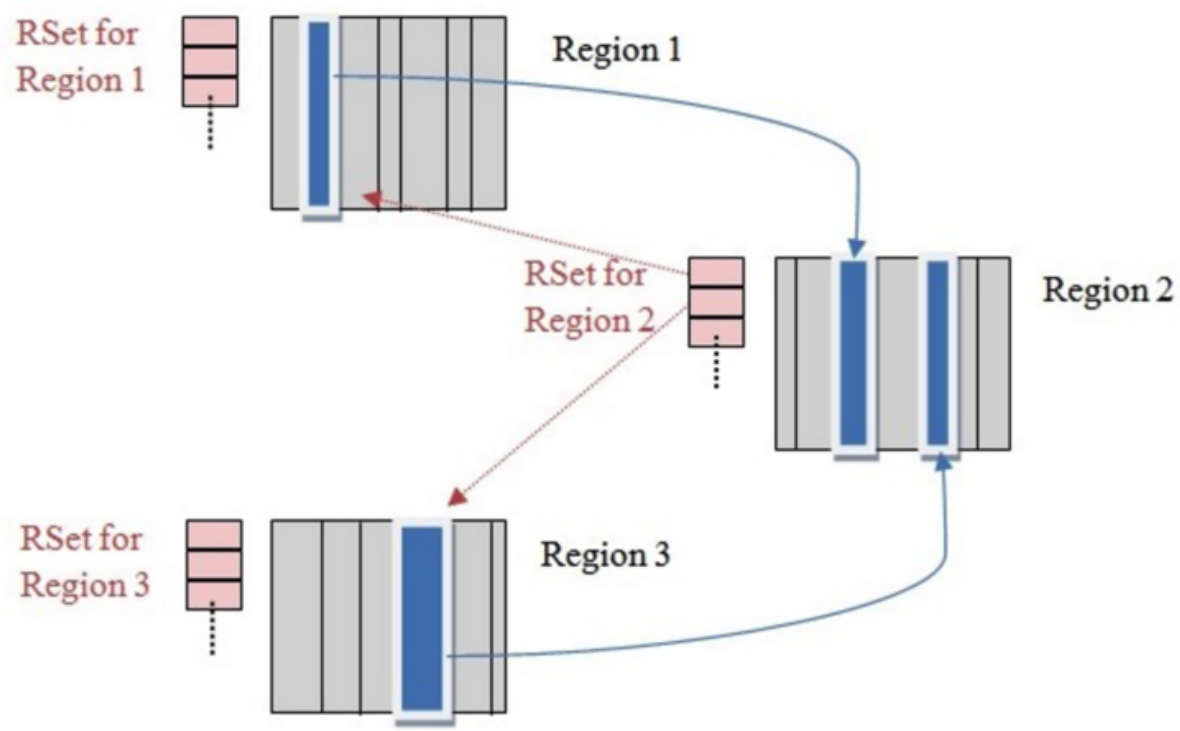
调整G1收集器窍门

这是由两部分组成的系列的第二篇关于G1垃圾回收器的文章，你可以在2013.07.17的InfoQ上找到第一部分：G1: One Garbage Collector To Rule Them All。

在我我们了解如何调整G1 GC之前，首先我必须了解G1定义的关键概念。在这篇文章里，我会首先介绍概念，然后讨论如何调整G1（适当的时候）。

Remembered Sets

从之前的文章回忆起它：Remembered Sets（RSets）是每一个region里面帮助G1 GC追踪外部指向这个region的引用。因此现在，取代因为引用指向这个region扫描整个heap区，G1只需要扫描RSets。



1: Remembered Sets

我们看一下示意图。上面的示意图向我们展示三个region(灰色)。Region 1, Region 2和Region 3和它们关联的RSets(粉红色)，RSets代表一些card的集合。Region 1和Region 3恰好引用Region2里的对象。因此，Region 2的RSets记录了两个引用Region2的对象，Region2就是“owning region”。

这里有两个概念帮助理解RSets：

1. Post-write barriers
2. Concurrent refinement threads

屏障代码在写操作之后（因此名称是“post-write barrier”），为了记录帮助追踪跨region更新。包含更新引用字段的card更新可靠的日志缓冲区。一旦这些缓冲区满了，它们就停止工作。Concurrent refinement threads处理这些缓冲区日志。

注意到Concurrent refinement threads通过并发更新它们来帮助维护RSets(通常在应用运行期间)。Concurrent refinement threads调度是分层的。开始只有少量的线程被部署，最终添加取决于更新充满缓冲区操作的数量。concurrent refinement threads的最大数量可以由**-XX:G1ConcRefinementThreads**或者**-XX:ParallelGCThreads**控制。如果concurrent refinement threads的数量赶不上装满缓冲区的数量，然后mutator threads处理缓冲区过程-通常你应该努力去避免这中情况。

OK,回到RSets-每一个Region有一个RSets。RSets由三种级别的粒度-Sparse, Fine和Coarse。一个Per-Region-Table (PRT)是RSet存储颗粒度级别一个抽象。sparse PRT是一个包含Card目录的hash table。G1 GC内部维护这些card。card包含来自region的引用，这个region的引用是card到“owning region”的关联的地址。fine-grain PRT是一个开放的hash table，每一个entry代表一个指向owning region的引用的region。region里面的card目录，是一个bitmap。当达到fine-grain PRT的最大容量，coarse grain bitmap里面的相应的coarse-grained bit被设置，相应地entry从 fine grain PRT删除。coarse bitmap有一个每个region对应的bit。coarse grain map设置bit意味着关联的region包含到“owning region”的引用。

Collection Set (CSet)是一个gc期间即将被回收的region的set。对于 Young gc,CSet只包含Young Region，对于混合回收，CSets包含Young Region和Old Region。

如果CSets包含许多携带coarsened RSets的Region(注意，“coarsening of RSets”是根据RSets贯穿不同级别颗粒度的过渡期定义的)，然后你会看到扫描RSets消耗时间的增长。GC阶段这些扫描时间就是GC日志里面的“Scan RS (ms)”。如果RSets扫描时间相当于GC阶段总时间很高，或者你的应用中它们表现很高，然后通过使用诊断选项**-XX:+G1SummarizeRSetStats**请观察你的 Young GC 日志输出的“Did xyz coarsenings”（你可以通过设置**-XX:G1SummarizeRSetStatsPeriod=period**指定周期频率报告（GCs的数量））。

如果你回想起之前的文章，GC阶段的“Update RS (ms)”展示了更新RSets花费时间，"Processed Buffers" 展示了GC期间更新缓冲区过程。如果你的日志中发现这些问题，然后使用上述的选项去进一步深入这些问题。

哪些选项通常可以帮助确定更新日志缓冲区和concurrent refinement threads的问题。

-XX:+G1SummarizeRSetStats 设置成一-XX:G1SummarizeRSetStatsPeriod=1的输出样例：

Concurrent RS processed 784125 cards

Of 4870 completed buffers:

4870 (100.0%) by concurrent RS threads.

0 (0.0%) by mutator threads.

Concurrent RS threads times (s)

0.64 0.30 0.26 0.18 0.17 0.16 0.17 0.15 0.15 0.12 0.13 0.08 0.13 0.13 0.12 0.13 0.12 0.11 0.12 0.11 0.12 0.13 0.11

Concurrent sampling threads times (s)

0.00

Total heap region rem set sizes = 199140K. Max = 661K.

Static structures = 660K, free_lists = 15052K.

1009422114 occupied cards represented.

Max size region = 313:(O)[0x000000054e400000,0x000000054e800000,0x000000054e800000], size = 662K, occupied = 1214K.

Did 2759 coarsenings.

上面的输出展示了已经处理过的card和已完成的buffer的数量。它展示concurrent refinement threads做了100%的工作，mutator threads 什么也没有做（这是我们说过的一个号的迹象！）。然后列出concurrent refinement thread的每一个线程摄入到工作的时间。

上面褐色的部分展示了自从HotSpot VM启动以来的累积状态。累积状态包括RSets的总和和最大RSets数量，已占用Card的数量，最大Region尺寸信息。它通常展示自VM启动以来任务完成的粗化总数。

此时此刻，介绍其他选项是合适的**-XX:G1RSetUpdatingPauseTimePercent=10**。设置GC evacuation（疏散）阶段期间G1 GC更新RSets消耗时间的百分比（默认是目标停顿时间的10%）。你可以增大或减小百分比的值，以便在stop-the-world(STW)GC阶段花费更多或更少的时间，让concurrent refinement thread处理相应的缓冲区。

记住，减少百分比的值，你在推迟concurrent refinement thread的工作；因此，你会看到并发任务增加。

Reference Processing

evacuation阶段期间和标记期间（并发标记多阶段的一部分）G1 GC处理引用。

evacuation阶段期间，扫描对象，复制，被处理后的时候找到引用对象。GC log里面，引用处理（Ref proc）时间和叫做“Other”下面的一组连续工作：

```
[Other: 0.2 ms]
  [Choose CSet: 0.0 ms]
  [Ref Proc: 0.2 ms]
  [Ref Enq: 0.0 ms]
  [Free CSet: 0.0 ms]
```

Note:无用的引用被添加进pending list,GC log里面展示的时间是reference enqueing time (Ref Enq)。

Remark阶段，发生在并发标记阶段之前。（note：多阶段并发标记周期的一部分。请参考上篇文章的更多细节。）remark阶段处理发现的引用过程。GC log里，你可以在GC remark部分看到reference processing (GC ref-proc)时间：

```
0.094: [GC remark 0.094: [GC ref-proc, 0.0000033 secs], 0.0004374 secs]
       [Times: user=0.00 sys=0.00, real=0.00 secs]
```

如果你看到引用处理期间时间很长，通过授权命令行选项** -XX:+ParallelRefProcEnabled**打开并行引用处理。

Evacuation Failure

如果你在GC日志发现"evacuation failure", "to-space exhausted", "to-space overflow"， "promotion failure"之类的字眼。这些术语的概念在G1 GC是相似的，请参考同一个。当没有更多的空闲region提升到Old代，或者复制到survivor空间，heap由于已经在最大值上而无法扩展，evacuation Failure聚会发生：

1. 如果对象被成功复制，G1需要更新对象引用，region必须是tenured。
2. 如果对象复制失败，G1会自己处理它们，在合适时机把region设置为tenured。

因此在G1 log 发生evacuation failure你应该怎么做：

1. 找出调整的一些影响导致失败-获取heap最大和最小的基线和真实的停顿时间目标：移除任何heap大小的设置例如**-Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio**等等。只使用**-Xms, -Xmx**，停顿时间目标**-XX:MaxGCPauseMillis**。
2. 如果问题在基线运行依然存在，大对象（看下面的章节）分配没有问题-矫正问题的方案是如果可以的话增加heap区大小。
3. 如果增加heap区大小行不通，如果你注意到为了G1 GC可以回收Old代标记阶段不会提早执行，然后你删掉**-XX:MaxGCPauseMillis**。这个选项默认占用你heap区的45%。删除这个选项可以帮助提早开始标记阶段。相反的，如果标记阶段提早开始并没有回收到很多空间，你应该在threshold默认值上增加来确保你的应用的存活数据可以适应。
4. 如果并发标记阶段准时启动，但是花了很长时间去完成；因此造成mixed gc周期延迟，最后导致evacuation失败，然后old代没有及时回收；使用选项**-XX:ConcGCThreads**增加并发标记线程数量。
5. 如果“to-space”Survivor区域有问题， 增加**-XX:G1ReservePercent**。默认是java heap的10%。G1 GC设置错误上限预留内存，以防万一"to-space"需要更多的空间。当然G1 GC只使用空间的50%，因此如果我们不想应用使用大的Young可以设置一个更大的值。

为了帮助解释evacuation failure的原因，我想介绍一个用户的选项：**-XX:+PrintAdaptiveSizePolicy**。这个选项会提供很多方法去阻止**-XX:+PrintGCDetails**选项。

让我看一个**-XX:+PrintAdaptiveSizePolicy**可用时的片段：

```
6062.121: [GC pause (G1 Evacuation Pause) (mixed) 6062.121: [G1Ergonomics (CSet Construction)
start choosing CSet, _pending_cards: 129059, predicted base time: 52.34 ms, remaining time:
147.66 ms, target pause time: 200.00 ms]

6062.121: [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 912 regions,
survivors: 112 regions, predicted young region time: 256.16 ms]

6062.122: [G1Ergonomics (CSet Construction) finish adding old regions to CSet, reason: old CSet
region num reached min, old: 149 regions, min: 149 regions]6062.122: [G1Ergonomics (CSet
Construction) finish choosing CSet, eden: 912 regions, survivors: 112 regions, old: 149 regions,
predicted pause time: 344.87 ms, target pause time: 200.00 ms]

6062.281: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: region allocation request
failed, allocation request: 2097152 bytes]

6062.281: [G1Ergonomics (Heap Sizing) expand the heap, requested expansion amount: 2097152 bytes,
attempted expansion amount: 4194304 bytes]

6062.281: [G1Ergonomics (Heap Sizing) did not expand the heap, reason: heap expansion operation
failed]

6062.902: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: recent GC overhead higher
than threshold after GC, recent GC overhead: 20.30 %, threshold: 10.00 %, uncommitted: 0 bytes,
calculated expansion amount: 0 bytes (20.00 %)]

6062.902: [G1Ergonomics (Concurrent Cycles) do not request concurrent cycle initiation, reason:
still doing mixed collections, occupancy: 9596567552 bytes, allocation request: 0 bytes,
threshold: 5798205810 bytes (45.00 %), source: end of GC]

6062.902: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions available,
candidate old regions: 1038 regions, reclaimable: 2612574984 bytes (20.28 %), threshold: 10.00 %]

(to-space exhausted), 0.7805160 secs]
```

上面的片段提供了很多信息-首先，让我们用上面GC log使用的命令行选项 server -Xms12g -Xmx12g -XX:+UseG1GC-** -XX:NewSize=4g -XX:MaxNewSize=5g**展示一些精彩的东西。

加粗部分展示用户限制region的范围在4-5G之间，因此限制了G1 GC的适应能力。如果G1需要去掉限制设置更小的值，它做不到；如果G1 GC需要增加空间范围，超过了给它分配的空间，它做不到！

这是evacuation结束时打印的heap明确信息：

```
[Eden: 3648.0M(3648.0M)->0.0B(3696.0M) Survivors: 448.0M->400.0M Heap: 11.3G(12.0G)-
>9537.9M(12.0G)]
```

阶段之后，G1不得不维持4096M作为最小范围(-XX:NewSize=4g)，由于基于G1的计算器，3696M用于Eden区，400M用户Survivor区。然而，heap区标记回收的数据已经达到9537.9M。因此，G1用完“to-space”。下面两次evacuation阶段的结果是evacuation failure:

混合evacuation阶段1:

```
[Eden: 2736.0M(3696.0M)->0.0B(4096.0M) Survivors: 400.0M->0.0B Heap: 12.0G(12.0G)->12.0G(12.0G)]
```

混合evacuation阶段2:

```
[Eden: 0.0B(4096.0M)->0.0B(4096.0M) Survivors: 0.0B->0.0B Heap: 12.0G(12.0G)->12.0G(12.0G)]
```

最终触发Full GC:

```
6086.564: [Full GC (Allocation Failure) 11G->3795M(12G), 15.0980440 secs]

   [Eden: 0.0B(4096.0M)->0.0B(4096.0M) Survivors: 0.0B->0.0B Heap: 12.0G(12.0G)->3795.2M(12.0G)]
```

Full GC 可以通过减少范围 / young代缩小至默认的minimum(Java heap的5%)。你也许会说old代足够容纳3795M的live data set (LDS)。然而， LDS明确耦合于设置young代minimum（4G）,压缩7891M以上空间。因此设置threshold为默认的heap的45%（也就是 5529M左右），标记阶段提早触发，混合回收期间回收很少的空间。heap使用保持增长，其他的标记阶段已经开始，但是标记阶段完成，踢开混合GC,已使用空间在11.3G（正如看到heap第一行的信息）。这次回收遭遇evacuation failure。因此，这个问题陷在 “starting marking cycle too early”上面。

Humongous Allocations

最后一个我想介绍的概念是，也许用户创建许多 humongous objects (H-objs)，G1 GC处理H-objs。

辣么，我们为什么需要不同途径去分配H-objs?

如果对象占用Region50%的区域或者更多那么被判定为humongous。分配humongous连续的空间。你可以想象一下，如果G1在Young代分配humongous，而且它们存活很长时间，然后它们需要一些必要而且昂贵（记住H-obj连续的region）的操作，复制这些H-obj导survivor空间，最终这些H-obj升迁到Old代。因此，为了避免上面的操作，H-obj直接分配在Old代，然后分类整理，映射作为humongousregion。

通过在Old代直接分配H-obj，G1避免在任何evacuation阶段包含它们，它们因此也不用移动。Full GC期间，压缩存活的H-obj。Full GC之后，multi-phased concurrent marking期间的清除阶段 死亡的H-obj被回收。换句话说，H-obj在清除阶段被回收，或者说它们在full gc期间被回收。

分配H-obj之前，G1 GC因为分配会交叉执行检查heap使用百分比，标记的threshold。如果允许，G1 GC然后初始化concurrent marking周期。由于我们想避免evacuation failures和可能多的Full GC,这样的方式被执行。结果在没有更过的可用region存活对象evacuations尽可能早检查以便给G1尽量多的时间去完成并发周期。

对于G1 GC,基本前提是没有太多H-obj和他们存活时间很长。然而，由于G1 GC的region尺寸取决于你的heap的minimum值，它的发生建立在分配的region的尺寸上，你的humongous可以看上去"normal"分配。这会导致需要H-obj分配在Old代的region上，甚至会导致evacuation failure，因为G1赶不上这些humongous分配。

现在你也许在思考如何找到humongous 分配导致evacuation failures的方法。这里，再一次-XX:+PrintAdaptiveSizePolicy会来到你的解决方案中。

你的GC日志中，可以看到类似下面的一些东西：

```
1361.680: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation, reason:
occupancy higher than threshold, occupancy: 1459617792 bytes, allocation request: 4194320 bytes,
threshold: 1449551430 bytes (45.00 %), source: concurrent humongous allocation]
```

因此，你可以看到一次并发周期被要求发生由于humongous分配需要4194320 bytes。

这些信息是有用的，因此你不但被告知你的应用有多少humongous分配（不管它们多还是少），而且还告诉你发配的大小。此外，如果你认为过多humongous分配，你能做的就是增加G1的region尺寸作为适合humongous的规则尺寸。因此，例如，分配尺寸仅仅在4M之上。所以，为了让这次分配可以规则分配，我们需要16M的region尺寸。所以，这里推荐明确设置命令行选项：-XX:G1HeapRegionSize=16M。

Note:回想一下我上篇文章，G1 region 跨越1M-32M(2的指数)，分配要求稍微超过4M。因此，8M的region的尺寸不足以避免humongous分配。问我们需要下一个2的指数，16MB。

ok。我认为这次我已经讲解了大部分重要问题和G1概念。再一次,谢谢阅读！