21 | 分代算法：基于生命周期的内存管理

2021-12-15 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳

时长 23:32 大小 21.56M

你好，我是海纳。

上节课，我们讲过了可达性分析中基于 copy 的垃圾回收算法，它比较适合管理短生命周期对象。那什么算法适合管理长生命周期对象呢？它就是可达性分析的 GC 算法中的另一大类：**Mark-Sweep 算法**。

为了发挥两种算法的优点，GC 的开发者就基于对象的生命周期引入了分代垃圾回收算法，它将堆空间划分为年轻代和老年代。其中年轻代使用 Copy GC 来管理，老年代则使用 Mark-Sweep 来管理。

所以这节课我们将先介绍传统 Mark-Sweep 算法的特点，在此基础上再引入分代垃圾回收。年轻代算法的原理你可以去上节课看看，这节课我们就重点介绍老年代的管理算法，

并通过 Hotspot 中的具体实现来进行举例。

通过这节课，你将从根本上把握住分代垃圾回收和老年代管理工作原理，在此基础上，你才能理解分代垃圾回收中比较晦涩的几个参数，从而可以对 GC 调优做到知其然且知其所以然。另外，CMS 算法是你以后掌握 G1 GC 和 Pauseless GC 的基础，尽管 CMS 在现实场景中应用得越来越少，但它的基本原理却仍然是学习 GC 的重要步骤。

好啦，不啰嗦了，我们先从朴素的 Mark-Sweep 算法开始讲起。

Mark-Sweep 算法

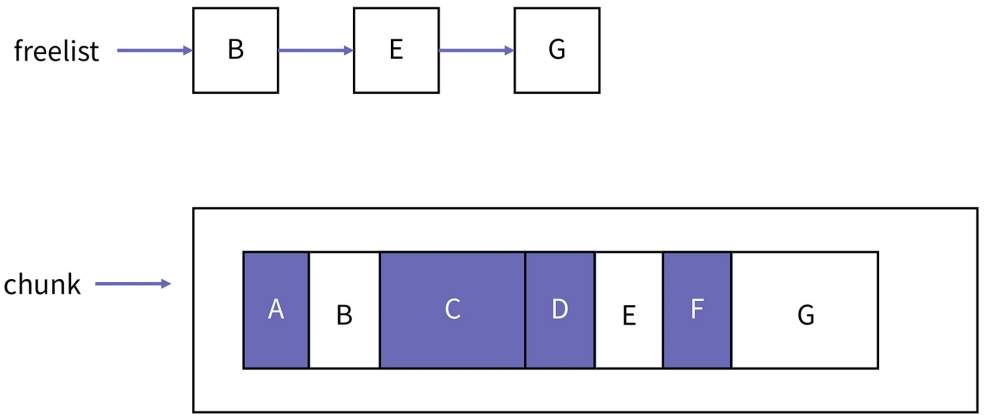
简单来讲，Mark-Sweep 算法由 **Mark 和 Sweep 两个阶段组成**。在 Mark 阶段，垃圾回收器遍历活跃对象，将它们标记为存活。在 Sweep 阶段，回收器遍历整个堆，然后将未被标记的区域回收。

当传统的 Mark-Sweep 算法在分配新的对象时，它的做法与第 9 节课所讲的 malloc 分配算法是一样的，但在具体实现上还是有一些细微的差别。

Mark-Sweep 算法和 malloc 相似的地方是，都是用一个链表将所有的空闲空间维护起来，这个链表就是**空闲链表（freelist）**。当内存管理器需要申请内存空间时，便向这个链表查询，如果找到了合适大小的空闲块，就把空闲块分配出去，同时将它从空闲链表中移除。如果空闲块比较大，就把空闲块进行分割，一部分用于分配，剩余的部分重新加到空闲链表中。

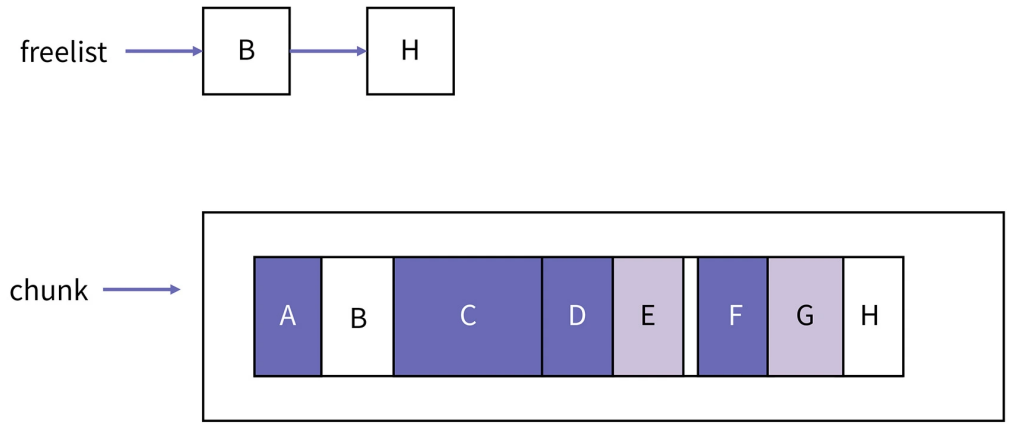
Hotspot 虚拟机的 Mark-Sweep 算法与 malloc 实现的不同之处在于：在 Hotspot 里，当一个空闲块分配给一个新对象后，如果剩余空间大于 24 字节，便将剩余的空间加入到空闲链表，当剩余空间不足 24 字节的话，就做为碎片空间填充一些无效值。而 malloc 则会分成多个空闲链表进行管理，更具体的实现可以参考第 9 节课。

Hotspot 中的 Mark-Sweep 的算法运行过程示意图如下：



图中的 A、C、D、F 是空间中的活跃对象，B、 E、G 三块区域是空闲区域，假设 B 的大小是 20，E 是 36，G 是 60。它们由空闲链表统一管理。

假设现在有两个分配请求，都要分配大小为 30 的空间，按照上面内容中的算法描述，具体的分配过程如下图所示：



E 的大小为 36，可以满足分配，但是 E 所剩下的区域已经比较小了，分配器不再将它加回到空闲链表，E 和 F 之间就产生了一块内存碎片。第二个请求，则会在 G 区域进行分配，在分配完以后，G 区域剩下的空闲区域还比较大，所以分配器会把分割后的空闲区域，也就是 H 区域再挂回空闲链表。上图中显示了分配完以后，堆里的最终结果。

介绍完空闲链表结构以后，我们来学习算法的原理。Mark-Sweep 算法主要包含 Mark 和 Sweep 两个阶段。按照先后顺序，我们先从第一阶段，也就是 Mark 阶段开始讲起。

Mark 阶段的核心任务是**从根引用出发，根据引用关系进行遍历，所有可以遍历到的对象就是活跃对象**。我们需要一边遍历，一边将哪些对象是活跃的记录下来。当遍历完成以后，我们就找到了所有的活跃对象。遍历的方法可以采用深度优先和广度优先两种策略。这一点和上一节课所讲的对象遍历的方法是一样的。

那如何对活跃对象进行标记呢？一般来说，常见的方法有两种。**一种是在每个对象前面增加一个机器字，采用其中的某一位作为“标记位”**。如果该位置位，就表示这个对象是活跃对象；如果该位未置位，那么表示这个对象是要回收的对象。

另一种方法是采用标记位图，将每一个机器字映射成位图中的一个比特。在真正的实现中，往往会采用两个位图，其中一个标记活跃对象的起始位置，另一个标记活跃对象的结束位置。

Sweep 阶段要做的事情就是把非活跃对象，也就是垃圾对象的空间回收起来，重新将它们放回空闲链表中。具体做法就是按照空间顺序，从头至尾扫描整个空间里的所有对象，如果一个对象没有被标记，这个对象所占用内存就会被添加回空闲链表。这个阶段的操作是比较简单的，只涉及了链表的添加操作，而且它和 malloc 的回收过程是一致的，所以我就不再啰嗦了。

Mark-Sweep 算法回收的是垃圾对象，如果垃圾对象比较少，回收阶段所做的事情就比较少。所以它适合于存活对象多，垃圾对象少的情况。而我们上节课讲的基于 copy 的垃圾回收算法 Scavenge，搬移的是活跃对象，所以它更适合存活对象少，垃圾对象多的情况。既然如此，那我们能不能把这两种算法的优点结合起来，用不同的算法管理不同类型的对象呢？

分代垃圾回收算法

自然是可以的，对于存活时间比较短的对象，我们可以用 Scavenge 算法回收；对于存活时间比较长的对象，就可以使用 Mark-Sweep 算法。这就是分代垃圾回收算法产生的动机。

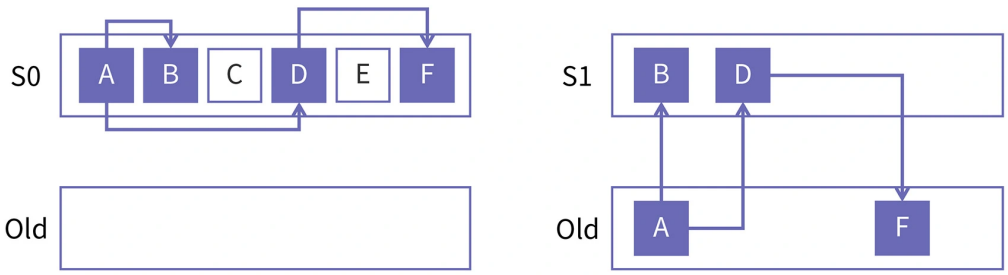
我们知道，Java 中的函数局部对象和临时对象也会在堆里进行分配，这就导致 Java 中的对象的生命周期都不长。所以，使用 Scavenge 对这些对象进行管理是合适的。可以说，**Scavenge 所管理的空间是“年轻代”**。

那怎么区分那些存活时间长的对象呢？我们可以在对象的头部记录一个名为 age 的变量，对象头部就是第 19 节课所介绍的 Mark word。age 变量不需要占据整个 Mark word 空间，只需要其中的几个比特就可以了。Scavenge GC 每做一次，就是把存活的对象往 Survivor 空间中复制一次，我们就相应把这个对象的 age 加一，以此来代表它的生命周期比较长。

当对象的 age 值到达一个上限以后，就可以将它搬移到由 Mark-Sweep 算法所管理的空间了。与“年轻代”相对应，我们称这个空间为“老年代”。而对象从年轻代到老年代的搬移过程，就称为**晋升 (Promotion)**。

记录集：维护跨代引用

可以想象，把对象放到两个空间，肯定会有跨空间的对象引用。如下图所示：



极客时间

上图中左边代表年轻代在 GC 之前，在幸存者空间 S0 中一共有 6 个对象，其中 4 个对象是活跃对象，两个白色框代表非活跃对象。假如此时发生年轻代 GC，垃圾回收器就会把 A、B、D、F 四个对象尝试向另外一个幸存者空间，也就是 S1 进行搬移。

在搬移的过程还会将对象的 age 加一，如果刚好 A 对象和 F 对象的 age 大于晋升阈值，那么这两个对象就会被搬到老年代中。如上图中的右侧所示，在年轻代的幸存者空间中，

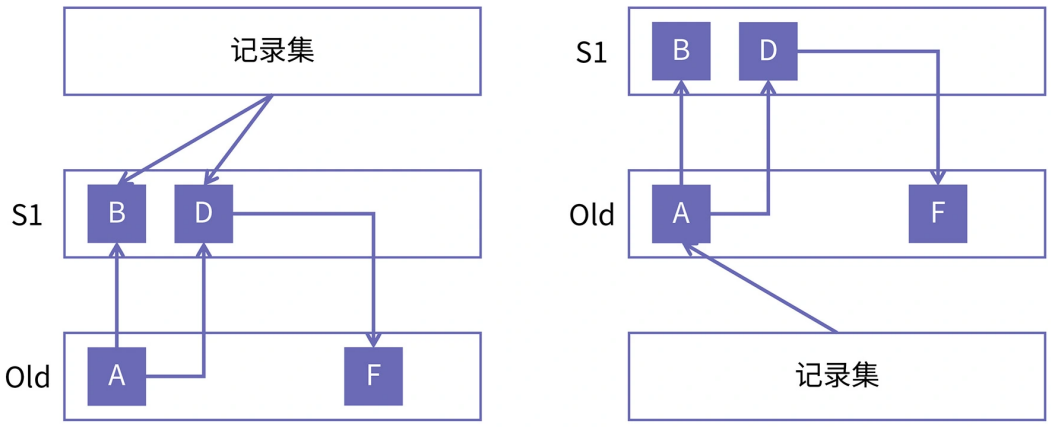
对象都是紧密排列的，所以 B 和 D 会靠在一起。而在老年代空间，由于对象是从空闲区域中分配的，所以 A 和 F 不一定是靠在一起的。

显然，这就会带来一个问题，在以后年轻代 GC 执行时，我们就要考虑从老年代到年轻代的引用了，也就是图中，A 指向 B 和 D 对象的引用。反过来说，当老年代 GC 执行时，也同样要考虑从年轻代到老年代的引用，也就是图中的 D 指向 F 的引用。

在进行年轻代垃圾回收时，为了找出从老年代到年轻代的引用，可以考虑对老年代对象进行遍历。但如果这么做的话，年轻代 GC 执行时，就会对全部对象进行遍历，分代就没意义了。

为了解决这个问题，我们可以想办法把这种跨代引用记录下来，**记录跨代引用的集合就是记录集（Remember Set，RS）。**

最直观的思路是，记录集里记录被引用到的新生代对象，如下图中的左侧所示：



极客时间

记录集中记下了被跨代引用的 B 和 D 两个对象，那么当发生年轻代 GC 时，就能正确地找到 B 和 D 对象是根对象。但是这样做的问题是，当 B 和 D 再经过一轮年轻代 GC，位置发生变化以后，A 对象对它们的引用却无法正确维护。

为了解决这个问题，我们可以再进一步将 A 放到记录集中，也就是上图中右侧所示，记录集中指向了 A 对象。这样，在做年轻代 GC 时，我们对 A 进行遍历，就可以访问到 B 和

D 了。因为 B 和 D 在年轻代中，它们的位置会发生变化，变化的地址也能正确地更新到对象 A 中。A 在老年代，所以它不会被搬移，我们只是借用它去访问 B 和 D。

搞清楚了记录集的作用，我来接着看如何维护记录集呢？

写屏障：维护记录集

维护记录集的手段，在第 19 节课我们已经接触过了，那就是写屏障（write barrier）。在第 19 节课，我们在对象的引用发生变化的时候，来维护对象的引用计数。在分代式垃圾回收算法中，我们可以使用同样的思路来维护跨代引用。

当对象的属性进行写操作时，跨代引用就有可能出现。所以，我们在这个时候可以检查是否存在跨代引用。写屏障的伪代码如下所示：

复制代码

```
1 void do_oop_store(Value obj, Value* field, Value value) {
2     if (in_young_space(value) && in_old_space(obj) &&
3         !rs.contains(obj)) {
4       rs.add(obj);
5     }
6     *field = &value;
7 }
```

上面代码中，obj 代表引用者对象；value 代表被引用的对象；field 代表 obj 对象中被修改的那个域，它是一个指针，代表了地址，这意味着我们要修改地址处存放的值，而不是修改指针本身。

在进行对象的域赋值时，我们要先做以下三个判断（第 2 行）：

1. **被引用的对象是否在年轻代**；
2. **发出引用的对象，也就是引用者，是否在老年代；以上两点都满足，就说明产生了跨代引用**；
3. **检查记录集中是否已经包含了 obj**。

如果以上三点都满足，就将 obj 添加到记录集中（第 4 行）。

还有一种情况可能产生跨代引用，那就是晋升。假设 A 对象引用 B 对象，它们都在年轻代里，经过一次年轻代 GC，A 对象晋升到老年代，那这也会产生跨代引用，所以在晋升的时候，我们也要对这种情况加以处理。晋升的完整伪代码，如下所示：

复制代码

```
void promote(obj) {
  new_obj = allocate_and_copy_in_old(obj);
  obj.forwarding = new_obj;

  for (oop in oop_map(new_obj)) {
    if (in_young_space(oop)) {
      rs.add(new_obj);
      return;
    }
  }
}
```

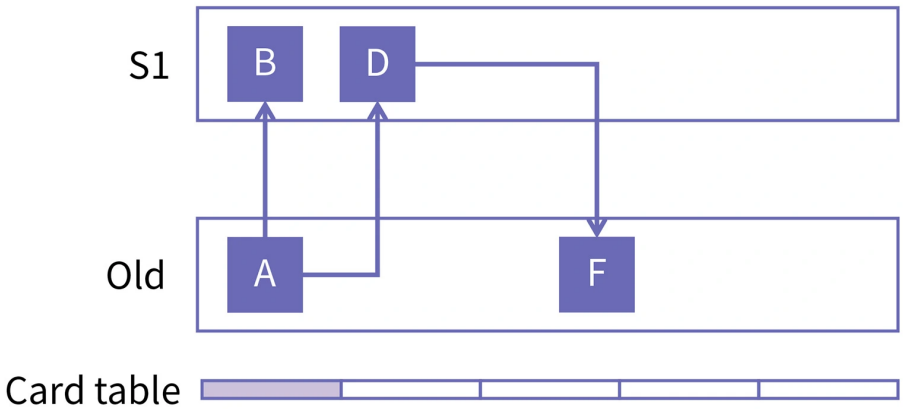
在上面的代码里，我们先在老年代中分配一块空间，把待晋升对象复制到老年代。然后把新地址 new_obj 记录到原来对象的 forwarding 指针。接着遍历从这个对象出发的所有引用，如果这个对象有引用年轻代对象，就把这个晋升后的对象添加到记录集中。

在上面所讲的写屏障实现里，一个对象写操作中要进行三个判断，如果条件成立，还要再执行一次记录集的添加对象操作，效率是比较差的。为了提升效率，又有人提出了 **Card table 这种实现方式来提升写屏障的效率。**

Card table：提升写屏障效率

随着对象的增多，记录集会变得很大，而且每次对老年代做 GC，正确地维护记录集也是一件复杂的事情。另外，写屏障的效率也不高，为了解决这个问题，可以借鉴位图的思路，这就是 Card table。

Hotspot 的分代垃圾回收将老年代空间的 512bytes 映射为一个 byte，当该 byte 置位，则代表这 512bytes 的堆空间中包含指向年轻代的引用；如果未置位，就代表不包含。这个 byte 其实就和位图中标记位的概念很相似，人们称它为 card。多个 card 组成的表就是 Card table。一个 card 对应 512 字节，压缩比达到五百分之一，Card table 的空间消耗还是可以接受的。



极客时间

如上图所示，图的下方就是 Card table，因为 A 对象包含指向年轻代的引用，所以 A 对象所对应的 card 就被置位，而 F 对象不存在指向年轻代的引用，所以它所对应的 card 就未被置位。

在明白了分代式垃圾回收是如何维护跨代引用的以后，我们就可以转向研究 Hotspot 中分代式垃圾回收的典型算法：**并发标记清除算法（Concurrent Mark Sweep，CMS）**。标记清除的概念我们已经介绍过了，并发又是什么意思呢？接下来，我们一起来研究一下。

并发标记算法

在并发标记算法中，当垃圾回收器在标记活跃对象的时候，我们肯定不希望业务线程同时还会修改对象之间的引用关系。所以，早期的 Mark-Sweep 算法会让业务线程都停下来，以便于垃圾回收器可以对活跃对象进行标记，这就是 GC 停顿产生的原因。由于业务线程停顿的时候，整个 Java 进程都不能再响应请求，**人们把这种情况形象地称为“世界停止”（Stop The World，STW）**。

为了减少 GC 停顿，我们可以在做 GC Mark 的时候，让业务线程不要停下来。这意味着 GC Mark 和业务线程在同时工作，这就是**并发**（Concurrent）的 GC 算法。

这里你要注意区别并发（Concurrent）和并行（Parallel）的区别。并发 GC 是指 GC 线程和业务线程同时工作，并行是指多个 GC 线程同时工作。

Hotspot 中在实现 Mark-Sweep 算法的时候，采用了并发的方式，这就是并发标记清除，简称为 CMS。它的特点是 GC 线程在标记存活对象的过程中，业务线程是不必停下来的。直觉上就是一边对图进行遍历，一边修改图中的边，这样肯定会产生问题。接下来，我们就详细地分析一下这么做到底会有什么问题。

为了方便描述，我们引入三种颜色来辅助算法的讲解。我们知道，图的遍历过程，就是不断地对结点进行搜索和扩展的过程。以标记算法来说，如果采用广度优先遍历，那么搜索就是对结点进行标记，扩展就是将结点所引用的其他对象都添加到辅助队列中。根据这个定义，我们可以将结点分为三类：

白色：还未搜索的对象；

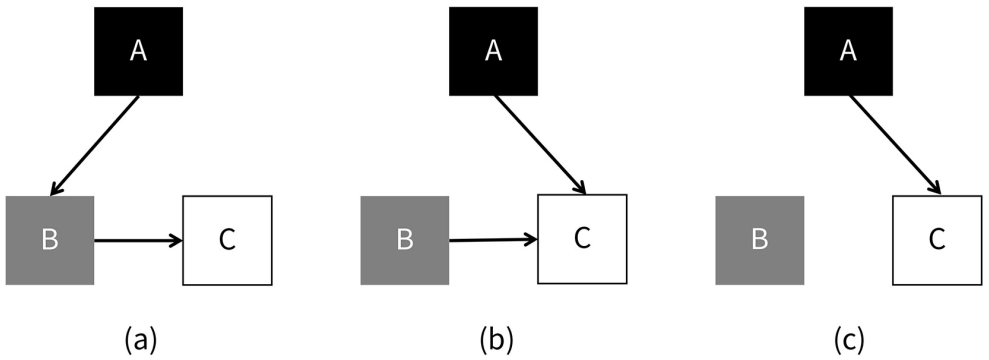
灰色：已经搜索，但尚未扩展的对象；

黑色：已经搜索，也完成扩展的对象。

这里要注意，三色标记并不意味着对象真的要为自己增加一个 color 属性，它只是一种抽象的概念，在不同的 GC 算法中，它代表不同的状态。

我们以广度优先搜索的标记算法为例，白色对象就是未被标记的对象；灰色则是已经被标记，但还没有完成扩展的对象，也就是还在队列中的对象；黑色则是已经扩展完的对象，即从队列中出队的对象。

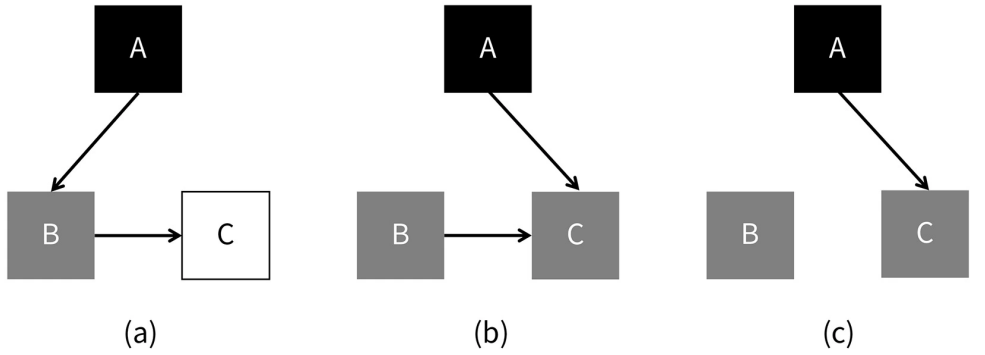
你要注意的是，并发标记中最严重的问题就是**漏标**。如果一个对象是活跃对象，但它没有被标记，这就是漏标。这就会出现活跃对象被回收的情况。例如下图中所示：



在上图中的最左边，标号（a）的子图中，一切都还是正常的，B 尚未扩展，在 B 扩展的时候，C 自然可以被标记。在（b）中，A 出发的引用指向了 C，这时由于 B 指向 C 的引用还存在，仍然没有什么问题。但在（c）中，B 指向 C 的引用消失了，因为 A 已经变成黑色，不会再被扩展了，所以 C 就没有机会再被标记了。这就产生了漏标。

总的来说，黑色对象引用了白色对象，而白色对象又没有其他机会再被访问到，所以白色对象就被漏标了。

漏标问题的解决方案主要有三种，我们这节课会介绍两种，第三种我们会在下一节课再详细介绍。第一种解决方案是**往前进**的方案，如下图所示：

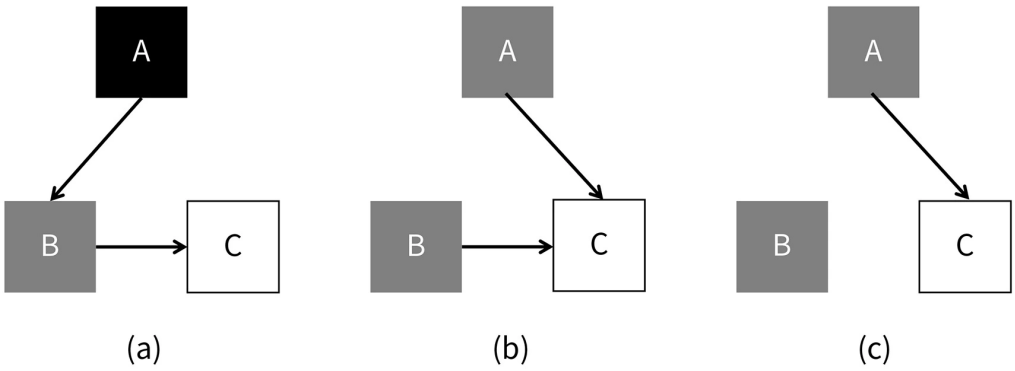


极客时间

解决漏标问题，还是要从写屏障入手。荷兰计算机科学家 Dijkstra 提出一种写屏障：**当黑色对象引用白色对象时，把白色对象直接标灰，也就是说将 C 对象直接标记，然后放入队列中待扩展**。这个过程的伪代码我建议你自己写一下，作为练习。

往前走的方案是比较符合直觉的，但是假如在 C 对象被标记以后，A 对 C 的引用又消失了，C 实际上是被多标了。多标并不会带来严重的后果，它只会导致原本应该被回收的对象没有被及时回收，这种对象被称为**浮动垃圾**。

为了解决浮动垃圾这个问题，我们可以使用第二种方案，那就是**往后退一步**：把黑色对象重新扩展一次，也就是说黑色结点变成灰色。如下图所示：



如果对象修改比较频繁，那么后退一步的方案会比前进一步的方案更好，因为它不会产生浮动垃圾。Lua 虚拟机中就采用了方案二，这是因为 Lua 中的 Table 结构的修改是比较频繁的。

在 Hotspot 中，CMS 的实现和上面的两种思路有关系，但又不完全一样，接下来，我们全面地分析一下在 Hotspot 中，CMS 是如何实现的。

Hotspot 中 CMS 的实现

在 Hotspot 中，CMS 是由多个阶段组成的，主要包括初始标记、并发标记、重标记、并发清除，以及最终清理等。其中：

初始标记阶段，标记老年代中的根对象，因为根对象中包含从栈上出发的引用等比较敏感的数据，并发控制难以实现，所以这一阶段一般都采用 Stop The World 的做法。这里一般不遍历年轻代对象，也就是不关注从年轻代指向老年代的引用。

并发标记阶段，这一阶段就是在上面内容中讲到的三色标记算法中做了一些改动，我们会在后面的内容中详细分析这一阶段的实现。

重标记阶段，这一阶段会把年轻代对象也进行一次遍历，找出年轻代对老年代的引用，并且为并发标记阶段扫尾。

并发清除阶段，这一阶段会把垃圾对象归还给 freelist，只要注意好 freelist 的并发访问，实现垃圾回收线程和业务线程并发执行是简单的。

最终清理阶段，清理垃圾回收所使用的资源。为下一次 GC 执行做准备。

CMS 中最复杂的还是并发标记阶段。如果在并发标记的过程中，业务线程修改了对象之间的引用关系，CMS 采用的办法是：**在 write barrier 中，只要一个对象 A 引用了另外一个对象 B，不管 A 和 B 是什么颜色的，都将 A 所对应的 card 置位。**

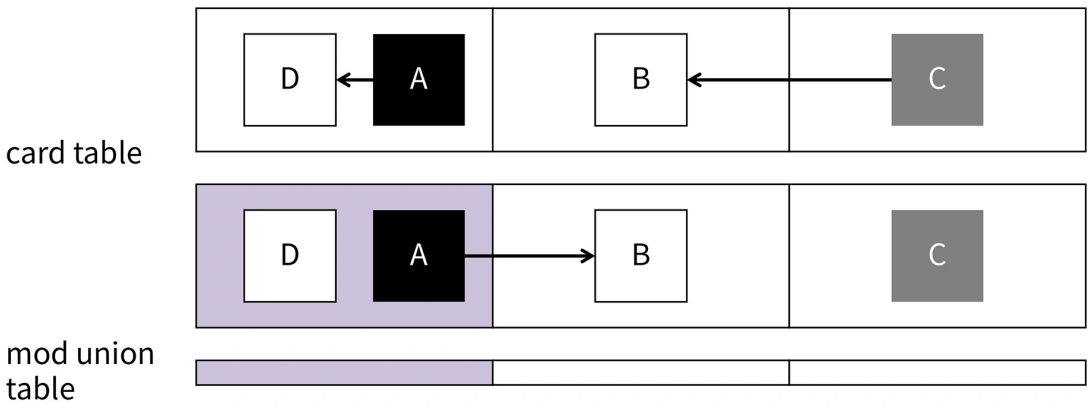
由于不必检查颜色，这个置位的过程就非常快，你可以自己思考一下，card 置位的具体实现。当一轮标记完成以后，如果还有置位的 card，那么垃圾回收器就会开启新一轮并发标记。新的一轮标记，会从上一轮置位的 card 所对应的对象开始进行遍历，遍历完成后再把 card 全部清零，所以这样的一轮并发标记也被称为**预清理（preclean）**。

如果恰好在某一轮并发标记的过程中，业务线程不再修改对象之间的引用关系了，那么就不会再产生 card 置位的情况了。这时就可以结束并发标记阶段了。

但是如果每一轮都有 card 置位，应该怎么办呢？CMS 也会在预清理达到一定次数以后停止，进入重标记阶段。重标记的作用是遍历新生代和 card 置位的对象，对老年代对象做一次重新标记。这一次是需要停顿的，因为这一次垃圾回收器将不允许 card 再被置位了。

一般来说，新生代指向老年代的引用不会太多，但是偶尔也会发生这种引用很多的情况，如果出现了这种情况，可以在重标记之前，进行一次年轻代 GC，这样可以减少年轻代中的对象数量，减少重标记的停顿时间。这个功能可以使用参数 -XX:+CMSScavengeBeforeReMark 来打开。

这个过程的示意图如下所示：



极客时间

因为 card table 有两个功能：维护跨代引用和标记灰色结点，所以，Hotspot 又引入了另外一个数据结构 mod union table（MUT）来用于维护跨代引用。在并发标记开始之前，card table 中的内容就会被复制到 MUT 里，如果在 Concurrent Mark 阶段，发生了年轻代的垃圾回收，则可以使用 MUT 来进行跨代扫描。

上图中展示了，A 对象在引用 B 对象时，A 对象所对应的 card 被置位。因为 D 对象所对应的 card 和 A 对象所对应的 card 是同一个 card，所以，GC 在清理 card 的过程中仍然会把 D 对象进行标记。所以 D 就是一个被多标记了的垃圾对象，也就是浮动垃圾。

到这时，Hotspot 中的具体实现我们就搞清楚了。

总结

好啦，这节课到这里就结束啦，我们一起来回顾一下这节课的重点内容吧。这节课，我们先介绍了最基本的 **Mark-Sweep 算法**，它使用空闲链表来组织空闲空间，它的分配过程与第 9 节课介绍的 malloc 的方法很相似。在标记阶段，垃圾回收器通过图遍历算法来标记对象，在活跃对象被标记以后，不活跃的区域会被集中地归还到空闲链表中。

Mark-Sweep 算法最大的特点是不移动活跃对象，只回收不活跃的空间。所以它更适合管理生命周期比较长的对象。它的特点与上节课所讲的基于 copy 的算法刚好互补，所以人们就把这两者结合起来，使用 copy 算法管理短生命周期对象，也就是**年轻代**；使用 Mark-Sweep 算法管理长生命周期对象，也就是**老年代**。

在分代式垃圾回收算法里，最大的挑战是如何维护跨代引用，以加速单独的某个代的垃圾回收的过程。我们介绍了**记录集**和 **Card table** 两种方式，并且介绍了如何使用 write barrier 对它们进行维护。

在分代垃圾回收算法里，**用于管理老年代的是 CMS 算法**。这种算法的主要挑战来自于垃圾回收线程在标记的过程中，业务线程还在不断地修改对象之间的引用关系。我们介绍了三色标记算法，并且介绍了两种解决漏标问题的手段：**“往前走”和“往后退一步”**。

思考题

我们已经知道标记的过程就是图遍历的过程，那你觉得在 Mark-Sweep 算法中，应该采用广度优先还是深度优先进行遍历呢？为什么呢？考虑到对象不必搬移，你还能不能想到更

省空间的做法？欢迎在留言区分享你的想法，我在留言区等你。

吊打面试官

- 你知道什么是提前晋升吗？怎么判断是否存在提前晋升的情况？如何处理呢？

提前晋升是指在分代式垃圾回收算法中，年轻代对象的年龄还没有达到晋升的阈值就因为年轻代的空间不够而不得不提前搬进老年代的现象。提前晋升不仅仅是年轻代的压力大，这会导致老年代空间快速消耗，从而触发CMS。

判断提前晋升最简单的办法是使用jstat工具，如果每一次年轻代GC以后，幸存者空间（S0或者S1）都是满的，而老年代的空闲空间明显下降，这就说明有一些年轻代中的对象在年龄没有达到阈值就升入老年代了。

解决的办法主要是可以考虑扩大年轻代的空间，一种是直接扩大堆的大小，如果堆的大小不能再扩大，也可以考虑将老年代空间配置得小一些，从而使得年轻代空间更大一些。当然，最好的办法还是尽量减少对象的创建，这可以从优化业务逻辑等方式入手。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 20 | Scavenge：基于copy的垃圾回收算法

下一篇 22 | G1 GC：分区回收算法说的是什么？

精选留言 (4)

写留言

费城的二鹏
2021-12-15

思考题
用广度优先搜索，可以减少维护一个内存空间。

2021-12-15

请问老师，

1. Remember Set 与 Card table 的关系，是因为 Remember Set 效率较差，复杂，所以最终在 hotspot 中使用了 Card table 这种方案吗；网上有些博客说 Remember Set 与 Card table 的关系是接口与实现的关系，老师怎么看呢...

展开

共 2 条评论 >

费城的二鹏
2021-12-15

老师有时间可以讲一下 ZGC 种染色指针的具体原理以及扫描过程的逻辑吗？

编辑回复: 小编有时间，你敢不敢听~~哈哈，开个玩笑，关注我们的23讲吧~ZGC马上就来

共 3 条评论 >

费城的二鹏

2021-12-15

老师有时间可以详细讲下 OopMap 吗？他的结构与生成原理。

共 1 条评论 >