G1垃圾回收器详解

在前一篇的文章《HotSpot垃圾回收算法概述》里面，对于Serial, Parallel和CMS几种垃圾回收器做了比较详细的描述。但是对于G1的叙述是比较粗糙的。这篇文章则是提供了G1垃圾回收器的详细分析。

概述

G1垃圾回收器是在Java7 update 4之后引入的一个新的垃圾回收器。G1是一个分代的，增量的，并行与并发的标记-复制垃圾回收器。它的设计目标是为了适应现在不断扩大的内存和不断增加的处理器数量，进一步降低暂停时间（pause time），同时兼顾良好的吞吐量。G1回收器和CMS比起来，有以下不同：
1. G1垃圾回收器是compacting的，因此其回收得到的空间是连续的。这避免了CMS回收器因为不连续空间所造成的问题。如需要更大的堆空间，更多的floating garbage。连续空间意味着G1垃圾回收器可以不必采用空闲链表的内存分配方式，而可以直接采用bump-the-pointer的方式；
2. G1回收器的内存与CMS回收器要求的内存模型有极大的不同。G1将内存划分一个个固定大小的region，每个region可以是年轻代、老年代的一个。内存的回收是以region作为基本单位的；

G1还有一个及其重要的特性：软实时（soft real-time）。所谓的实时垃圾回收，是指在要求的时间内完成垃圾回收。“软实时”则是指，用户可以指定垃圾回收时间的限时，G1会努力在这个时限内完成垃圾回收，但是G1并不担保每次都能在这个时限内完成垃圾回收。通过设定一个合理的目标，可以达到90%,99.9%甚至以上的垃圾回收时间都在这个时限内的目标。

数据结构

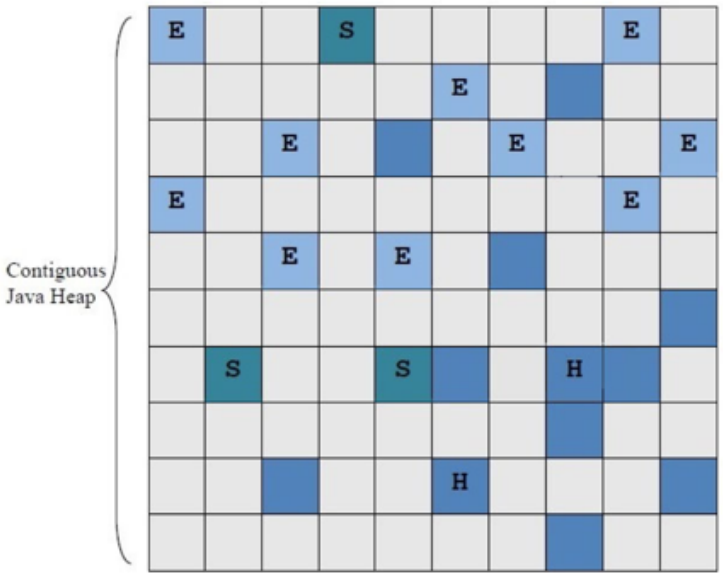
在了解G1垃圾回收器的算法前，先熟悉在G1中使用的一些数据结构和概念是有用的。如果只是想大概了解一下G1垃圾回收器，那么此节可以跳过。即便想详细了解G1回收器的细节也可以先跳过这一节，直接阅读下面的算法详解，遇到了不明白的数据结构和概念，再回来此处寻找解释。

G1垃圾回收器的复杂难懂，有很大一部分原因是因为这些数据结构。

Heap Region

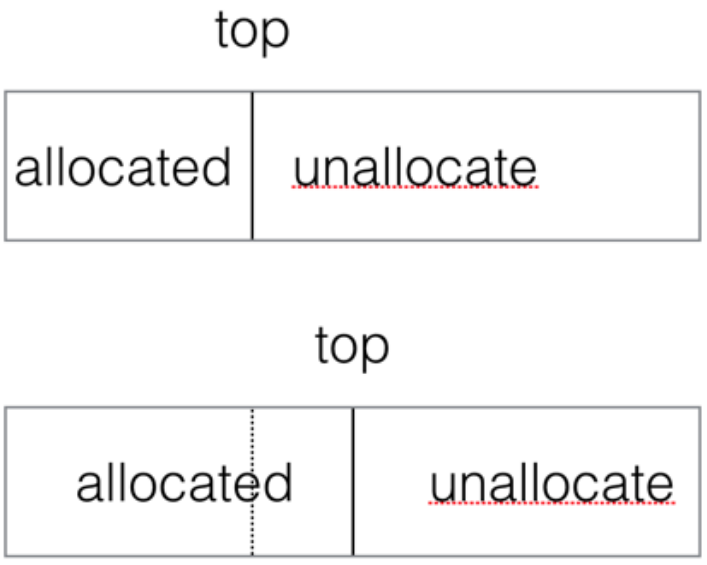
本质上来说，G1垃圾回收器依然是一个分代垃圾回收器。但是它与一般的回收器所不同的是，它引入了额外的概念，Region。G1垃圾回收器把堆划分成一个个大小相同的Region。在HotSpot的实现中，整个堆被划分成2048左右个Region。每个Region的大小在1-32MB之间，具体多大取决于堆的大小。

G1垃圾回收器的分代也是建立在这些Region的基础上的。对于Region来说，它会有一个分代的类型，并且是唯一一个。即，每一个Region，它要么是young的，要么是old的。还有一类十分特殊的Humongous。所谓的Humongous，就是一个对象的大小超过了某一个阈值——HotSpot中是Region的1/2，那么它会被标记为Humongous。如果我们审视HotSpot的其余的垃圾回收器，可以发现这种对象以前被称为大对象，会被直接分配老年代。而在G1回收器中，则是做了特殊的处理。
G1并不要求相同类型的region要相邻。换言之，就是G1回收器不要求它们连续。当然在逻辑上，分代依旧是连续的。因此，一种典型的分配可能是：



注：图片来自资料9

其中E代表的是Eden，S代表的是Survivor，H代表的是Humongous，剩余的深蓝色代表的是Old（或者Tenured），灰色的代表的是空闲的region。
每一个分配的Region，都可以分成两个部分，已分配的和未被分配的。它们之间的界限被称为top。总体上来说，把一个对象分配到Region内，只需要简单增加top的值。这个做法实际上就是bump-the-pointer。过程如下：



Region可以说是G1回收器一次回收的最小单元。即每一次回收都是回收N个Region。这个N是多少，主要受到G1回收的效率和用户设置的软实时目标有关。每一次的回收，G1会选择可能回收最多垃圾的Region进行回收。与此同时，G1回收器会维护一个空间Region的链表。每次回收之后的Region都会被加入到这个链表中。
每一次都只有一个Region处于被分配的状态中，被称为current region。在多线程的情况下，这会带来并发的问题。G1回收器采用和CMS一样的TLABs的手段。即为每一个线程分配一个Buffer，线程分配内存就在这个Buffer内分配。但是当线程耗尽了自己的Buffer之后，需要申请新的Buffer。这个时候依然会带来并发的问题。G1回收器采用的是CAS（Compate And Swap）操作。

> *为线程分配Buffer的过程大概是：*
>
> *1. 记录top值；*

> *2. 准备分配；*
> *3. 比较记录的top值和现在的top值，如果一样，则执行分配，并且更新top的值；否则，重复1；*

显然的，采用TLABs的技术，就会带来碎片。举例来说，当一个线程在自己的Buffer里面分配的时候，虽然Buffer里面还有剩余的空间，但是却因为分配的对象过大以至于这些空闲空间无法容纳，此时线程只能去申请新的Buffer，而原来的Buffer中的空闲空间就被浪费了。Buffer的大小和线程数量都会影响这些碎片的多寡。

Remember Set和Card Table

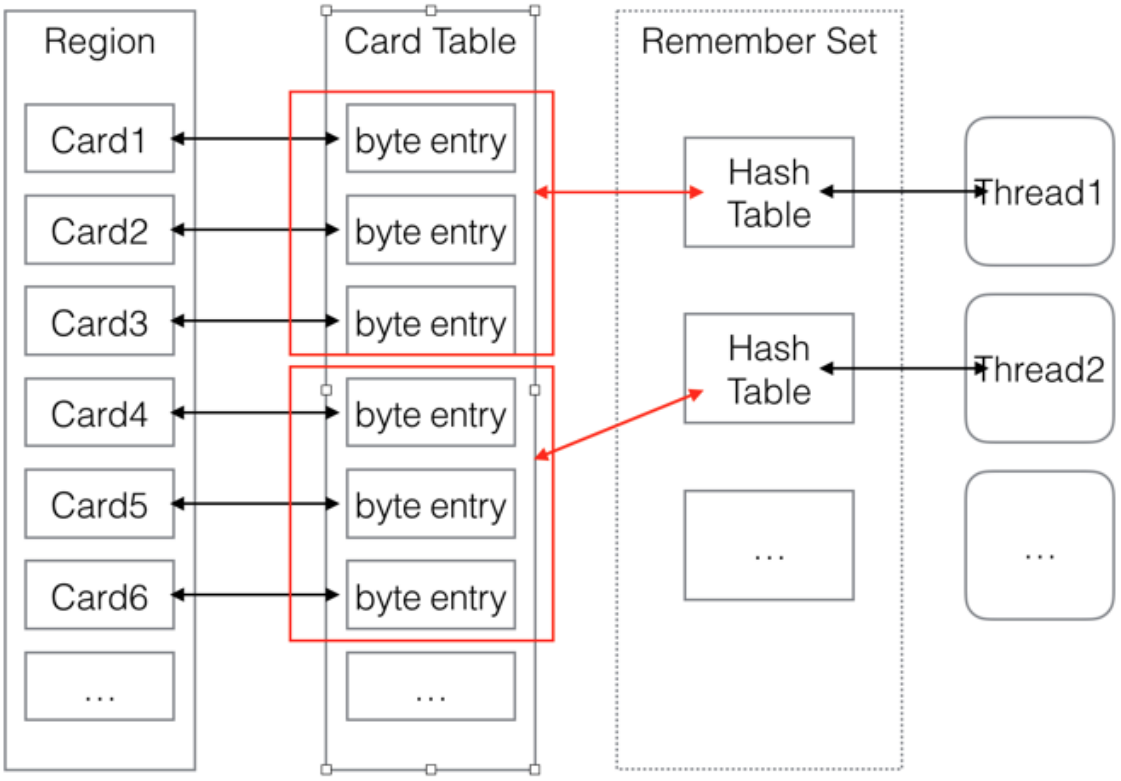
RS(Remember Set)是一种抽象概念，用于记录从非收集部分指向收集部分的指针的集合。
在传统的分代垃圾回收算法里面，RS(Remember Set)被用来记录分代之间的指针。在G1回收器里面，RS被用来记录从其他Region指向一个Region的指针情况。因此，一个Region就会有一个RS。这种记录可以带来一个极大的好处：在回收一个Region的时候不需要执行全堆扫描，只需要检查它的RS就可以找到外部引用，而这些引用就是initial mark的根之一。

那么，如果一个线程修改了Region内部的引用，就必须要去通知RS，更改其中的记录。为了达到这种目的，G1回收器引入了一种新的结构，CT(Card Table)——卡表。每一个Region，又被分成了固定大小的若干张卡(Card)。每一张卡，都用一个Byte来记录是否修改过。卡表即这些byte的集合。实际上，如果把RS理解成一个概念模型，那么CT就可以说是RS的一种实现方式。

> *从第一感觉，或者出于直觉的考虑，使用一个bit来记录一张卡是否被修改过，就已经足够了。而使用一个byte会造成更多的空间开销。但是实际上，使用一个byte来记录一张卡是否被修改过，会比使用一个bit来记录效率更高。更多细节参阅资料3。*

在RS的修改上也会遇到并发的问题。因为一个Region可能有多个线程在并发修改，因此它们也会并发修改RS。为了避免这样一种冲突，G1垃圾回收器进一步把RS划分成了多个哈希表。每一个线程都在各自的哈希表里面修改。最终，从逻辑上来说，RS就是这些哈希表的集合。哈希表是实现RS的一种通常的方式之一。它有一个极大的好处就是能够去除重复。这意味着，RS的大小将和修改的指针数量相当。而在不去重的情况下，RS的数量和写操作的数量相当。

整个关系如下：



图中RS的虚线表名的是，RS并不是一个和Card Table独立的，不同的数据结构，而是指RS是一个概念模型。实际上，Card Table是RS的一种实现方式。

Remember Set的写屏障

写屏障是指，在改变特定内存的值（实际上也就是写入内存）的时候额外执行的一些动作。在大多数的垃圾回收算法中，都利用到了写屏障。写屏障通常用于在运行时探测并记录回收相关指针(interesting pointer)，在回收器只回收堆中部分区域的时候，任何来自该区域外的指针都需要被写屏障捕获，这些指针将会在垃圾回收的时候作为标记开始的根。JAVA使用的其余的分代的垃圾回收器，都有写屏障。举例来说，每一次将一个老年代对象的引用修改为指向年轻代对象，都会被写屏障捕获，并且记录下来。因此在年轻代回收的时候，就可以避免扫描整个老年代来查找根。

G1垃圾回收器的写屏障和RS是相辅相成的，也就是记录Region内部的指针。这种记录发生在写操作之后。对于一个写屏障来说，过滤掉不必要的写操作是十分有必要的。这种过滤既能加快赋值器的速度，也能减轻回收器的负担。G1垃圾回收器采用的双重过滤
1. 过滤掉同一个Region内部引用；
2. 过滤掉空引用；

过滤掉这两个部分之后，可以使RS的大小大大减小。

G1的垃圾回收器的写屏障使用一种两级的log buffer结构：
1. global set of filled buffer：所有线程共享的一个全局的，存放填满了的log buffer的集合；
2. thread log buffer：每个线程自己的log buffer。所有的线程都会把写屏障的记录先放进去自己的log buffer中，装满了之后，就会把log buffer放到 global set of filled buffer中，而后再申请一个log buffer；

> *可以内容可以参阅资料4的11.8节*

Collect Set

Collect Set(CSet)是指，在Evacuation阶段，由G1垃圾回收器选择的待回收的Region集合。G1垃圾回收器的软实时的特性就是通过CSet的选择来实现的。对应于算法的两种模式fully-young generational mode和partially-young mode，CSet的选择可以分成两种：
1. 在fully-young generational mode下：顾名思义，该模式下CSet将只包含young的Region。G1将调整young的Region的数量来匹配软实时的目标；
2. 在partially-young mode下：该模式会选择所有的young region，并且选择一部分的old region。old region的选择将依据在Marking cycle phase中对存活对象的计数。G1选择存活对象最少的Region进行回收。

SATB(snapshot-at-the-beginning)

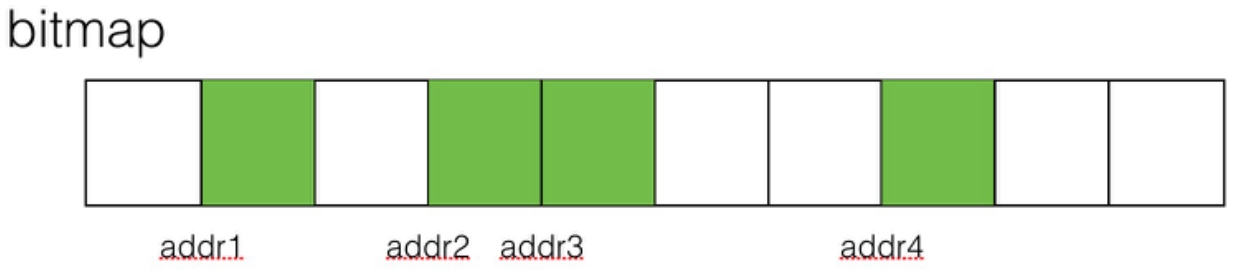
SATB(snapshot-at-the-beginning)，是最开始用于实时垃圾回收器的一种技术。G1垃圾回收器使用该技术在标记阶段记录一个存活对象的快照(“logically takes a snapshot of the set of live objects in the heap at the start of marking cycle”)。然而在并发标记阶段，应用可能修改了原本的引用，比如删除了一个原本的引用。这就会导致并发标记结束之后的存活对象的快照和SATB不一致。G1是通过在并发标记阶段引入一个写屏障来解决这个问题的：每当存在引用更新的情况，G1会将修改之前的值写入一个log buffer（这个记录会过滤掉原本是空引用的情况），在最终标记(final marking phase)阶段扫描SATB，修正SATB的误差。

SATB的log buffer如RS的写屏障使用的log buffer一样，都是两级结构，作用机制也是一样的。

> *细节可以参阅资料2，6*

Marking bitmaps和TAMS

Marking bitmap是一种数据结构，其中的每一个bit代表的是一个可用于分配给对象的起始地址。举例来说：



其中addrN代表的是一个对象的起始地址。绿色的块代表的是在该起始地址处的对象是存活对象，而其余白色的块则代表了垃圾对象。
G1使用了两个bitmap，一个叫做previous bitmap，另外一个叫做next bitmap。previous bitmap记录的是上一次的标记阶段完成之后的构造的bitmap；next bitmap则是当前正在标记阶段正在构造的bitmap。在当前标记阶段结束之后，当前标记的next bitmap就变成了下一次标记阶段的previous bitmap。
TAMS(top at mark start)变量，是一对用于区分在标记阶段新分配对象的变量，分别被称为previous TAMS和next TAMS。在previous TAMS和next TAMS之间的对象则是本次标记阶段时候新分配的对象。如图：

白色region代表的是空闲空间，绿色region代表是存活对象，橙色region代表的在此次标记阶段新分配的对象。注意的是，在橙色区域的对象，并不能确保它们都事实上是存活的。

算法详解

整个算法可以分成两大部分：
1. Marking cycle phase：标记阶段，该阶段是不断循环进行的；
2. Evacuation phase：该阶段是负责把一部分region的活对象拷贝到空Region里面去，然后回收原本的Region空间，该阶段是STW(stop-the-world)的；

而算法也可以分成两种模式：
1. fully-young generational mode：有时候也会被称为young GC，该模式只会回收young region，算法是通过调整young region的数量来达到软实时目标的；
2. partially-young mode：也被称为Mixed GC，该阶段会回收young region和old region，算法通过调整old region的数量来达到软实时目标；

有趣的地方是不论处在何种模式之下，yong region都在被回收的范围内。而old region只能期望于Mixed GC。但是，如同在CMS垃圾回收器中遇到的困境一样，Mixed GC可能来不及回收old region。也就说，在需要分配老年代的对象的时候，并没有足够的空间。这个时候就只能触发一次full GC。

算法会自动在young GC和mixed GC之间切换，并且定期触发Marking cycle phase。HotSpot的G1实现允许指定一个参数InitiatingHeapOccupancyPercent，在达到该参数的情况下，就会执行marking cycle phase。

算法并不使用在对象头增加字段来标记该对象，而是采用bitmap的方式来记录一个对象被标记的情况。这种记录方法的好处就是在使用这些标记信息的时候，仅仅需要扫描bitmap而已。G1统计一个region的存活的对象，就是依赖于bitmap的标记。

Marking Cycle Phase

算法的Marking cycle phase大概可以分成五个阶段：
1. Initial marking phase：G1收集器扫描所有的根。该过程是和young GC的暂停过程一起的；
2. Root region scanning phase：扫描Survivor Regions中指向老年代的被initial mark phase标记的引用及引用的对象，这一个过程是并发进行的。但是该过程要在下一个young GC开始之前结束；
3. Concurrent marking phase：并发标记阶段，标记整个堆的存活对象。该过程可以被young GC所打断。并发阶段产生的新的引用（或者引用的更新）会被SATB的write barrier记录下来；
4. Remark phase：也叫final marking phase。该阶段只需要扫描SATB(Snapshot At The Beginning)的buffer，处理在并发阶段产生的新的存活对象的引用。作为对比，CMS的remark需要扫描整个mod union table的标记为dirty的entry以及全部根；
5. Cleanup phase：清理阶段。该阶段会计算每一个region里面存活的对象，并把完全没有存活对象的Region直接放到空

闲列表中。在该阶段还会重置Remember Set。该阶段在计算Region中存活对象的时候，是STW(Stop-the-world)的，而在重置Remember Set的时候，却是可以并行的；

Initial marking phase

该阶段扫描所有的根，与CMS类似。所不同的是，该阶段是和young GC一起的。这里的young GC实际上是指的就是fully-young generational mode。

> *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Guide的原文是"This phase is piggybacked on a normal (STW) young garbage collection"。*

Root region scanning phase

该过程主要是扫描Survivor region中指向老年代的，在initial mark phase标记的引用及其引用的对象。这是一个很奇怪的步骤，因为在前面不论是Parallel Collector还是CMS，都没有这么一个步骤。

要理解这一点，要注意的是，算法的两种模式，不论是young GC还是mixed GC，都需要回收young region。因为实际上RS是不记录从young region出发的指针，例如，这部分指针包括young region – young region，也包括young-region – old region指针。那么就可能出现一种情况，一个老年代的存活对象，只被年轻代的对象引用。在一次young GC中，这些存活的年轻代的对象会被复制到Survivor Region，因此需要扫描这些Survivor region来查找这些指向老年代的对象的引用，作为并发标记阶段扫描老年代的根的一部分。

在理解了这一点的基础上，那么对于阶段必须在下一次young GC启动前完成的要求，也就理解了。因为如果第二次的young GC启动了，那么这个过程中，survivor region就可能发生变化。这个时候执行root region phase就会产生错误的结果。

Concurrent marking phase

在标记阶段，会使用到一个marking stack的东西。G1不断从marking stack中取出引用，递归扫描整个堆里的对象图，并且在bitmap上进行标记。这个递归过程采用的是深度遍历，会不断把对象的域入栈。

在并发标记阶段，因为应用还在运行，所以可能会有引用变更，包括现有引用指向别的对象，或者删除了一个引用，或者创建了一个新的对象等。G1采用的是使用SATB的并发标记算法。

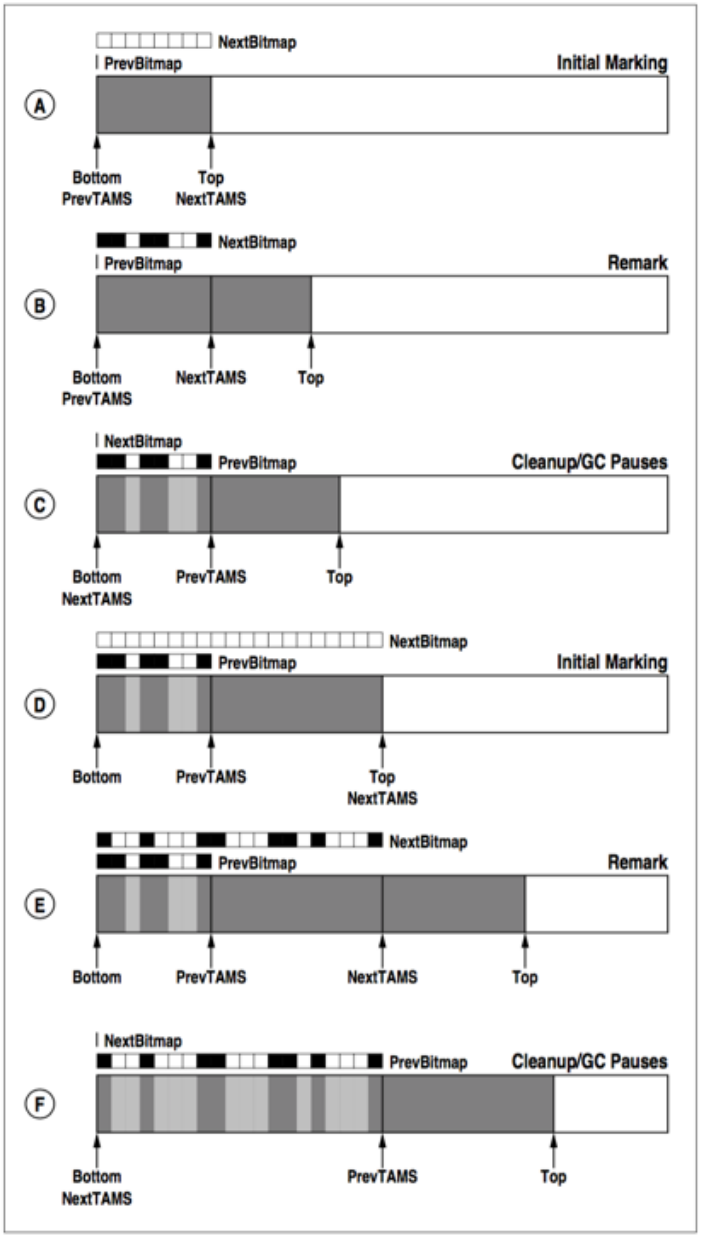
> *在资料6中记录了使用SATB的两条原则：*
> *1. All accessible cells at the beginning of the garbage collection are eventually marked during the marked phase;*
> *2. Newly alocated cells during the garbage collection are never collected during the sweep phase of that garbage collection*

在G1中，该算法的关键在于，如果在并发标记的时候，出现了引用修改（不包含新分配内存给对象），那么写屏障会把这些引用的原始值捕获下来，记录在log buffer中。而后再处理。后续的所有的标记，都是从原来的值出发，而不是从新的值出发的。

SATB是一个逻辑上存在概念，在实际中并没有任何真的实际的数据结构与之对应。叫这个名字，是因为，一旦进入了concurrent marking阶段，那么该在该阶段的运行过程中，即便应用修改了引用，但是因为SATB的写屏障记录下来了原始的值，在遍历整个堆查找存活对象的时候，使用的依然是原来的值。这就是在逻辑上保持了一个snapshot at the beginning of concurrent marking phase。

在处理新创建的对象，G1采用了不同的方式。G1用了两个TAMS变量了判断新创建的对象。一个叫做previous TAMS，一个叫做next TAMS。位于两者之间的对象就是新分配的对象。

并发标记阶段，bitmap和TAMS的作用如图：



注：图片引自资料2

该图的详细解释如下：
1. A是第一次marking cycle的initial marking阶段。next bitmap尚未标记任何存活对象，而此时的previous TAMS被初始化为region内存地址起始值，next TAMS被初始化为top。top实际上就是一个region未分配区域和已分配区域的分界点；

2. B是经过concurrent marking阶段之后，进入了remark阶段。此时存活对象的扫描已经完成了，因此next bitmap构造好了，刚好代表的是当下状态中region中的内存使用情况。注意的是，此时top已经不再与next TAMS重合了，top和next TAMS之间的就是在前面标记阶段之时，新分配的对象；
3. C代表的是clean up阶段。C和B比起来，next bitmap变成了previous bitmap，而在bitmap中标记为垃圾（也就是白色区域的）的对应的region的区域也被染成了浅灰色。这并不是指垃圾对象已经被清扫了，仅仅是标记出来了。同时next TAMS和previous TAMS也交换了角色；
4. D代表的是下一个marking cycle的initial marking阶段，该阶段和A类似，next TAMS重新被初始化为top的值；
5. EF就是BC的重复；

Remark phase

该阶段是一个STW的阶段。引入该阶段的目的，是为了能够达到结束标记的目标。要结束标记的过程，要满足三个条件：
1. concurrent marking已经追踪了所有的存活对象；
2. marking stack是空的；
3. 所有的log都被处理了；

前两个条件是很容易达到的，但是最后一个是很困难的。如果不引入一个STW的remark过程，那么应用会不断的更新引用，也就是说，会不断的产生log，因而永远也无法达成完成标记的条件。

Clean up

该阶段主要完成：
1. 统计存活对象，这是利用RS和bitmap来完成的，统计的结果将会用来排序region，以用于下一次的CSet的选择；
2. 重置RSet；
3. 把空闲region放到空闲region列表中；

该阶段比较容易引起误解地方在于，Clean up并不会清理垃圾对象，也不会执行存活对象的拷贝。也就是说，在极端情况下，该阶段结束之后，空闲Region列表将毫无变化，JVM的内存使用情况也毫无变化。

Evacuation

Evacuation阶段STW的，大概可以分成两个步骤：第一个步骤是从Region中选出若干个Region进行回收，这些被选中的Region称为Collect Set（简称CSet）；而第二个步骤则是把这些Region中存活的对象复制到空闲的Region中去，同时把这些已经被回收的Region放到空闲Region列表中。

这两个步骤又可以被分解成三个任务：
1. 根据RS的日志更新RS：只有在处理完了RS的日志之后，RS才能够保证是准确的，完整的，这也是Evacuation是STW的重要原因；
2. 扫描RS和其余的根来确定存活对象：该阶段实际上最主要依赖于RS；
3. 拷贝存活对象：该阶段只要从2中确定的根触发，沿着引用链一直追溯下去，将存活对象复制到新的region就可以。这个过程中，可能有一部分的年轻代对象会被提升到老年代；

Evacuation的时机

Evacuation的触发时机在不同的模式下会有一些不同。在不同的模式下都相同的是，只要堆的使用率达到了某个阈值，就必然会触发Evacuation。这是为了确保在Evacuation的时候有足够的空闲Region来容纳存活对象。

在young GC的情况下，G1会选择N个region作为CSet，该CSet首先需要满足软实时的要求，而一旦已经有N个region已经被分配了，那么就会执行一次Evacuation。

G1会尽可能的执行mixed GC。唯一的限制就是mix GC也需要满足软实时的要求。

G1触发Evacuation的原则大概是：
1. 如果被分配的young region数量满足young GC的要求，那么就会触发young GC；
2. 如果被分配的young region数量不满足young GC，就会进一步考察加上old region的数量，能否满足old GC的要求；

为了理解这一点，可以举例来说，假如回收一个old region的时间是回收一个young region的两倍，也就是young region花费时间T，old region花费2T，在满足软实时目标的情况下，GC只能回收8T的region，那么：
1. 假如应用现在只分配k（k<8）块young region，没有分配任何old region。这个时候又分配了一个old region，那么这个时候会立刻触发一次mixed GC，此次GC会选择k块young region和一块old region；
2. 因此，在这种假设下，只要有可以回收的old region的时候，总是会先回收old region；
3. 在没有任何old region的情况下，才有可能触发young region。

当然，在一般情况下，这些假设是不成立的。读者可以思考一下，在young GC和mixed GC达到软实时的要求下，young region和old region之间回收的花销不同会导致young GC和mixed GC会在什么情况下触发。

资料

1. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide
2. David Detlefs, Christine Flood, Steve Heller, Tony Printezis. Garbage-First Garbage Collection
3. Urs Hölzle. A Fast Write Barrier for Generational Collectors
4. 垃圾回收算法手册——自动内存管理的艺术
5. 请教G1算法的原理——RednaxelaFX的回答
6. Taichi Yuasa. Real-time garbage collection on general-purpose machines.
7. Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors
8. 名词连接贴-RednaxelaFX对Remeber Set和Card Table的解释
9. G1: One Garbage Collector To Rule Them All
10. Poonam Parhar. Understanding G1 GC Logs
11. Getting Started with the G1 Garbage Collector