G1收集器

G1设计的一个重要目标是设置stop-the-world阶段的持续时长和频率，因为垃圾收集器可预测，可配置。事实上，G1是一款软实时的收集器，意味着你可以给它设置明确的运行目标。你可以要求stop-the-world阶段不超过 x milliseconds在给定的y milliseconds时长范围之内，比如，在给定的s内不超过5s。G1收集器尽自己最大努力高概率实现目标（但不是必然，它会是硬实时）。

为了实现它，G1建立在一系列的观察上。首先，heap去不是必须Young和Old代分配连续的空间。相反，heap区分成一定数量（代表性的2048）的小的heap区来分配对象。单个的区域可能是Eden区，Survivor区，Old区。所有逻辑的Eden区和Survivor区合称为Young代，所有的Old区组合在一起称为Old代：

这允许GC避免一次回收整个heap区，取而代之递增处理问题：每次只有collection set调用region的子集。每个阶段期间所有的Young region被回收，但同样的只包含一部分old region：

G1的另一个新特性是并发阶段期间估算每一个region里包含存活数据的数量。这个被用于建立collection set：region包含的垃圾越多，越先被回收。因此名称是：garbage-first 收集器。

为了激活JVM中G1收集器，按照下面的命令执行你的应用：

```
java -XX:+UseG1GC com.mypackages.MyExecutableClass
```

疏散（Evacuation）阶段：Fully Young

在应用程序生命周期的开始阶段，在并发阶段执行之前，G1获取不到任何附加信息，因此它的最初功能是full-yong模式。当Young代塞满了，应用线程暂停，Young区的存活数据被复制到Survivor区域，任何空闲区域因此变成Survivor区。

复制对象过程被叫做疏散（Evacuation）， 它的工作方式和我们之前看到其他Young收集器几乎是一样的。疏散阶段full logs相当大，因此在第一次full-young 疏散阶段我们略去一些不相关的片段。并发阶段之后我们会解释大量细节。补充一点，由于log记录的全量尺寸，并行阶段和“其他”阶段的细节被抽取成独立的片段：

```
0.134: [GC pause (G1 Evacuation Pause) (young), 0.0144119 secs]1
    [Parallel Time: 13.9 ms, GC Workers: 8]2
        …3
    [Code Root Fixup: 0.0 ms]4
    [Code Root Purge: 0.0 ms]5
    [Clear CT: 0.1 ms]
    [Other: 0.4 ms]6
        …7
    [Eden: 24.0M(24.0M)->0.0B(13.0M) 8Survivors: 0.0B->3072.0K 9Heap: 24.0M(256.0M)-
>21.9M(256.0M)]10
     [Times: user=0.04 sys=0.04, real=0.02 secs] 11
```

1. G1阶段清理Young区域。JVM启动之后的134ms阶段开始，通过钟墙时间检测阶段持续了0.0144s。
2. 表明下列活动被8个并行GC线程实施耗费13.9ms(real time)。
3. 省略部分，细节在下面的系列片段。
4. 释放数据结构用于管理并行活动。通常应该是靠近zero。这通常顺序完成。
5. 清除更多数据结构，通常应该非常快，不是几乎等于0。顺序完成。
6. 混杂其他活动，它们的很多是并行的。
7. 细节可以看下面的章节。
8. 阶段前后的Eden区使用大小和容量大小。
9. 阶段前后被用于Survivor区的空间。
10. 阶段前后的heap区总使用大小和容量大小。
11. GC时间期间，不同类别的时长：

    .user-回收期间GC线程消耗的总的cpu时间。
    .sys-调用系统或等待系统事件的耗费时长。
    .应用程序的停顿的时钟时间。GC期间并发活动时长理论上接近（user time+sys time）GC线程数量消费的时长,这种情况下用了8个线程。注意的是由于一些活动不是并行执行，它会超过一定比率。

大多数重大事件被多个专用GC线程完成。它们的活动在如下面片段的描述：

```
[Parallel Time: 13.9 ms, GC Workers: 8]1
     [GC Worker Start (ms)2: Min: 134.0, Avg: 134.1, Max: 134.1, Diff: 0.1]
    [Ext Root Scanning (ms)3: Min: 0.1, Avg: 0.2, Max: 0.3, Diff: 0.2, Sum: 1.2]
    [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: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
    [Code Root Scanning (ms)4: Min: 0.0, Avg: 0.0, Max: 0.2, Diff: 0.2, Sum: 0.2]
    [Object Copy (ms)5: Min: 10.8, Avg: 12.1, Max: 12.6, Diff: 1.9, Sum: 96.5]
    [Termination (ms)6: Min: 0.8, Avg: 1.5, Max: 2.8, Diff: 1.9, Sum: 12.2]
        [Termination Attempts7: Min: 173, Avg: 293.2, Max: 362, Diff: 189, Sum: 2346]
    [GC Worker Other (ms)8: Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
    GC Worker Total (ms)9: Min: 13.7, Avg: 13.8, Max: 13.8, Diff: 0.1, Sum: 110.2]
    [GC Worker End (ms)10: Min: 147.8, Avg: 147.8, Max: 147.8, Diff: 0.0]
```

1. 表明下列活动被8个并行GC线程实施耗费13.9ms(real time)。
2. 线程开始活动的合计时间，在阶段的开始时间匹配时间戳。如果Min和Max差别很大，它也许表明太多线程被使用或者JVM里的GC进程CPU时间被机器上其他进程盗用。
3. 扫描外部（非heap）Root消耗的时间例如clasloader,JNI引用，JVM系统等等。展示消耗时间，“Sum”是cpu时间。
4. 扫描来自真实code Root的时长：局部变量等等。
5. 从回收区域复制存活对象花费的时间。
6. GC线程确定它们到达安全点消耗的时间，没有多余工作完成，然后终止。
7. 工作线程尝试终止的次数。实际上线程发现有任务需要完成的时候尝试失败，过早去终止。
8. 其他琐碎的活动不值得在日志里独立片段展示。
9. 任务线程总共花费的时间。
10. 任务线程完成工作的时间戳。通常它们因该大值相等，另一方面它也许显示出太多线程无所事事，或者繁复的上下文工作。

此外，Evacuation阶段期间一些混杂活动被执行。我们会讲解它们的一部分在下面的片段。剩余部分随后讲解。

```
[Other: 0.4 ms]1
    [Choose CSet: 0.0 ms]
    [Ref Proc: 0.2 ms]2
    [Ref Enq: 0.0 ms]3
    [Redirty Cards: 0.1 ms]
    [Humongous Register: 0.0 ms]
    [Humongous Reclaim: 0.0 ms]
    [Free CSet: 0.0 ms]4
```

1. 混杂其他的活动，大多数并行执行。
2. 处理非强引用的时间：清除或者确定不需要清理。
3. 顺序处理将剩下的非强引用从引用队列中移除出去。
4. 释放收集集合里面区域花费的时间以便它们适用于下一次分配。

并发标记

从上面章节看出G1借鉴了CMS的许多理念，因此可以方便你充分理解之前的阶段。虽然在一些方式上不尽相同，但是并发标记的目标非常类似。G1并发标记使用STAB(Snapshot-At-The-Beginning)方法，意味着在周期的开始阶段标记所有存活对象，即使在收集期间已经调整。存活对象被允许建立在每个区域（region）活跃性上，以便收集结合快速选择。

这些信息随后被用于执行Old代GC。它可以完全并发执行，一个仅仅包含垃圾的region被标记，或者一个Old region “stop-the-world”evacuation阶段包含垃圾和存活对象。

并发标记开始于heap区已使用空间足够大。默认的，占45%，但是可以被JVM 选项 InitiatingHeapOccupancyPercent 改变。类似CMS，G1并发标记有一些小阶段组成，它们中一些完全并发，一些则不得不暂停应用线程。

阶段1：初始标记。

这个阶段标记所有从GC Root可达的对象。在CMS里，他需要“stop-the-world”,但是在G1， Evacuation阶段它可以并行执行，因此它的上限事最小的。你可以通过evacuation阶段第一行添加“(initial-mark)”留意下GC日志里的这个阶段：

```
1.631: [GC pause (G1 Evacuation Pause) (young) (initial-mark), 0.0062656 secs]
```

阶段2：Root region扫描。

这个阶段标记从所谓的root区域可达所有的存活对象，也就是标记周期中间没有必要分配的非空的对象。因为移除标记周期中的填充会导致异常，这个阶段必须在下一个阶段开始之间完成。如果它提前开始，它会提前中止root扫描，然后等待完成。在目前的实现中，root区域在syrvivor区中：它们占据Young代小部分空间，在下一个 Evacuation 阶段被禁止回收。

```
1.362: [GC concurrent-root-region-scan-start]
1.364: [GC concurrent-root-region-scan-end, 0.0028513 secs]
```

阶段3：并发标记。

这个阶段和CMS的阶段非常相似：它简单统计对象，在特别的bitmap中标记可以访问的对象。为了保证STAB语义论，为了达到标记目的，通过可感知应用线程放弃先前的引用G1 GC需要所有的并发线程更新到对象统计模式。

通过使用写屏障（Pre-Write barriers，不要混淆于Post-Write barriers，以及涉及多线程的memory barriers，随后进行分析）。它们的职责是，每当G1并发标记期间你写进一个字段，在所谓的log buffer里面存储之前结果，被并发标记线程处理。

```
1.364: [GC concurrent-mark-start]
1.645: [GC concurrent-mark-end, 0.2803470 secs]
```

阶段4：再标记。

这个阶段需要“stop-the-world”,类似之前的CMS里面看到，在标记阶段完成。对于G1，对于遗留的部分它短暂的暂停应用线程去阻塞并发更新日志和处理它们，标记并发标记开始的时候没有标记的存活对象。这个阶段执行一些额外的清理工作，例如，引用（查看Evacuation阶段日志）处理，或者卸载的class。

```
1.645: [GC remark 1.645: [Finalize Marking, 0.0009461 secs] 1.646: [GC ref-proc, 0.0000417 secs]
1.646: [Unloading, 0.0011301 secs], 0.0074056 secs]
[Times: user=0.01 sys=0.00, real=0.01 secs]
```

阶段5：清除。

最后的阶段准备即将到来的Evacuation阶段，计算heap区所有的存活对象，通过预期的GC效率排序这些region。它通常执行所有活动的整理工作，为了下一次并发标记迭代维持内部状态。

最后但同样重要的是，包含不再使用的对象的region在这个阶段被回收。这个阶段的一些部分是并发执行的，例如回收空region,大多数活跃性估算，但你在应用线程不干涉的期间通常需要短暂的“stop-the-world”阶段区确定方案。日志“stop-the-world”阶段和下图类似：

```
1.652: [GC cleanup 1213M->1213M(1885M), 0.0030492 secs]
[Times: user=0.01 sys=0.00, real=0.00 secs]
```

一旦发现只包含垃圾对象的region，日志格式会有些区别，类似于：

```
1.872: [GC cleanup 1357M->173M(1996M), 0.0015664 secs]
[Times: user=0.01 sys=0.00, real=0.01 secs]
1.874: [GC concurrent-cleanup-start]
1.876: [GC concurrent-cleanup-end, 0.0014846 secs]
```

疏散阶段：混合

并发清除可以空出来整个old代是令人兴奋，但是事实情况是不是每次都这样。并发标记已经成功完成之后，G1会安排一次混合回收，不仅仅是回收young region的垃圾，还把old region的一部分放进collection set中。

一次混合疏散阶段不是经常紧随并发标记阶段结束之后。有一系列的规则和试探影响这个阶段。例如，并发空出来old region中的大块区域是可能的，然而根本没有必要去做。

它们可能因此在并发标记结束和混合evacuation阶段之间简单发生一系列full-young evacuation阶段。

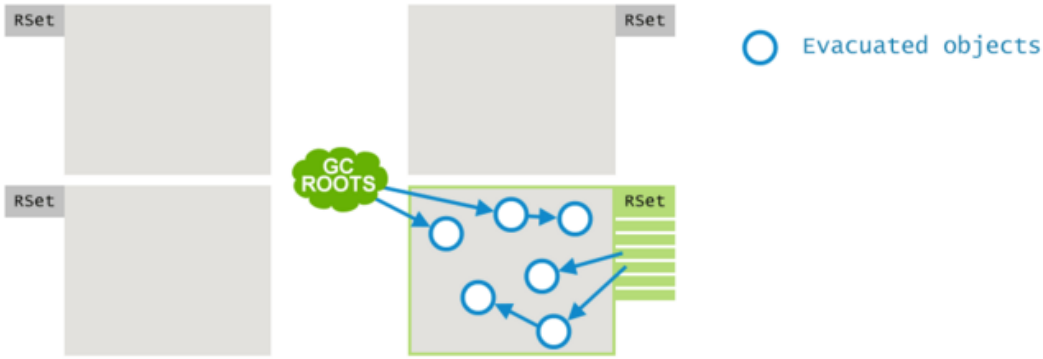
old区准确的对象被添加进collection set，保证其添加顺序，根据一些规则挑选出顺序。这些包含了为了达到应用程序的软实时性能目标。并发标记期间，活跃的和GC 效率的数据被回收，还有一些JVM配置选项。混合回收进程和初期分析的full-young gc一样庞大，但是这次我们会包含remembered set的子集。

Remembered set运行heap不同region使用独立的收集器。例如，当回收区域A,B和C,我们必须知道D和E中任何一个引用它们，去确定它们的活跃性。但是统计整个heap区花费很长时间，销毁整个增量收集，因此最佳化被破坏。很像为了使用其他GC算法独立手机Young region我们使用卡表（ Card Table），在G1中我们使用Remembered Sets。

正如上面插图展示那样，每一个region有一个remembered sets列出外部引用指向这个区域。这些被看作额外的GC Roots。注意的是并发标记期间old区域被判断为垃圾的对象，即使外部引用它们会被忽略：那种情况下被当作垃圾的参照图：

下一步操作和其他收集器一样：多个并行GC线程计算出哪些对象存活，哪些是垃圾：

最后，存活对象被移到Survior区域中，如果有必要则新建。空的orgion被释放出来，用户再次存储对象：



为了维护Remembered Sets，应用运行期间，每当写入操作执行的时候触发一个Post-Write Barrier。如果一个引用跨越region，也就是一个region指向另一个region，目标region的 Remembered Set存入相对应的entry中。为了减少write barrier,将card放进Remember Set过程异步执行，突出性能最优化。但是基本上将dirty card信息放进本地缓存的方式存入Write barrier，一个专门GC线程将信息引用region的Remember set中。

混合模式中，对照fully young模式log展示一些有趣的方面：

```
[Update RS (ms)1: Min: 0.7, Avg: 0.8, Max: 0.9, Diff: 0.2, Sum: 6.1]
[Processed Buffers2: Min: 0, Avg: 2.2, Max: 5, Diff: 5, Sum: 18]
[Scan RS (ms)3: Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 0.8]
[Clear CT: 0.2 ms]4
[Redirty Cards: 0.1 ms]5
```

1. 自并发执行Remember Set以后，我必须确保真正收集开始之前still-buffered cards被执行。如果数量很多，并发GC线程无法负载。他可能是，举例来说，势不可挡的到来的字段修改的数量，或者CPU资源不足。
2. 每一个任务线程操作local buffer的数量。
3. 从Remember Set中扫描引用的数量。
4. 清理card table中的card花费的时间。Remember set通过简单移除“dirty”（表示filed被修改）状态进行清理工作。
5. card table超着ditry card的占用位置花费的时间。GC自己操作通过heap中发生突变被定义为位置占用，例如引用队列。

总结

这个应该建立在充分理解G1如果工作基础之上，这些当然为了简介，需要我们忽略相当多的一些实现细节，像humongous objects的细节。综合考虑，G1是HotSpot中现有的最先进的收集器产品，在G1上，他被HotSpot的工程师无所不用其极地改进，在即将到来的Java 新版本。

正如他我所看到的，G1修正了CMS的广为人知的问题，从阶段可预测到heap碎片。使得应用不再受限于CPU利用率，但是对个别选项十分敏感。G1很可能是对HotSpot用户来说最好的选择，尤其是运行最新版本的Java。然而，这性能升不是毫无代价：G1吞吐量归功于附加的write barrier和更多后台活动线程。如果应用是吞吐量优先或者CPU使用率100%，不关注个别阶段，CMS,甚至Parallel或许是更好的选择。

唯一可行选择合适的GC算法和设置的方式通过尝试和错误，但是我还在下一章节给出一般的参考。

注意的是G1很有可能是java 9默认的GC收集器：http://openjdk.java.net/jeps/248