June 18, 2012

Understanding G1 GC Logs

Poonam Parhar
CONSULTING MEMBER OF TECHNICAL STAFF

The
purpose of this post is to explain the meaning of GC logs
generated with some tracing and diagnostic options for G1 GC. We will

take a look at the output generated with *PrintGCDetails* which is a product flag and provides the most detailed level of information. Along with that, we will also look at the output of two diagnostic flags that get enabled with -XX:+*UnlockDiagnosticVMOptions* option - *G1PrintRegionLivenessInfo*
that prints the occupancy and the amount of space used by live objects in each region at the end of the marking cycle and *G1PrintHeapRegions* that provides detailed information on the heap regions being allocated and reclaimed.

We
will be looking at the logs generated with JDK 1.7.0_04 using these options.

Option
-XX:+PrintGCDetails

Here's a sample log of G1 collection generated with PrintGCDetails.

```
0.522: [GC pause (young), 0.15877971 secs]
   [Parallel Time: 157.1 ms]
      [GC Worker Start (ms):  522.1  522.2  522.2  522.2
       Avg: 522.2, Min: 522.1, Max: 522.2, Diff:   0.1]
      [Ext Root Scanning (ms):  1.6  1.5  1.6  1.9
       Avg:   1.7, Min:   1.5, Max:   1.9, Diff:   0.4]
      [Update RS (ms):  38.7  38.8  50.6  37.3
       Avg:  41.3, Min:  37.3, Max:  50.6, Diff:  13.3]
         [Processed Buffers : 2 2 3 2
          Sum: 9, Avg: 2, Min: 2, Max: 3, Diff: 1]
      [Scan RS (ms):  9.9  9.7  0.0  9.7
```

```
         Avg:   7.3, Min:   0.0, Max:   9.9, Diff:   9.9]
        [Object Copy (ms):  106.7  106.8  104.6  107.9
         Avg: 106.5, Min: 104.6, Max: 107.9, Diff:   3.3]
        [Termination (ms):  0.0  0.0  0.0  0.0
         Avg:   0.0, Min:   0.0, Max:   0.0, Diff:   0.0]
           [Termination Attempts : 1 4 4 6
            Sum: 15, Avg: 3, Min: 1, Max: 6, Diff: 5]
        [GC Worker End (ms):  679.1  679.1  679.1  679.1
         Avg: 679.1, Min: 679.1, Max: 679.1, Diff:   0.1]
        [GC Worker (ms):  156.9  157.0  156.9  156.9
         Avg: 156.9, Min: 156.9, Max: 157.0, Diff:   0.1]
        [GC Worker Other (ms):  0.3  0.3  0.3  0.3
         Avg:   0.3, Min:   0.3, Max:   0.3, Diff:   0.0]
     [Clear CT:   0.1 ms]
     [Other:   1.5 ms]
        [Choose CSet:   0.0 ms]
        [Ref Proc:   0.3 ms]
        [Ref Enq:   0.0 ms]
        [Free CSet:   0.3 ms]
     [Eden: 12M(12M)->0B(10M) Survivors: 0B->2048K Heap:
 13M(64M)->9739K(64M)]
  [Times: user=0.59 sys=0.02, real=0.16 secs]
```

This is the typical log of an *Evacuation Pause* (G1 collection) in which live objects are copied from one set of regions (young OR young+old) to another set. It is a stop-the-world activity and all the application threads are stopped at a safepoint during this time.

This pause is made up of several sub-tasks indicated by the indentation in the log entries. Here's is the top most line that gets printed for the Evacuation Pause.

0.522:

[GC pause (young), 0.15877971 secs]

This is the highest level information telling us that it is an Evacuation Pause that started at 0.522 secs from the start of the process, in which all the regions being evacuated are Young i.e. Eden and Survivor regions. This collection took 0.15877971 secs to finish.

Evacuation
Pauses can be mixed as well. In which case the set of regions selected include all of the young regions as well as some old regions.

1.730:
[GC pause (mixed), 0.32714353 secs]

Let's take a look at all the sub-tasks performed in this Evacuation Pause.

[Parallel Time: 157.1 ms]

Parallel Time is the total elapsed time spent by all the parallel GC worker threads. The following lines correspond to the parallel tasks performed by these worker threads in this total parallel time, which in this case is 157.1 ms.

[GC Worker Start (ms): 522.1 522.2 522.2 522.2
Avg: 522.2, Min: 522.1, Max: 522.2, Diff: 0.1]

The first line tells us the start time of each of the worker thread

in milliseconds. The start times are ordered with respect to the worker thread ids – thread 0 started at 522.1ms and thread 1 started at 522.2ms from the start of the process. The second line tells the Avg, Min, Max and Diff of the start times of all of the worker threads.

[Ext Root Scanning (ms): 1.6 1.5 1.6 1.9
Avg: 1.7, Min: 1.5, Max: 1.9, Diff: 0.4]

This gives us the time spent by each worker thread scanning the roots (globals, registers, thread stacks and VM data structures). Here, thread 0 took 1.6ms to perform the root scanning task and thread 1 took 1.5 ms. The second line clearly shows the Avg, Min, Max and Diff of the times spent by all the worker threads.

[Update RS (ms): 38.7 38.8 50.6 37.3
Avg: 41.3, Min: 37.3, Max: 50.6, Diff: 13.3]

Update
RS gives us the time each thread spent in updating the Remembered Sets. Remembered Sets are the data structures that keep track of the references that point into a heap region. Mutator threads keep changing the object graph and thus the references that point into a particular region. We keep track of these changes in buffers called Update Buffers. The
Update RS sub-task processes the update buffers that were not able to be processed concurrently, and updates the corresponding remembered sets of all regions.

[Processed Buffers : 2 2 3 2
Sum: 9, Avg: 2, Min: 2, Max: 3, Diff: 1]

This tells us the number of Update Buffers (mentioned above) processed by each worker thread.

[Scan RS (ms): 9.9 9.7 0.0 9.7
Avg: 7.3, Min: 0.0, Max: 9.9, Diff: 9.9]

These are the times each worker thread had spent in scanning the Remembered Sets. Remembered Set of a region contains cards that correspond to the references pointing into that region. This phase scans those cards looking for the references pointing into all the regions of the collection set.

[Object Copy (ms): 106.7 106.8 104.6 107.9
Avg: 106.5, Min: 104.6, Max: 107.9, Diff: 3.3]

These are the times spent by each worker thread copying live objects from the regions in the Collection Set to the other regions.

[Termination (ms): 0.0 0.0 0.0 0.0
Avg: 0.0, Min: 0.0, Max: 0.0, Diff: 0.0]

Termination time is the time spent by the worker thread offering to terminate. But before terminating, it checks the work queues of other threads and if there are still object references in other work queues, it tries to steal object references, and if it succeeds in stealing a reference, it processes that and offers to terminate again.

[Termination Attempts : 1 4 4 6
Sum: 15, Avg: 3, Min: 1, Max: 6, Diff: 5]

This gives the number of times each thread has offered to terminate.

[GC Worker End (ms): 679.1 679.1 679.1 679.1
Avg: 679.1, Min: 679.1, Max: 679.1, Diff: 0.1]

These are the times in milliseconds at which each worker thread stopped.

[GC Worker (ms): 156.9 157.0 156.9 156.9
Avg: 156.9, Min: 156.9, Max: 157.0, Diff: 0.1]

These are the total lifetimes of each worker thread.

[GC Worker Other (ms): 0.3 0.3 0.3 0.3
Avg: 0.3, Min: 0.3, Max: 0.3, Diff: 0.0]

These are the times that each worker thread spent in performing some other tasks that we have not accounted above for the total Parallel Time.

[Clear CT: 0.1 ms]

This is the time spent in clearing the Card Table. This task is performed in serial mode.

[Other: 1.5 ms]

Time
spent in the some other tasks listed below. The
following sub-tasks (which individually may be parallelized) are
performed serially.

[Choose CSet: 0.0 ms]

Time spent in selecting the regions for the Collection Set.

[Ref Proc: 0.3 ms]

Total time spent in processing Reference objects.

[Ref Enq: 0.0 ms]

Time spent in enqueuing references to the ReferenceQueues.

[Free CSet: 0.3 ms]

Time spent in freeing the collection set data structure.

[Eden: 12M(12M)->0B(13M) Survivors: 0B->2048K Heap: 14M(64M)->9739K(64M)]

This
line gives the details on the heap size changes with the Evacuation
Pause. This shows that Eden had the occupancy of 12M and its capacity

was also 12M before the collection. After the collection, its occupancy got reduced to 0 since everything is evacuated/promoted from Eden during a collection, and its target size grew to 13M. The new Eden capacity of 13M is not reserved at this point. This value is the target size of the Eden. Regions are added to Eden as the demand is made and when the added regions reach to the target size, we start the next collection.

Similarly,
Survivors had the occupancy of 0 bytes and it grew to 2048K after the collection. The total heap occupancy and capacity was 14M and 64M receptively before the collection and it became 9739K and 64M after the collection.

Apart from the evacuation pauses, G1 also performs concurrent-marking to build the live data information of regions.

```
1.416: [GC pause (young) (initial-mark), 0.62417980 secs]
….....
2.042: [GC concurrent-root-region-scan-start]
2.067: [GC concurrent-root-region-scan-end, 0.0251507]
2.068: [GC concurrent-mark-start]
3.198: [GC concurrent-mark-reset-for-overflow]
4.053: [GC concurrent-mark-end, 1.9849672 sec]
4.055: [GC remark 4.055: [GC ref-proc, 0.0000254 secs],
0.0030184 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
4.088: [GC cleanup 117M->106M(138M), 0.0015198 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
4.090: [GC concurrent-cleanup-start]
```

```
4.091: [GC concurrent-cleanup-end, 0.0002721]
```

The first phase of a marking cycle is Initial Marking where all the objects directly reachable from the roots are marked and this phase is piggy-backed on a fully young Evacuation Pause.

2.042:
[GC concurrent-root-region-scan-start]

This marks the start of a
concurrent phase that scans the set of root-regions which are directly reachable from the survivors of the initial marking phase.

2.067:
[GC concurrent-root-region-scan-end, 0.0251507]

End of the concurrent root region scan phase and it lasted for 0.0251507 seconds.

2.068:
[GC concurrent-mark-start]

Start of the concurrent marking at 2.068 secs from the start of the

process.

3.198:
[GC concurrent-mark-reset-for-overflow]

This indicates that the global marking stack had became full and there was an overflow of the stack. Concurrent marking detected this overflow and had to reset the data structures to start the marking again.

4.053:
[GC concurrent-mark-end, 1.9849672 sec]

End of the concurrent marking phase and it lasted for 1.9849672 seconds.

4.055:
[GC remark 4.055: [GC ref-proc, 0.0000254 secs], 0.0030184 secs]

This corresponds to the remark phase which is a stop-the-world phase. It completes the left over marking work (SATB buffers processing) from the previous phase. In this case, this phase took 0.0030184 secs and out of which 0.0000254 secs were spent on Reference processing.

4.088:
[GC cleanup 117M->106M(138M), 0.0015198 secs]

Cleanup phase which is again a stop-the-world phase. It goes through the marking information of all the regions, computes the live data information of each region, resets the marking data structures and sorts the regions according to their gc-efficiency. In this example, the total heap size is 138M and after the live data counting it was found that the total live data size dropped down from 117M to 106M.

4.090:
[GC concurrent-cleanup-start]

This concurrent cleanup phase frees up the regions that were found to be empty (didn't contain any live data) during the previous stop-the-world phase.

4.091:
[GC concurrent-cleanup-end, 0.0002721]

Concurrent cleanup phase took 0.0002721 secs to free up the empty regions.

Option
-XX:G1PrintRegionLivenessInfo

Now, let's look at the output generated with the flag G1PrintRegionLivenessInfo. This is a diagnostic option and gets enabled with -XX:+UnlockDiagnosticVMOptions.

G1PrintRegionLivenessInfo prints the live data information of each region during the *Cleanup* phase of the concurrent-marking cycle.

26.896:
[GC cleanup
###
PHASE Post-Marking @ 26.896
###
HEAP committed: 0x02e00000-0x0fe00000 reserved: 0x02e00000-0x12e00000 region-size: 1048576

Cleanup phase of the concurrent-marking cycle started at 26.896 secs from the start of the process and this live data information is being printed after the marking phase. Committed G1 heap ranges from 0x02e00000 to 0x0fe00000 and the total G1 heap reserved by JVM is from 0x02e00000 to 0x12e00000. Each region in the G1 heap is of size 1048576 bytes.

###
type address-range used prev-live
next-live gc-eff
###
(bytes) (bytes) (bytes)
(bytes/ms)

This is the header of the output that tells us about the type of the region, address-range of the region, used space in the region, live bytes in the region with respect to the previous marking cycle, live

bytes in the region with respect to the current marking cycle and the GC efficiency of that region.

###
FREE 0x02e00000-0x02f00000 0 0 0
0.0

This is a Free region.

###
OLD 0x02f00000-0x03000000 1048576 1038592 1038592
0.0

Old region with address-range from 0x02f00000 to 0x03000000. Total used space in the region is 1048576 bytes, live bytes as per the previous marking cycle are 1038592 and live bytes with respect to the current marking cycle are also 1038592. The GC efficiency has been computed as 0.

###
EDEN 0x03400000-0x03500000 20992 20992 20992
0.0

This is an Eden region.

###
HUMS 0x0ae00000-0x0af00000 1048576 1048576 1048576

0.0
###
HUMC 0x0af00000-0x0b000000 1048576 1048576 1048576
0.0
###
HUMC 0x0b000000-0x0b100000 1048576 1048576 1048576
0.0
###
HUMC 0x0b100000-0x0b200000 1048576 1048576 1048576
0.0
###
HUMC 0x0b200000-0x0b300000 1048576 1048576 1048576
0.0
###
HUMC 0x0b300000-0x0b400000 1048576 1048576 1048576
0.0
###
HUMC 0x0b400000-0x0b500000 1001480 1001480 1001480
0.0

These are the continuous set of regions called Humongous regions for storing a large object. HUMS (Humongous starts) marks the start of the set of humongous regions and HUMC (Humongous continues) tags the
subsequent regions of the humongous regions set.

###
SURV 0x09300000-0x09400000 16384 16384 16384

0.0

This is a Survivor region.

###
SUMMARY capacity: 208.00 MB used: 150.16 MB / 72.19 % prev-live: 149.78 MB / 72.01 % next-live: 142.82 MB / 68.66 %

At the end, a summary is printed listing the capacity, the used space and the change in the liveness after the completion of concurrent marking. In this case, G1 heap capacity is 208MB, total used space is 150.16MB which is 72.19% of the total heap size, live data in the previous marking was 149.78MB which was 72.01% of the total heap size and the live data as per the current marking is 142.82MB which is 68.66% of the total heap size.

Option
-XX:+G1PrintHeapRegions

G1PrintHeapRegions option
logs the regions related events when regions are committed, allocated into or are reclaimed.

COMMIT/UNCOMMIT events

G1HR COMMIT

[0x6e900000,0x6ea00000]
G1HR COMMIT [0x6ea00000,0x6eb00000]

Here, the heap is being initialized or expanded and the region (with bottom: 0x6eb00000 and end: 0x6ec00000) is being freshly committed. COMMIT events are always generated in order i.e. the next COMMIT event will always be for the uncommitted region with the lowest address.

G1HR UNCOMMIT
[0x72700000,0x72800000]
G1HR UNCOMMIT [0x72600000,0x72700000]

Opposite to COMMIT. The heap got shrunk at the end of a Full GC and the regions are being uncommitted. Like COMMIT, UNCOMMIT
events are also generated in order i.e. the next UNCOMMIT event will always be for the committed region with the highest address.

GC Cycle events

G1HR #StartGC 7
G1HR CSET
0x6e900000
G1HR REUSE 0x70500000
G1HR ALLOC(Old)
0x6f800000
G1HR RETIRE 0x6f800000 0x6f821b20
G1HR #EndGC 7

This shows start and end of an Evacuation pause. This event is followed by a GC counter tracking both evacuation pauses and Full GCs. Here, this is the 7th GC since the start of the process.

G1HR #StartFullGC 17
G1HR
UNCOMMIT [0x6ed00000,0x6ee00000]
G1HR POST-COMPACTION(Old)
0x6e800000 0x6e854f58
G1HR #EndFullGC 17

Shows start and end of a Full GC. This event is also followed by the same GC counter as above. This is the 17th GC since the start of the process.

ALLOC events

G1HR ALLOC(Eden) 0x6e800000

The region with bottom 0x6e800000 just started being used for allocation. In this case it is an Eden region and allocated into by a mutator thread.

G1HR ALLOC(StartsH) 0x6ec00000
0x6ed00000
G1HR ALLOC(ContinuesH) 0x6ed00000 0x6e000000

Regions being used for the allocation of Humongous object. The object spans over two regions.

G1HR ALLOC(SingleH) 0x6f900000
0x6f9eb010

Single region being used for the allocation of Humongous object.

G1HR COMMIT
[0x6ee00000,0x6ef00000]
G1HR COMMIT [0x6ef00000,0x6f000000]
G1HR
COMMIT [0x6f000000,0x6f100000]
G1HR COMMIT
[0x6f100000,0x6f200000]
G1HR ALLOC(StartsH) 0x6ee00000
0x6ef00000
G1HR ALLOC(ContinuesH) 0x6ef00000 0x6f000000
G1HR
ALLOC(ContinuesH) 0x6f000000 0x6f100000
G1HR ALLOC(ContinuesH)
0x6f100000 0x6f102010

Here, Humongous object allocation request could not be satisfied by the free committed regions that existed in the heap, so the heap needed to be expanded. Thus new regions are committed and then allocated into for the Humongous object.

G1HR ALLOC(Old) 0x6f800000

Old region started being used for allocation during GC.

G1HR ALLOC(Survivor) 0x6fa00000

Region being used for copying old objects into during a GC.

Note that Eden and Humongous ALLOC events are generated outside the GC boundaries and Old and Survivor ALLOC events are generated inside the GC boundaries.

Other Events

G1HR RETIRE 0x6e800000 0x6e87bd98

Retire and stop using the region having bottom 0x6e800000 and top 0x6e87bd98 for allocation.

Note that most regions are full when they are retired and we omit those events to reduce the output volume. A region is retired when another region of the same type is allocated or we reach the start or end of a GC(depending on the region). So for Eden regions:

For example:

1. ALLOC(Eden) Foo
2. ALLOC(Eden) Bar
3. StartGC

At point 2, Foo has just been retired and it was full.
At point 3, Bar was retired and it was full. If they were not full

when they were retired, we will have a RETIRE event:

1. ALLOC(Eden) Foo
2. RETIRE Foo top
3.
ALLOC(Eden) Bar
4. StartGC

G1HR CSET 0x6e900000

Region (bottom: 0x6e900000) is selected for the Collection Set. The region might have been selected for the collection set earlier (i.e. when it was allocated). However, we generate the CSET events for all regions in the CSet at the start of a GC to make sure there's no confusion about which regions are part of the CSet.

G1HR POST-COMPACTION(Old)
0x6e800000 0x6e839858

POST-COMPACTION event is generated for each non-empty region in the heap after a full compaction. A full compaction moves objects around, so we don't know what the resulting shape of the heap is (which regions were written to, which were emptied, etc.). To deal with this, we generate a POST-COMPACTION event for each non-empty region with its type (old/humongous) and the heap boundaries. At this point we should only have Old and Humongous regions, as we have collapsed the young generation, so we should not have eden and survivors.

POST-COMPACTION events are generated within the Full GC boundary.

G1HR CLEANUP 0x6f400000
G1HR
CLEANUP 0x6f300000
G1HR CLEANUP 0x6f200000

These regions were found empty after remark phase of Concurrent Marking and are reclaimed shortly afterwards.

G1HR #StartGC 5
G1HR CSET
0x6f400000
G1HR CSET 0x6e900000
G1HR REUSE 0x6f800000

At the end of a GC we retire the old region we are allocating into. Given that its not full, we will carry on allocating into it during the next GC. This is what REUSE means. In the above case 0x6f800000 should have been the last region with an ALLOC(Old) event during the previous GC and should have been retired before the end of the previous GC.

G1HR ALLOC-FORCE(Eden) 0x6f800000

A specialization of ALLOC which indicates that we have reached the max desired number of the particular region type (in this case: Eden), but we decided to allocate one more. Currently it's only used for Eden regions when we extend the young generation because we

cannot do a GC as the GC-Locker is active.

G1HR EVAC-FAILURE 0x6f800000

During a GC, we have failed to evacuate an object from the given region as the heap is full and there is no space left to copy the object. This event is generated within GC boundaries and exactly once for each region from which we failed to evacuate objects.

When Heap Regions are reclaimed ?

It is also worth mentioning when the heap regions in the G1 heap are reclaimed.

- All regions that are in

  the CSet (the ones that appear in CSET events) are reclaimed at the

  end of a GC. The exception to that are regions with EVAC-FAILURE

  events.

- All regions with

  CLEANUP events are reclaimed.

- After a Full GC some regions get reclaimed (the

ones from which we moved the objects out). But that is not shown explicitly, instead the non-empty regions that are left in the heap are printed out with the POST-COMPACTION events.

Join the discussion

Comments (8)

Recent Content

StackOverflowError and threads waiting for ReentrantReadWriteLock

If the terms 'Stuck Threads' or 'Hang' sound familiar and scary to you, please read on. We can sometimes come across situations where the...

POONAM PARHAR

Can young generation size impact the application response times?

I was recently involved in a performance issue where the application's response times were decreasing over time, and I was tasked to...

Clarifying some confu Java Flight Recording

In this blog, I am goir about two confusion Flight Recordings tha across from multiple users/customers....