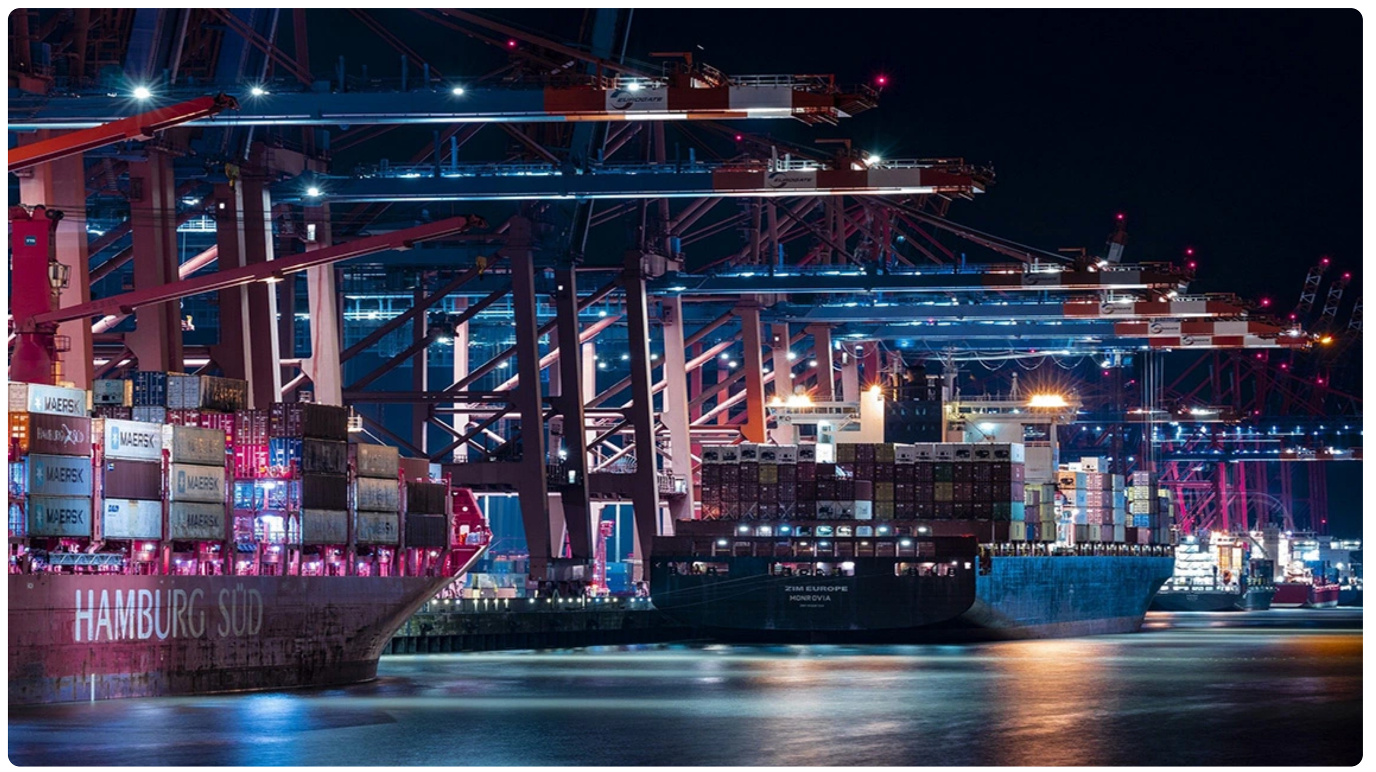
下载APP

22 | G1 GC：分区回收算法说的是什么？

2021-12-17 海纳

《编程高手必学的内存知识》 课程介绍 >



讲述：海纳
时长 20:44 大小 18.99M

你好，我是海纳。

在上一节课，我们介绍了分代式垃圾回收算法。把对象分代以后，可以大大减轻垃圾回收的压力，进而就减少了停顿时长。在这种思路的启发下，人们进一步想，如果把对象分到更多的空间中，根据内存使用的情况，每一次只选择其中一部分空间进行回收不就好了吗？根据这个思路，GC 开发者设计了**分区回收算法**。

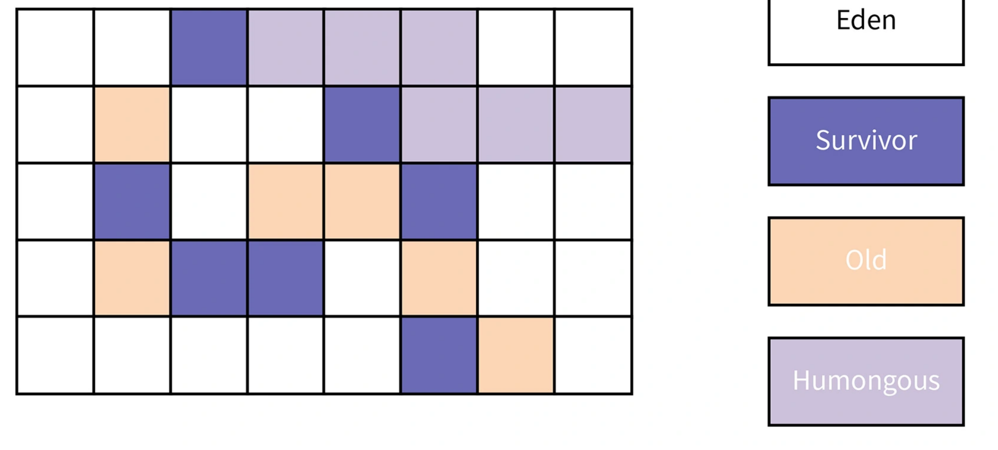
它在实际场景中应用非常广泛，比如说 Hotspot 中的 G1 GC 就是分区回收算法的一种具体实现，Android 上的 art 虚拟机也采用了分区回收算法。而且从 JDK9 开始，G1 GC 是 JDK 的默认垃圾回收算法了，所以在将来很长时间里，对 G1 GC 进行合理的调优，将是 Java 程序员要重点掌握的知识。

那么这节课，我们就来深入地讲解分区回收算法的基本原理，掌握 G1 GC 的若干重要参数，从而对 G1 GC 进行合理的参数调优。

要想理解分区垃圾回收的原理，还得从它的结构讲起。

分区算法的堆结构

首先，我们来了解一下分区回收算法的堆空间是如何划分的。下图是 G1 GC 的堆结构：



G1 也是一个分代的垃圾回收算法，不过，和之前介绍的 CMS、Scavenge 算法不同的是：**G1 的老年代和年轻代不再是一块连续的空间，整个堆被划分成若干个大小相同的 Region，也就是区**。Region 的类型有 **Eden、Survivor、Old、Humongous** 四种，而且每个 Region 都可以单独进行管理。

Humongous 是用来存放大对象的，如果一个对象的大小大于一个 Region 的 50%（默认值），那么我们就认为这个对象是一个大对象。为了防止大对象的频繁拷贝，我们可以将大对象直接放到 Humongous 中。

而 Eden、Survivor、Old 三种区域和我们前面课程中介绍的 Eden 分区、Survivor 分区以及老年代的作用是类似的。也就是说，对象会在 Eden Regions 中分配，当进行年轻代 GC 时，会将活跃对象拷贝到 Survivor Regions；当对象年龄超过晋升阈值时，就把活跃对象

复制进 Old Regions。如果你不清楚，可以看看第 19 节课和第 20 节课，这里我就不再啰嗦了。

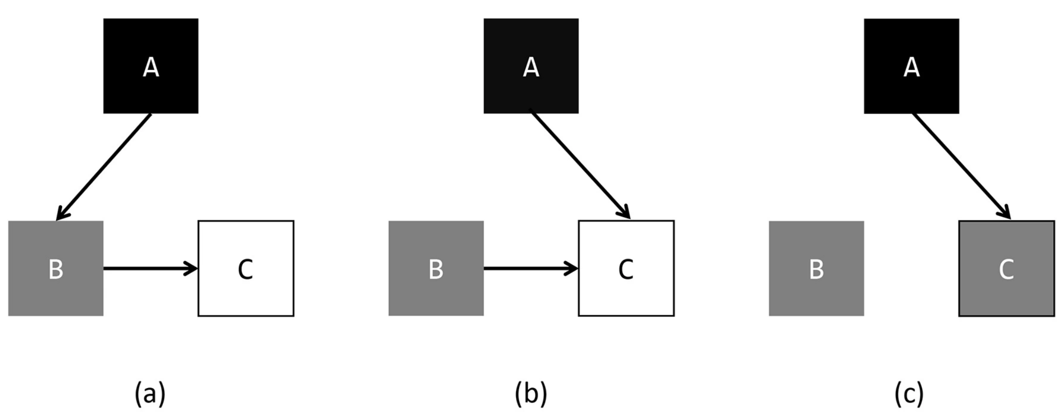
在了解了 G1 的堆空间划分之后，我们就可以开始学习 G1 算法的回收原理了。实际上，分区垃圾回收算法最大的特点是**维护跨分区引用**，这也是它实现起来最难的地方。下面我们就以此为切入点，来探寻 G1 算法的原理。

写屏障

维护跨分区引用，其中的关键就是写屏障。我们在上节课讲 CMS 时提到，写屏障是对象在修改引用关系时，额外做一些操作来维护相关信息。在 CMS 中，写屏障主要有两个作用：

1. **在并发标记阶段解决活跃对象漏标问题；**
2. **在写屏障里使用 card table 维护跨代引用。**

我们先来看第一个作用，也就是解决活跃对象漏标的问题。上一节课介绍了解决漏标问题的两种方法，分别是 **“往前走”和“往后退一步”** 。今天这节课，我们就来介绍第三种解法，这个解法呢，是由日本学者汤浅太一提出的，具体的算法如下图所示：



在这张图中，我们可以看到，当对象 B 对 C 的引用关系消失以后，再将 C 标记为灰色，即便将来 A 对 C 的引用消失了，也会在当前 GC 周期内被视为活跃对象。也就是说，C 有可能变成浮动垃圾。我们把这种在删除引用的时候进行维护的屏障叫做 **deletion barrier**。

G1 中采用的就是这种做法。这种做法的特点是，在 GC 标记开始的一瞬间，活跃的对象无论在标记期间发生怎样的变化，都会被认为是活跃的对象。

我们知道，当一个对象的全部引用被删除时，才会被当做垃圾。而如果使用我们前面讲到的 deletion barrier，在并发标记阶段，即便对象的全部引用被删除，也会被当做活跃对象来处理。就好像在 GC 开始的瞬间，内存管理器为所有活跃对象做了一个快照一样，所以人们给了这种技术一个很形象的名字：**开始时快照（Snapshot At The Beginning，SATB）**。

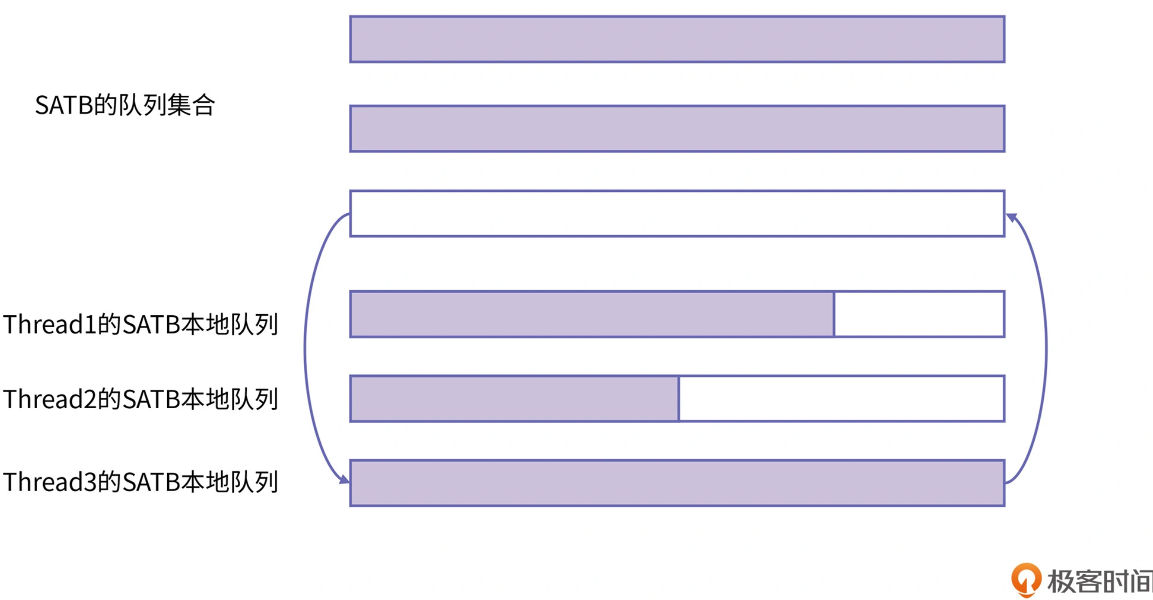
你要注意的是，有些文章对 SATB 的解释是：在 GC 开始时将堆做一个内存快照，存放到磁盘上。这种说法就是望文生义了。因为快照这个词在计算机领域通常是指压缩，索引等技术，所以就有人把这里的快照理解成了对堆对象的一种压缩。由此，我们就知道这种错误的说法是怎么来的了。

言归正传，我们在理解 SATB 的含义之后，再来看看 SATB 具体的工作原理吧。

我们在讲写屏障时提到，当 B 对象对 C 对象的引用消失时，C 对象将会被标记为灰色。这个动作的效率是比较低的，如果都放在写屏障中做，会极大地影响程序性能。**因为写屏障的逻辑是由业务线程执行的。**

为了解决这个问题，GC 开发者将“C 对象标记为灰色”这件事情往后推迟了。业务线程只需要把 C 对象记录到一个本地队列中就可以了。每个业务线程都有一个这样的线程本地队列，它的名字是 **SATB 队列**。

当业务线程发现对象 C 的引用被删除之后，直接将 C 放到 SATB 队列中，并不去做标记，真正做标记的工作交给 GC 线程去做，这样就减少了写屏障的开销。



如上图所示，每个线程有自己的本地 SATB 队列，当本地队列满了之后，就把它交给 SATB 队列集合，然后再领取一个空队列当做线程的本地 SATB 队列。GC 线程则会将 SATB 队列集合中的对象标记为灰色，至于什么时候标记，并不需要业务线程关心。

在学习了 SATB 相关知识后，我们继续来定义 G1 的两种垃圾回收模式，以方便后面详细地介绍算法的执行过程。

垃圾回收模式

G1 的垃圾回收模式有两种：分别是 **young GC 和 mixed GC**。

- young GC：只回收年轻代的 Region。
- mixed GC：回收全部的年轻代 Region，并回收部分老年代的 Region。

我要告诉你的是，无论是 young GC 还是 mixed GC，都会回收全部的年轻代，mixed 回收的老年代 Region 是需要进行决策的（Humongous 在回收时也是当做老年代的 Region 处理的）。那么决定老年代 Region 是否被回收的因素具体有哪些呢？

我们把 mixed GC 中选取的老年代对象 Region 的集合称之为**回收集合（Collection Set，CSet）**。CSet 的选取要素有以下两点：

1. 该 Region 的垃圾占比。垃圾占比越高的 Region，被放入 CSet 的优先级就越高，这就是**垃圾优先策略（Garbage First），也是 G1 GC 名称的由来**。
2. 建议的暂停时间。建议的暂停时间由 -XX:MaxGCPauseMillis 指定，G1 会根据这个值来选择合适数量的老年代 Region。

MaxGCPauseMillis 默认是 200ms，一般不需要进行调整，如果需要停顿时间更短可以对它进行设置，不过需要注意的是，MaxGCPauseMillis 设置的越小，选取的老年代 Region 就会越少，如果 GC 压力居高不下，就会触发 G1 的 Full GC。

触发 G1 的 Full GC 代价是很高的。最早的实现是一个单线程的 Mark-Compact GC，停顿时间非常长，虽然后来也改进成多线程，但还是需要尽量避免触发 G1 的 Full GC。如果一个应用会频繁触发 G1 GC 的 Full GC，那么说明这个应用的 GC 参数配置是不合理的，理想情况下 G1 是没有 Full GC 的。在这节课的最后，我会介绍几个常用的 G1 参数，方便你在实践中对 G1 进行调参。

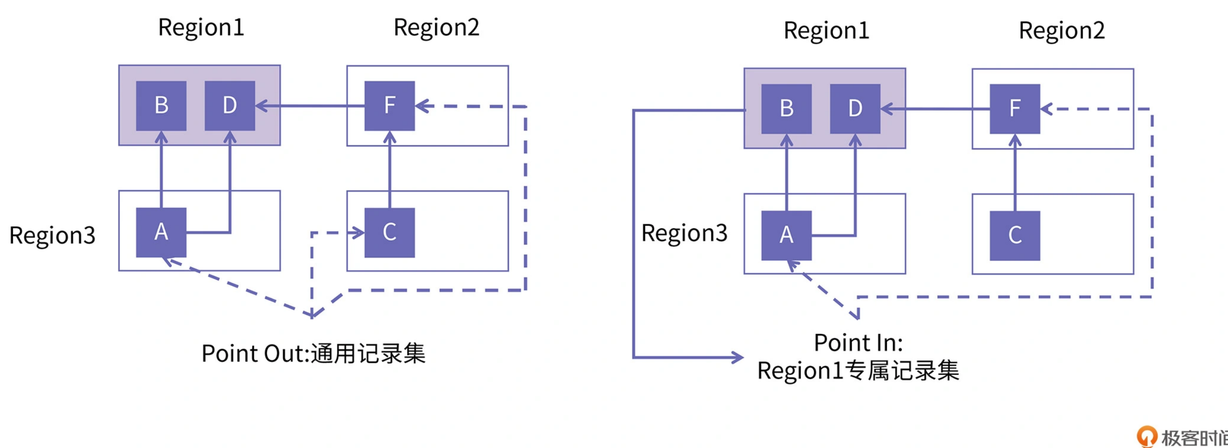
在学习了 G1 的垃圾回收模式之后，我们需要解决的问题还有不少，首先就是跨区引用的问题。

维护跨区引用

在上面的内容中，我们提到了写屏障的两个功能，第二个功能就是**维护跨区引用**。在第 21 节课中，我们已经学习了 CMS 的跨代引用，实际上，CMS 的跨代引用和 G1 的跨区引用的原理是相同的。不同的是，CMS 的跨代引用它的回收空间是固定的，例如 young GC 只回收年轻代，Concurrent Mark Sweep 只回收老年代，这样只需要维护一张卡表就可以了。

但是像 G1 这种分区回收算法，有些 Region 可能被选入 CSet，有些则不会。所以，我们需要知道当一个 Region 需要被回收时，有哪些其他的 Region 引用了自己。相应地，为了加快定位速度，分区回收算法为每个 Region 都引入了**记录集（Remembered Set，RSet）**，每个 Region 都有自己的专属 RSet。

和 Card table 不同的是，RSet 记录谁引用了我，这种记录集被人们称为 **point-in 型**的，而 Card table 则记录我引用了谁，这种记录集被称为 **point-out 型**。



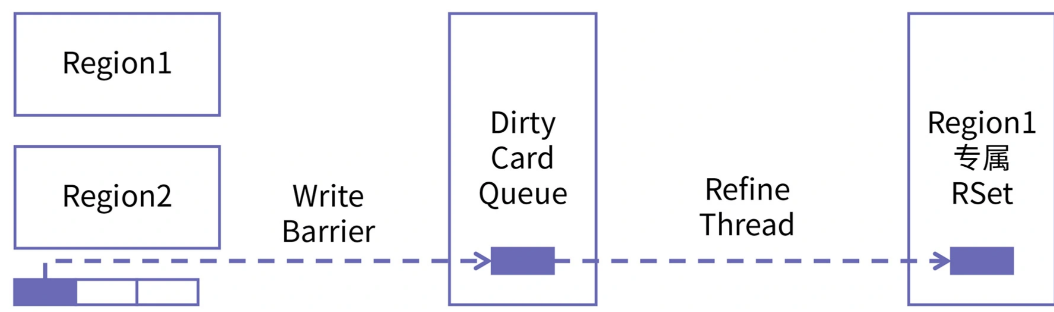
如图所示，图中的左侧展示了一个维护跨区引用的通用记录集，而右侧则展示了只对应于一个 Region 的专属记录集。

接下来我们继续分析 RSet 的维护策略，也就是说哪些引用关系需要加入到 RSet：

1. 如果是同一个 Region 的对象，它们之间相互引用是不必维护的，这个很好理解，因为不存在跨 Region 的问题；
2. 由年轻代 Region 出发到其他 Region 的，无论目标是年轻代还是老年代，这一类引用也都不用维护。因为结合 young GC 和 mixed GC 的策略可以知道，无论是什么回收模式，年轻代的全部 Region 都会被清理，这就意味着一定会对年轻代的所有对象进行遍历；
3. 从 CSet 集合的 Region 出发指向其他 Region 的，也不需要维护，理由和第 2 点是一样的。

总的来说，RSet 需要维护的引用关系只有两种，**非 CSet 老年代 Region 到年轻代 Region 的引用，和非 CSet 老年代 Region 到 CSet 老年代 Region 的引用。**

那么，RSet 具体是何时被记录的呢？答案也是写屏障，写屏障的这个作用，我们在上面的内容中已经提到过。如下图所示：

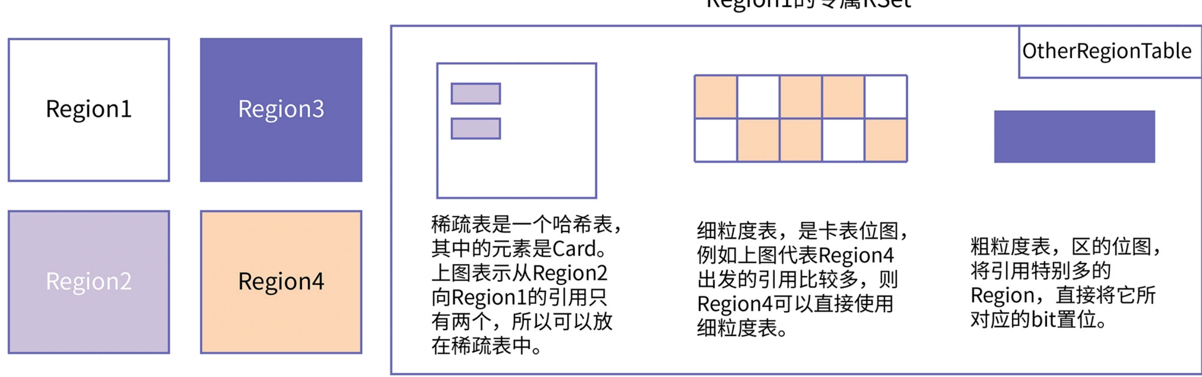


G1 在 RSet 中记录的也是 card。比如 Region1 中的对象 A 引用了 Region2 的对象 B，那么对象 A 所对应的 card 就会被记录在 Region2 的 RSet 中（注意！不是 Region1 的 RSet）。

在 G1 中，我们把这种 card 称为 dirty card。和 SATB 相似，业务线程也不是直接将 dirty card 放到 RSet 中的。而是在业务线程中引入一个叫做 **dirty card queue（DCQ）** 的队列，在写屏障中，业务线程只需要将 dirty card 放入 DCQ 中，而不做非常细致的检查。

接下来，GC 线程中，有一类特殊的线程，它们会从 DCQ 中找到这种 dirty card，然后再去做更精细的检查，只有确实不属于上面所描述的三种情况的跨区引用，才真正放到专属 RSet 中去。这一类特殊的线程就是 G1 GC 中的 **Refine 线程**。

下面我们再来继续剖析 RSet 存放的形式是怎样的。考虑某个 Region 的 RSet，它可能会因为引用关系比较多，而变得很大。根据另一个 Region 对这个 Region 的引用数量，可以分为少、中、多三种情况。针对这三种情况，RSet 准备了三种不同的数据结构来应对，分别是**稀疏表、细粒度表和粗粒度表**。三种表之间的关系是不断粗化的，如下图所示：



从上图中，我们可以看到：

1. 稀疏表是一个哈希表，当 Region A 对 Region B 的引用很少时，就可以将相关的 card 放到稀疏表里；
2. 细粒度表则是一个真正的 card table，当 Region 之间的引用比较多时，就可以直接使用位图来代替哈希表，因为这能加快查找的速度（使用位操作代替哈希表的查找）；
3. 粗粒度表则是一个区的位图，因为相对来说，区是比较少的，所以粗粒度表的大小也很小。当 Region A 对 Region B 的引用非常多时，就不用再使用 card table 来进行管理了，在回收 Region B 时，直接将 Region A 的全部对象都遍历一次就可以了。

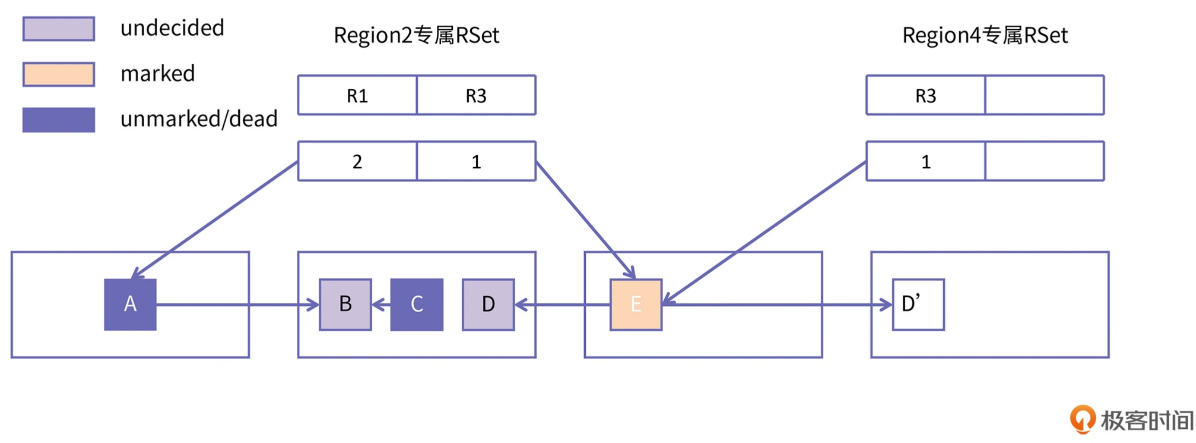
总之，随着其他 Region 对本 Region 的引用关系越多，RSet 存放引用关系使用的表粒度就越粗，这样做主要是为了减少 RSet 记录数，提高定位效率。

在解决了跨区引用的问题之后，接下来我们就可以学习 G1 的垃圾清理过程了，这是垃圾回收器真正回收内存的过程，所以它的重要性不言而喻。

垃圾回收的过程

G1 的垃圾清理是通过把活跃的对象，从一个 Region 拷贝到另一个空白 Region，这个空白 Region 隶属于 Survivor 空间。这个过程在 G1 GC 中被命名为**转移（Evacation）**。它和之前讲到的基于 copy 的 GC 的最大区别是：**它可以充分利用 concurrent mark 的结果快速定位到哪些对象需要被拷贝。**

接下来让我们通过一个例子，来看看 G1 Evacation 的具体过程吧。



在上图中，Region2 是一个待回收的 Region，隶属于 CSet。在它的专属 RSet 中记录了 Region1 的第二个 card 和 Region3 的第一个 card，说明 Region1 和 Region3 有对 Region2 的对象引用，Region4 是一个被选为 Survivor 的空白 Region。

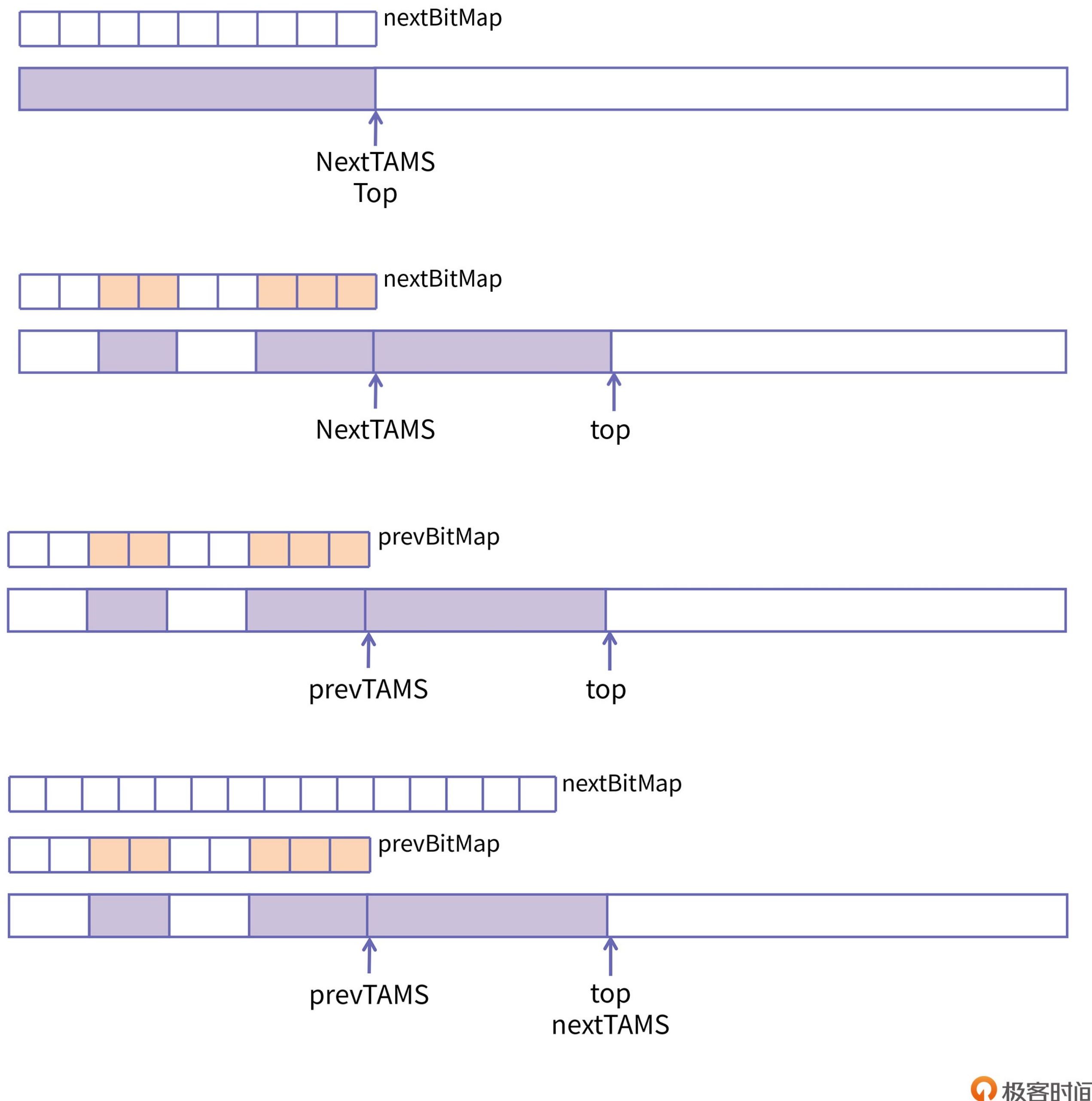
假如 Region1 和 Region3 都经过了并发标记，识别出 A 对象是垃圾对象，而 E 对象是活跃对象。那么，我们就可以从活跃对象 E 开始进行遍历。注意，这一次遍历的目标是把 Region2 中的对象搬移到 Region4。

Region1 中的 A 是垃圾对象，这在并发标记阶段就已经发现了，所以在转移阶段就不会再起作用了。进而，Region2 中的 B、C 也不会被标记到，最终只有对象 D 被拷贝到了 Region4，与此同时，原始 Region2 的 RSet 也会被维护到 Region4。

因为 Evacation 发生的时机是不确定的，在并发标记阶段也可能发生。所以并发标记要使用一个 BitMap 来记录活跃对象，而 Evacation 也需要使用一个 BitMap 来将活跃的对象进行搬移。这就产生了读和写的冲突：**并发标记需要写 BitMap，而 Evacation 需要读 BitMap。**

为了解决这个问题，G1 维护了两个 BitMap，一个名为 nextBitMap，一个名为 prevBitMap。其中，**prevBitMap 是用于搬移活跃对象**，而 **nextBitMap 则用于并发标记记录活跃对象。**

当并发标记开始以后，新的对象仍然有可能会被继续分配。内存管理器把这些对象全部认为是活跃对象。我们来看下面的这个示意图：



在上图中，TAMS 指针，是 Top At Mark Start 的缩写。初始时，prevTAMS，nextTAMS 和 top 指针都指向一个分区的开始位置。

随着业务线程的执行，top 指针不断向后移动。并发标记开始时（图 1），nextTAMS 记录下当前的 top 指针，并且针对 nextTAMS 之前的对象进行活跃性扫描，扫描的结果就存放在 nextBitMap 中（图 2）。

当并发标记结束以后，nextTAMS 的值就记录在 prevTAMS 中，并且 nextBitMap 也赋值给 prevBitMap。如果此时发生了 Evacation，则 prevBitMap 已经可用了。如果没有发生 Evacation，那么 nextBitMap 就会清空，为下一轮并发标记做准备。这样就可以保证，在任意时刻开启 Evacation 的话，prevBitMap 总是可用的（图 3）。

在并发标记开始以后，再创建的对象，其实就是 nextTAMS 指针到 top 指针之间的对象，这些对象全部认为是活跃的（注意观察图中紫色部分）。

我们再从对象活跃性的角度理解两个 TAMS 指针和 top 的关系。当并发标记开始时，nextTAMS 就固定了，但是 top 还是可能继续向后移，所以 nextTAMS 和 top 之间的对象在这次标记过程中都被认为是活跃对象。当 Evacation 开始时，它只使用 prevBitMap 的信息，显然 prevBitMap 中的信息只能覆盖到 prevTAMS 处，所以从 prevTAMS 到 top 的对象就都认为是活跃的。

top 指针是一个 Region 内已分配区域和未分配区域的界限。通过 TAMS 和 BitMap，GC 线程可以清楚地知道一个 Region 内活跃对象的分布，不仅可以确定 Evacation 的范围，还可以用来计算一个 Region 的垃圾比例，为 CSet 选择提供参考。

好啦，关于 G1 的算法原理，我们就先介绍到这里吧，下面让我们一起看看 G1 有哪些常用参数吧，因为掌握 G1 中重要的参数的意义，才能帮助你对 G1 GC 进行参数调优。

G1 常用参数

G1 的默认参数已经被调整得很好了，大多数情况下，不需要再调整。但是，也不排除特殊情况，因此我们还是需要掌握一些 GC 参数，具体列表如下：

选项	默认	作用
G1HeapRegionSize	32m	指定每个区的size
G1ConcRefinementThreads	0	指定Refine线程的数量，GC根据CPU核数进行合理推断
MaxGCPauseMillis	200ms	G1 GC的最大停顿时间，JVM会努力做到，但不能保证绝对满足
ParallelGCThreads	0	并行执行GC的线程数，JVM会根据CPU核数自行推断
InitiatingHeapOccupancyPercent	45	老年代内存占用总空间达到45%以后启动并发标记
ParallelRefProcEnabled	false	是否要开启并发处理弱引用。当系统中弱引用比较多时建议打开
UseAdaptiveIHOP	false	JDK9以后引入，表示自适应地采用合适的IHOP

极客时间

这个表格中最重要、也是你平时最有可能用到的参数，就是 **MaxGCPauseMillis**。它设置了期望的最大停顿时间。MaxGCPauseMillis 设置的越小，可以控制的停顿时间就越短。但是如果设置得太短，可能会引起 Full GC，代价十分昂贵。

其次，比较关键的参数是 **InitiatingHeapOccupancyPercent（IHOP）**，它的作用是在老年代的内存空间达到一定百分比之后，启动并发标记。当然，这更进一步是为了触发 mixed GC，以此来回收老年代。如果一个应用老年代对象产生速度较快，可以尝试适当调小 IHOP。

总结

好了，今天这节课就到这里来，我们一起来回顾一下这节课的重点内容。这节课，我们首先介绍了 G1 特点，明确了分区的意义。然后我们讲到了 G1 的堆空间划分策略，G1 的每个分区都可以单独管理，空**闲 Region 可以用来当做 Survivor 空间，Homongous 区是用来存放大对象的**，在回收过程中，和老年代同等对待。

然后，我们重点分析了 write barrier 的两个作用，一个是**维护记录集**，一个是**解决漏标问题**。

G1 的记录集是与 Region 一一对应的，是一种 point-in 类型的记录集。它仍然采用 dirty card 的设计，将 dirty card 存放在记录集中。记录集为了管理 dirty card，区分了三种粒度，分别是**稀疏表，细粒度表和粗粒度表。**

解决漏标问题则是采用了 SATB 的设计，保证了在 GC 开始的瞬间活跃的对象就始终是活跃的。

接下来，我们解释了 G1 的两种垃圾回收模式，分别是 **young GC** 和 **mixed GC**。young GC 只回收年轻代，mixed GC 回收全部年轻代和部分老年代。

G1 的垃圾清理过程与普通的 copy-based 不同。我们说，G1 的 Evacation 可能发生在并发标记阶段，为了保证 Evacation 在并发标记阶段可以知道哪些对象是活的、需要被拷贝，我们介绍了两个 BitMap 和 TAMS 指针。这样一来，在 Evacation 进行的过程中，管理器就有依据判断一个 Region 中哪些对象是应该被拷贝的。

最后，我还给你讲了在实际工作中需要掌握的几个 G1 参数，尤其是 **MaxGCPauseMillis 和 IHOP**。MaxGCPauseMillis 用来设置期望最大停顿时间，IHOP 用来调整并发标记的处理时机，调整老年代回收的及时性。

思考题

请你思考：如何可以进一步减少垃圾回收的最大停顿时间？欢迎在留言区分享你的想法，我在留言区等你。

吊打面试官

- G1 中使用的SATB是什么意思，它的主要功能是什么样的?

G1中使用的SATB意思是说，在并发标记开始那一时刻活跃的对象，在并发标记阶段都会被认为是活跃对象。就好比在并发开始的瞬间对所有活跃对象做了一个逻辑上的“快照”，并不需要额外的内存来存放对象，具体实现方式是使用了deletion barrier，即便一个对象的引用关系全部消失，仍然把它当做活跃对象来对待。

因为使用了SATB队列，不需要应用线程来完成标记的工作，将对象放到SATB队列等候GC线程处理即可，所以SATB的主要功能是可以减少write barrier的开销，从而减少应用线程的压力。

高频面试真题

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！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 分代算法：基于生命周期的内存管理

下一篇 不定期福利第一期 | 海纳：我是如何学习计算机知识的？

训练营推荐

Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等大厂真题

18 大知识点专项练习

大厂面试流程解析

可复用的面试方法

面试前要做的准备

精选留言（1）

写留言

一子三木

2021-12-17

Garbage First原来是指老年代 Region 的垃圾占比高，就有可能优先被回收。因为年轻代是全部回收。