G1垃圾收集器调研

基本介绍

G1是一种服务器端的垃圾收集器，应用在多处理器和大容量内存环境中，在实现高吞吐量的同时，尽可能的满足**垃圾收集暂停时间**的要求。它是专门针对以下应用场景设计的:

- 像CMS收集器一样，能与应用程序线程并发执行。
- 整理空闲空间更快。
- 需要GC停顿时间更好预测。
- 不希望牺牲大量的吞吐性能。
- 不需要更大的Java Heap。

G1收集器的设计目标是取代CMS收集器，它同CMS相比，在以下方面表现的更出色：

- G1是一个有整理内存过程的垃圾收集器，不会产生很多内存碎片。
- G1的Stop The World(STW)更可控，G1在停顿时间上添加了预测机制，用户可以指定期望停顿时间。

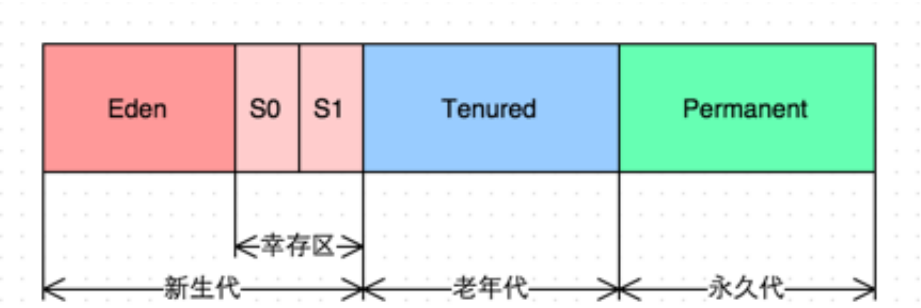
官方原文：

> 1. The Garbage-First (G1) collector is a server-style garbage collector, targeted for multi-processor machines with large memories. It meets garbage collection (GC) pause time goals with a high probability, while achieving high throughput. *The G1 garbage collector is fully supported in Oracle JDK 7 update 4 and later releases*. The G1 collector is designed for applications that:
> 2.
>   - Can operate concurrently with applications threads like the CMS collector.
>   - Compact free space without lengthy GC induced pause times.
>   - Need more predictable GC pause durations.
>   - Do not want to sacrifice a lot of throughput performance.
>   - Do not require a much larger Java heap.
> 3. G1 is planned as the long term replacement for the Concurrent Mark-Sweep Collector (CMS). Comparing G1 with CMS, there are differences that make G1 a better solution. One difference is that G1 is a compacting collector. G1 compacts sufficiently to completely avoid the use of fine-grained free lists for allocation, and instead relies on regions. This considerably simplifies parts of the collector, and mostly eliminates potential fragmentation issues. Also, G1 offers more predictable garbage collection pauses than the CMS collector, and allows users to specify desired pause targets.

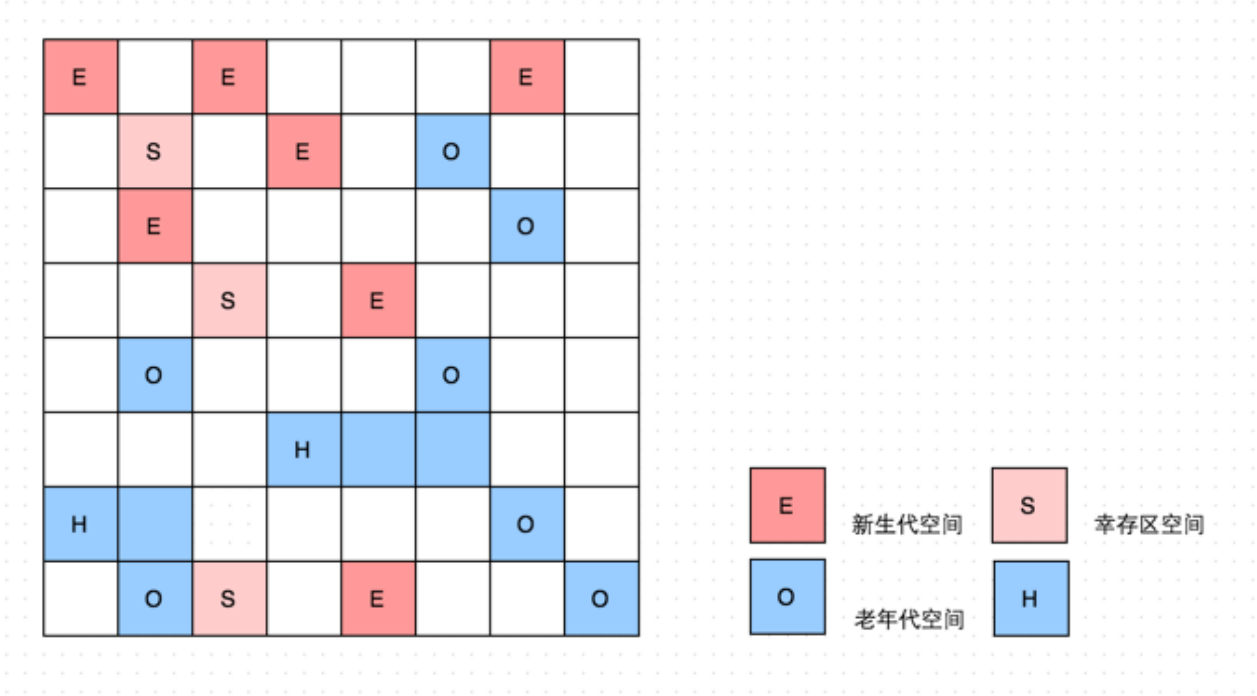
重要概念

Region

传统的GC收集器将连续的内存空间划分为新生代、老年代和永久代（jdk8已去除），这种划分的特点是各代的存储地址是**连续**的。



G1的各代存储地址是**不连续**的，每一代都使用了n个不连续的大小相同的Region，每个Region占有一块连续的虚拟内存地址。在物理上不需要连续，带来了额外的好处：`有的分区内垃圾对象特别多，有的分区内垃圾对象很少，G1会优先回收垃圾对象特别多的分区，这样可以花费较少的时间来回收这些分区的垃圾。`这也就是G1名字的由来，即首先收集垃圾最多的分区。



其中部分Region标记了H，表示这些Region存储的是巨大对象`humongous object、H-obj`，即对象的大小 >= Region一半。

H-obj的特征如下：

> 1.H-obj直接分配到了老年代，防止了反复拷贝移动。
>
> 2.H-obj在`并发标记阶段（global concurrent marking）`的 cleanup 和 full gc 阶段回收。
>
> 3.在分配H-obj之前先检查是否超过`-XX:InitiatingHeapOccupancyPercent`，如果超过的话，就启动"并发标记"，以提早回收，防止 evacuation failures 和 full gc。

Region的大小可以通过参数`-XX:G1HeapRegionSize`设定，取值范围从1M到32M，且是2的指数。如果不设定，那么G1会根据Heap大小自动决定。

```
// 最小值 1M
#define MIN_REGION_SIZE  (      1024 * 1024 )
// 最大值 32M
#define MAX_REGION_SIZE  ( 32 * 1024 * 1024 )
#define TARGET_REGION_NUMBER          2048
void HeapRegion::setup_heap_region_size(size_t initial_heap_size, size_t
max_heap_size) {
  uintx region_size = G1HeapRegionSize;
  // 使用默认值
  if (FLAG_IS_DEFAULT(G1HeapRegionSize)) {
    size_t average_heap_size = (initial_heap_size + max_heap_size) / 2;
    region_size = MAX2(average_heap_size / TARGET_REGION_NUMBER,
                       (uintx) MIN_REGION_SIZE);
  }
  int region_size_log = log2_long((jlong) region_size);
  // 保证region大小是2的幂
  region_size = ((uintx)1 << region_size_log);

  if (region_size < MIN_REGION_SIZE) {
    region_size = MIN_REGION_SIZE;
  } else if (region_size > MAX_REGION_SIZE) {
    region_size = MAX_REGION_SIZE;
  }
}
```

CSet

全称是Collection Set，是辅助GC过程的一种结构。

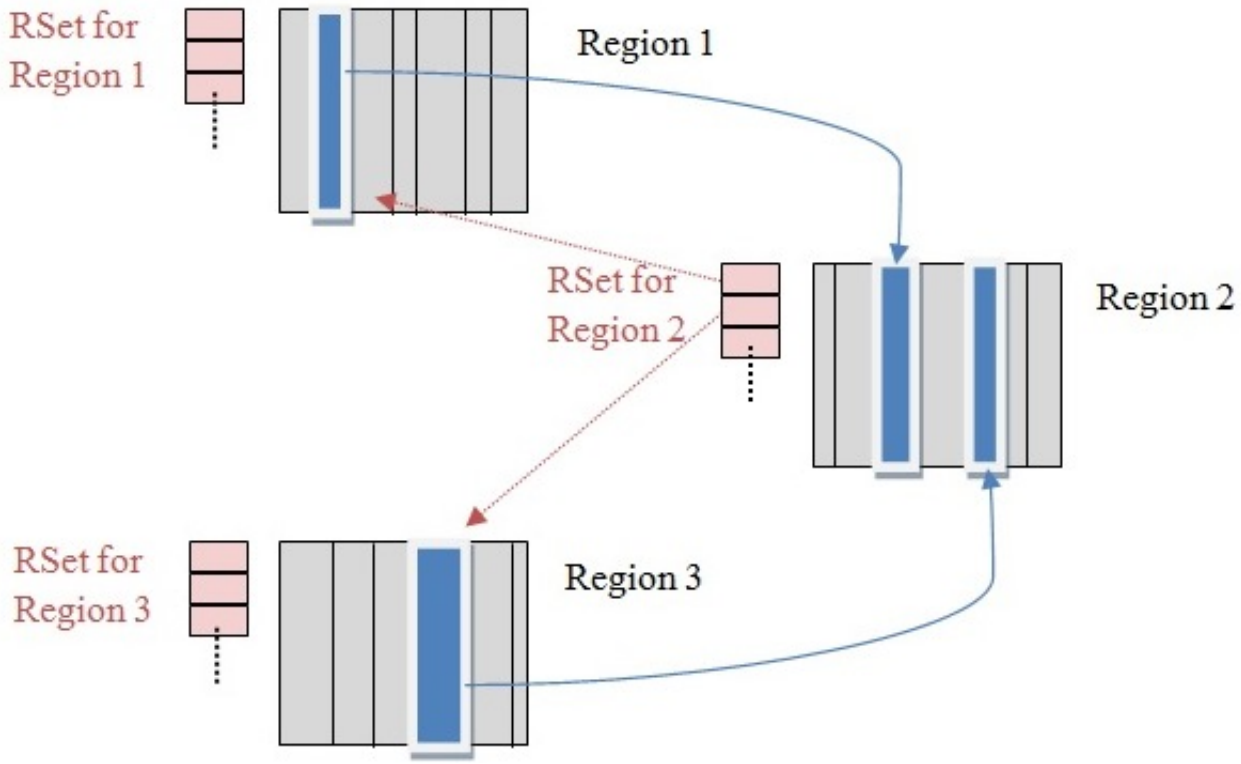
CSet记录了GC要收集的Region集合，集合里的Region可以是任意年代的。在GC的时候，对于`old -> young`和`old -> old`的跨代对象引用，只要扫描对应的CSet中的RSet即可。

RSet

全称是Remembered Set，也是辅助GC过程的一种结构。

逻辑上每个Region都有一个RSet，RSet记录了其他Region中的对象引用本Region中对象的关系，属于points-into结构（谁引用了我的对象）。RSet的价值在于使得垃圾收集器不需要扫描整个堆找到谁引用了当前分区中的对象，只需要扫描RSet即可。

另外Card Table是一种points-out（我引用了谁的对象）的结构，每个Card 覆盖一定范围的Heap（一般为512Bytes）。RSet是在Card Table的基础上实现的：每个Region会记录下别的Region指向自己的指针，并标记这些指针分别在哪些Card的范围内。 RSet其实是一个Hash Table，Key是别的Region的起始地址，Value是一个集合，里面的元素是Card Table的Index。



上图中有三个Region，每个Region被分成了多个Card，在不同Region中的Card会相互引用，Region1中的Card中的对象引用了Region2中的Card中的对象，蓝色实线表示的就是points-out的关系，而在Region2的RSet中，记录了Region1的Card，即红色虚线表示的关系，这就是points-into。（Region2的RSet中会有一条 key = Region1的起始地址， vlaue = 元素包含该Card的index的集合）

对象漏标

垃圾回收的并发标记阶段，GC线程和应用线程是并发执行的，所以一个对象被标记之后，应用线程可能篡改对象的引用关系，从而造成对象的漏标、误标。误标没什么关系，顶多造成浮动垃圾，在下次gc还是可以回收的，但是漏标的后果是致命的，把本应该存活的对象给回收了，从而影响的程序的正确性。

为了解决在并发标记过程中，存活对象漏标的情况，GC HandBook把对象分成三种颜色`即三色标记算法`：

- 黑色：自身以及可达对象都已经被标记
- 灰色：自身被标记，可达对象还未标记
- 白色：还未被标记

所以，漏标的情况只会发生在白色对象中，且满足以下任意一个条件：

- 并发标记时，应用线程给一个黑色对象的引用类型字段赋值了该白色对象
- 并发标记时，应用线程删除所有灰色对象到该白色对象的引用

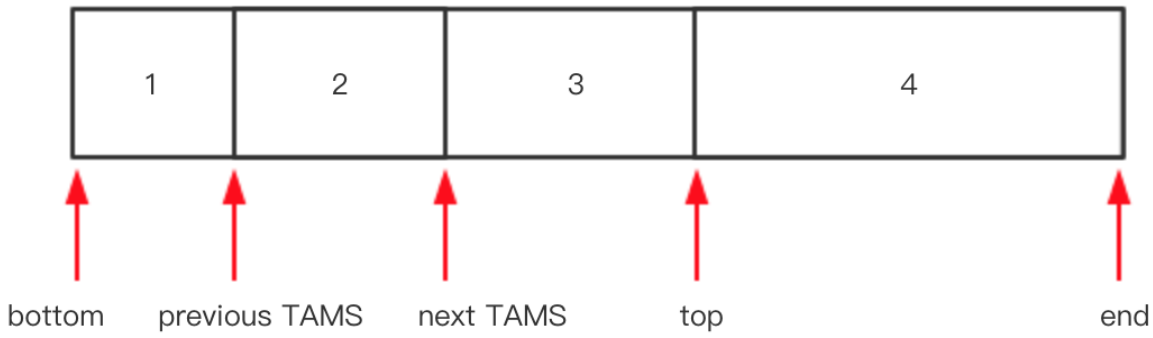
对于第一种情况，利用post-write barrier，记录所有新增的引用关系，然后根据这些引用关系为根重新扫描一遍

对于第二种情况，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重新扫描一遍

SATB

全称是Snapshot-At-The-Beginning，是由Taiichi Yuasa为增量式标记清除垃圾收集器设计的一个标记算法。字面表示GC开始时存活对象的一个快照，用于维持并发GC的正确性。

Region包含了5个指针，分别是bottom、previous TAMS、next TAMS、top和end。



其中previous TAMS、next TAMS是前后两次发生并发标记时的位置，全称`top-at-mark-start`

- 假设第n轮并发标记开始，将该Region当前的top指针赋值给next TAMS，在并发标记标记期间，分配的对象都在[next TAMS, top]之间，SATB能够确保这部分的对象都会被标记，默认都是存活的。
- 当并发标记结束时，将next TAMS所在的地址赋值给previous TAMS，SATB给 [bottom, previous TAMS] 之间的对象创建一个快照Bitmap，所有垃圾对象能通过快照被识别出来。
- 第n+1轮并发标记开始，过程和第n轮一样

> SATB保证了在并发标记过程中新分配对象不会漏标!

但如果在TAMS之前有一个白色对象W，被一个灰色对象G引用，在并发标记扫描到这个字段之前被赋值为null，切断了对象W和对象G之间的引用关系，对象W就有可能漏标，这就是白色对象被漏标的第二种情况。

G1的解决办法则是在引用关系被修改之前，插入一层pre-write barrier

```
template <class T> static void write_ref_field_pre_static(T* field, oop newVal)
{
  T heap_oop = oopDesc::load_heap_oop(field);
  if (!oopDesc::is_null(heap_oop)) {
    // 核心方法
    enqueue(oopDesc::decode_heap_oop(heap_oop));
  }
}

void G1SATBCardTableModRefBS::enqueue(oop pre_val) {
  if (!JavaThread::satb_mark_queue_set().is_active()) return;
  Thread* thr = Thread::current();
  if (thr->is_Java_thread()) {
    JavaThread* jt = (JavaThread*)thr;
    // 入队
    jt->satb_mark_queue().enqueue(pre_val);
  } else {
```

```
    MutexLockerEx x(Shared_SATB_Q_lock, Mutex::_no_safepoint_check_flag);
    JavaThread::satb_mark_queue_set().shared_satb_queue()->enqueue(pre_val);
  }
}
```

通过`G1SATBCardTableModRefBS::enqueue(oop pre_val)`把原引用保存到satb mark queue中，和RSet的实现类似，每个应用线程都自带一个satb mark queue.在下一次的并发标记阶段，会依次处理satb mark queue中的对象，确保这部分对象在本轮GC是存活的。

垃圾回收

G1提供了两种GC模式，Young GC和Mixed GC。

Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。

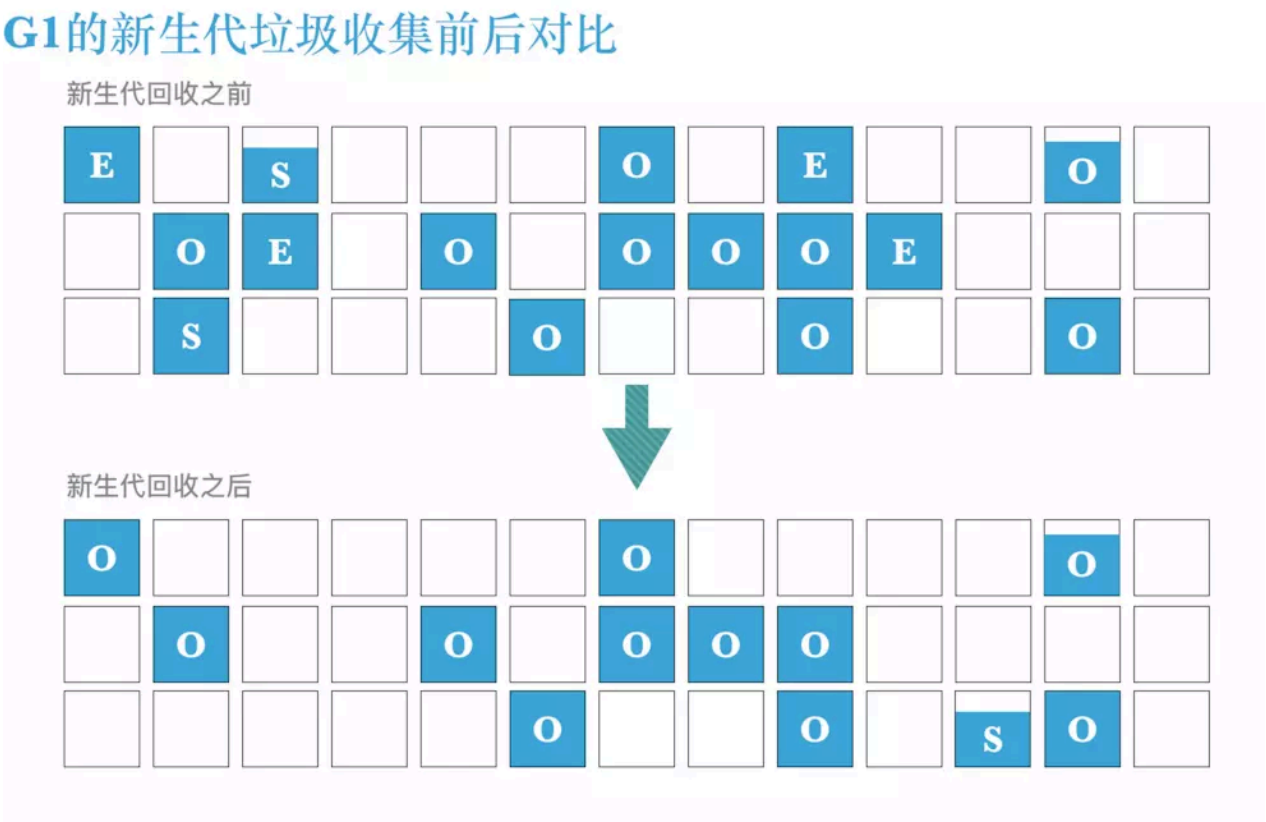
Mixed GC：选定所有年轻代里的Region + 并发标记阶段统计得出收集收益高的若干老年代Region。在用户指定的开销目标范围内尽可能选择收益高的老年代Region。

Mixed GC不是full GC，它只能回收部分老年代的Region，如果mixed GC实在无法跟上程序分配内存的速度，导致老年代填满无法继续进行Mixed GC，就会使用serial old GC（full GC）来收集整个GC heap。

Young GC

Eden区耗尽的时候就会触发Young GC，新生代垃圾收集会对整个新生代进行回收。

> Young GC会stop-the-world。
>
> Young GC是多线程并行执行的。



Young GC过程日志：

```
2019-06-25T15:22:12.407+0800: 21.553: [GC pause (G1 Evacuation Pause) (young), 0.0247670 secs]
   [Parallel Time: 18.7 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 21553.0, Avg: 21553.1, Max: 21553.2, Diff: 0.2]
      [Ext Root Scanning (ms): Min: 0.2, Avg: 1.0, Max: 3.1, Diff: 2.9, Sum: 8.4]
      [Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0]
      [Scan RS (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.5]
      [Code Root Scanning (ms): Min: 0.0, Avg: 1.2, Max: 2.6, Diff: 2.6, Sum: 9.6]
      [Object Copy (ms): Min: 15.5, Avg: 16.2, Max: 17.7, Diff: 2.2, Sum: 129.4]
      [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.0, Sum: 0.2]
      [GC Worker Total (ms): Min: 18.4, Avg: 18.5, Max: 18.6, Diff: 0.2, Sum: 148.3]
      [GC Worker End (ms): Min: 21571.6, Avg: 21571.6, Max: 21571.6, Diff: 0.0]
   [Code Root Fixup: 1.4 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.2 ms]
   [Other: 4.4 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 3.4 ms]
      [Ref Enq: 0.0 ms]
```

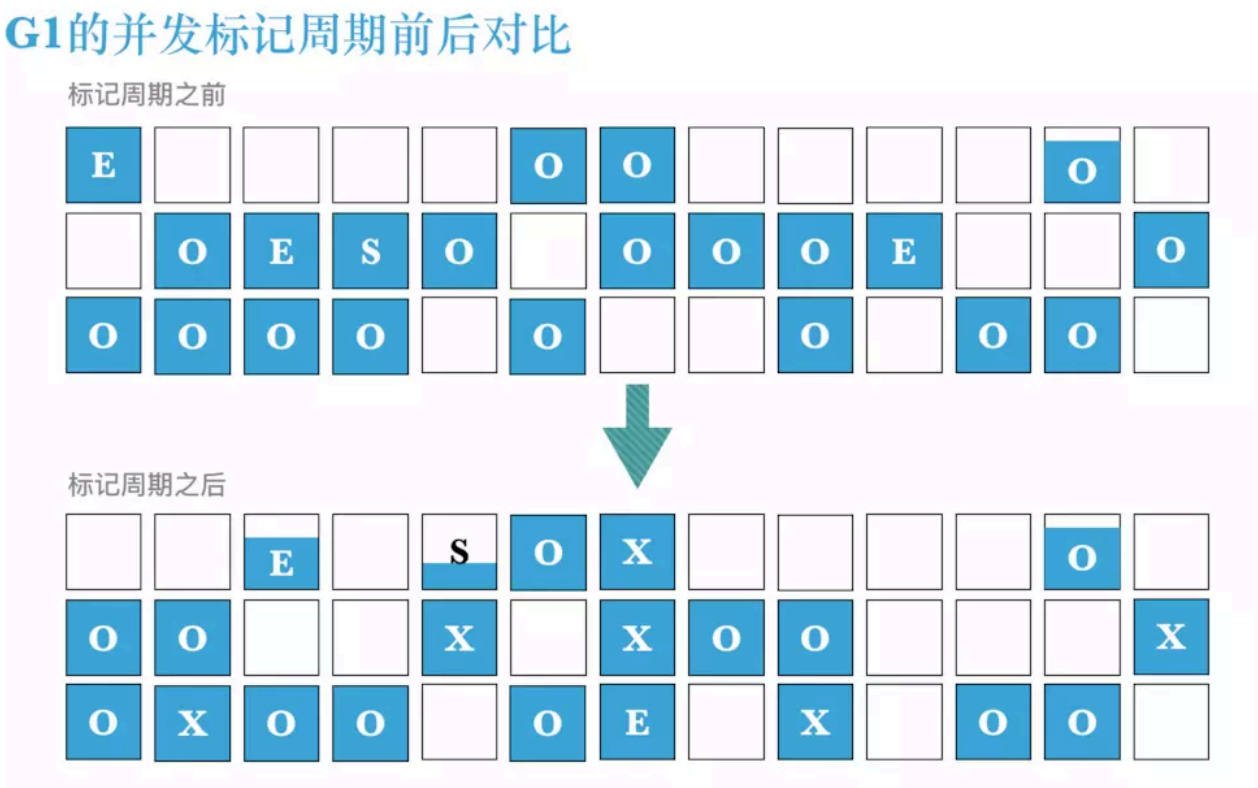
```
      [Redirty Cards: 0.1 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 0.7 ms]
   [Eden: 597.0M(597.0M)->0.0B(585.0M) Survivors: 17.0M->29.0M Heap:
614.0M(1024.0M)->29.0M(1024.0M)]
 [Times: user=0.14 sys=0.01, real=0.02 secs]
```

简要说明：

> GC pause (G1 Evacuation Pause) (young)：新生代对象分配失败，触发Young GC
>
> Update RS：更新RSet的耗时统计，内部Processed Buffers表示每个线程的耗时
>
> Object Copy：将存活的对象拷贝到新Region耗时
>
> Clear CT：清理Card Table耗时

并发标记周期（global concurrent marking）

G1设计了一个标记阈值`the marking threshold`，它描述的是总体Java堆大小的百分比，默认值是45，这个值可以通过命令`-XX:InitiatingHeapOccupancyPercent`来调整，一旦达到这个阈值就回触发一次`并发标记周期`。



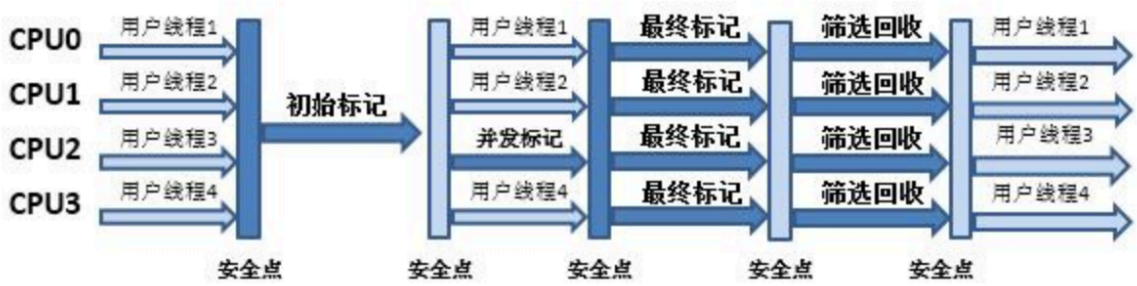
从图中可以注意到：

- 新生代的空间占用情况发生了变化——在并发收集周期中，至少有一次（很可能是多次）新生代垃圾收集；
- 注意到一些分区被标记为X，这些分区属于老年代，它们就是标记周期找出的包含最多垃圾的分区（注意：它们内部仍然保留着数据）；

- 老年代的空间占用在标记周期结束后变得更多，这是因为在标记周期期间，新生代的垃圾收集会晋升对象到老年代，而且标记周期中并不会是否老年代的任何对象。

并发标记周期的多个阶段：

- 初始标记（initial-mark），在这个阶段，应用会经历STW，通常初始标记阶段会跟一次新生代收集一起进行，换句话说——既然这两个阶段都需要暂停应用，G1 GC就重用了新生代收集来完成初始标记的工作。在新生代垃圾收集中进行初始标记的工作，会让停顿时间稍微长一点，并且会增加CPU的开销。初始标记做的工作是设置两个TAMS变量（NTAMS和PTAMS）的值，所有在TAMS之上的对象在这个并发周期内会被识别为隐式存活对象；
- 根分区扫描（root-region-scan），这个过程不需要暂停应用，在初始标记或新生代收集中被拷贝到survivor分区的对象，都需要被看做是根，这个阶段G1开始扫描survivor分区，所有被survivor分区所引用的对象都会被扫描到并将被标记。survivor分区就是根分区，正因为这个，该阶段不能发生新生代收集，如果扫描根分区时，新生代的空间恰好用尽，新生代垃圾收集必须等待根分区扫描结束才能完成。如果在日志中发现根分区扫描和新生代收集的日志交替出现，就说明当前应用需要调优。
- 并发标记阶段（concurrent-mark），并发标记阶段是多线程的，我们可以通过`-XX:ConcGCThreads`来设置并发线程数，默认情况下，G1垃圾收集器会将这个线程总数设置为并行垃圾线程数（`-XX:ParallelGCThreads`）的四分之一；并发标记会利用trace算法找到所有活着的对象，并记录在一个bitmap中，因为在TAMS之上的对象都被视为隐式存活，因此我们只需要遍历那些在TAMS之下的；记录在标记的时候发生的引用改变，SATB的思路是在开始的时候设置一个快照，然后假定这个快照不改变，根据这个快照去进行trace，这时候如果某个对象的引用发生变化，就需要通过pre-write barrier logs将该对象的旧的值记录在一个SATB缓冲区中，如果这个缓冲区满了，就把它加到一个全局的列表中——G1会有并发标记的线程定期去处理这个全局列表。
- 重新标记阶段（remarking），重新标记阶段是最后一个标记阶段，需要暂停整个应用，G1垃圾收集器会处理掉剩下的SATB日志缓冲区和所有更新的引用，同时G1垃圾收集器还会找出所有未被标记的存活对象。这个阶段还会负责引用处理等工作。
- 清理阶段（cleanup），清理阶段真正回收的内存很小，截止到这个阶段,G1垃圾收集器主要是标记处哪些老年代分区可以回收，将老年代按照它们的存活度（liveness）从小到大排列。这个过程还会做几个事情：识别出所有空闲的分区、RSet梳理、将不用的类从metaspace中卸载、回收巨型对象等等。识别出每个分区里存活的对象有个好处是在遇到一个完全空闲的分区时，它的RSet可以立即被清理，同时这个分区可以立刻被回收并释放到空闲队列中，而不需要再放入CSet等待混合收集阶段回收；梳理RSet有助于发现无用的引用。



G1收集器运行示意图

并发标记周期日志：

```
2019-06-25T17:34:37.194+0800: 7966.340: [GC pause (G1 Humongous Allocation)
(young) (initial-mark), 0.0156129 secs]
   [Parallel Time: 13.9 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 7966339.8, Avg: 7966339.9, Max: 7966340.0,
Diff: 0.2]
      [Ext Root Scanning (ms): Min: 1.0, Avg: 1.1, Max: 1.3, Diff: 0.3, Sum:
8.9]
      [Update RS (ms): Min: 11.8, Avg: 11.9, Max: 12.0, Diff: 0.2, Sum: 94.8]
         [Processed Buffers: Min: 157, Avg: 165.3, Max: 170, Diff: 13, Sum:
1322]
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.4]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum:
0.0]
      [Object Copy (ms): Min: 0.5, Avg: 0.6, Max: 0.6, Diff: 0.1, Sum: 4.8]
      [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
      [GC Worker Total (ms): Min: 13.6, Avg: 13.6, Max: 13.8, Diff: 0.2, Sum:
109.1]
      [GC Worker End (ms): Min: 7966353.5, Avg: 7966353.5, Max: 7966353.6,
Diff: 0.0]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.5 ms]
   [Other: 1.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.1 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.6 ms]
      [Humongous Reclaim: 0.1 ms]
      [Free CSet: 0.1 ms]
   [Eden: 328.0M(610.0M)->0.0B(609.0M) Survivors: 4096.0K->5120.0K Heap:
762.8M(1024.0M)->149.1M(1024.0M)]
 [Times: user=0.05 sys=0.00, real=0.02 secs]
2019-06-25T17:34:37.209+0800: 7966.355: [GC concurrent-root-region-scan-start]
2019-06-25T17:34:37.211+0800: 7966.357: [GC concurrent-root-region-scan-end,
0.0016811 secs]
2019-06-25T17:34:37.211+0800: 7966.357: [GC concurrent-mark-start]
2019-06-25T17:34:37.242+0800: 7966.388: [GC concurrent-mark-end, 0.0308061
secs]
2019-06-25T17:34:37.242+0800: 7966.388: [GC remark 7966.388: [Finalize Marking,
0.0004156 secs] 7966.389: [GC ref-proc, 0.0001542 secs] 7966.389: [Unloading,
0.0072260 secs], 0.0085303 secs]
 [Times: user=0.00 sys=0.00, real=0.01 secs]
2019-06-25T17:34:37.251+0800: 7966.397: [GC cleanup 225M->217M(1024M),
0.0013890 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
2019-06-25T17:34:37.253+0800: 7966.399: [GC concurrent-cleanup-start]
2019-06-25T17:34:37.253+0800: 7966.399: [GC concurrent-cleanup-end, 0.0000115
secs]
```

简要说明

> GC pause (G1 Humongous Allocation) (young) (initial-mark) : 在一次young gc中进行初识标记
>
> concurrent-root-region-scan-start：根分区扫描开始
>
> concurrent-mark-start：并发标记阶段开始
>
> remark：重新标记阶段
>
> cleanup：清理阶段

Mixed GC

Mixed GC触发的时机并不像Young GC那么明显，它由一些参数控制。

参数	说明	默认值
-XX:G1HeapWastePercent	该值表示愿意浪费的堆百分比。当可回收的垃圾占比大于该值时，则会触发Mixed GC。	10
-XX:G1MixedGCLiveThresholdPercent	该值表示如果一个分区中的存活对象比例超过n，就不会被当作垃圾分区。	65
-XX:G1MixedGCCountTarget	一次并发标记周期后，最多执行Mixed GC的次数。	8
-XX:G1OldCSetRegionThresholdPercent	一次Mixed GC期间可被回收的最大的老年代的比例上限（选入CSet的最多old generation region数量）	10

Mixed GC日志：

```
2019-06-25T17:37:56.925+0800: 8166.070: [GC pause (G1 Evacuation Pause)
(mixed), 0.0439753 secs]
   [Parallel Time: 41.4 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 8166071.0, Avg: 8166071.2, Max: 8166071.3,
Diff: 0.3]
      [Ext Root Scanning (ms): Min: 1.3, Avg: 1.5, Max: 2.3, Diff: 1.0, Sum:
12.3]
      [Update RS (ms): Min: 38.2, Avg: 38.6, Max: 38.7, Diff: 0.5, Sum: 309.1]
         [Processed Buffers: Min: 103, Avg: 112.4, Max: 124, Diff: 21, Sum:
899]
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum:
0.0]
      [Object Copy (ms): Min: 0.4, Avg: 0.6, Max: 0.7, Diff: 0.3, Sum: 4.6]
```

```
      [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
      [GC Worker Total (ms): Min: 40.7, Avg: 40.8, Max: 41.0, Diff: 0.3, Sum:
326.4]
      [GC Worker End (ms): Min: 8166112.0, Avg: 8166112.0, Max: 8166112.0,
Diff: 0.0]
   [Code Root Fixup: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.8 ms]
   [Other: 1.7 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.2 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.5 ms]
      [Humongous Reclaim: 0.1 ms]
      [Free CSet: 0.1 ms]
   [Eden: 51.0M(51.0M)->0.0B(601.0M) Survivors: 0.0B->2048.0K Heap:
513.8M(1024.0M)->313.3M(1024.0M)]
 [Times: user=0.38 sys=0.00, real=0.04 secs]
```

JDK各版本修改

JDK版本	修改	说明
JDK 9	G1成为默认垃圾回收器	http://openjdk.java.net/jeps/248
JDK 9	对G1中的大对象进行了测试	http://openjdk.java.net/jeps/278
JDK 10	引入并行Full GC	之前的G1执行Full GC采用单线程mark-sweep-compact算法http://openjdk.java.net/jeps/307
JDK 12	可终止的Mixed GC	http://openjdk.java.net/jeps/344
JDK 12	及时返回未使用的已分配内存	该功能默认关闭 http://openjdk.java.net/jeps/346

参数介绍

参数	说明
-XX:+UseG1GC	使用 G1 收集器
-XX:MaxGCPauseMillis	指定目标停顿时间，默认值 200 毫秒
-XX:InitiatingHeapOccupancyPercent	整堆使用达到这个比例后，触发并发标记周期，默认45%
-XX:G1HeapWastePercent	该值表示愿意浪费的堆百分比。当可回收的垃圾占比大于该值时，则会触发Mixed GC。默认10%
-XX:G1MixedGCLiveThresholdPercent	该值表示如果一个分区中的存活对象比例超过n，就不会被当作垃圾分区。默认65%
-XX:G1MixedGCCountTarget	一次并发标记周期后，最多执行Mixed GC的次数。默认8
-XX:G1OldCSetRegionThresholdPercent	一次Mixed GC期间可被回收的最大的老年代的比例上限（选入CSet的最多old generation region数量）。默认10%