简单介绍jvm的垃圾回收算法

1. 标记清理整理
2. 半区复制
3. 分区方案
4. 分代回收
5. 并发标记清理

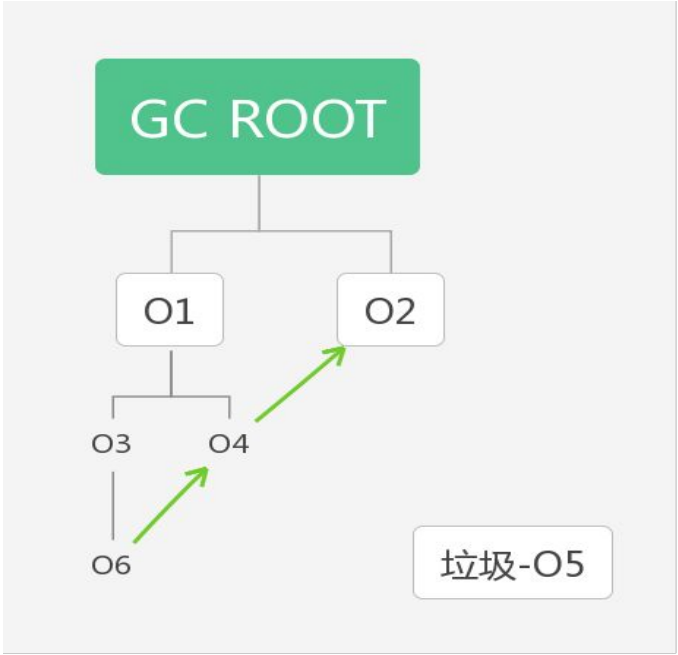
author:余杰水

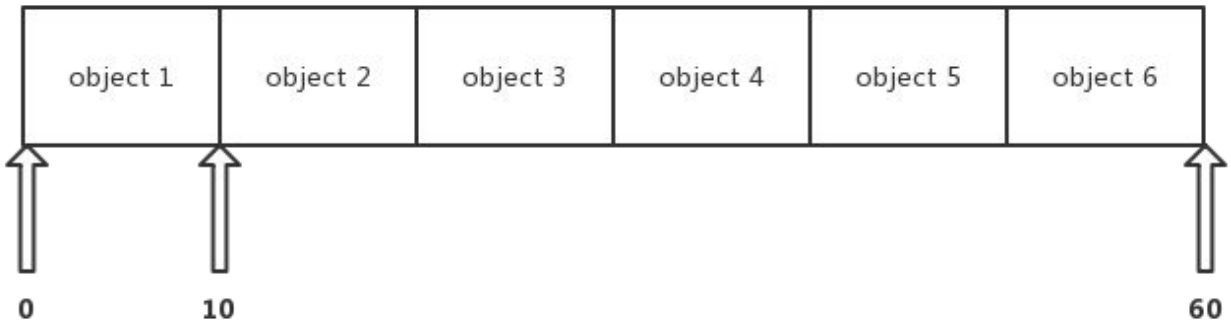
一个简单对象

```
class Obj {
  val a = new Object
  val b = new Object
  val i = 0
}
```

对象的内存结构

程序运行时的对象引用情况



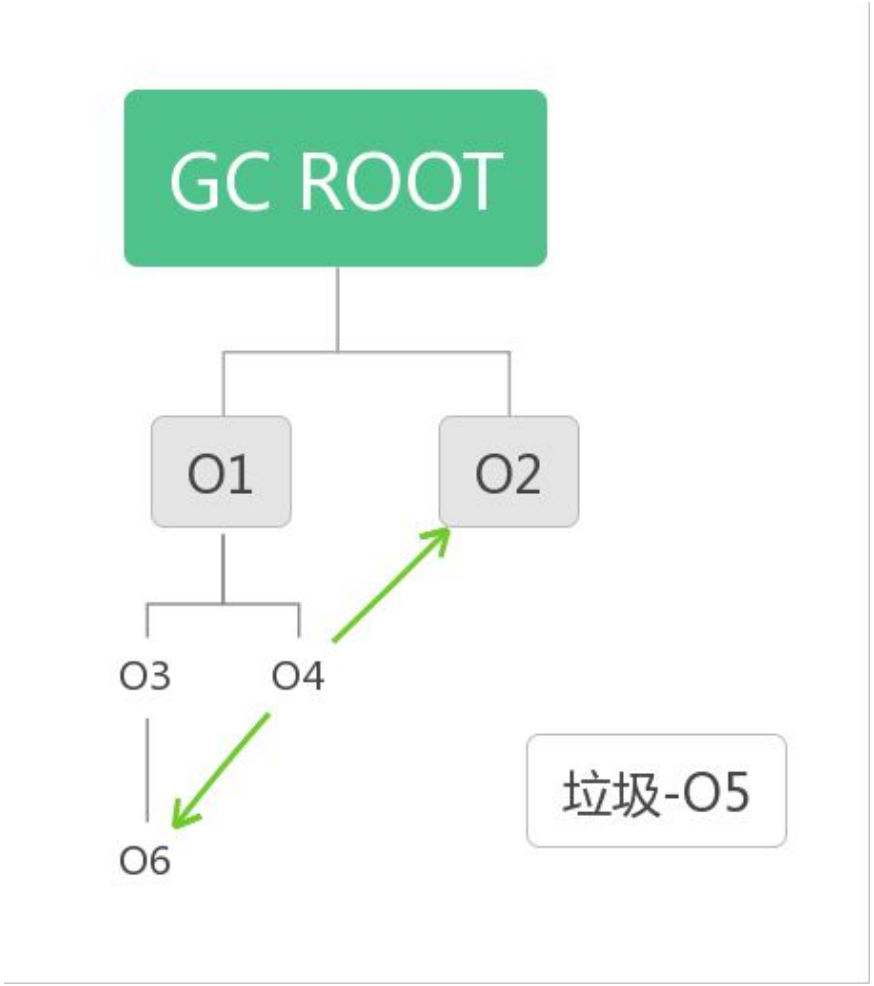


标记清理算法

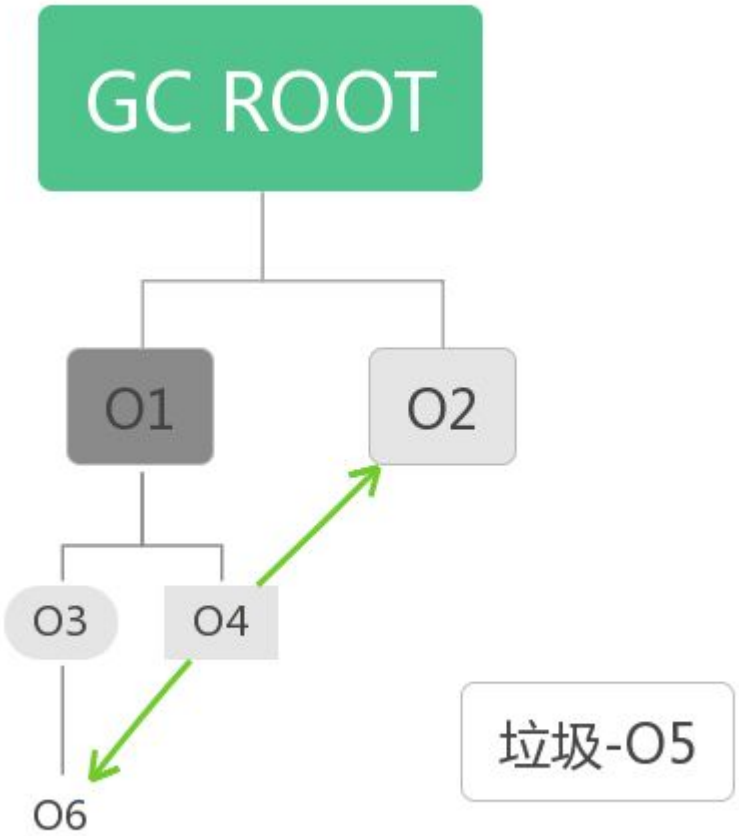
val workList = GcRoot

```
def mark(workList):
    if (workList isEmpty) return
    val obj = head(workList)
    if (header(obj).is_mark) return mark(tail(workList))
    header(obj).is_mark = true
    return mark(fields(obj) + tail(workList)) //深度优先搜索
```

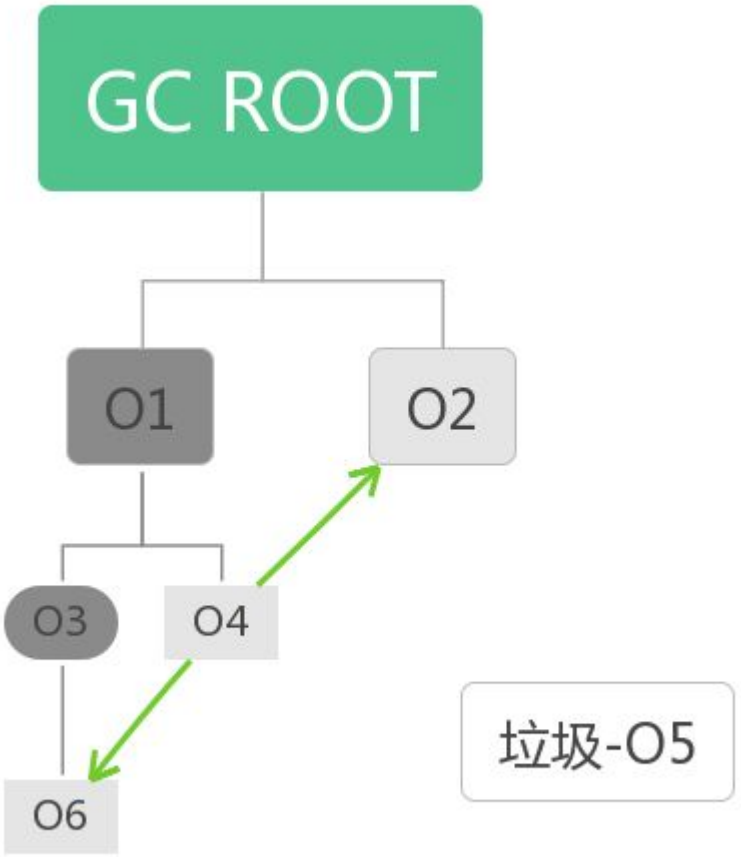
workList = O1, O2

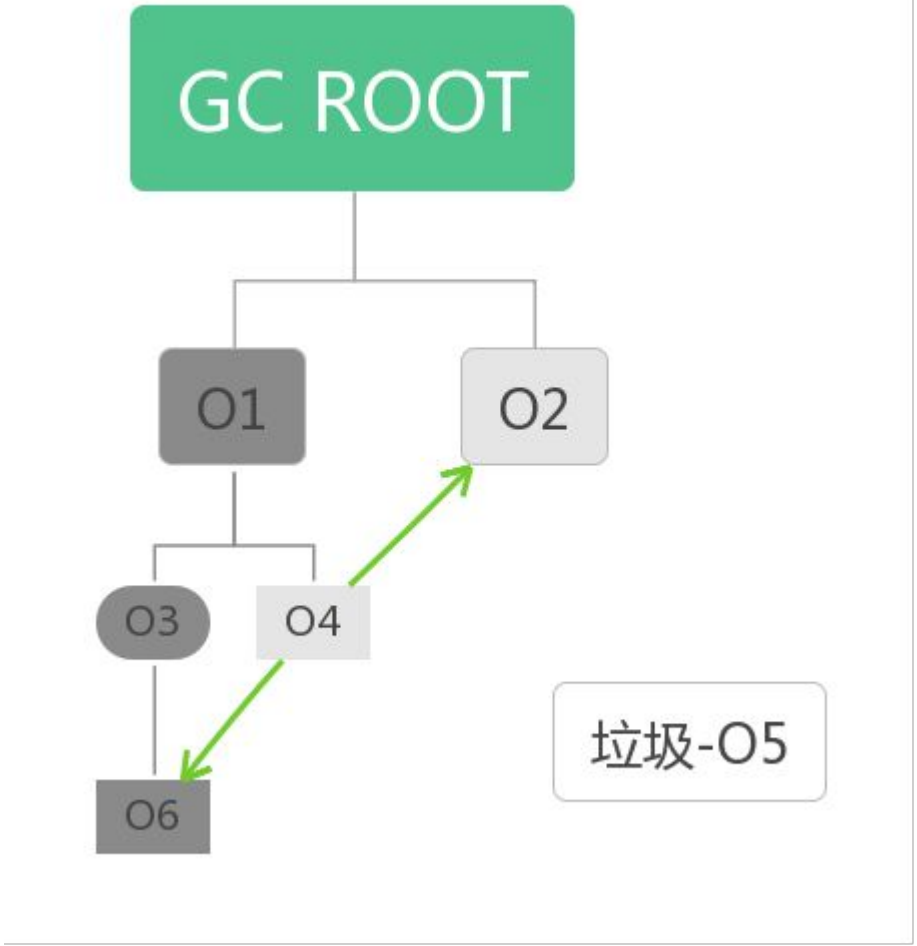


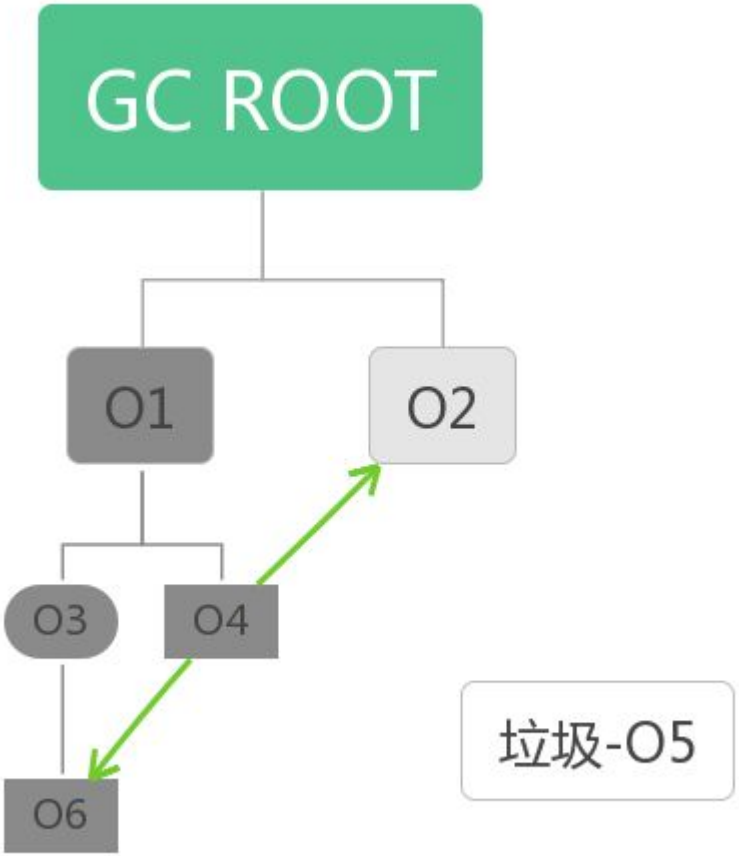
workList = O3, O4, O2,

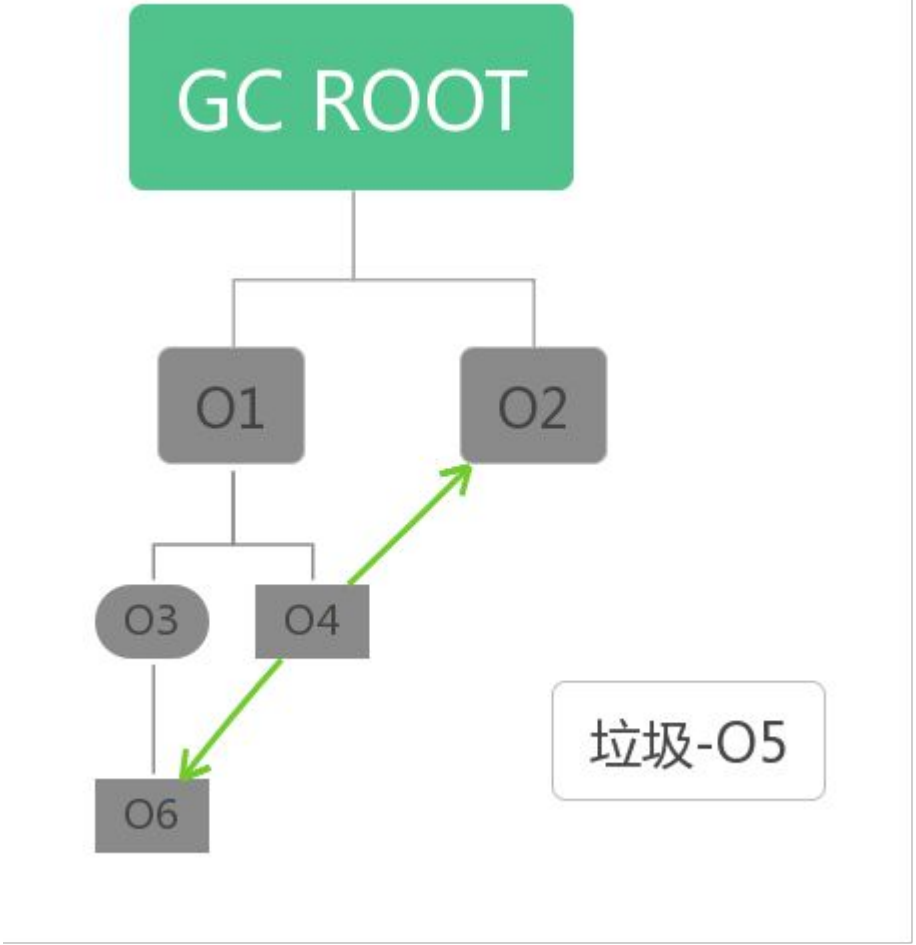


workList = O6, O4, O2



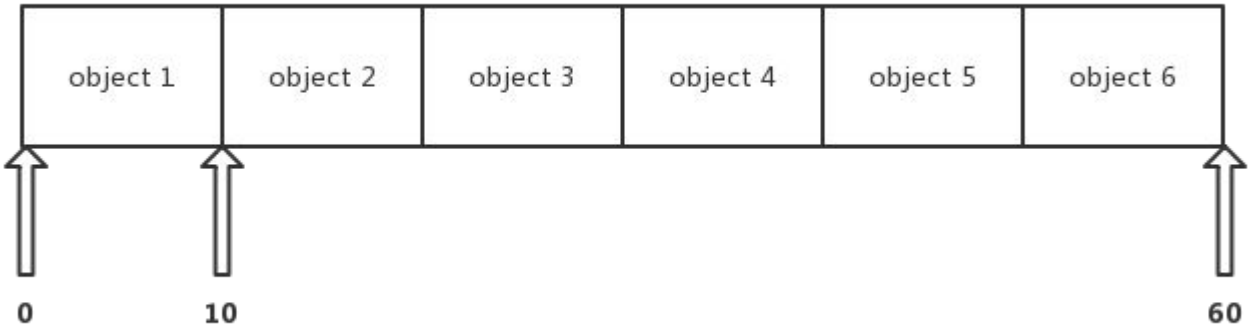

GC ROOT
O1
O2
O3
O4
O6
垃圾-O5

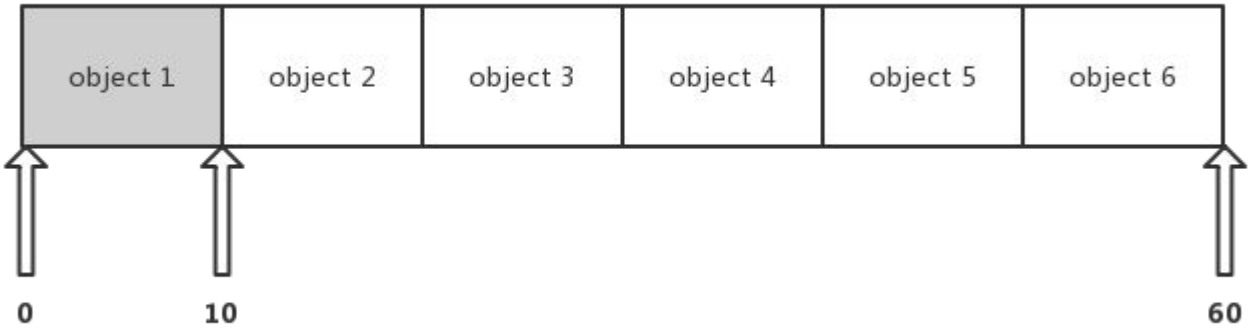

GC ROOT
O1
O2
O3
O4
O6
垃圾-O5

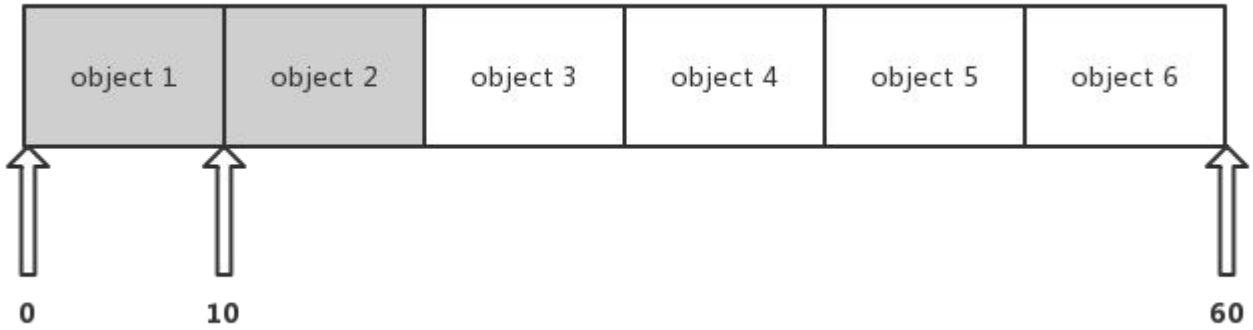

GC ROOT
O1
O2
O3
O4
O6
垃圾-O5

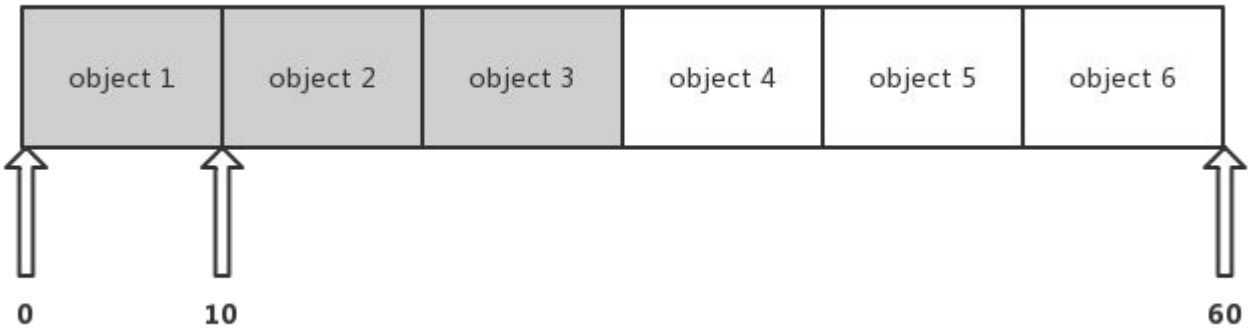
清理

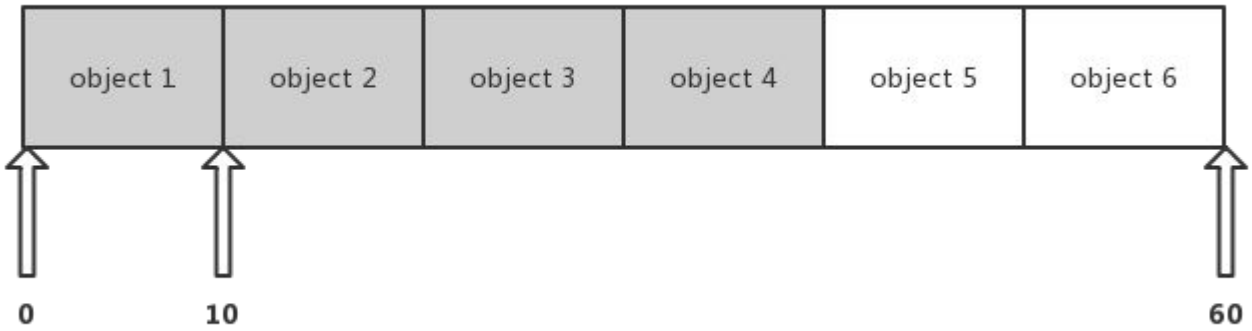
```
def clear(start_ptr,end_ptr):
    if (start_ptr >= end_ptr) return
    size = header(start_ptr).size
    if (header(start_ptr).is_mark == false))
        free_memory(start_prt,size)
    clear(start_ptr + size,end_ptr)
```

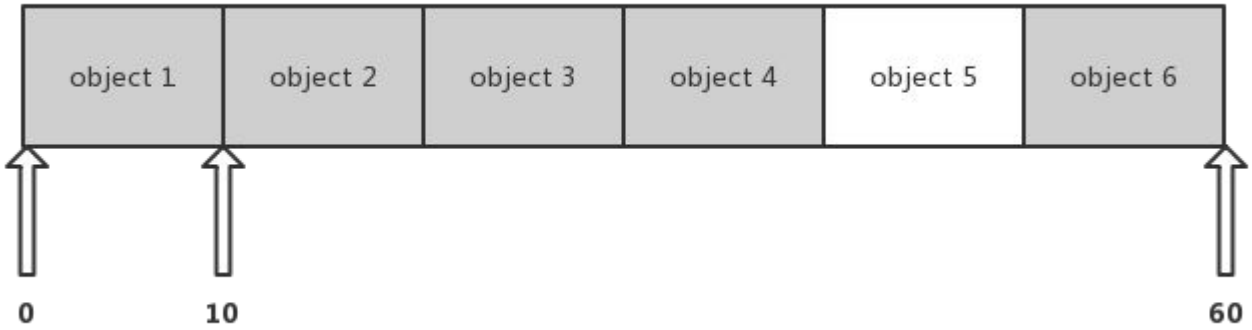

object 1
object 2
object 3
object 4
object 5
object 6
0
10
60


object 1
object 2
object 3
object 4
object 5
object 6
0
10
60


object 1
object 2
object 3
object 4
object 5
object 6
0
10
60


object 1
object 2
object 3
object 4
object 5
object 6
0
10
60


object 1
object 2
object 3
object 4
object 5
object 6
0
10
60

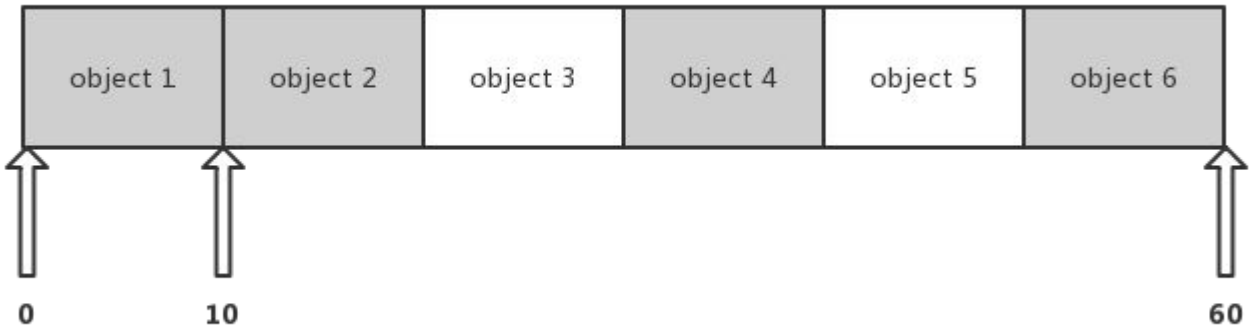

object 1
object 2
object 3
object 4
object 5
object 6
0
10
60

标记整理 - 解决内存碎片

总共的空闲内存为20

但如果申请一块大小为20的连续的内存会申请失败

因为空闲的内存是两块独立大小为10的内存块

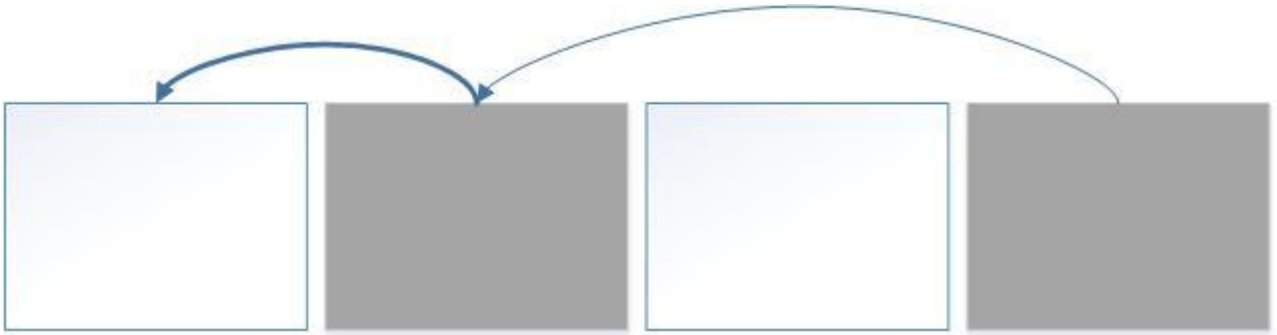


整理

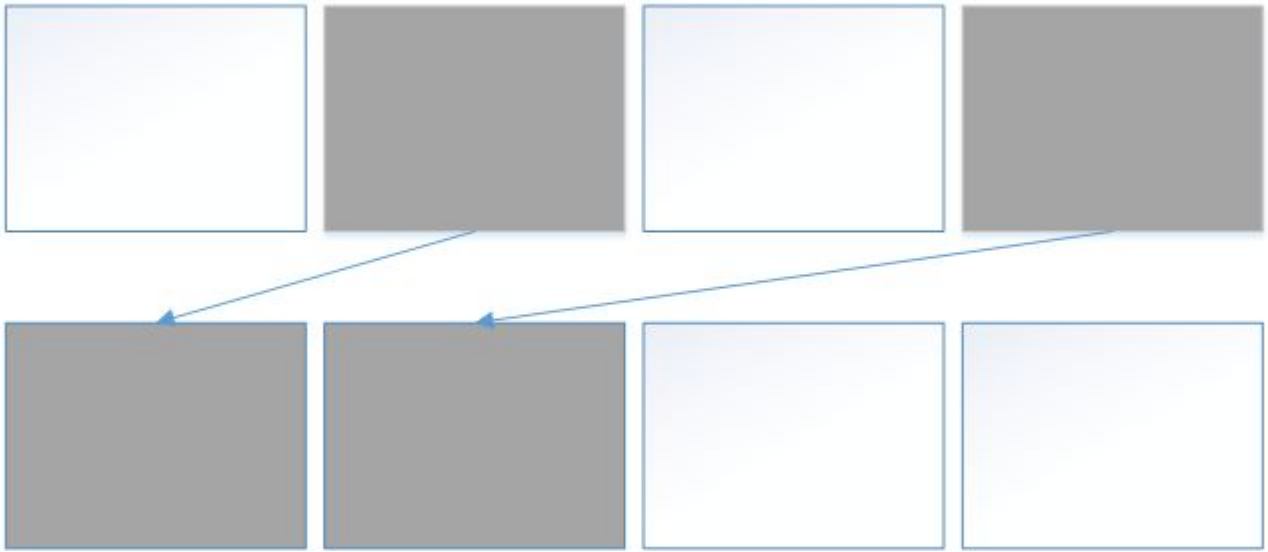
1. ptr_map = 对象当前的指针->对应整理之后的指针
2. 根据 ptr_map 对对象的fields中的指针进行修改
3. 移动对象

复制整理

滑动整理

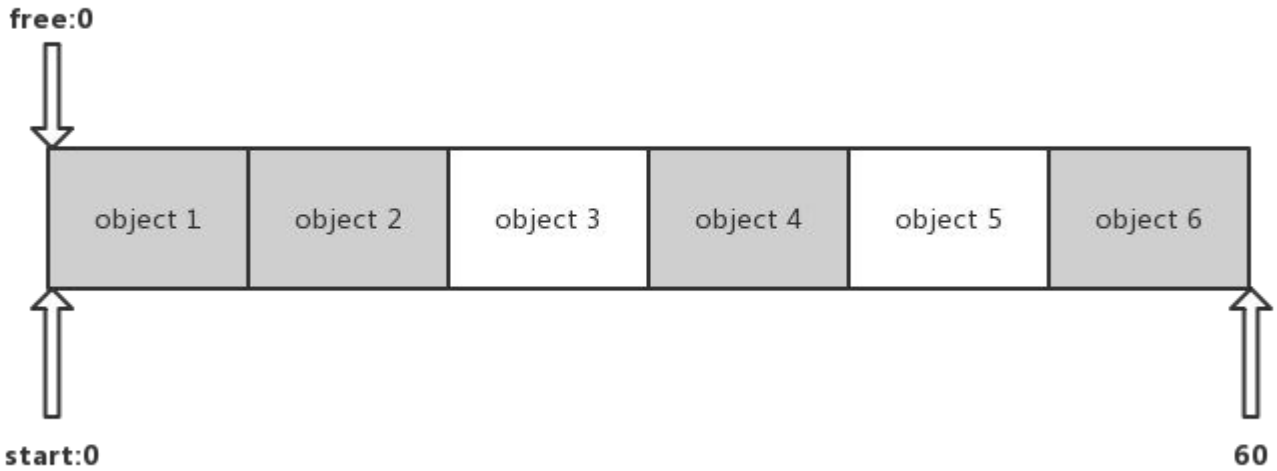


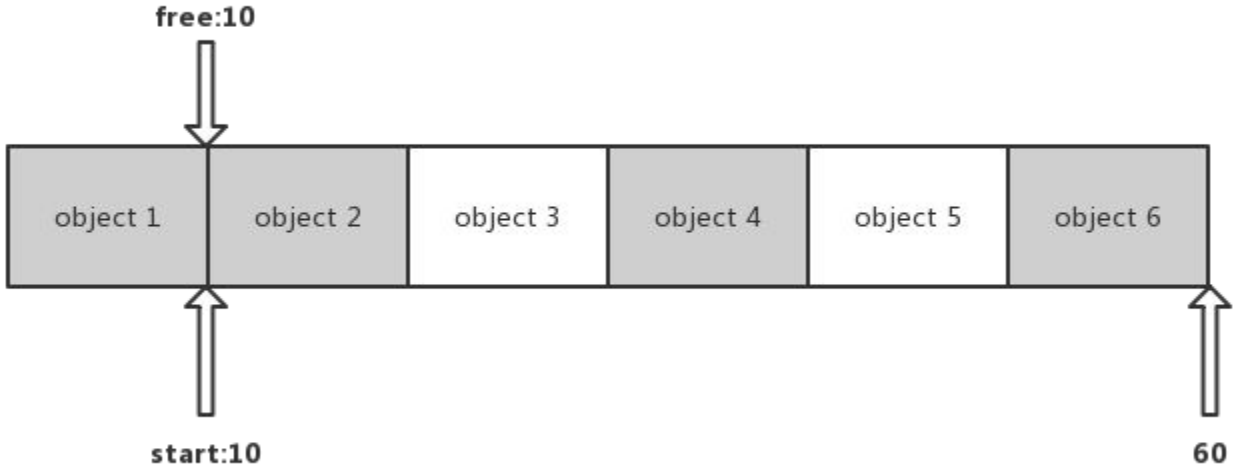
复制整理

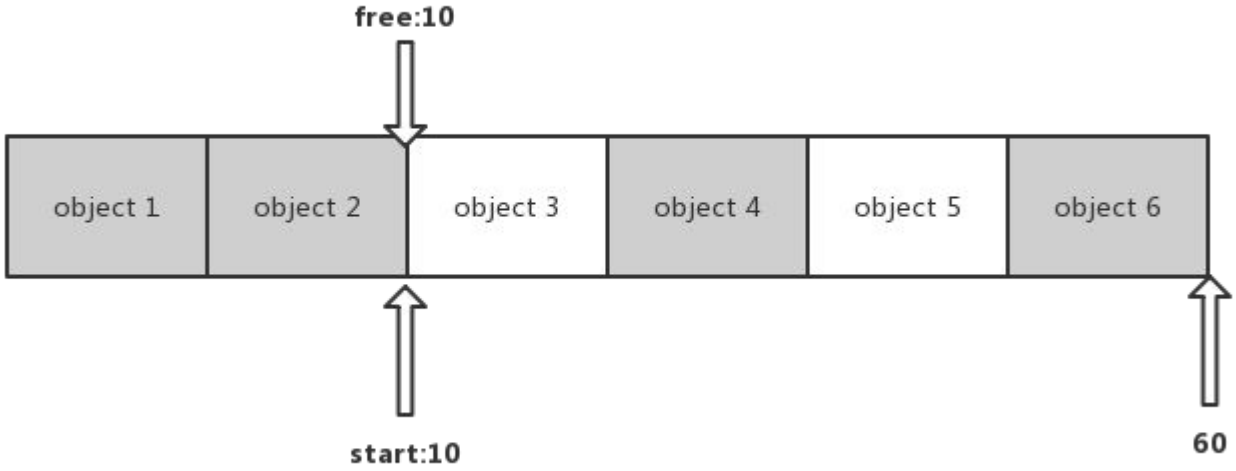


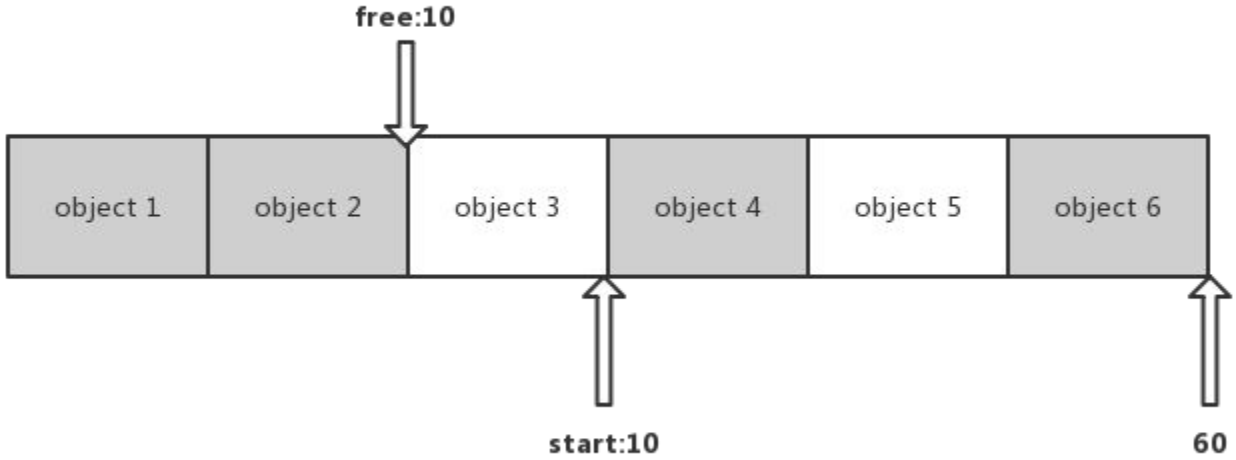
```
ptr_map = {}

def gen_ptr_map(free_ptr,start_ptr,end_ptr):
        if (start_prt >= end_ptr) return
        size = header(start_ptr).size
        if(header(start_ptr).is_mark)
                ptr_map += start_ptr -> free_ptr
                free_ptr += size
                header(start_ptr).is_move = true
        return oranization(free_ptr,start_ptr+size,end_ptr)
```

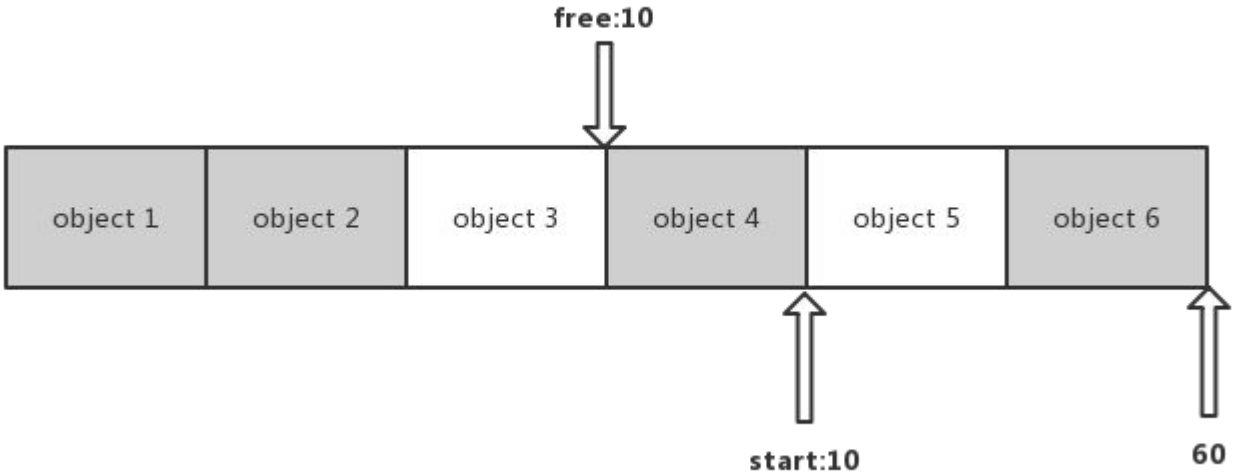

free:0
object 1
object 2
object 3
object 4
object 5
object 6
start:0
60


free:10
object 1
object 2
object 3
object 4
object 5
object 6
start:10
60

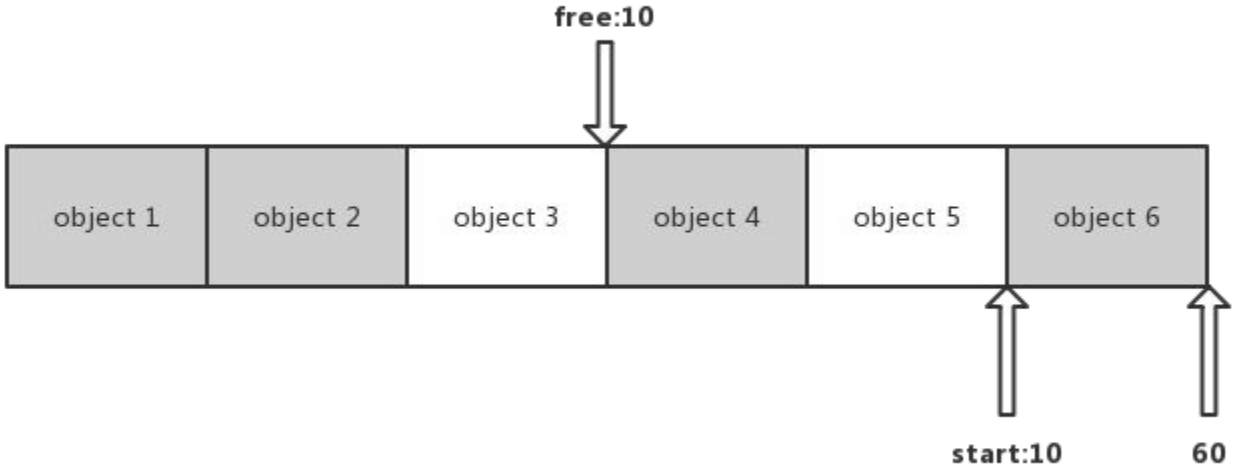

free:10
object 1
object 2
object 3
object 4
object 5
object 6
start:10
60


free:10
object 1
object 2
object 3
object 4
object 5
object 6
start:10
60

map = O4 -> O3

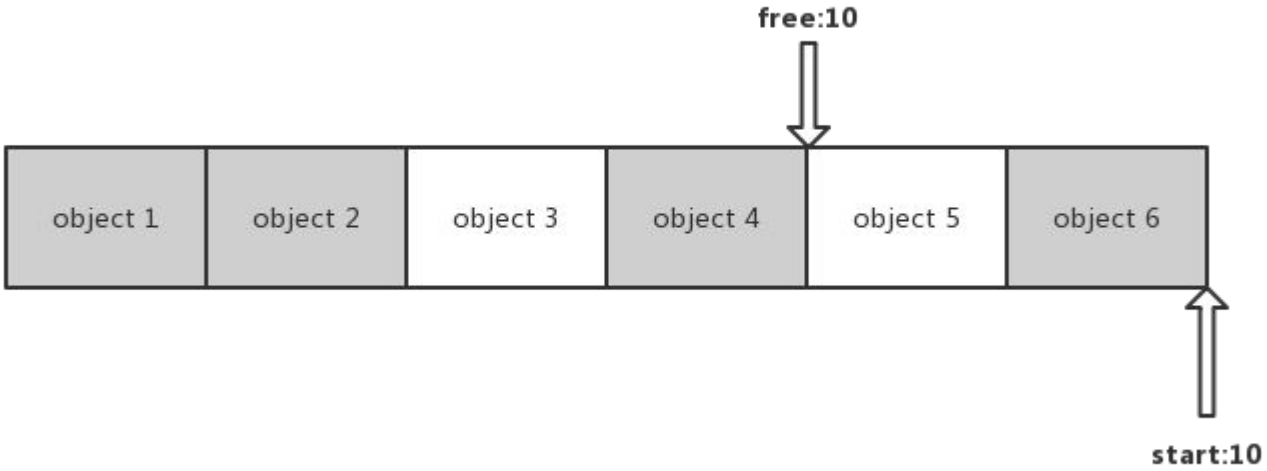

free:10
object 1
object 2
object 3
object 4
object 5
object 6
start:10
60

map = O4 -> O3


free:10
object 1
object 2
object 3
object 4
object 5
object 6
start:10
60

map = O4 -> O3,

O6 -> O4



```
def update_ptr(start_ptr,end_ptr):
    if(start_ptr >= end_ptr) return
    size = header(start_prt).size
    if (header(start_prt).is_mark)
        fields(start_prt) foreach{field=>
            if (header(field).is_move)
                start_prt[field] = prt_map(field)
        }
    update_ptr(start_prt + size, end_ptr)
```

1. 标记整理的开销通常大于标记清理
2. 内存碎片通常要运行垃圾回收多次之后才会成为问题
3. 所以通常标记清理和标记整理会交替进行

阶段	耗时
标记	存活对象数量
清理	堆大小
整理	堆大小 / 存活对象数量
复制	堆大小 / 存活对象数量

分代回收

将内存中划分成不同区域;对不同的区域使用不同的回收算法;

问题:

1. 如何划分区域
2. 对象如何在不同区域之间移动

划分成固定大小的区域

0-32的对象放入32列表中

32-64的对象放入64列表中

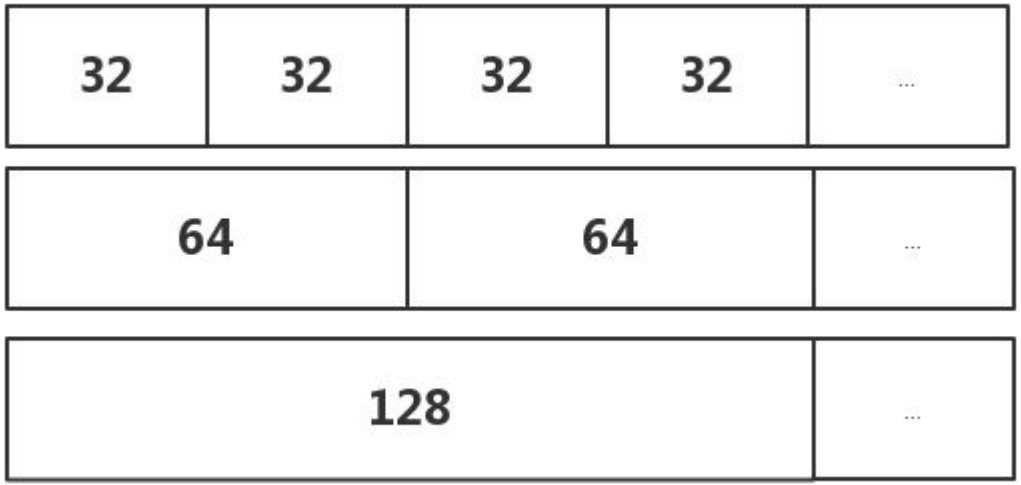
64-128的对象放入128列表中

其余的放入大对象列表中

优点:

对于固定大小的对象列表

可以迅速分配和回收内存/无需整理



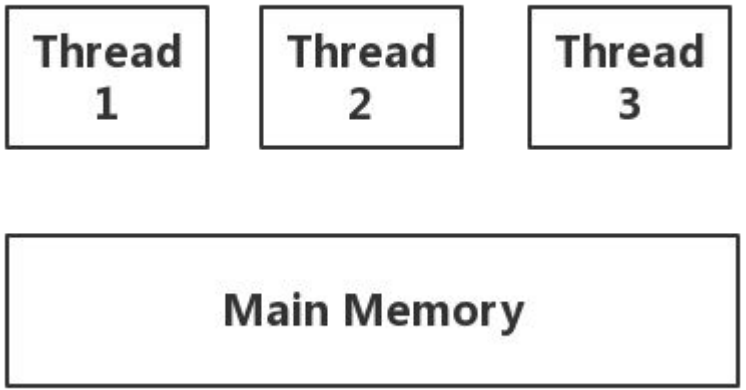
根据线程划分

为每个线程分配一个独立的私有内存堆

可以缓解多线程申请内存时同步开销

使用逃逸分析确保分配在私有堆的对象

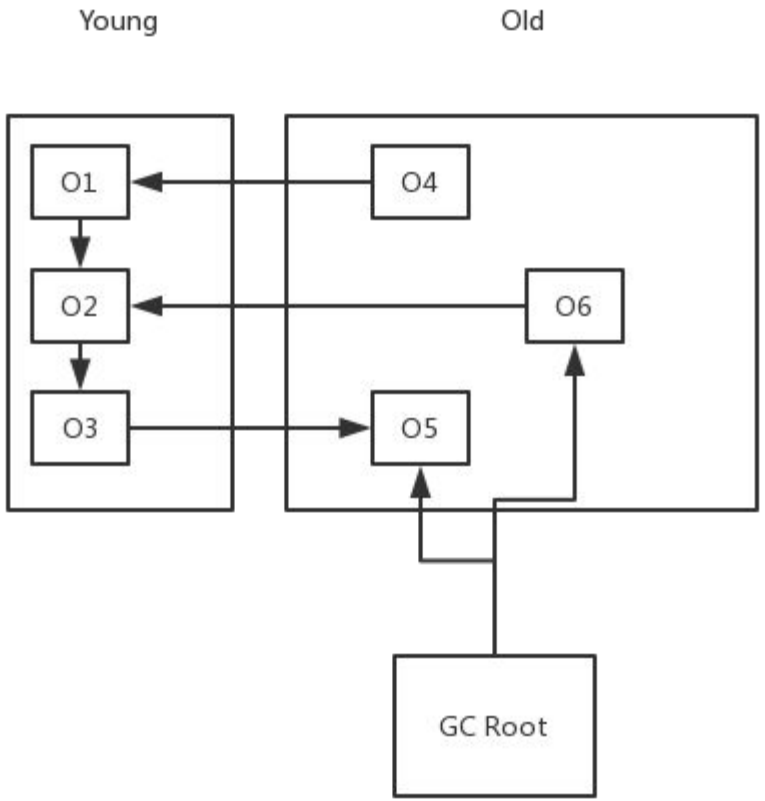
不会被其他线程访问

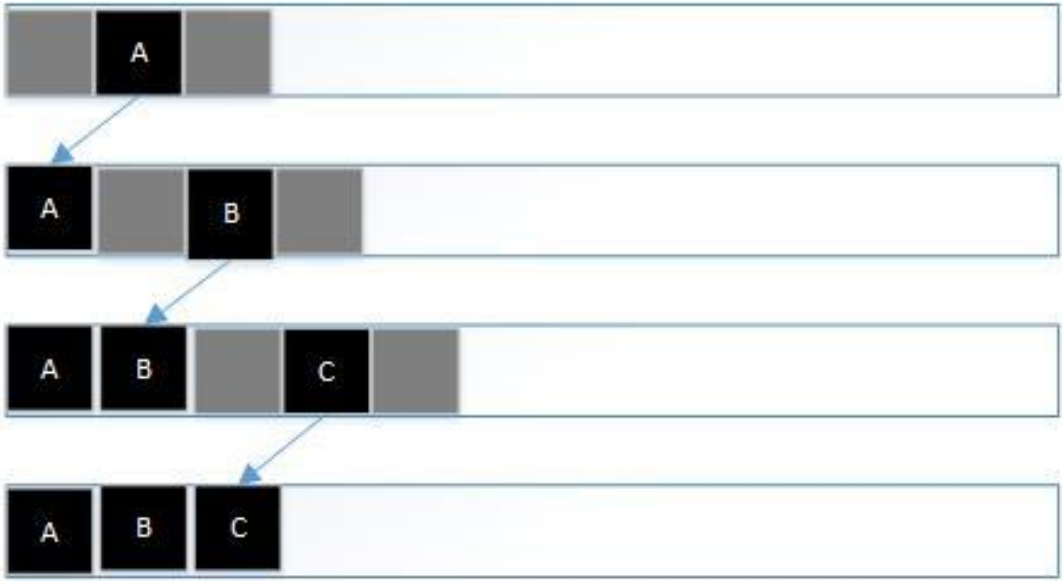


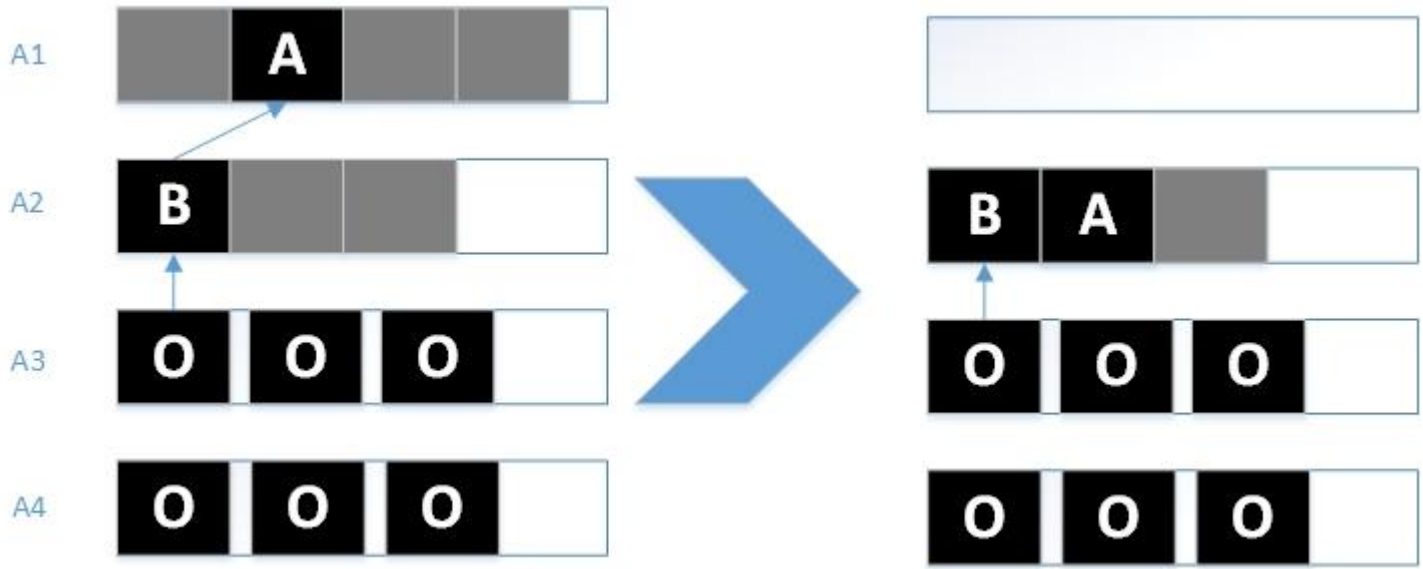
根据对象存活时间划分

young区域 :新创建的对象,

old区域: 长寿的对象




A
A
B
A
B
C
A
B
C

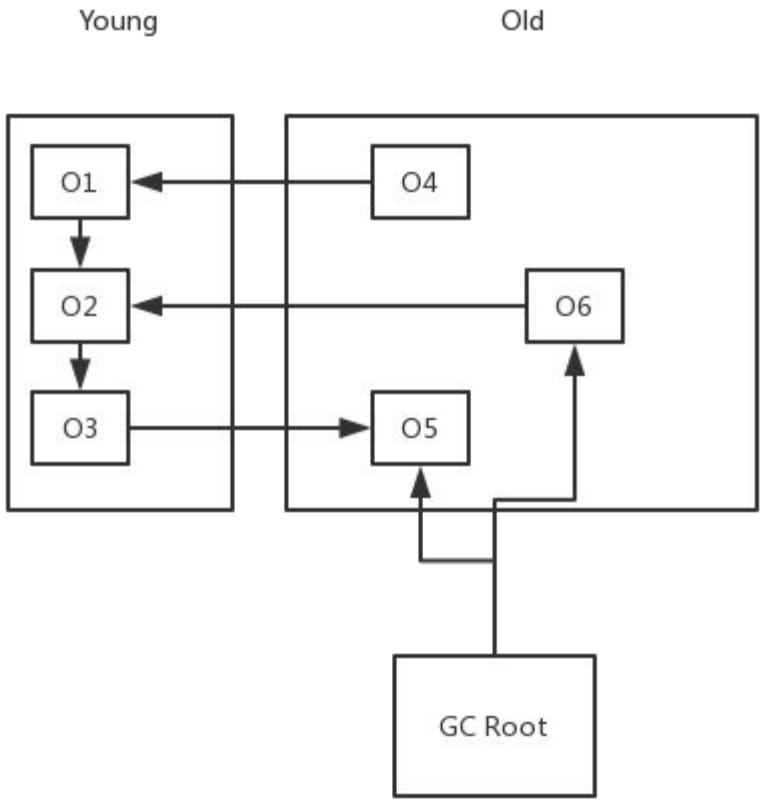

A1
A
A2
B
A3
O
O
O
A4
O
O
O
B
A
O
O
O
O
O
O

跨代指针集

需要记录下所有Old区域中

引用了Young区域的对象

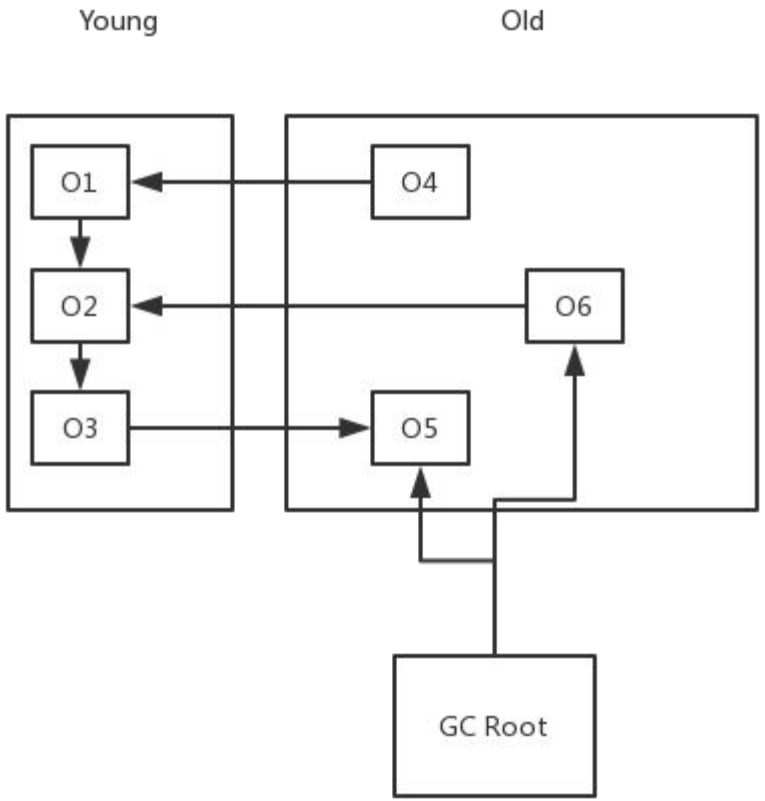
如: { O4, O6 }

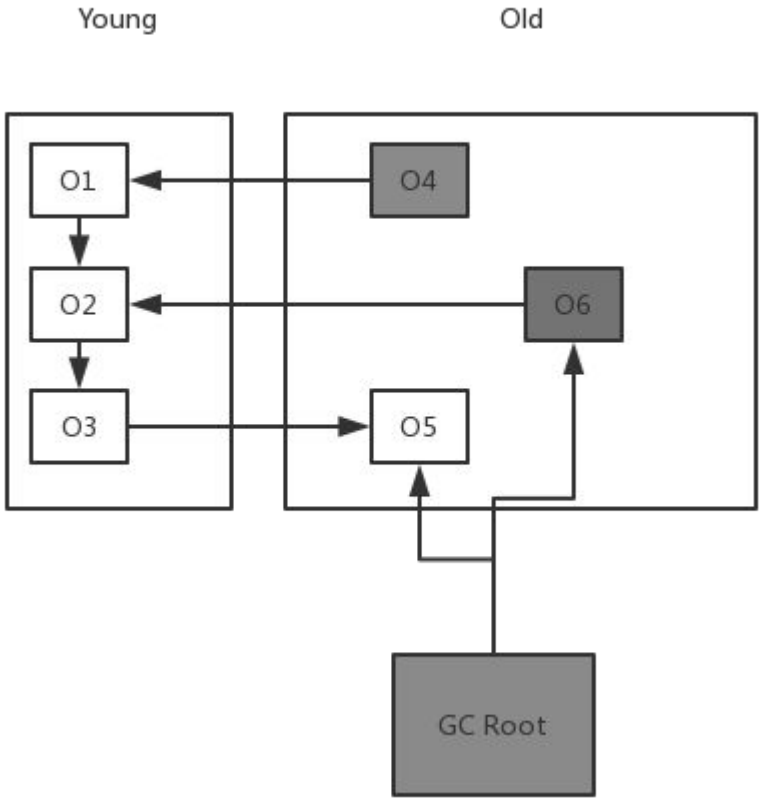


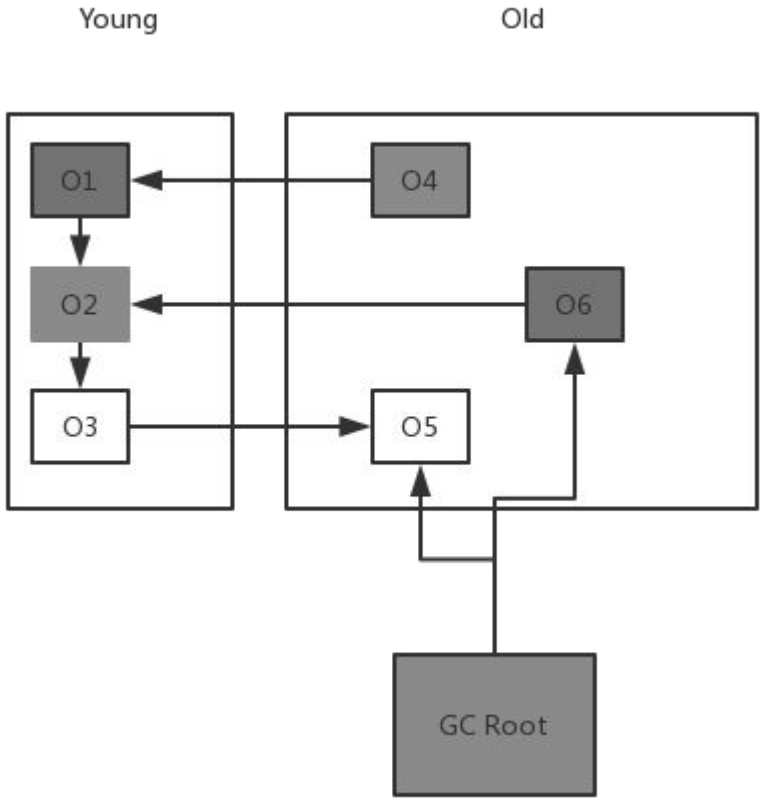
young标记

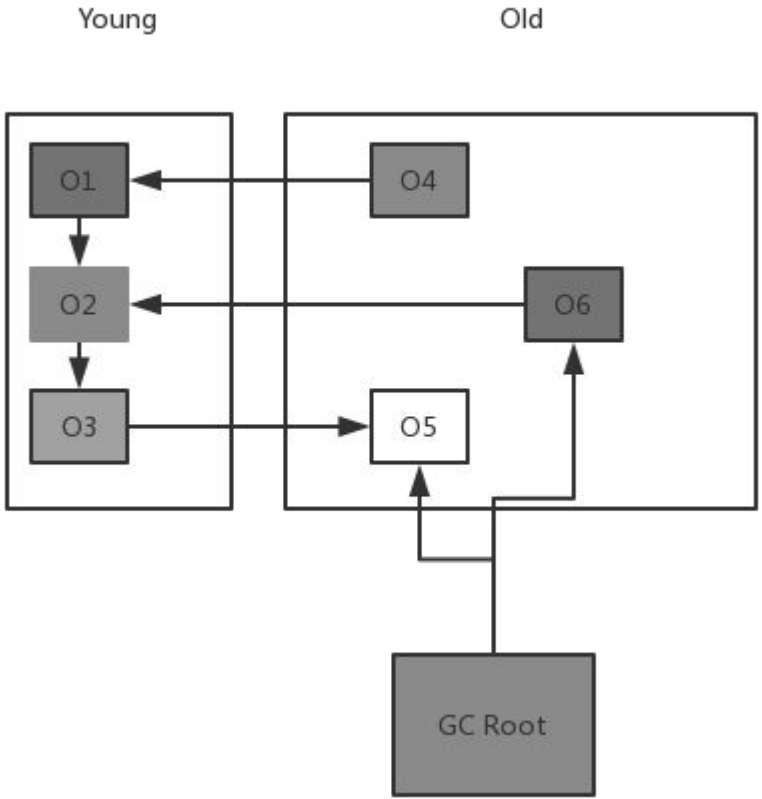
val workList = GcRoot + 跨代指针集

```
def mark(workList):
    if (workList isEmpty) return
    val obj = head(workList)
    if (obj is in Old) return mark(tail(workList))
    ...
```



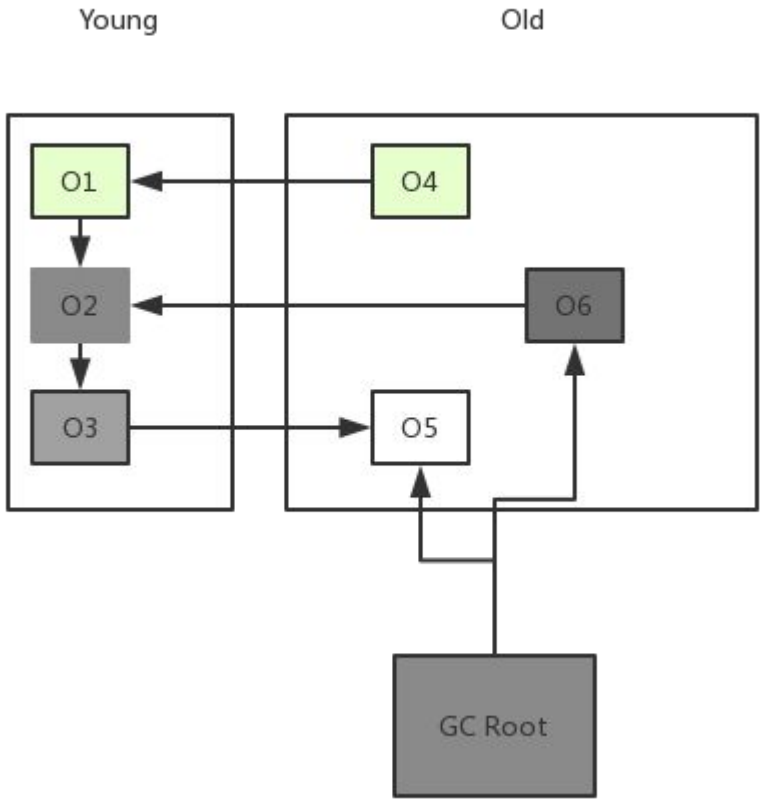

Young
Old
O1
O2
O3
O4
O5
O6
GC Root


Young
Old
O1
O4
O2
O6
O3
O5
GC Root


Young
Old
O1
O2
O3
O4
O5
O6
GC Root

垃圾 O1 , O4 没有正常回收

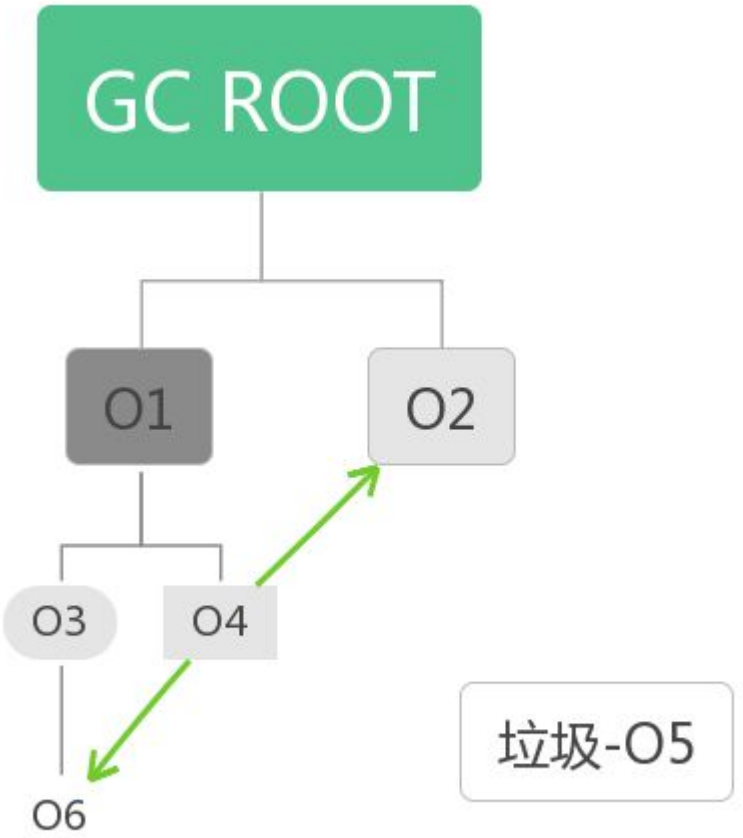
要等到执行一次完整标记清理才会回收



并发回收

垃圾回收与应用程序同时运行

降低程序停顿时间

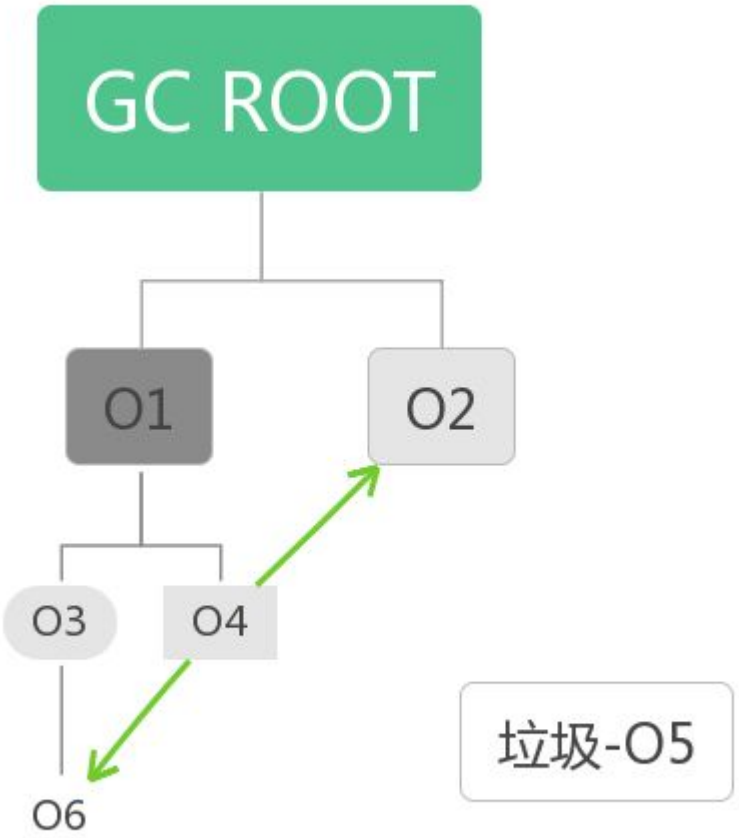


在标记阶段

程序修改了未扫描的对象O6

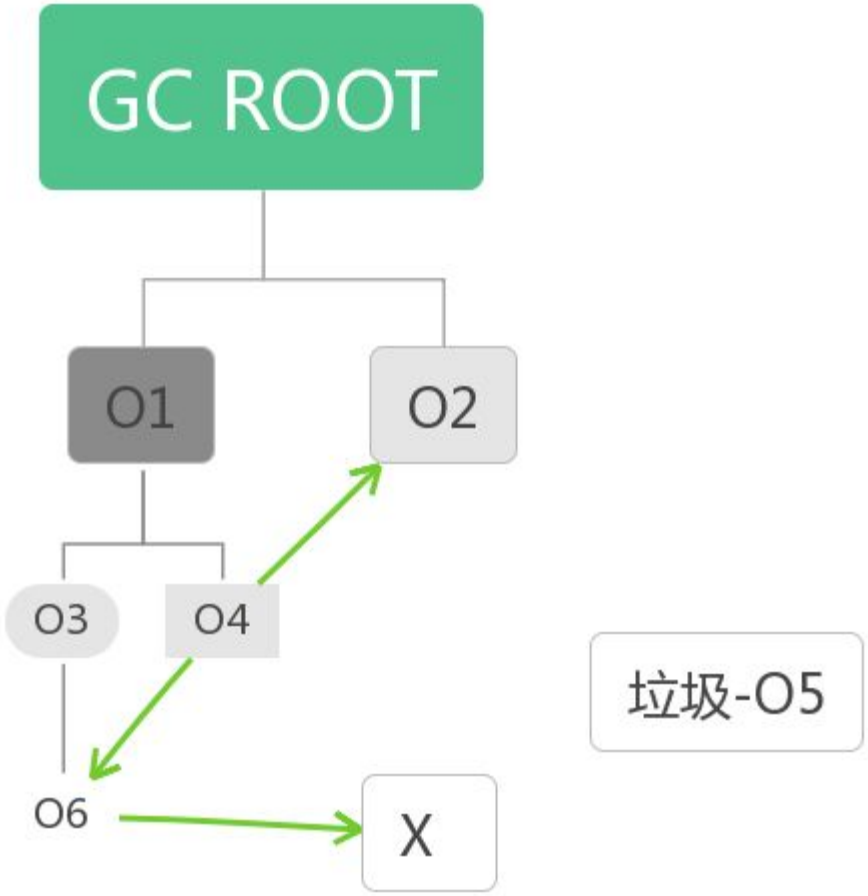
程序修改了待扫描的对象O2,O3,O4

程序修改了已扫描的对象O1

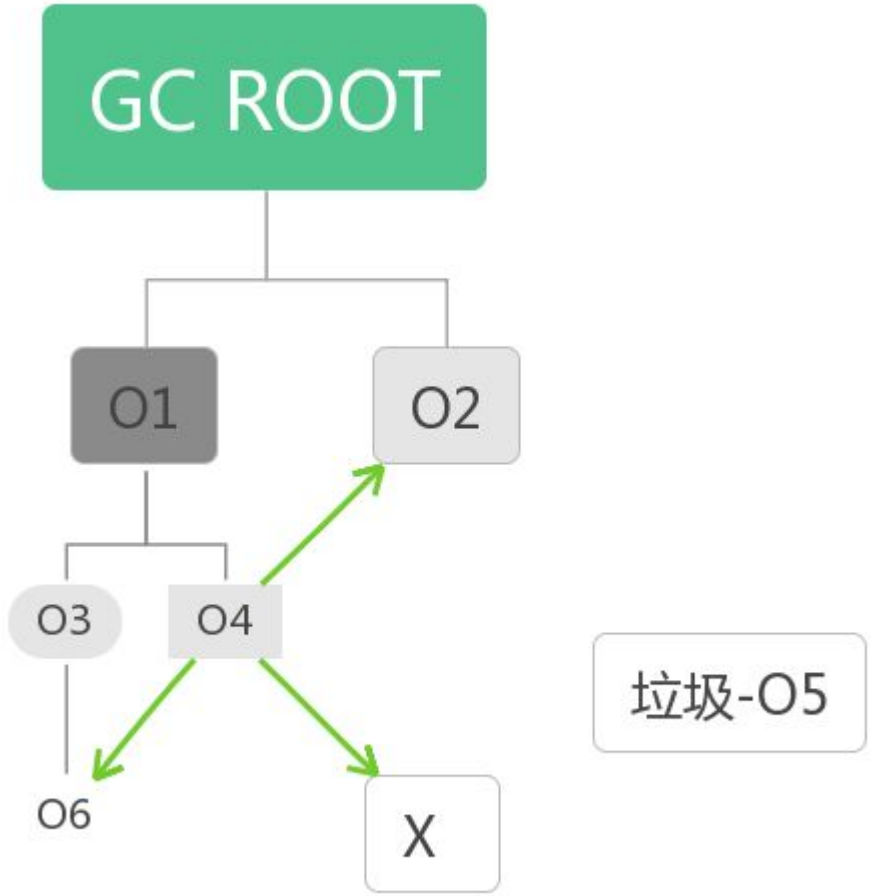


程序修改了未扫描的对象O6

无影响



程序修改了待扫描的对象O2,O3,O4



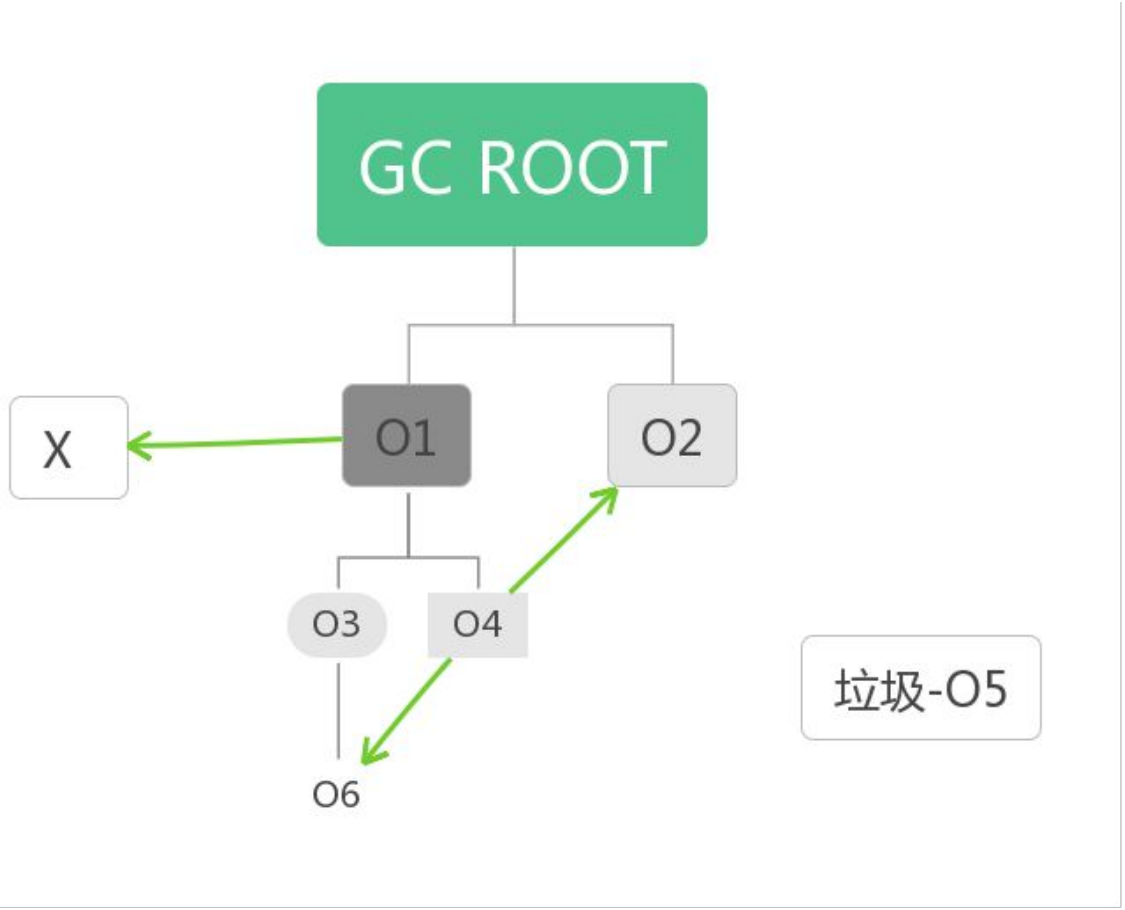
程序修改了已扫描的对象O1

如果不做任何处理

会无法标记X

而导致回收非垃圾的X

将O1重新加入WorkList中



并发清理

1. 标记了的存活的对象
2. 没标记的垃圾
3. 空闲内存

问题:从空闲内存分配对象,但未标记

方法:在对象头添加一个新的字段

并发清理

在清理过程中对象存在三种状态

1. 未被标记的垃圾
2. 被标记的存活对象
3. 新分配的对象 - 需要在对象头中加入一个标记
4. 空闲内存

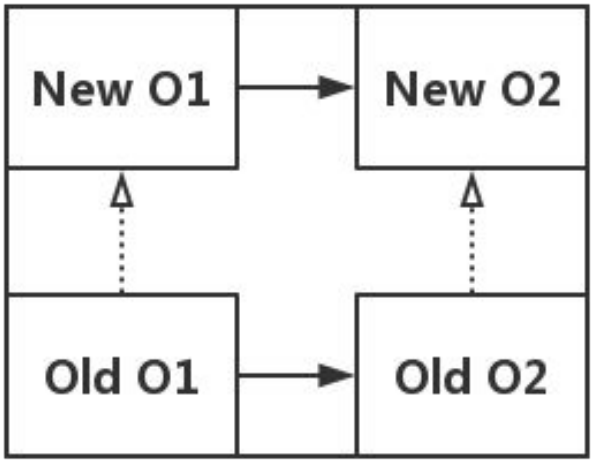
并发整理

生成转发指针

启动整理线程和应用程序

问题:

1. 读取对象时,没有复制到新的位置上;



并发整理

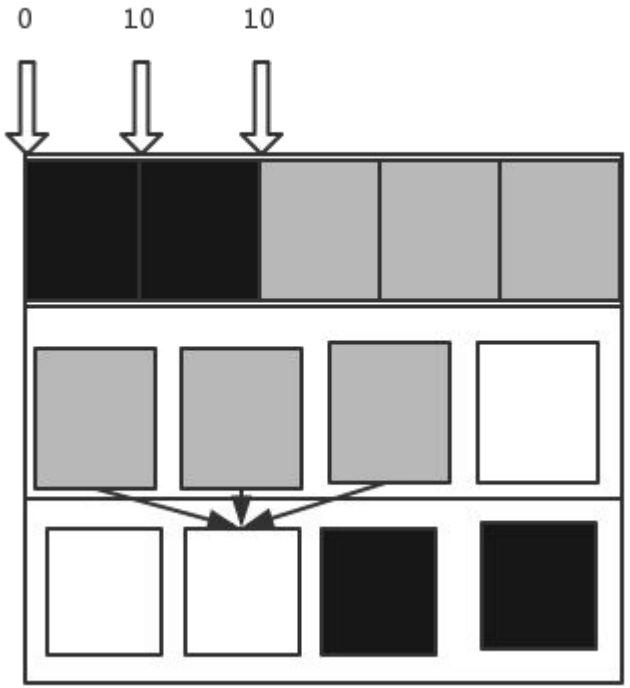
黑色:有大量存货对象

灰色:有少量存货对象

白色:空闲内存

将内存按块划分

- 如12M的内存划分成12块1M的内存块
- 可以跳过黑色块

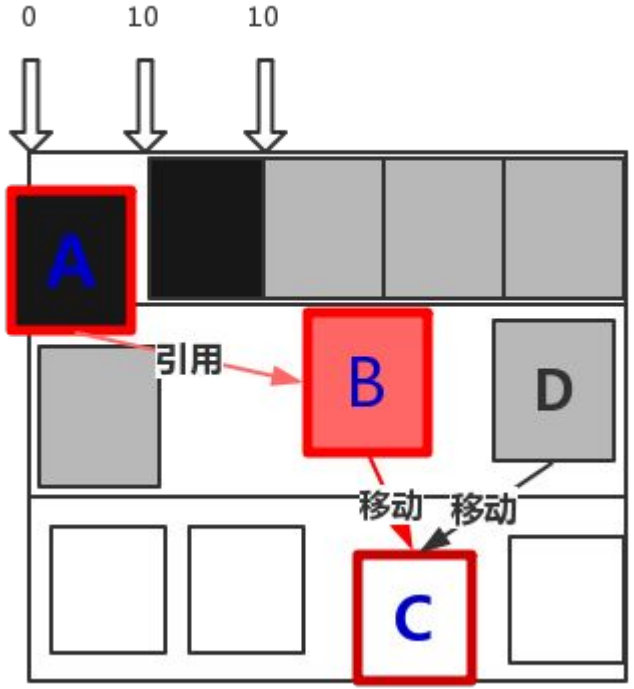


为黑色/白色内存块设置读写陷阱

- 即读取或写入该内存块的时候
- 中止该线程,将控制转交给回收线程

A:有不整理/移动的内存块

B,D:需要移动到C

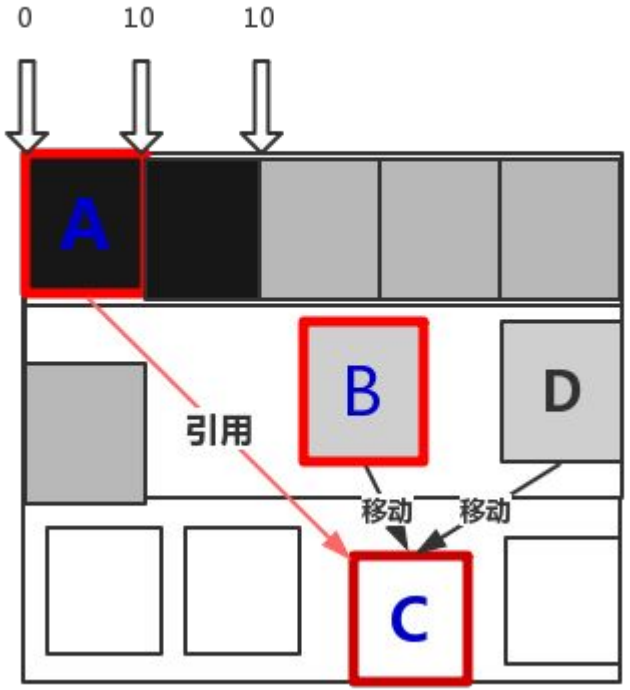


读取黑色块A的时候会陷入陷阱;

会根据转发指针

将更新A中所有存活的对象的field

取消A块上的陷阱设置;

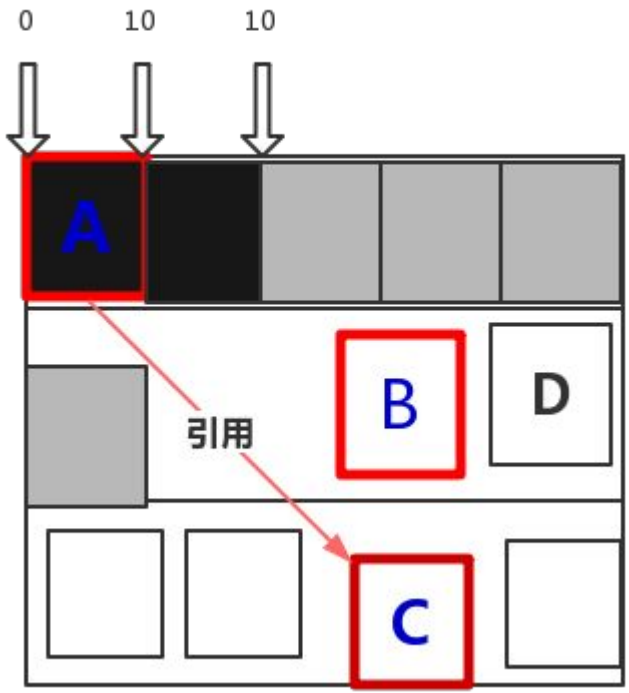


当读取C块是再次陷入陷阱:

根据转发指针将B,D块中的对象复制到C中

释放B,D块的内存

取消陷阱设置



end