JVM G1GC的算法与实现

结合实用JVM，图解Java垃圾回收机制的关键技术

JVM G1 GC设计1

Mjh啊哈
让世界倾斜

Parallel GC/Concurrent GC

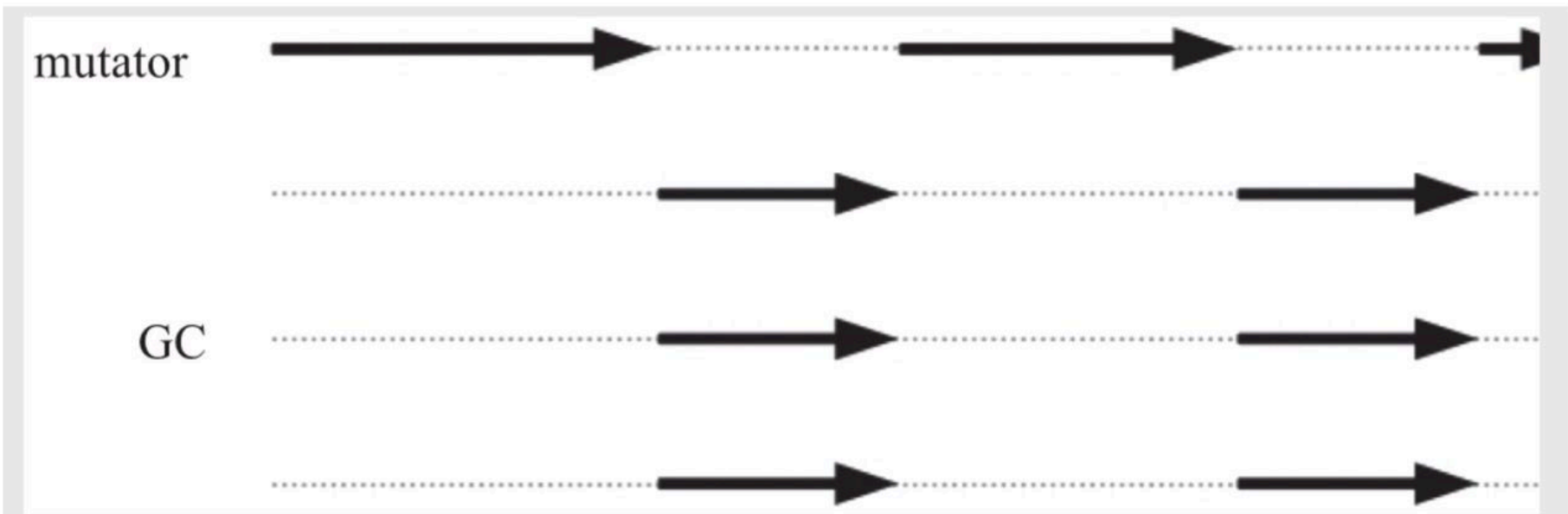
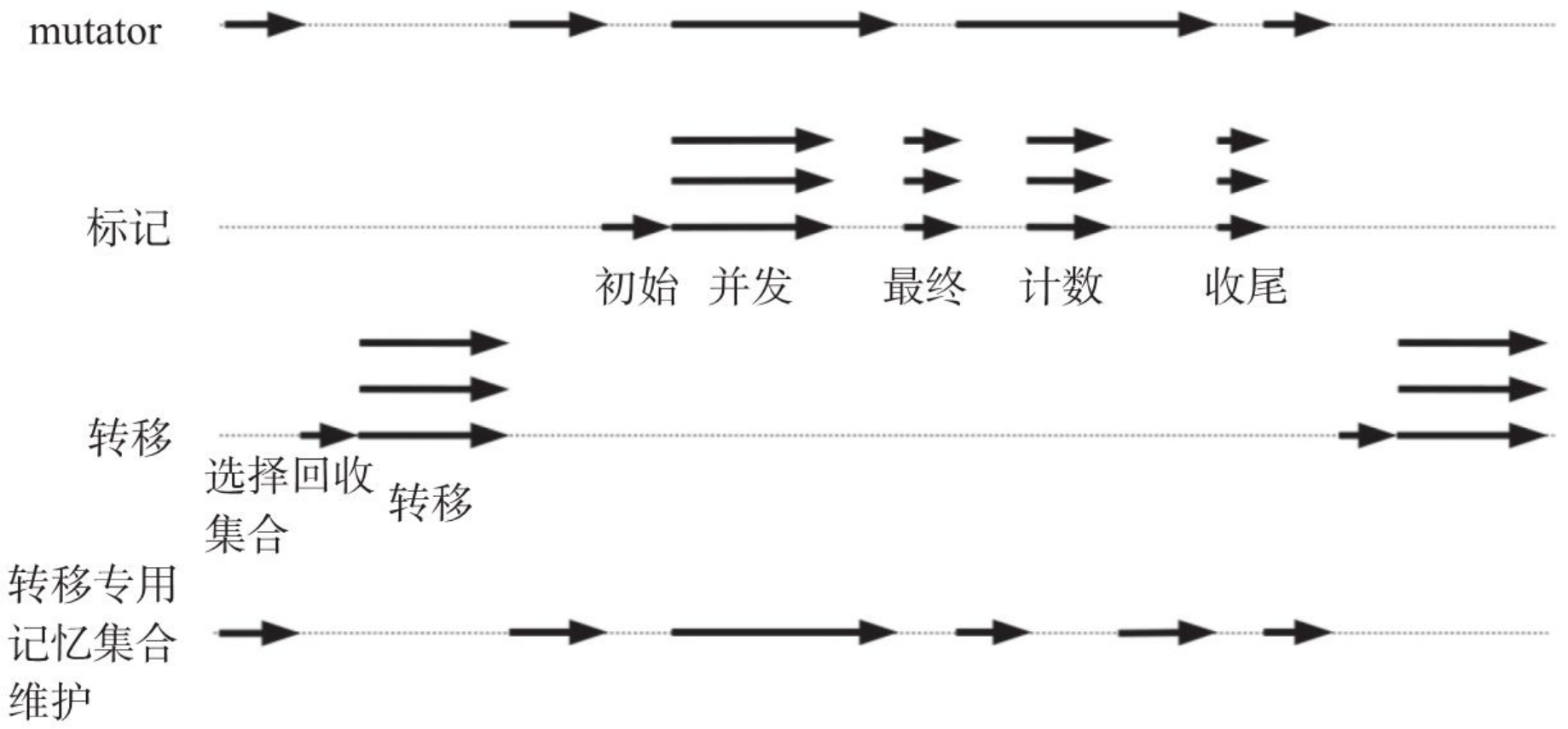


图0.2　并行GC的执行示意图

图0.3　并发GC的执行示意图

- GC算法目标：应用吞吐量（单位时间内回收垃圾的量）最大化、GC暂停时间最小化、内存占用最小化、并发开销最小化。
- **以多线程执行的GC**有并行GC/ 并发GC 2类。并行GC的目标是尽量缩短mutator（应用）的暂停时间，而并发GC的目标是消除mutator的暂停时间。
- 并发GC相比于并行GC，实现更为复杂。因为并发GC是和mutator并发执行的，所以在标记存活对象的过程中，对象的引用关系可能会被mutator改变。GC线程需要知道到这种引用关系的变化，于是并发GC采用了增量式GC中也使用的**写屏障**。
- 并行与并发2种策略都会使用。在G1中大多数时候GC线程和mutator会并发地执行GC，但是在个别阶段的处理中则需要暂停mutator，这时会启动多线程通过并行处理来缩短暂停时间。

G1



- OpenJDK 7（即Java 7）引入。
- 最大特征是重视实时性，更偏向服务端开发。实时处理必须尽力保证不超出最后期限，因此相比高速性，**可预测性**更重要，如果有超出期限的可能，就可以事先采取应对措施。
- 高效实现软实时性（尽量减少超出用户期望暂停时间的频率）。1.支持用户自定义mutator暂停时间；2.预测下次GC会导致mutator暂停多长时间。根据预测出来的结果，G1会通过延迟执行GC、拆分GC目标对象等手段来遵守设置的期望暂停时间。
- Java先前的GC算法都在一味地尝试缩短最大暂停时间，通常以对象为单位的精细粒度来处理。而G1GC则是让用户去设置期望暂停时间，在确保吞吐量（以区域卡片为粒度）比以往的GC更好的前提下，实现了软实时性（可预测并延迟回收）。
- 同时由于服务端应用程序大都运行在拥有巨大堆和多处理器上，因此GC算法必须能够在短时间内以高吞吐量来处理巨大的堆，而且还要高效地发挥多处理器的优势。
- 执行过程主要是Concurrent Marking（并发标记）与Evacuation（转移复制压缩，降低区域内碎片化），2个过程可以是独立的，转移过程可能发生在标记过程中。
- 用户可设置：1.可用内存上限；2.GC暂停时间上限；3.GC单位时间，在单位时间内遵守暂停上限，注意是任选一个单位时间（不会出现上次GC与下次GC时间段上邻近）。

三色标记

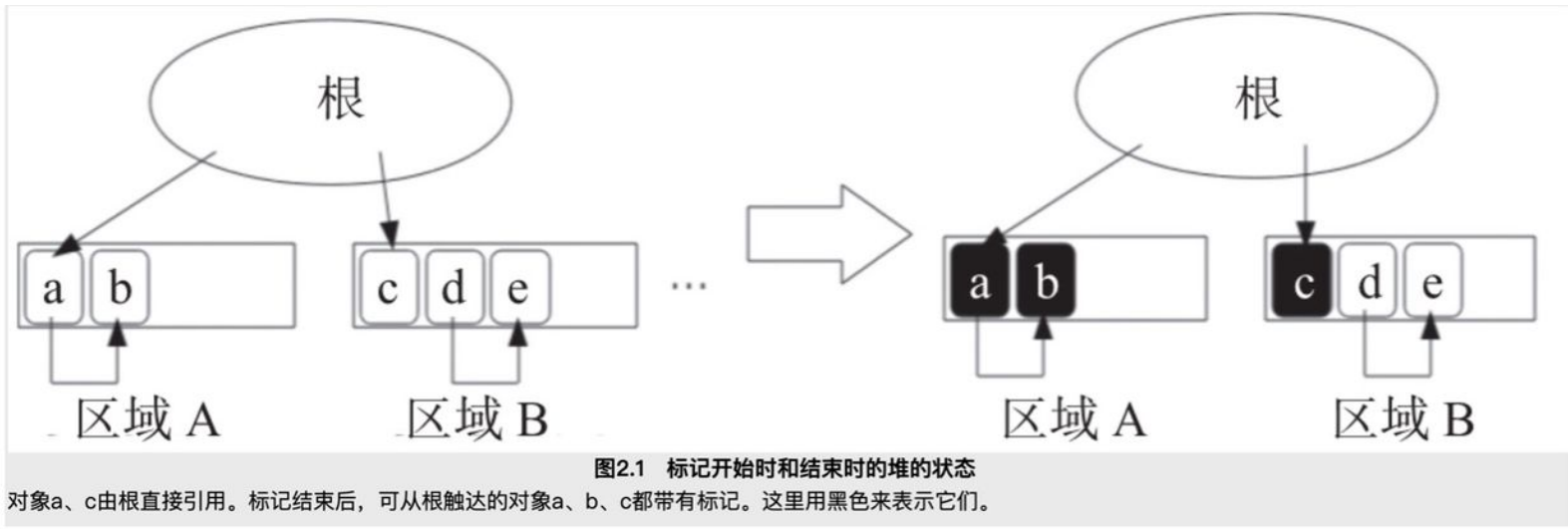
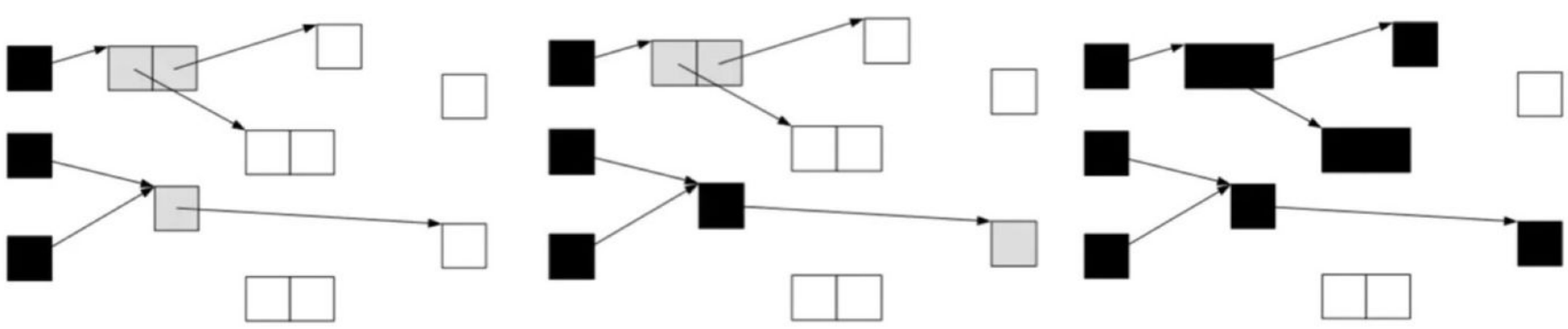


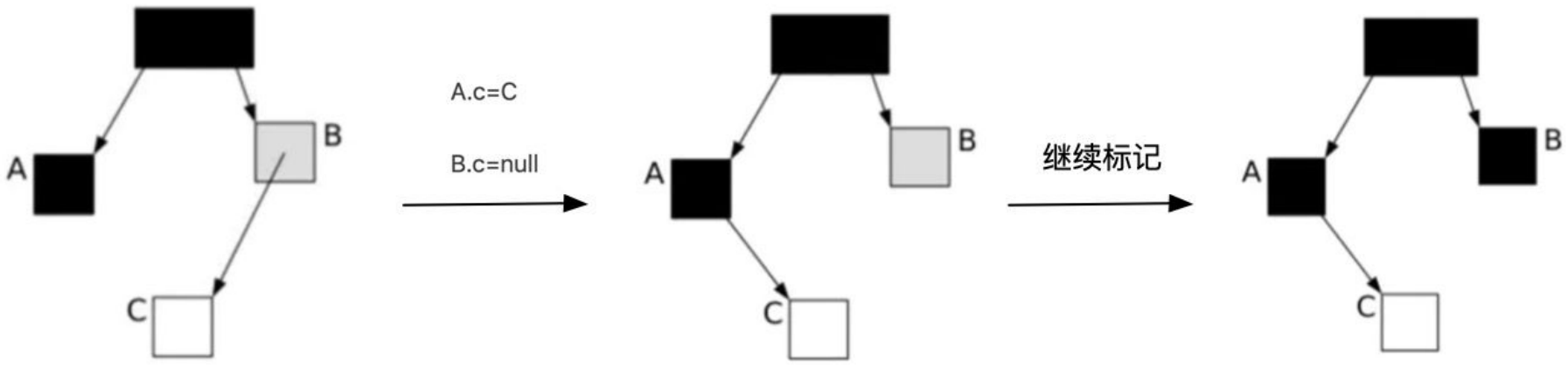
图2.1　标记开始时和结束时的堆的状态

对象a、c由根直接引用。标记结束后，可从根触达的对象a、b、c都带有标记。这里用黑色来表示它们。



- 黑开始为root对象，最终为黑表示对象及其引用的所有子对象都被标记了，是存活的可达对象；灰为未完成扫描的对象，其子对象还未完成标记，为中间状态；白为还未完成标记的对象，最终为白表示不可达对象，需要被回收。
- CMS和G1都涉并发标记。并发标记的问题是collector在标记对象的过程中mutator可能正在改变对象引用关系，mutator可能会修改对象引用（可能**漏标**），或者产生新对象（可能**错**

标）。比如下面漏标会造成错误，假设A完成扫描标记为黑，灰B与白C还未完成，但此时mutator修改了引用，A引用了C，A不会继续扫描导致本阶段漏扫了C，因而回收C从而出错。而错标不会影响程序的正确性，只是造成浮动垃圾，下次回收即可。



- 一个白对象在并发标记阶段被漏标的**充分必要条件**是：

1. 引用关系插入：mutator插入了一个从黑对象到该白对象的新引用（A->C）；
2. 引用关系消失：mutator删除了所有从灰对象到该白对象的直接或者间接引用（B->C）。

- 意思是本阶段后，白对象却被黑对象引用了（A->C），从而从黑出发不会被继续扫描标记，同时缺少了灰对象到白对象的路径（B->C），进而无法从灰出发去标记，从而使得白对象C不会被标记而被回收。
- 由于是充分必要条件，因此要避免对象的漏标，**只需要打破上述2个条件中的任何一个**，继续扫描标记即可。解决方式有2种：

1. 在插入关系的时候记录对象（打破第1条）。CMS的 Incremental Update 采用此方式，其利用write barrier将所有新插入的引用关系都记录下来，最后以新增关系的源为根STW地重新扫描一遍即可。比如从A开始重新扫。
2. 在删除关系的时候记录对象（打破第2条）。G1的SATB 采用此方式，SATB利用pre write barrier将所有即将被删除的引用关系的旧引用记录下来，最后以这些旧引用为根STW地重新扫描一遍即可。比如从C开始重新扫。

- 这里要注意的是，插入与消失不是对称的。除了上述情况，插入关系不一定意味着有关系删除，结果可能是共同引用，而删除关系可能会留下孤立对象。因此插入关系可能比删除关系角度去扫描的对象更多。

区域

堆的内部被划分为大小相等的区域，所有区域排列成一排。G1GC以区域为单位进行GC。用户可以随意设置区域大小，但是内部会将用户设置的值向上调整为2的指数幂（2^n），并以该正数作为区域的大小（图1.1）。

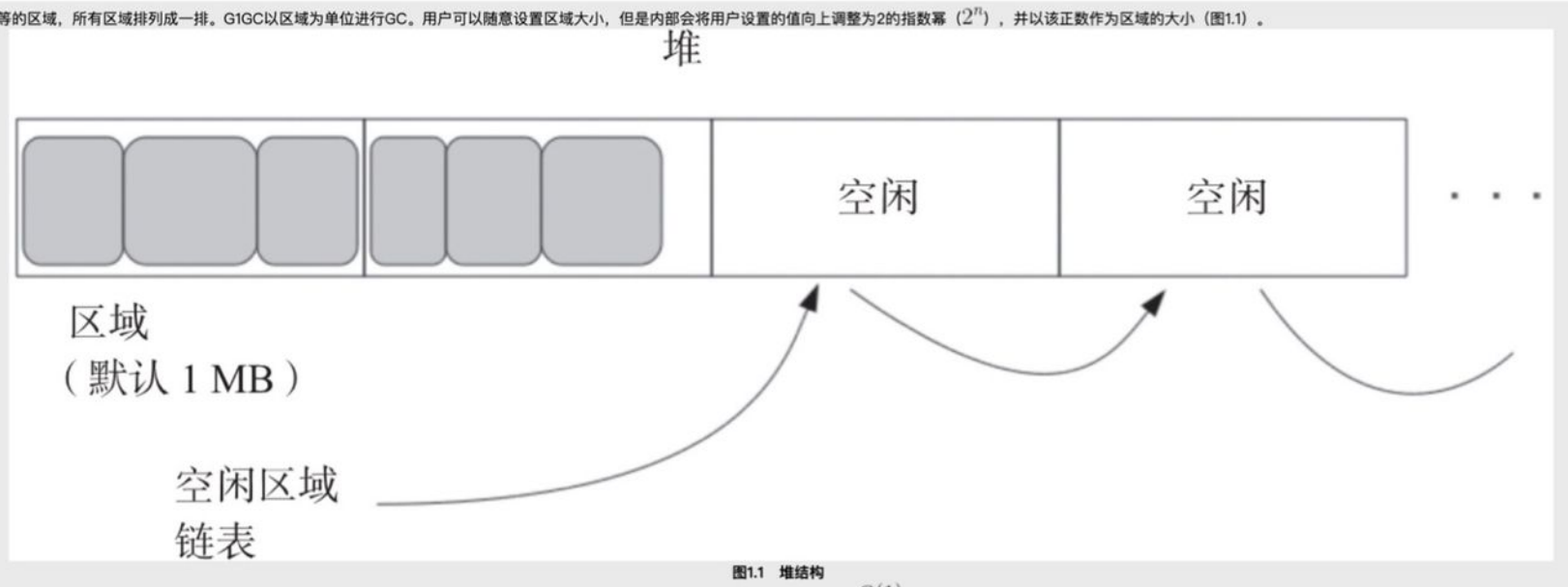


图1.1　堆结构

如果正在分配对象的某个区域已经满了，GC线程会寻找下一个空闲的区域来继续分配。空闲区域是通过链表进行管理的，因此查找的时间复杂度是固定的 $O(1)$。

首先，从众多区域中选择一个进行GC操作。如果该区域中有存活对象，则将其复制到其他空闲区域中（图1.2）。

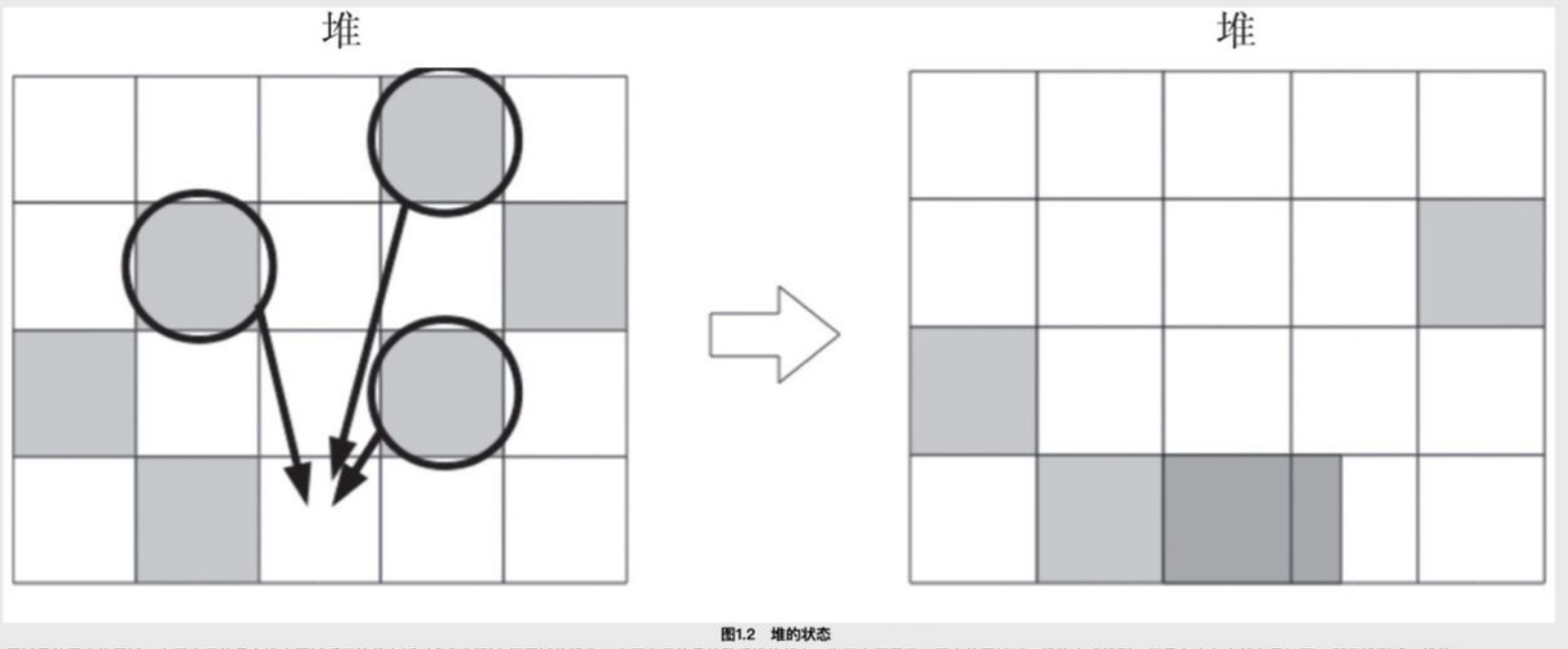


图1.2　堆的状态

白色区域是空闲区域，灰色区域是使用中的区域。左图表示的是在选中区域后开始将存活对象复制到空闲区域的操作；右图表示的是转移后堆的状态。为了方便展示，图中的区域以二维的方式排列，但是在内存中其实是如图1.1所示排列成一排的。

当选择的空闲区域也满了的时候，GC线程会再次选择其他空闲区域来存放存活对象。对象复制完成之后，只剩下死亡对象[7]的区域会被重置为空闲区域以便复用。

- G1将堆分为多个大小相等的**区域Region**。
- 空闲区域用链表管理，查找更快。

标记位图

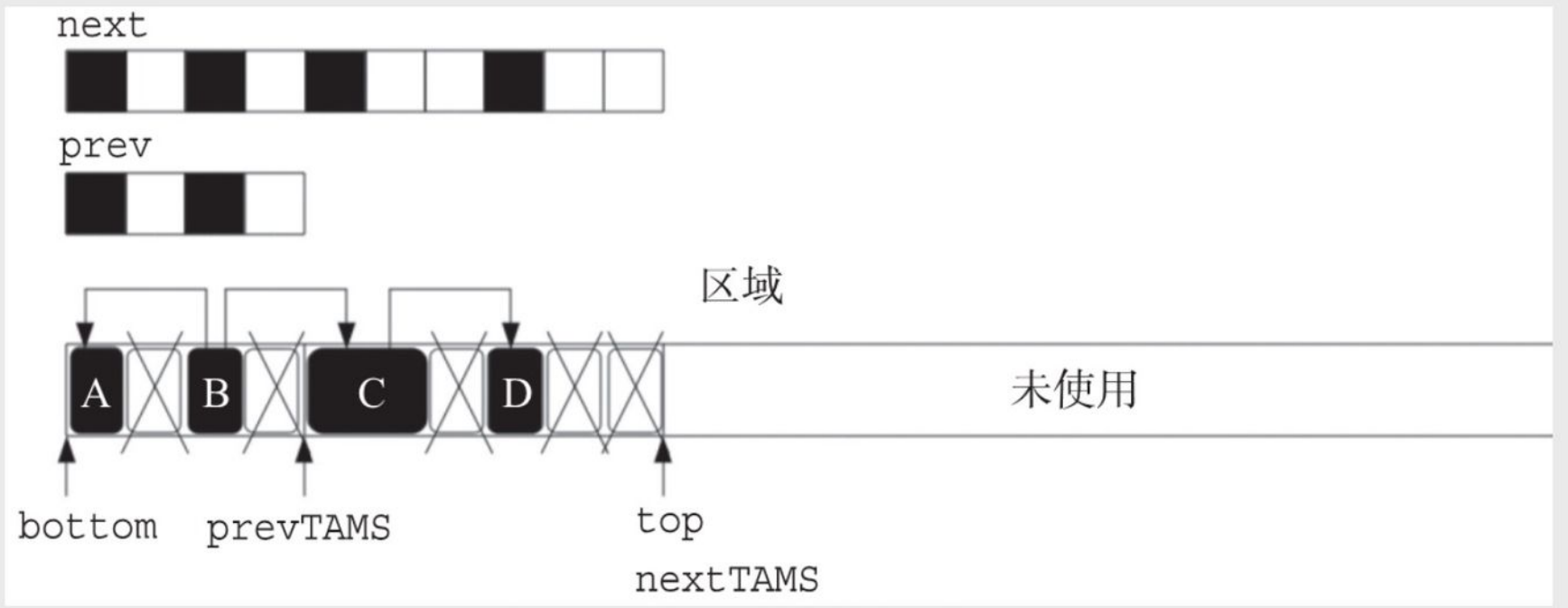


图2.2　标记位图和区域的内部结构

位图中的黑色表示已标记，白色表示未标记。黑色的是存活对象，带有叉号的是死亡对象。

每个区域都带有两个标记位图：`next`和`prev`。`next`是本次标记的标记位图，而`prev`是上次标记的标记位图，保存了上次标记的结果。

- 并发标记并不是直接在对象上添加标记，而在**标记位图**上添加标记表示是否存活。
- 每个区域有2个标记位图。next是本次标记的位图，prev是上次标记的。
- TAMS是（Top At Marking Start）。nextTAMS保存了本次标记开始时的top，而prevTAMS保存了上次标记开始时的top，用来分隔标记开始后的新生对象区域。

SATB (Snapshot At The Beginning)

图2.6表示的是obj1未被SATB专用写屏障获知时对象之间的关系。我们假定并发标记进行到了obj3。由于obj1不会被添加到SATB本地队列中，所以会保持为白色。而obj0会被添加到SATB本地队列中，所以会变成灰色。但是在后续扫描obj4时，obj1最终还是会被标记，所以不存在标记遗漏。

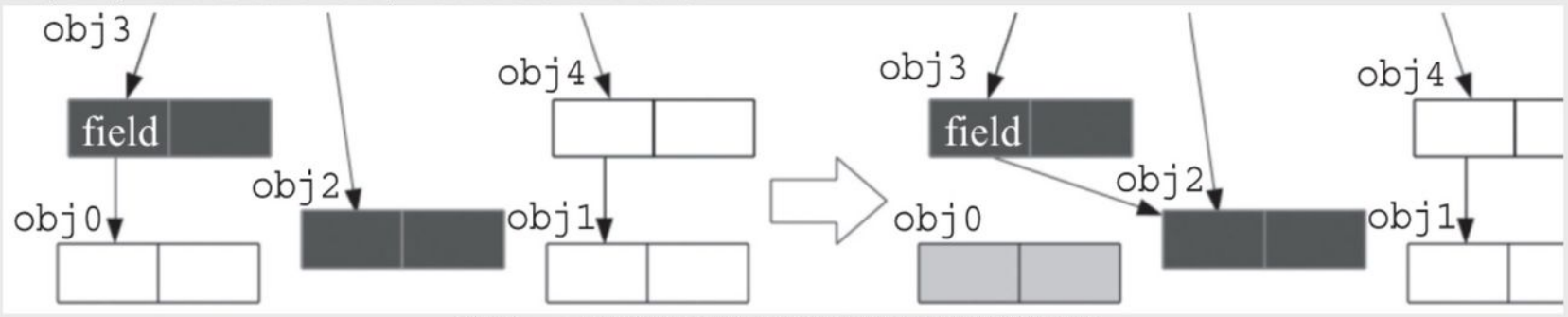


图2.6　obj1未被SATB专用写屏障获知时对象之间的关系

标记完成的对象用黑色表示；添加到SATB本地队列中的对象用灰色表示；其余对象用白色表示。

那么，如果obj1不再被obj4引用，而变为被obj2引用时，情况又是怎样的呢？图2.7进行了演示。

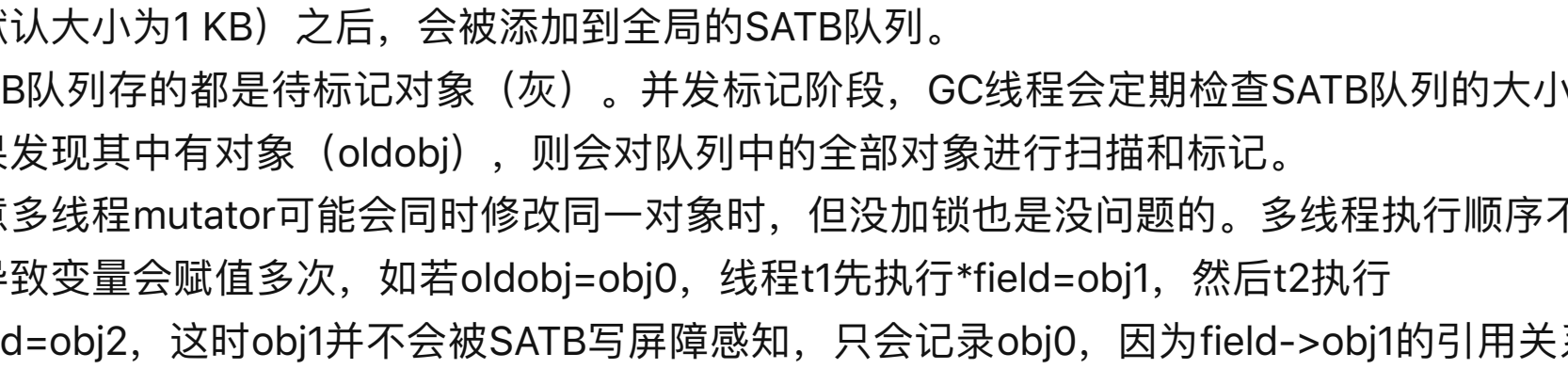


图2.7　obj1不再被obj4引用，变为被obj2引用时对象之间的关系

当来自obj4的引用消失时，obj1就会变成灰色。

在这种情况下，来自obj4的引用消失会被SATB专用写屏障获知，obj1会变成灰色，所以也不会有问题。

SATB专用写屏障会记录下并发标记阶段开始时对象之间的引用关系。这么来看，因为obj3对obj1的引用在并发标记阶段开始时并不存在，所以根本没有必要记录obj1。相反，因为obj3对obj0的引用在并发标记阶段开始时就存在，所以记录obj0是有必要的。

代码清单2.2中 (a) 到 (c) 的步骤虽然没有加锁，但是SATB专用写屏障技术严格遵守了前面这些约束条件，所以即使不记录obj1也是没有问题的。

- 以逻辑快照的形式，来保存并发标记阶段开始时对象间的引用关系图，用于后续判别是否有引用关系的消失。**什么是逻辑快照，什么时候构建？**对象本身即有保存前向引用关系。
- SATB后标记过程中新生成的对象，会在[nextTAMS, top]间，被看作“已完成扫描和标记”（快照中没有其引用关系），直接当作存活对象（解决错标），也不必为他们创建标记位图。
- 由于标记与mutator并发，引用关系会变化，因此**SATB专用写屏障**用来获知被改写的引用关系（解决漏标）。stab_local_queue是各个mutator持有的线程本地队列。SATB本地队列在装满（默认大小为1 KB）之后，会被添加到全局的SATB队列。
- SATB队列存的都是待标记对象（灰）。并发标记阶段，GC线程会定期检查SATB队列的大小，如果发现其中有对象（oldobj），则会对队列中的全部对象进行扫描和标记。
- 注意多线程mutator可能会同时修改同一对象时，但没加锁也是没问题的。多线程执行顺序不同导致变量会赋值多次，如若oldobj=obj0，线程t1先执行*field=obj1，然后t2执行*field=obj2，这时obj1并不会被SATB写屏障感知，只会记录obj0，因为field->obj1的引用关系在开始快照中并不存在，而SATB写屏障记录的是**快照内引用关系消失**。

```
1: def satb_write_barrier(field, newobj):
2:   if $gc_phase == GC_CONCURRENT_MARK:
3:     oldobj = *field // (a)
4:     if oldobj != Null:
5:       enqueue($current_thread.stab_local_queue, oldobj) // (b)
6:
7:     *field = newobj // (c)
```

并发标记过程

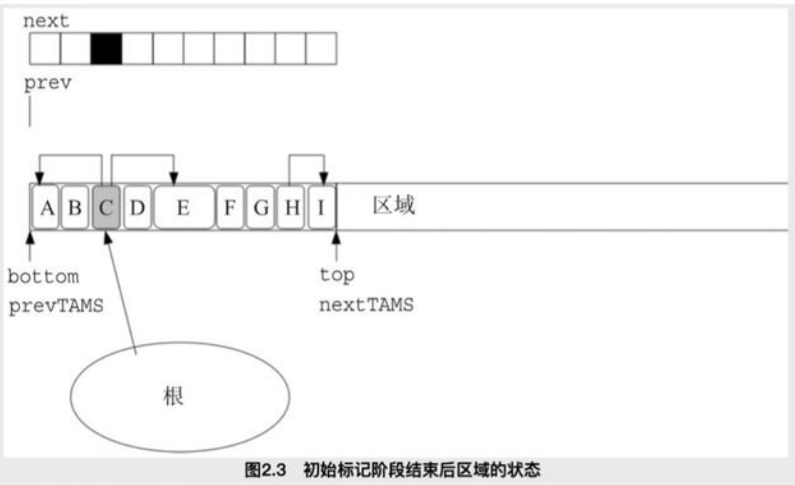


图2.3　初始标记阶段结束后区域的状态

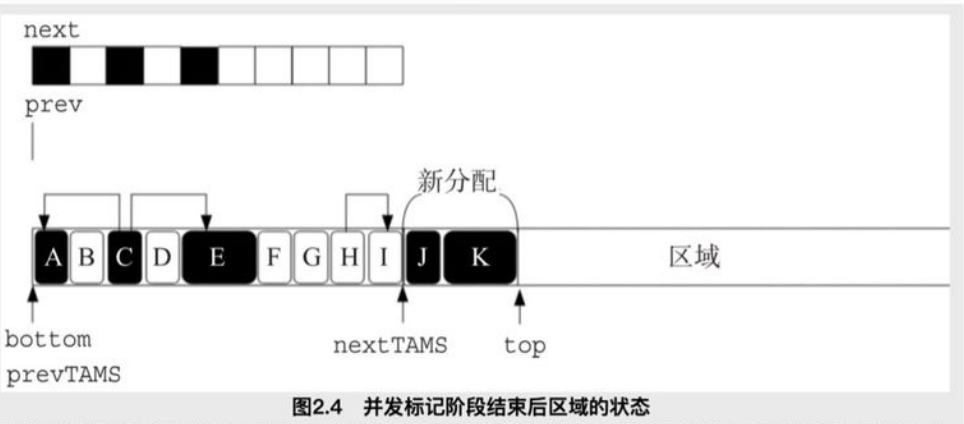


图2.4　并发标记阶段结束后区域的状态

对象C的子对象A和E被涂成了黑色。在并发标记执行期间新创建的对象J和K也在区域内被分配了空间。对象J和K也被涂成了黑色，因此会被GC当成存活对象。

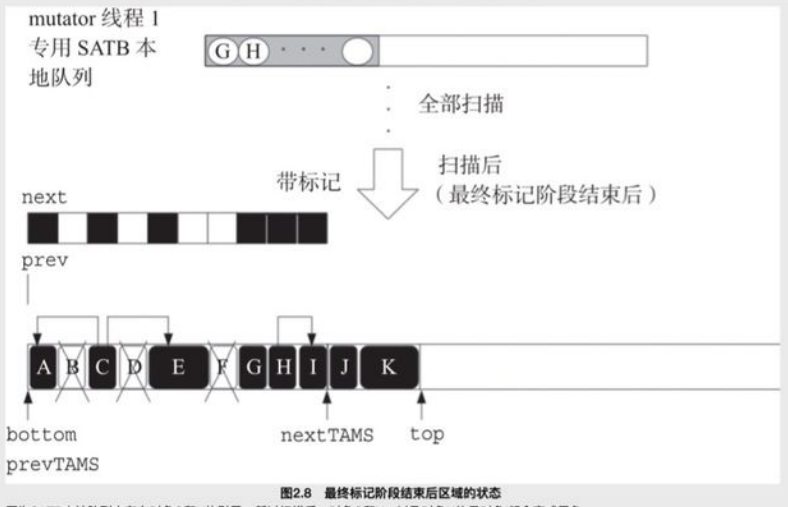


图2.8　最终标记阶段结束后区域的状态

因为SATB本地队列中存在对象G和H的引用，所以扫描后，对象G和H，以及对象H的子对象I都会变成黑色。

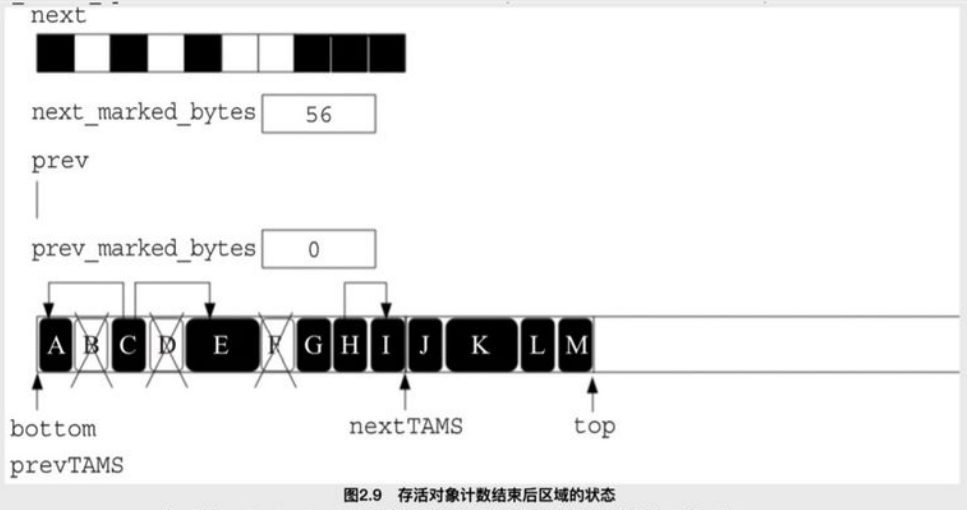


图2.9　存活对象计数结束后区域的状态

next_marked_bytes表示对象A、C、E、G、H和I的总字节数，一共56字节。计数过程中新创建了对象L和M。

- 并发标记所有存活的对象，为转移过程提供所需信息。并发标记阶段分为5步：

1. 初始标记阶段。创建所有区域的标记位图next与nextTAMS是并发，其后是STW。标记可由根直接引用的对象（灰）。由于大多数根不是对象，因此这里不用写屏障获取对象的修改，直接暂停来扫描。
2. 并发标记阶段。开启并发标记线程。并发标记线程扫描1中标记的对象，完成对大部分存活对象的标记。这里应该是利用对象的前向引用关系来搜索，并可以多线程并发。由于与mutator并发，这里利用写屏障技术记录对象间引用关系的变化。
3. 最终标记阶段。STW。扫描2中未完成标记的对象（各个线程的SATB本地队列中可能仍有待扫描的对象），最终完成后堆内所有存活对象都会被标记。
4. 存活对象计数。与mutator并发。扫描标记位图next，统计各个区域（[bottom, nextTAMS]）内存活对象的字节数存入区域内的next_marked_bytes中。计数过程中会停止转移专用记忆集合维护线程。转移处理也可能启动中断这个过程，但要当前区域计数完成才可进行转移处理。
5. 收尾。STW。扫描每个区域，进行没有存活对象区域的清除，还会统计每个区域的转移效率（一般死亡对象多，转移效率高）并排序，后续转移会利用此选择回收集合。更新prevTAMS=nextTAMS与prev位图=next位图，并重制next。

转移专用记忆集合

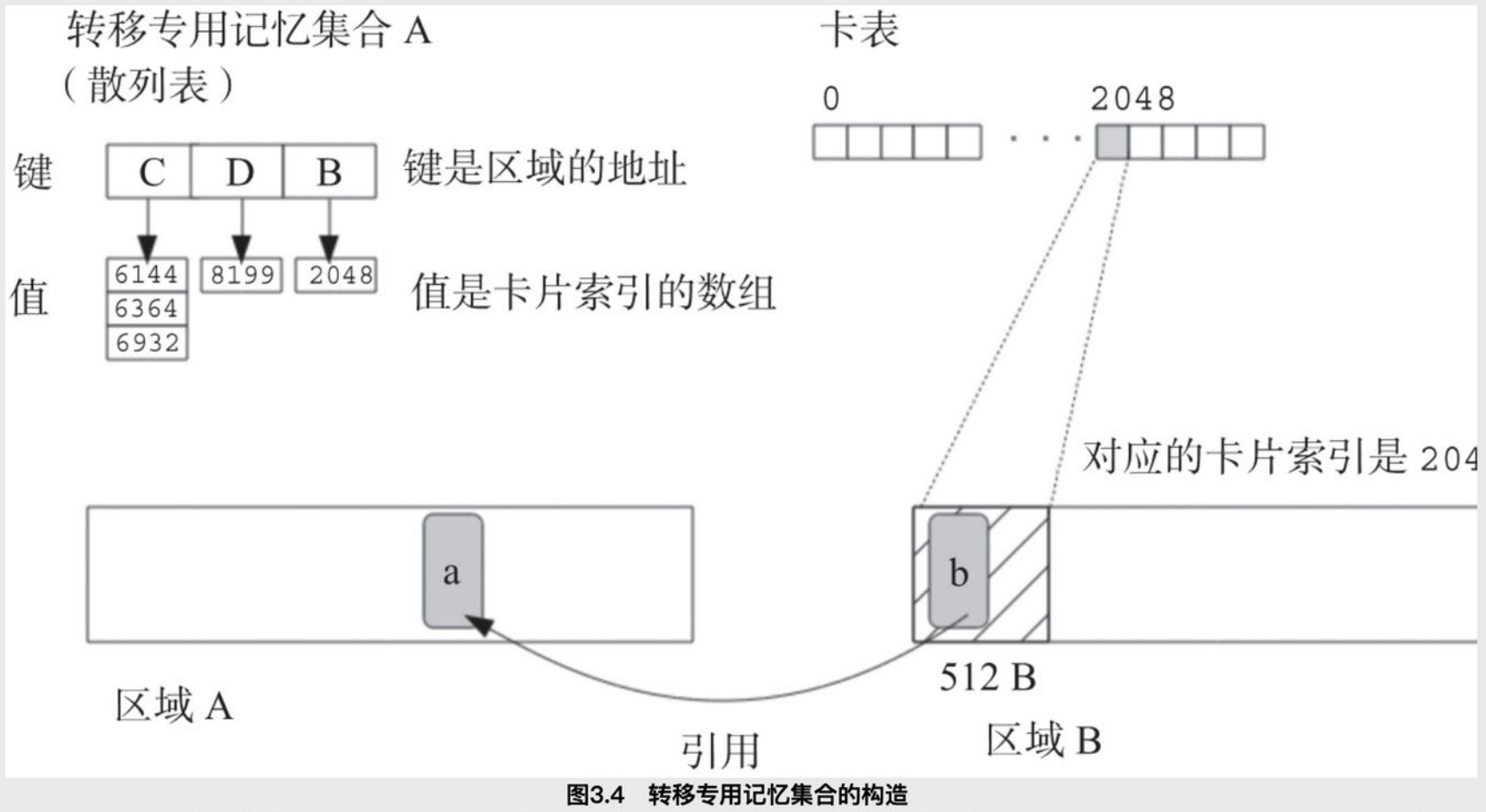


图3.4　转移专用记忆集合的构造

每个区域都有一个转移专用记忆集合，它是通过散列表实现的。图中对象b引用了对象a，因此对象b所对应的卡片索引就被记录在了区域A的转移专用记忆集合中。

- 转移操作需要转移所选区域内的所有存活对象到空闲区域。
- SATB队列集合用来记录标记过程中对象之间引用关系的变化，而**转移专用记忆集合RSet**（Remember Set）中记录了来自其他区域的粗粒度的**反向引用关系**，因此在转移时即使不扫描所有区域内的对象（基于前向引用搜索），也可以确定待转移对象所在区域内的存活对象（基于RSet的反向引用搜索）。注意RSet记录的不是完整的对象反向引用关系，在分代回收中新生代间对象引用关系不计入CSet。
- 维护RSet有开销，为什么要用RSet来反向引用？按理可从CSet的对象根据本身的前向引用搜索（类似于标记过程），也可多线程处理，并且活对象引用的对象也是活对象，活对象反向引用的对象不一定是活对象。个人理解是因为，转移后的对象需要更新引用方的域地址，这个通过反向关系比较好找父对象，并且容易判断父对象是否在CSet。而通过前向关系需要从根开始扫描整个区域来搜索路径，有很多无效路径判断。
- **RSet什么时候构建？SATB开始时?**
- 卡表Card Table粗粒度地表示区域，由数组构成，每个索引表示堆中一段大小区域的卡片（比为1B:512B）。有净卡片与脏卡片2种。**脏卡片**指的是已经被转移专用写屏障添加到转移专用记忆集合日志中的卡片。
- 每个区域有RSet，通过散列表实现。points-into结构（**反向引用或者依赖**），散列表的键是引用本区域的其他区域的地址，而散列表的值是一个数组，数组的元素是引用方的对象所对应的卡片索引。区域间完整的对象的引用关系是由RSet以卡片为单位粗略记录的，注意RSet没有记录来自根的引用。
- 当对象的引用被修改时（非删除），被修改对象所对应的卡片会被**转移专用写屏障**记录到RSet中。每个mutator线程都持有本地的**转移专用记忆集合日志RSLog**的缓冲区，其中存放的是卡片索引的数组。当对象引用被修改时，写屏障就会获知，并会将对象所对应的卡片索引添加到RSLog中。RSLog会在写满后被添加到**全局的RSLog日志集合**中，这是多线程的竞争操作。

```
1: def evacuation_write_barrier(obj, field, newobj):
2:   check = obj ^ newobj  //obj与newobj是否在同一个区域
3:   check = check >> LOG_OF_HEAP_REGION_SIZE //区域地址为LOG_OF_HEAP_REGION_SIZE的
4:   if newobj == Null:  //对象不是被删除
5:     check = 0
6:   if check == 0:
7:     return
8:
9:   if not is_dirty_card(obj):  //检查脏卡片，避免向转移专用记忆集合日志中添加重复的卡片
10:    to_dirty(obj)  //从净卡片变成脏卡片
11:    enqueue($current_thread.rs_log, obj)  //添加到转移专用记忆集合日志rs_log，保证转
12:
13:  *field = newobj
```

- **转移专用记忆集合维护线程**是和mutator并发执行的线程，它的作用是基于RSLog集合，来维护RSet。转移专用记忆集合维护线程和mutator在大多数时间中是并发执行的（包括并发标记时），但是在存活对象计数时是暂停的。
- 频繁发生修改的存储空间所对应的卡片称为**热卡片**。热卡片可能会多次被RSLog集合维护线程处理成脏卡片，从而加重线程处理负担。可通过卡片计数表记录卡片变脏的次数，其内容在下次转移时被清空。次数超过阈值的卡片会变成热卡片，添加到**热队列**（1KB）尾部，队列满则头部淘汰当作普通卡片处理。热队列中的卡片不会被RSLog集合维护线程处理，因为即使处理了，它也有可能马上又变成脏卡片，而是被留到转移的时候再处理。

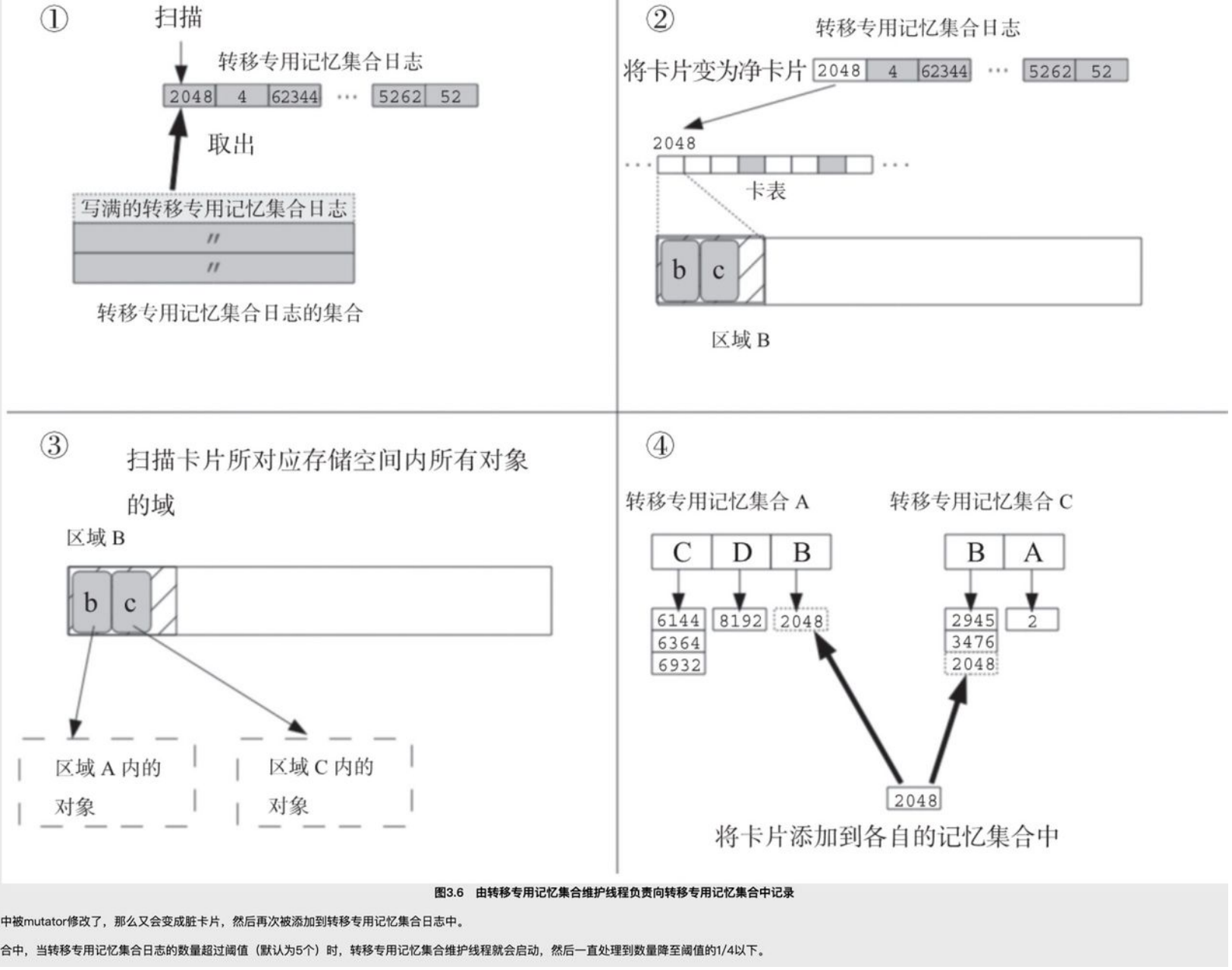


图3.6　由转移专用记忆集合维护线程负责向转移专用记忆集合中记录

如果卡片在③和④的处理过程中被mutator修改了，那么又会变成脏卡片，然后再次被添加到转移专用记忆集合日志中。

在转移专用记忆集合日志的集合中，当转移专用记忆集合日志的数量超过阈值（默认为5个）时，转移专用记忆集合维护线程就会启动，然后一直处理到数量降至阈值的1/4以下。

转移过程

- 转移处理可能发生在并发标记中暂停处理以外的所有时刻。比如在并发标记阶段或者存活对象计数的过程中，都可能中断并执行转移。

1. 选择回收集合。STW。根据并发标记提供的信息来依次选择被转移的区域，即**回收集合CSet**（Collection Set），注意转移只转移CSet内所有存活对象。以转移效率从高到低排序（垃圾越多的区域效率越高）选择CSet，同时当已选区域预测暂停时间的总和快要超过用户的容忍范围时，选择会停止。转移开始会中断并发标记过程。
2. 根转移。STW，多线程。这时已有完整的最新引用关系图，需要解决怎么找到CSet内所有存活对象，包括CSet内由根直接引用的对象、CSet内对象的其他区域的直接引用方并也在CSet的对象。直接扫描CSet内所有对象代价太大，这里根据CSet内区域的RSet指向的卡片的引用关系进行初步搜索。
3. 转移转移队列内对象。STW，多线程。BFS迭代搜索CSet内所有对象，此步骤完成后回收集合内的所有存活对象就转移完成了。

```
1: def evacuate_roots():
   //转移CSet内被根引用的对象。注意并发标记使用的SATB队列内对象引用也要转移，需要被修改为转移后
2:   for r in $roots:
3:     if is_into_collection_set(r):
4:       *r = evacuate_obj(r)  //转移对象操作，返回转移后对象的新地址，更新引用地址
5:
     //扫描RSLog集合与热卡片所对应的脏卡片更新到RSet。这是由于转移开始时，转移专用记忆集合维护
6:   force_update_rs()
     //虽然只转移CSet内区域对象，但这里先通过RSet反向查找区域卡片，而不是直接扫描CSet内所有对
     //是因为直接扫描区域内所有对象再迭代搜索代价太高，因此只选择有引用关系的区域。
7:   for region in $collection_set: //只扫描转移CSet内区域的对象
8:     for card in region.rs_cards: //转移RSet对应的的其他区域的直接引用方。通过区域间引用
9:       scan_card(card) //扫描区域卡片内每个对象并转移
10:
11: def scan_card(card):
12:   for obj in objects_in_card(card): //扫描区域卡片内每个对象。card相当于BFS的根。
13:     if is_marked(obj):  //标记的存活对象才会被转移。
```

```
        //前向扫描引用的直接子对象，若也存在于CSet则也需要转移。注意非递归，后续会BFS处理。
14:     for child_ref in children(obj):
15:       if is_into_collection_set(child_ref):
16:         *child_ref = evacuate_obj(child_ref)
```

转移对象

1. 针对转移对象a引用了CSet内的对象b（a-[a.b_ref]->b），a.b_ref是a内对b的引用feild，a.b_ref需要被添加到转移队列，因为b接下来会被转移，a.b_ref后续需要更新为新地址。这里有个问题，前面evacuate_roots根扫描了所有CSet内的对象，为什么这里还要加入转移队列？这是因为转移后会使前面转移的引用方的域变旧，比如a->b->c，转移a后再转移b，b转移后a中a.b_ref需要修改为b的新地址。
2. 针对转移对象a引用了CSet外（区域不会转移因而不会变）的对象c（a->c），需要更新c所在区域的RSet，维护引用关系。
3. 针对转移对象a被引用的引用方d（d->a），需要更新d到转移后的a所在区域的RSet，维护引用关系。
4. 通过*ref=evacuate_obj(ref)更新旧位置的引用域（可能因为共同引用有多个）。

```
1: def evacuate_obj(ref): //注意从引用方开始找
2:   from = *ref //ref为待转移对象的引用域（注意不要理解为对象），from为待转移对象
3:   if not is_marked(from): //若是未标记的死亡对象直接返回
4:     return from
5:   if from.forwarded: //对象已经被转移。转移后的目标地址指针是forwarding
6:     add_reference(ref, from.forwarding) //更新ref到forwarding/to所在区域的RSet
7:     return from.forwarding
8:
9:   to = allocate($free_region, from.size) //将对象复制到转移目标区域
10:  copy_data(new, from, from.size) //保存了前向引用关系，后面要更新反向引用关系RSet
11:
12:  from.forwarding = to //将新地址存入到forwarding指针
13:  from.forwarded = True
14:  //ref->(from/to)-[child_ref]->child，注意child_ref为to内child的引用field。
15:  for child_ref in children(to): //扫描已转移完成的对象的引用树的子对象。活对象引用的对
16:    if is_into_collection_set(child_ref): //CSet中child所在区域会变化，后续处理
17:      enqueue($evacuate_queue, child_ref) //子对象ref添加到转移队列进行BFS搜索
18:    else:
19:      add_reference(to, child_ref) //更新to到child所在区域的RSet
20:
21:  add_reference(ref, to) //更新引用方ref到to所在区域的RSet
22:  return to


1: def evacuate(): // 步骤3，转移转移队列内对象，BFS处理。
2:   while $evacuate_queue != Null:
3:     ref = dequeue($evacuate_queue)
4:     *ref = evacuate_obj(ref) //更新引用地址
```

软实时性

在G1GC中，由GC导致的mutator的暂停时间称为**消耗**。转移回收集合的消耗，等于扫描转移专用记忆集合中的卡片时的消耗与对象转移时的消耗之和。具体公式如下所示。

$$V(\mathrm{cs}) = V_{\mathrm{fixed}} + U \cdot d + \sum_{r \in \mathrm{cs}} (S \cdot rsSize(r) + C \cdot liveBytes(r))$$

公式中各个数值的含义如下。

cs：回收集合
$V(\mathrm{cs})$：转移回收集合（cs）的消耗
V_{fixed}：固定消耗
U：扫描脏卡片的平均消耗
d：转移开始时的脏卡片数
S：查找卡片内指向回收集合的引用的消耗
r：区域
$rsSize$：区域的转移专用记忆集合中的卡片总数
C：对象转移时（每个字节）的消耗
$liveBytes$：区域内存活对象的总字节数（大概的值）

$V(\mathrm{cs})$ 表示某个回收集合（cs）的转移时间。V_{fixed} 表示转移过程中的固定消耗，主要是选择和释放回收集合时的消耗。

V_{fixed}、U、S、C 这几个值的大小受实现方法、运行平台以及应用程序特性等各种环境因素的影响，是可变的。因此可以先根据经验设置一些初始值，再通过测量各自的实际处理时间来进行修正，以提高精度。

$liveBytes$ 使用2.7节中设置过的值。对于在并发标记结束后被分配（allocated）的对象，即使是死亡对象，也要将其当作存活对象来计数。因此 $liveBytes$ 并非精确值，只是大概的值。

理解了各个变量的值之后，公式的含义就简单易懂了。$S \cdot rsSize(r) + C \cdot liveBytes(r)$ 是一个区域的转移消耗。$\sum_{r \in \mathrm{cs}} (\dots)$ 是回收集合内所有区域的转移消耗的总和。$U \cdot d$ 是对3.8节中介绍的剩余脏卡片进行扫描的消耗。这些消耗再加上 V_{fixed}，就是总体的消耗了。

- 在添加CSet时，会根据公式预测区域的转移暂停时间。
- GC暂停调度。堆内空间充足时，可以根据需要扩展堆，从而延迟转移处理。同时并发标记结束后不一定转移处理。根据调度队列处理，元素是暂停时间段，保证单位时间内暂停时间不超过上限。
- 并发标记阶段的暂停时间，不像转移暂停时间那样可控，可根据经验设置。
- 但是GC预测时间不准或者堆空间不足时，导致GC会提前开始，GC暂停处理还是会超过暂停时间上限。

分代G1GC

完全新生代GC不会选择老年代区域，而是将所有新生代区域都选入回收集合，然后转移回收集合内的存活对象。晋升的对象会被转移到老年代区域，其余对象则被转移到存活区域（图5.2）。

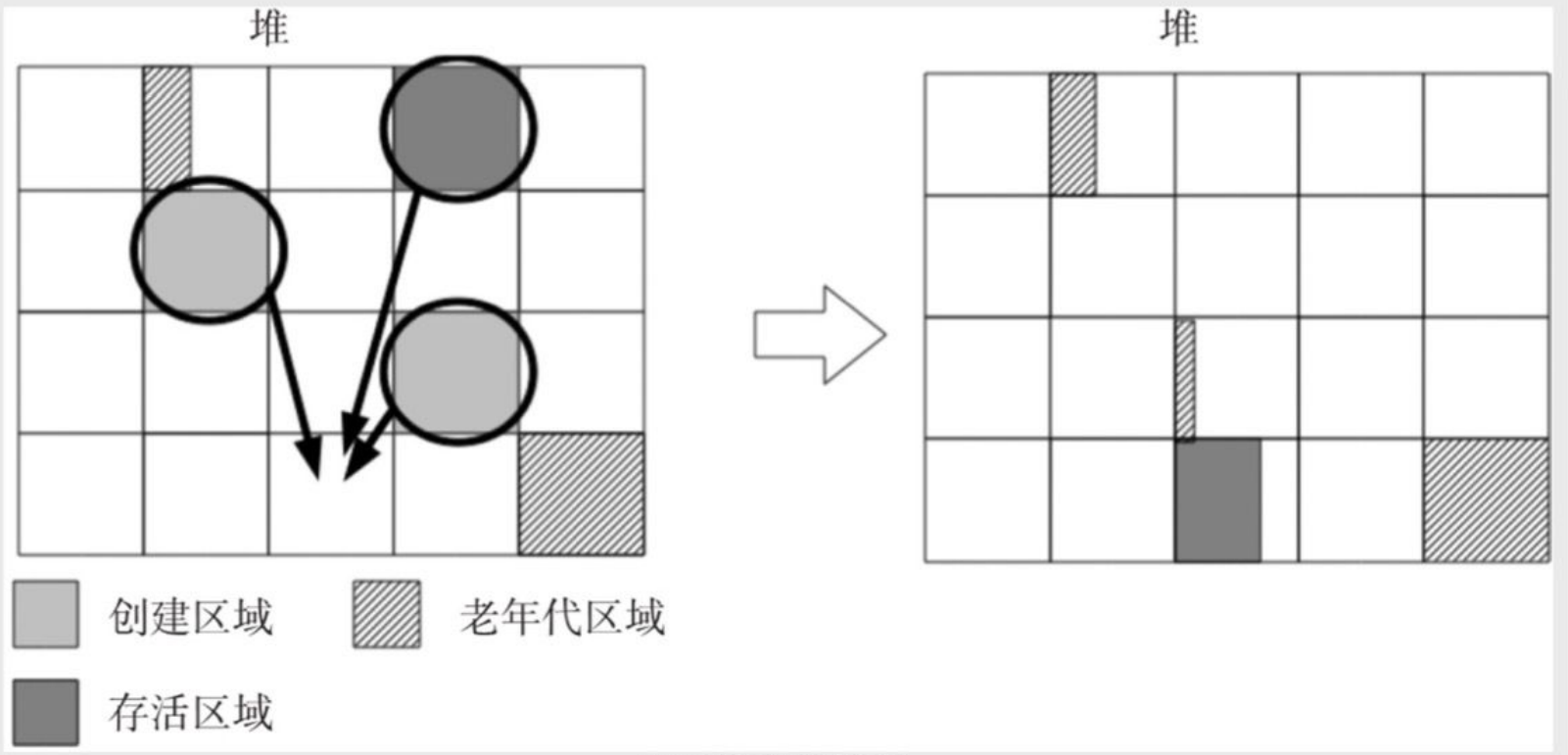


图5.2　完全新生代GC的执行过程简述

只有新生代区域会被选入回收集合，老年代区域则不会被选入。

部分新生代GC则是除了所有新生代区域外，还会选择一部分老年代区域进入回收集合。除了回收集合的选择方式，部分新生代GC和完全新生代GC的执行过程是一样的（图5.3）。

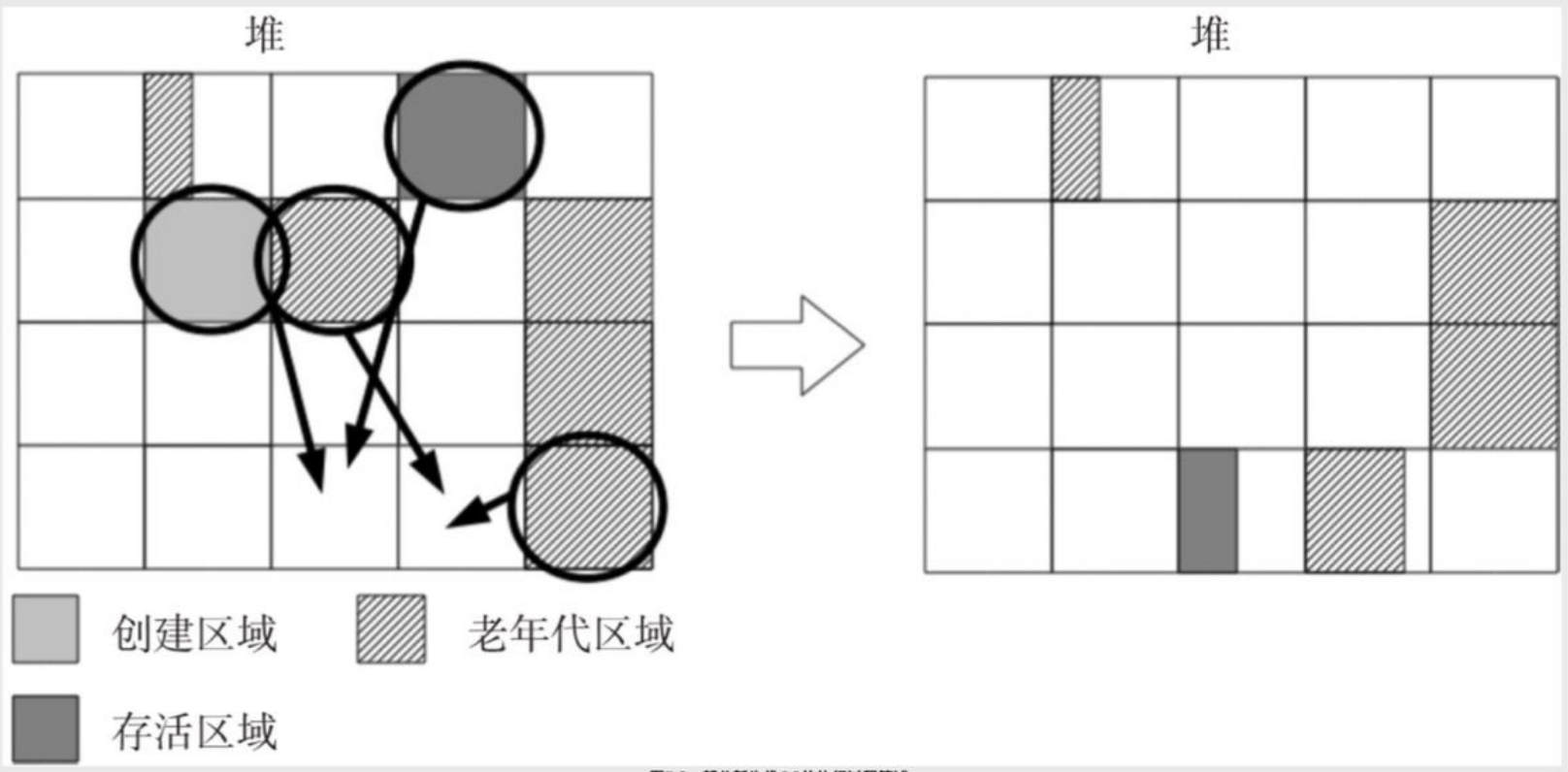


图5.3　部分新生代GC的执行过程简述

所有新生代区域和一部分老年代区域会被选入回收集合。

- 区域被分为新生代与老生代。回收集合也是分代的。
- 完全新生代回收只回收新生代，而部分新生代回收会回收部分老生代，但是所有新生代区域都会被选入CSet，因此CSet可能有很多新生代对象。大多数对象都是朝生夕死，优先转移新生代区域，积极释放空间。
- 新生代区分为Eden与Survior。Eden区域只存放刚生成且从未转移的对象，Survivor区存放至少转移过一次的对象。

- 存活对象保存有**被转移次数Age**。Age低于阈值则转移到Survior区，否则转移到Old区，转移到老年代区就是**晋升**。
- 新生代区对老年代区的引用，转移专用写屏障是无效的，老年代区域的RSet不会记录。但是来自老年代区域对象的引用，会计入新生代区域的RSet。同时新生代区域都会被选入到CSet，而老年代未必，因此在转移新生代区域时所有对象的引用关系都会被检查，RSet不必记入重复信息，因此只有Old->Young的RSet反向引用，同时有RSet不必扫描所有的Old区。
- 有时选择太多新生代区域到CSet，可能打破软实时性，因此需要计算**最大新生代区域**。完全新生代回收会尽可能转移更多新生代区域，根据过去转移记录获取转移单个新生代区域所需时间，然后计算出不超过暂停时间上限的最大新生代区域数。部分新生代回收会尽可能少的转移新生代区域，先根据暂停调度计算出下次GC调度的时机，然后计算出此时机前可回收的区域数。调节最大新生代区域数，可控制转移执行。
- 并发标记结束后，选择最大新生代区域设置，以及是否采用完全新生代回收还是部分新生代回收。根据过去经验预测完全新生代转移效率，若更高则采用完全新生代回收。
- 转移完成后，通过以下检查可开始并发执行：1.不再并发标记过程中；2.并发标记结果上次转移已使用；3.已使用一定量堆内存（默认45%）；4.相比上次转移完成后，堆内存使用量有所增加。

参考

- 深入Java虚拟机：JVM G1GC的算法与实现。[日]中村成洋
- authorNari / g1gc-impl-book
- Java GC：干掉 CMS，G1 才是未来

发布于 16 分钟前

Java 虚拟机（JVM） Java GC垃圾回收（计算机科学）

还没有评论

写下你的评论...

赞同 添加评论 分享 喜欢 收藏 设置 投稿

知乎工作？请发送邮件到 jobs@zhihu.com