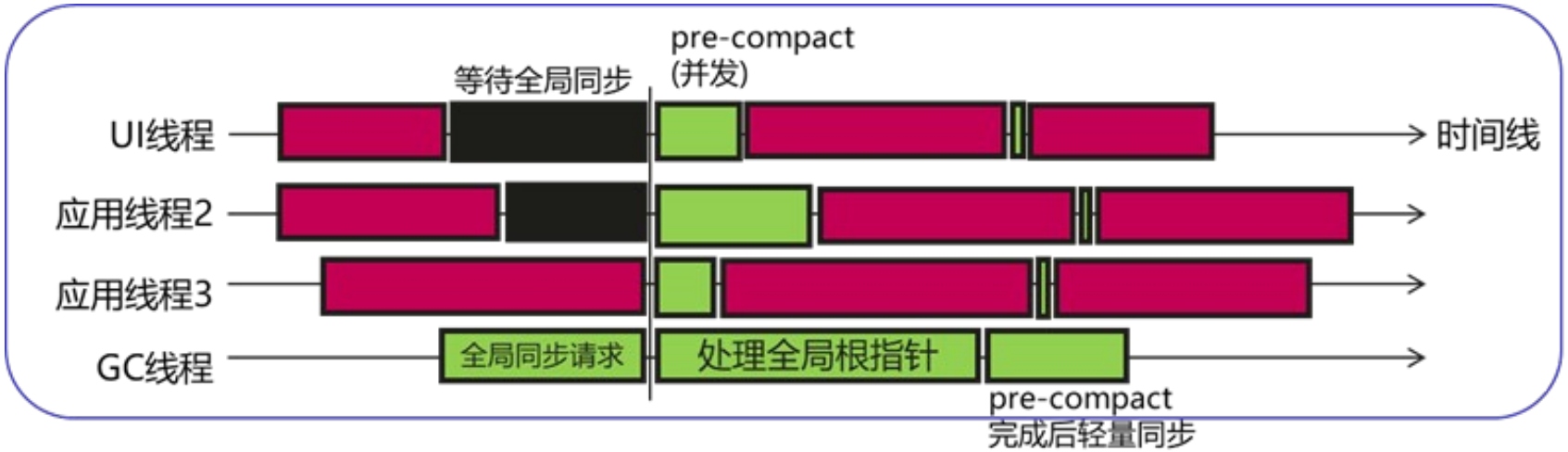


仓颉并发方案：
应用线程到达同步点后
即可恢复独立执行

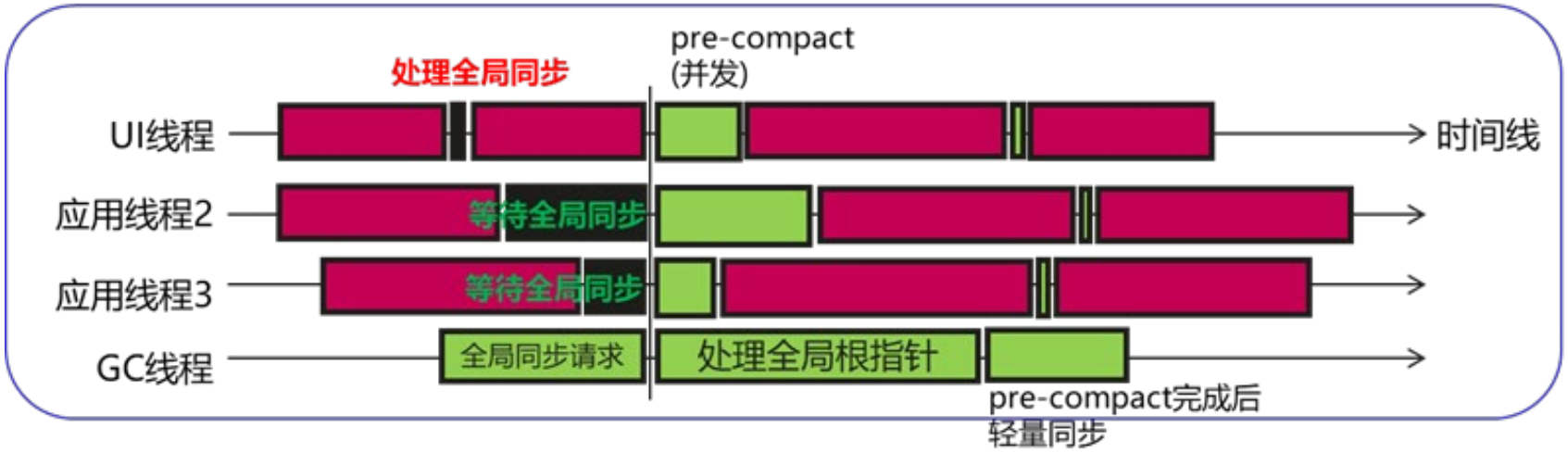
同步等待

GC任务

UI线程高优先调度优化暂停时间



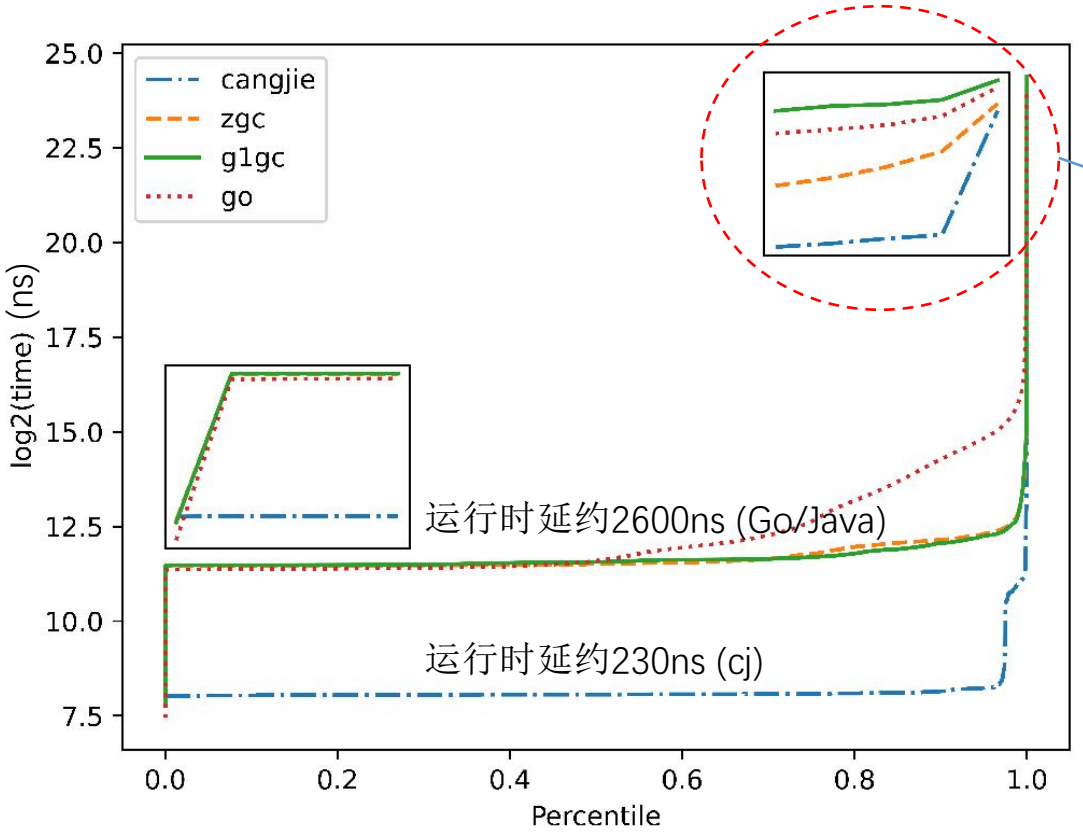
1. 低优先级应用线程进入同步等待状态暂停等待完成GC同步。
2. UI线程进入**同步等待**状态后“主动脱离”该状态(高优先调度)继续运行，缩短UI线程暂停时间。

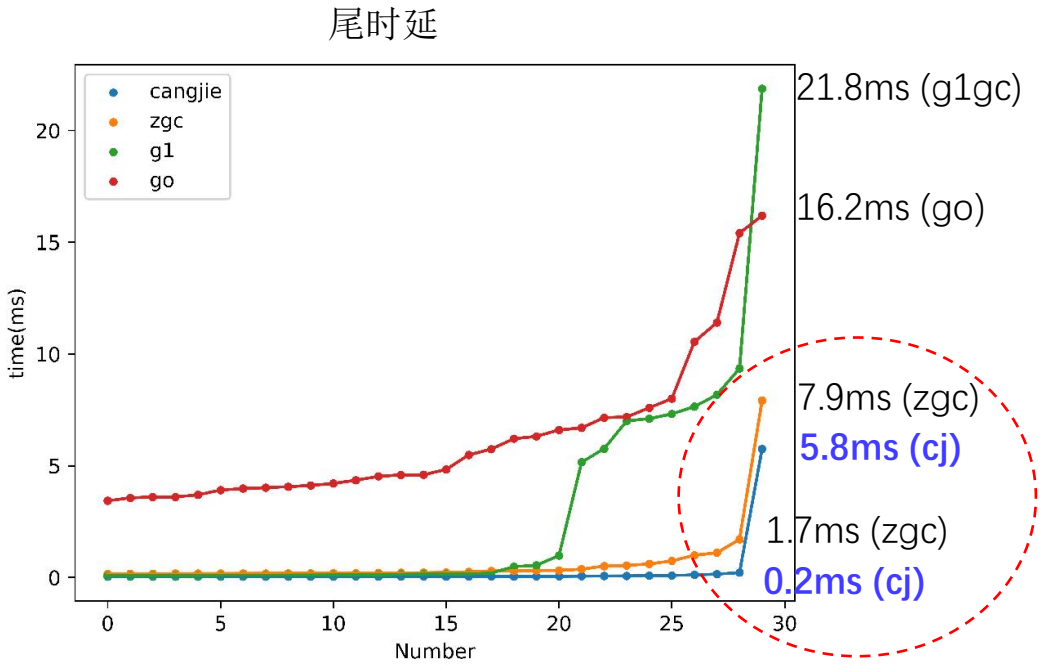


同步等待

GC任务

全并发GC达成更低时延

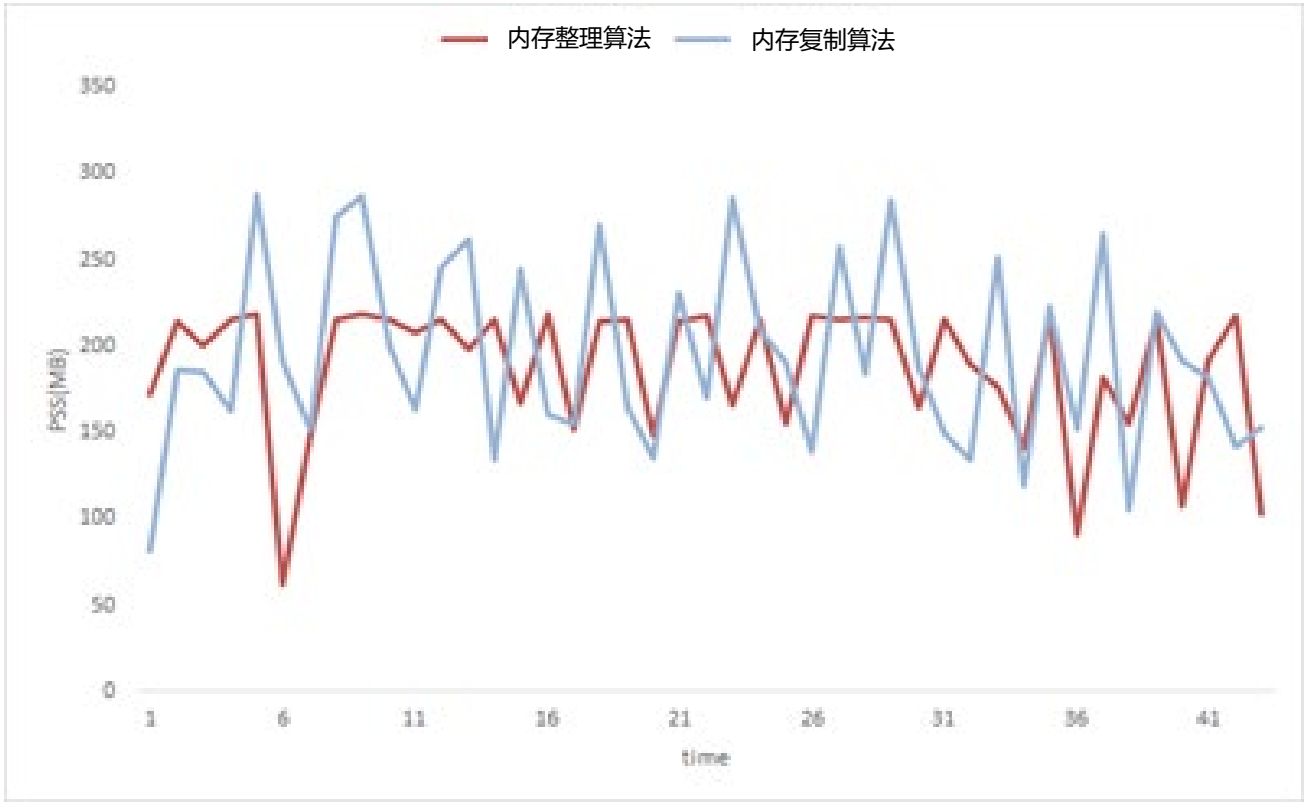




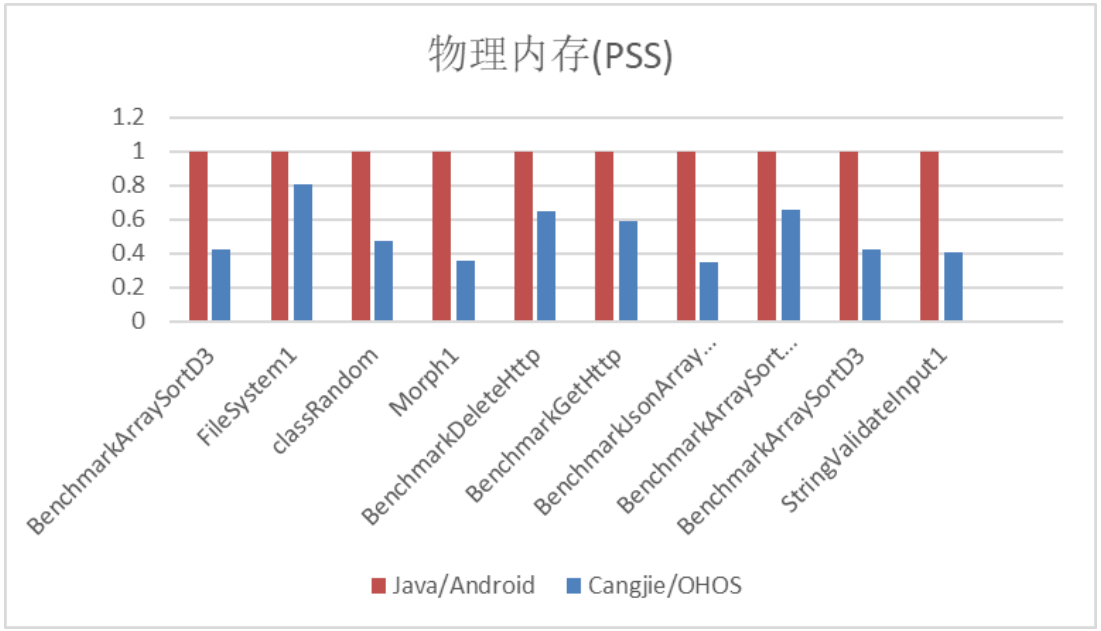
1. 测试用例是Benchmarks Game的binarytrees
2. 运行时延：在应用中创建一个新线程，新线程执行一段无限循环代码，记录上一次循环离开的时刻和当前循环进入的时刻，二者时间差称为“运行时延”，运行时延可以综合反映GC和线程调度对程序的影响。

内存整理降低内存峰值

内存整理算法“削峰”

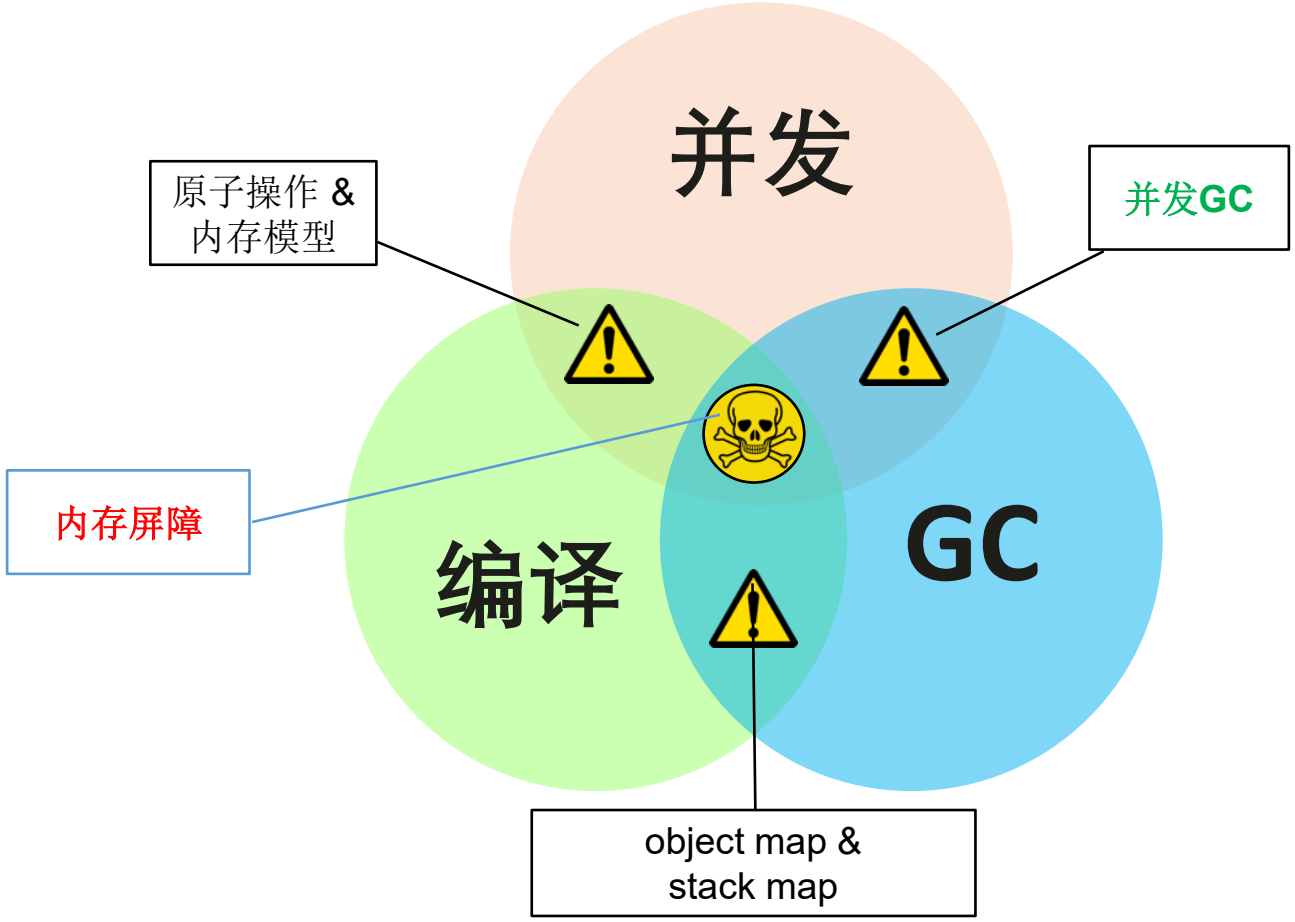


内存用量平均减少25%以上



优化仓颉GC

为了实现高性能，编译器与垃圾回收必是紧耦合的



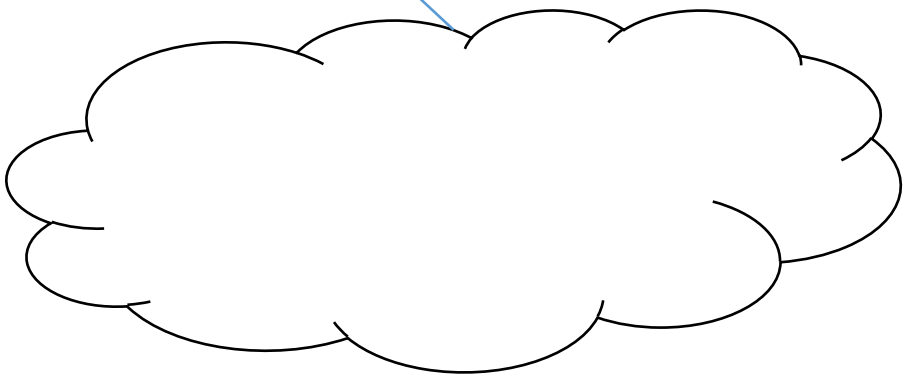
来源：Steve Blackburn. Micro Virtual Machines. In PLISS’ 17. https://youtu.be/T2WViuD5GrQ

读屏障和写屏障优化机会

- **读屏障**

对应源码：target = obj.ref
编译后伪码：target = LoadObjectField(obj, offset)

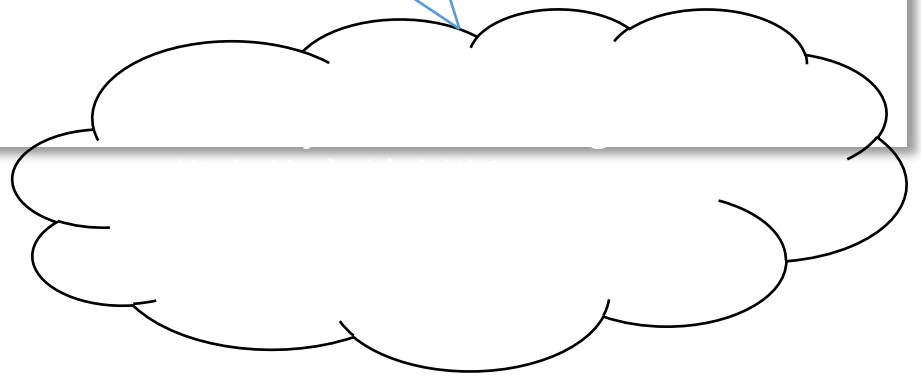
```
Object* LoadObjectField(
      Object* object,
      Offset field) {
  xx = object[field];
  if (...) { // load barrier slow path
    ...;
  }
}
```



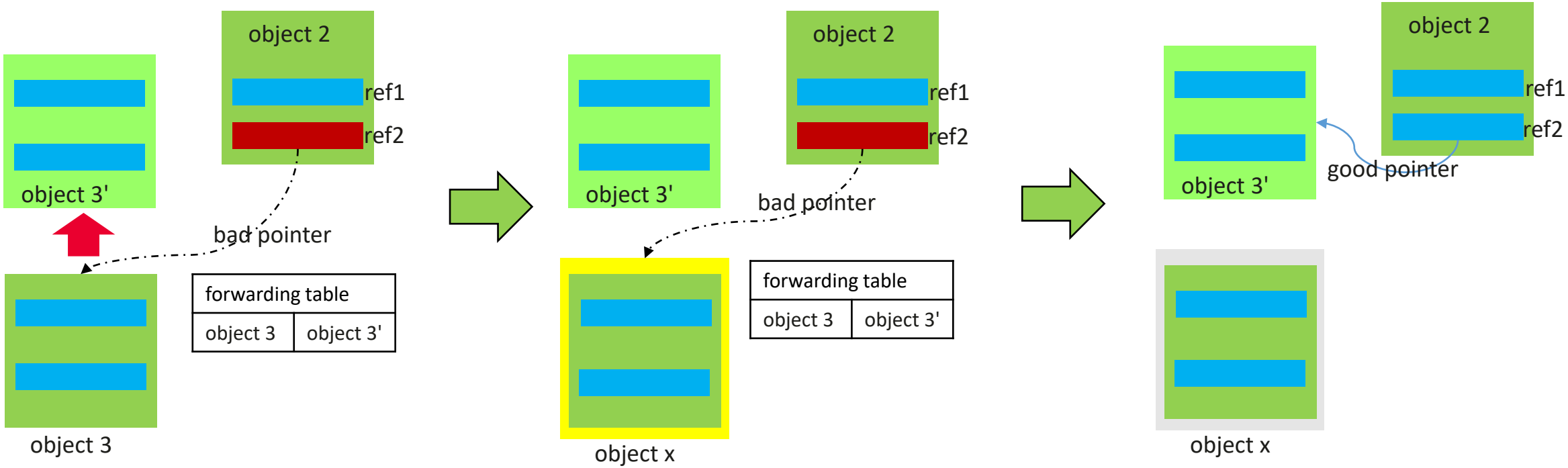
- **写屏障**

对应源码：obj.ref = target
编译后伪码：StoreObjectField(obj, offset, target)

```
void StoreObjectField(
      Object* object,
      Offset field,
      Object* target) {
  if (...) { // pre-write barrier slow path
    ...;
  }
  object[field] = target;
  if (...) { // post-write barrier slow path
    ...;
  }
}
```



指针标记的作用：辅助读屏障和GC修复旧指针



活对象object 3被搬移到其它地方，(object 2).ref2指向旧对象

object 3占用的内存被回收用于分配object x，(object 2).ref2指向错误的内存

修复(object 2).ref2使其指向新对象

在并发内存整理算法里，通过指针标记区分bad pointer和good pointer

指针标记的典型编码

通过不同的指针状态区分ghost pointer和live pointer

位域	63		…	…		56	…	…	47						…						0
值	?	?	?	?	?	?	?	?	x	x	x	x	x	x	x	x	x	x	x	x	x
	指针状态								内存地址												

64位平台指针标记示例

1. good pointer

位域	63		…	…		56	…	…	47						…						0
值	0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x
	指针状态								内存地址												

2. bad pointer for **previous gc**

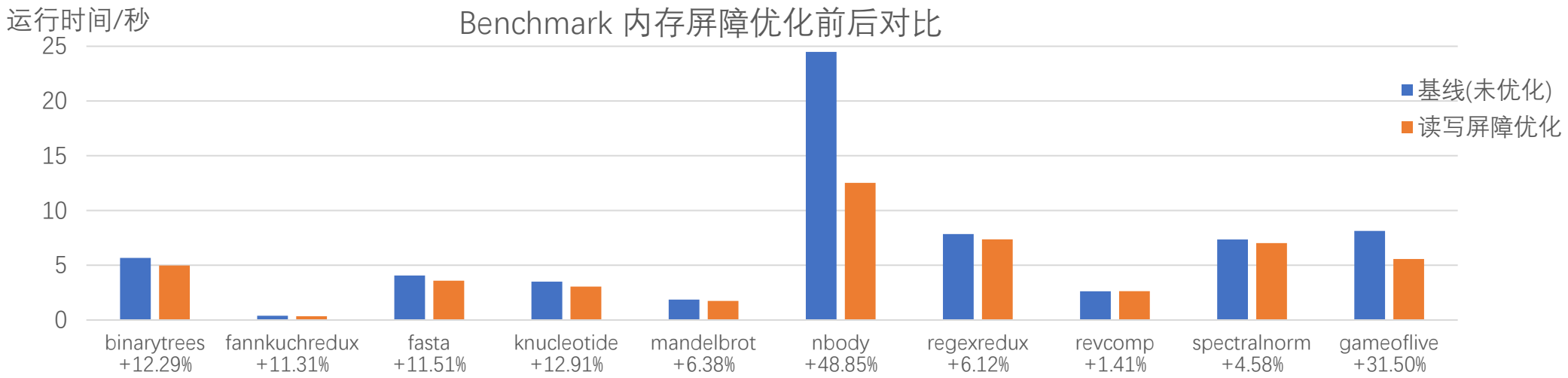
位域	63		…	…		56	…	…	47						…						0
值	0	0	0	0	0	0	V	1	x	x	x	x	x	x	x	x	x	x	x	x	x
	指针状态								内存地址												

3. bad pointer for **current gc**

位域	63		…	…		56	…	…	47						…						0
值	0	0	0	0	0	0	~V	1	x	x	x	x	x	x	x	x	x	x	x	x	x
	指针状态								内存地址												

V和~V构成乒乓结构，
轮流表示previous gc和current gc.

内存屏障优化




```
union {
        struct {
                uintptr_t tag : 16;
                uintptr_t address: 48;
        };
        uintptr_t value;
};
```

```
Object* LoadObjectField(Object* obj, Field* f)
{
        auto tag = f->tag;
        if(tag == 0) {
                return f->address;
        }
        return readfieldslow(obj, f);
}
```

在编译时内联

```
void StoreObjectField(Object* obj, Field* f, Object* target)
{
    if (gc_is_running) {
        writeSlow(obj, f, target);
    } else {
        f->value = target;
    }
}
```

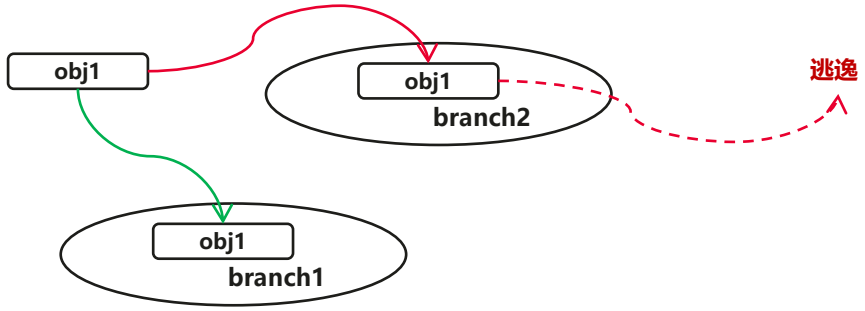
在编译时内联

其它重要优化

➢ **PEA (Partial Escape Analysis)**：基于控制流信息分析对象是否逃逸，通过将未逃逸的对象分配到栈上而不是堆上减少GC压力。

```
class State {……}
func getValue(id: Int64) {
    var state: State = State(……)
    if (id > 0) {
        return state.value + id
    } else if (id == 0) {
        return getValue(id, state)
    }
    return state.value
}
```

True分支state**没有逃逸**，
可以在**栈上分配**



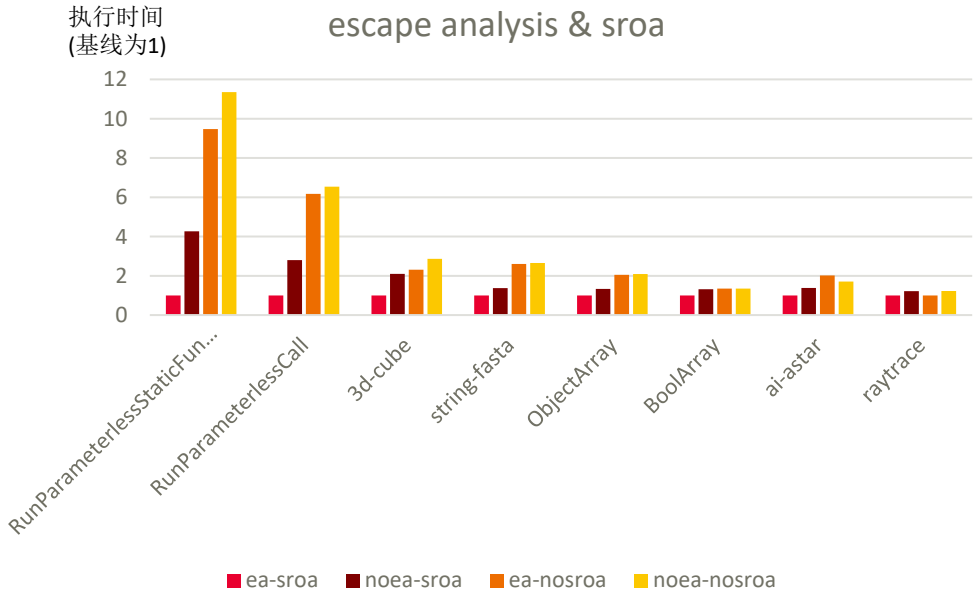
➢ **SROA (Scalar Replacement Of Aggregates)**

：基于栈上对象指针的作用范围，将不逃逸的栈上对象拆成多个标量，这些标量值存储在寄存器里，不需要保存在栈内存里通过访存指令获取。

```
struct B {
  int a = 5;
  int b = 15;
};

struct A {
  struct B b;
  int c = 10;
  class d = new d();
  double e = 2.0;
};
```

Before:
```
Foo() {
  var a : A = A()
  bar(a.b)
  gA += a.c
  gD += a.e
  …
}
```

After:
```
Foo() {
  var b : B = B()
  bar(b)
  gA += 10
  gD += 2.0
  …
}
```



手工内存管理 vs. 自动内存管理

手工内存管理和安全性问题

```
typedef struct Dummy { int a; int b; } Dummy;

void foo(void) {
  Dummy *ptr = (Dummy *) malloc(sizeof(struct Dummy));
  Dummy *alias = ptr;
  free(ptr);
  int a = alias.a;
  free(alias);
}
```

Use after free

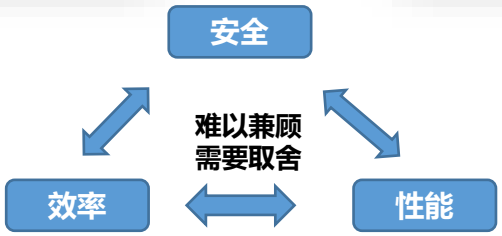
Aliasing ✚ *Mutation*

Double free

开发痛苦，容易出错

Garbage Collection

- Runtime frees heap memory that is no longer in use
- How do we determine what is no longer in use?
- Ideally: any piece of memory that will not be used in the future of the computation
- In practice: any piece of memory that is not **reachable**
  - Reachable = can be accessed through some chain of pointers starting from program variables
  - This is a subset of the memory that will not be used in the future



性能开销、卡顿、内存开销等

第三条路径：类型系统和编译器支持

Rust's solution:

~~*Aliasing*~~ ✚ ~~*Mutation*~~

Compiler enforces:

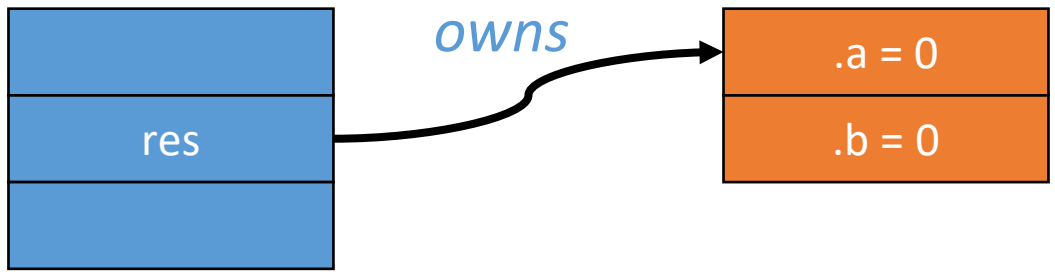
- Every resource has a unique *owner*.
- Others can *borrow* the resource from its owner.
- Owner *cannot* free or mutate its resource while it is borrowed.

No need for runtime | Memory safety | Data-race freedom

Ownership

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                      a: 0,
                      b: 0
                  });
}
```

res is out of scope and its resource is freed automatically



Ownership: Lifetime

Lifetime that res owns the resource.

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res: Box<Dummy>;
    {
        res = Box::new(Dummy {a: 0, b: 0});
    }
    res.a = 2048;
}
```

Compiling Error: res no longer owns the resource

- Lifetime is determined and checked statically.

Ownership: Unique Owner

Aliasing ✚ *Mutation*

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy {
                    a: 0,
                    b: 0
                });
    take(res);
    println!("res.a = {}", res.a);
}

fn take(arg: Box<Dummy>) {
}
```

Compiling Error!

Ownership is moved from res to arg

arg is out of scope and the resource is freed automatically

Immutable/Shared Borrowing (&)

Aliasing ✚ ~~*Mutation*~~

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{
                    a: 0,
                    b: 0
                });
    take(&res);
    res.a = 2048;
}

fn take(arg: &Box<Dummy>) {
    arg.a = 2048;
}
```

Resource is returned from arg *to* res

Resource is immutably borrowed by arg *from* res

Compiling Error: Cannot mutate via an immutable reference

Resource is still owned by res*. No free here.*

Immutable/Shared Borrowing (&)

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});
    {
        let alias1 = &res;
        let alias2 = &res;
        let alias3 = alias2;
        res.a = 2048;
    }
    res.a = 2048;
}
```

- Read-only sharing

Mutable Borrowing (&mut)

```
struct Dummy { a: i32, b: i32 }

fn foo() {
    let mut res = Box::new(Dummy{a: 0, b: 0});

    take(&mut res);
    res.a = 4096;

    let borrower = &mut res;
    let alias    = &mut res;
}

fn take(arg: &mut Box<Dummy>) {
    arg.a = 2048;
}
```

~~Aliasing~~ + Mutation

Mutably borrowed by arg from res

Multiple mutable borrowings are disallowed

Returned from arg to res