SATB的一些理解

空无 发布于 2021-02-28

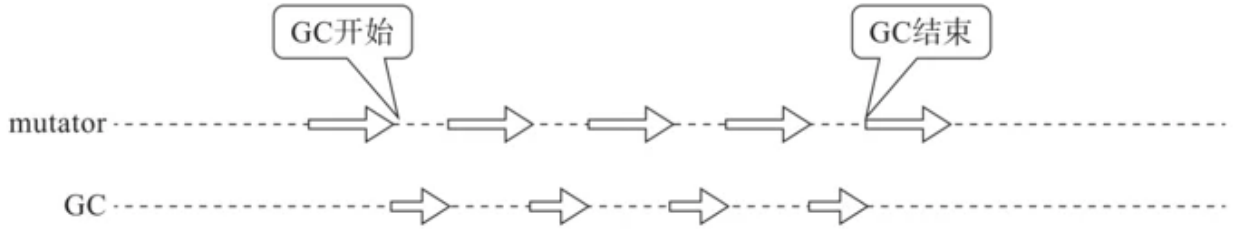
啃G1的时候，SATB(Snapshot At The Beginning)这个术语看我的很是迷糊。简单解释是：“GC开始时对象关联的快照”，但这个解释……貌似有点歧义。查阅了不少资料，也全都一笔带过不解释

下面结合一些前置的垃圾回收知识，总结一下我对SATB的理解

前置知识

增量式垃圾回收

增量（incremental）式垃圾回收，是指GC和Mutator交替运行的一种垃圾回收工作方式，如下图所示



Hotsopt中的G1就是一款增量式的垃圾回收器，老一代的CMS也有增量模式。在增量式垃圾回收下，可以降低Mutator的暂停时间，只是吞吐量没那么高（只是说增量回收下，并不是指G1）

三色标记法

三色标记法（Tri-color marking）是Edsger W. Dijkstra 等人提出的，在增量式垃圾回收里非常有用，用于标记GC过程中不同阶段的对象的状态。

- 白色：还未搜索过的对象
- 灰色：正在搜索的对象
- 黑色：搜索完成的对象

> GC 开始运行前所有的对象都是白色。GC 一开始运行，所有从根能到达的对象都会被标记，然后被堆到栈里。GC 只是发现了这样的对象，但还没有搜索完它们，所以这些对象就 成了灰色对象。

> 灰色对象会被依次从栈中取出，其子对象也会被涂成灰色。当其所有的子对象都被涂成 灰色时，对象就会被涂成黑色。 当 GC 结束时已经不存在灰色对象了，活动对象全部为黑色，垃圾则为白色。 这就是三色标记算法的概念。

> 有一点需要我们注意，那就是为了表现黑色对象和灰色对象，不一定要在对象头里设置标志（事实上也有通过标志来表现黑色对象和灰色对象的情况）。 在这里我们根据对象的情况，更抽象地把对象用三个颜色表现出来。每个对象是什么样的状 况，意味着什么颜色，这些都根据算法的不同而不同。

比如在增量式的标记-清除(Mark-Sweep)算法中，可以分为以下3个阶段：

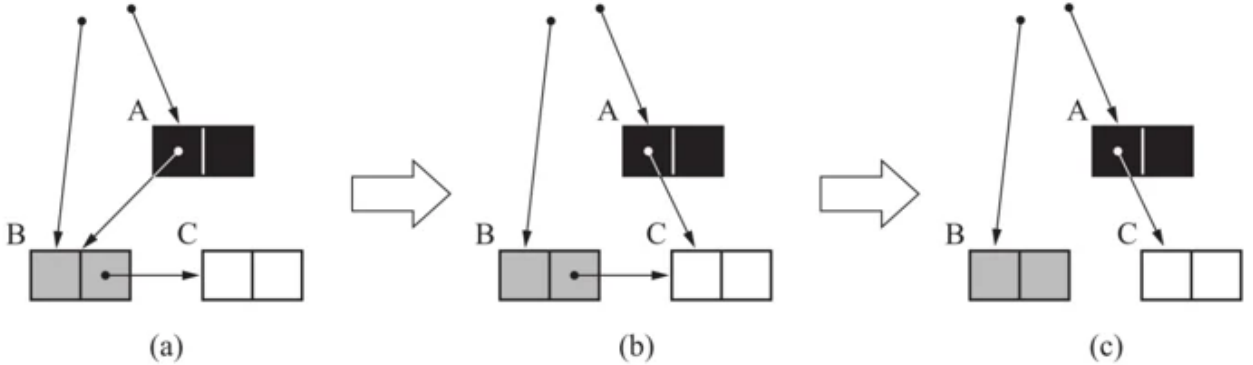
1. 根查找阶段
2. 标记阶段
3. 清除阶段

在根查找阶段，把所有能从根直接引用的对象标记为灰色。在标记阶段查找灰色对象，将其引用的对象也标记为灰色，查找结束（tracing完成）后将灰色对象标记为黑色。在最后的清除阶段，将黑色对象再标记为白色

标记遗漏

如果在标记阶段执行一半暂停后，Mutator更新了对象的引用关系，**就可能会导致活动对象（reachable objects）的“标记遗漏”，一旦发生了标记遗漏，就可能会在最后的清除阶段造成误回收活动对象的严重问题**

比如下面这个场景，在标记过程中暂停之后，Mutator修改了引用关系：



(a)是刚暂停的状态，A被标记为黑色，B被标记为灰色，接下来会对B进行遍历。此时继续执行Mutator

在(b)中，Mutator将A->B的引用，修改为A->C，然后删除了B->C的引用关系，本来是A->B->C，变成了A-C，就成了(c)图的情况

这个时候如果进行重新标记阶段就会出现问题：B本来是灰色对象，经过遍历后就被标记为了黑色。虽然此时C是活动对象，但也不会进行搜索了，因为没有一个灰色对象关联它，所以C就不会被标记为活动对象；那么在最后的清除阶段时，就会造成误清除。这种情况就成为标记遗漏

在上面这个例子里，造成标记遗漏的关键原因是，B->C的引用被移除了

写入屏障

写入屏障(Write Barrier)在GC里并不是一个具体的算法，只是一个“抽象”，其作用是在对象引用更新时增加一个“屏障”，在屏障中做一些增强操作。

下面是一段Edsger W. Dijkstra 等人提出的写入屏障实现（伪代码），在更新对象之间的引用时，需要调用write_barrier函数，从而实现“屏障”的功能。

```
write_barrier(obj, field, newobj){
    if(newobj.mark == FALSE)
        //标记新对象
         newobj.mark = TRUE
        //将新对象记录至标记栈
        push(newobj, $mark_stack)
    *field = newobj
}
```

在write_barrier函数中，更新引用的同时，将新对象标记后再记录至标记栈里，这样的话在待会的清除阶段，发生引用变化的对象活动对象也会被正常的标记了

通过这个写入屏障的方式，就可以解决上面的标记遗漏问题，因为记录了新的引用对象，新的引用对象也会被标记为活动对象，就不会出现标记遗漏了

汤浅太一 的算法

为 “Snapshot GC”。

> 这是因为这种算法是以 GC 开始时对象间的引用关系（snapshot）为基础来执行 GC 的。因此，根据汤浅的算法，在 GC 开始时回收垃圾，保留 GC 开始时的活动对象和 GC 执行过程中被分配的对象。

刚看这段时有点乱，SATB/Snapshot GC/Write Barrier几个概念有点混淆了

这里说的汤浅太一的算法，指的是它提出的一种实时垃圾回收的技术，和Edsger W. Dijkstra 等人提出不同。

“GC 开始时对象间的引用关系（snapshot）”这个也并不是说在标记阶段前，再新增一个快照的流程去记录引用关系。

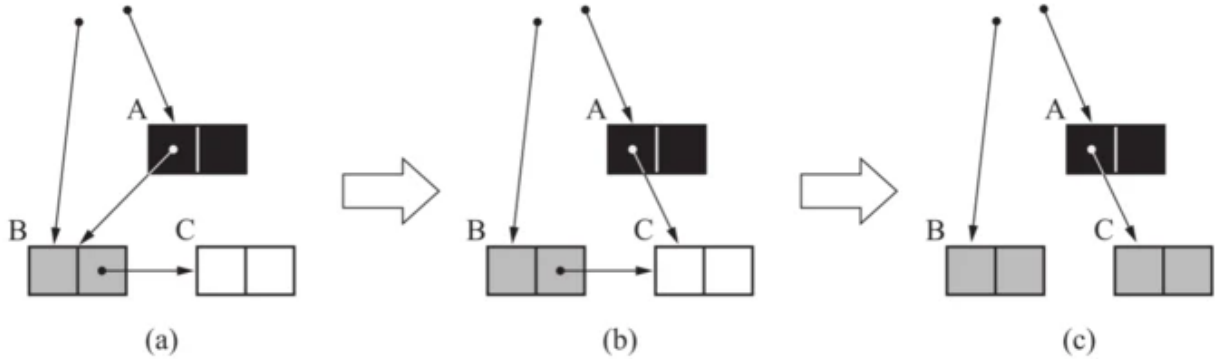
这个算法的设计理念是“Snapshot”数据，它认为“在标记阶段中**新的从根引用的对象**在 GC 开始时应该会被别的对象所引用”（别的对象引用这里由写入屏障来记录）

所以这种基于初始引用关系作为基准数据，忽略了GC过程中的变化（新的从根引用的对象）这种方式，称为Snapshot
**
下面是汤浅太一 的写入屏障实现的伪代码，和上面介绍的写入屏障不同，在这个版本里，**标记并添加到标记栈的对象变成了oldobj**

```
write_barrier(obj, field, newobj){
    oldobj = *field
    if(gc_phase == GC_MARK && oldobj.mark == FALSE)
        //标记老对象
        oldobj.mark = TRUE
        //将老对象记录至标记栈
        push(oldobj, $mark_stack)

    *field = newobj
}
```

通过写入屏障，记录引用变化前的对象（关系），从而构成“快照”。在汤浅的算法中，上面的ABC引用例子过程是这样：



在B->C的引用删除后，将C（oldobj）也标记为灰色，这样在标记阶段时，C还是会被遍历，这样就避免了标记遗漏的问题。

不过还是会有可能在并发标记过程中，某些对象已经没有引用不可达了，但是仍然会被标记，少回收几个也没什么关系……

G1 GC中的SATB

Hotspot G1 GC中的SATB，是汤浅太一写屏障算法的增强版，其核心还是基于快照理念“在标记阶段中**新的从根引用的对象**在 GC 开始时应该会被别的对象所引用”和写屏障来完成完整的标记

参考

- Taiichi Yuasa, _Real-time garbage collection on general-purpose machines_, Journal of Systems and Software, v.11 n.3, p.181-198, Mar. 1990

- 《垃圾回收的算法与实现》 中村成洋 , 相川光 , 竹内郁雄 (作者) 丁灵 (译者)
- 《深入Java虚拟机：JVM G1GC的算法与实现》中村成洋 (作者) 吴炎昌 , 杨文轩 (译者)
- HotSpot VM 请教G1算法的原理 - 资料 - 高级语言虚拟机 - ITeye群组

jvm gc 垃圾回收 g1gc

阅读 2.6k • 更新于 2021-08-21

赞 3 收藏 1 分享

空无

坚持原创，专注分享 JAVA、网络、IO、JVM、GC 等技术干货

2.9k 声望 **4.3k** 粉丝

关注作者

3 条评论 得票 最新