Collecting and reading G1 garbage collector logs - part 2

May 18, 2017 | Matt Robson

< Back to all posts

Tags: *Technical Account Managers*

Welcome back! I hope you enjoyed part 1 of this series where we took an in-depth look at how G1 works and makes its decisions. In the second part of this ongoing series, we will dive into the extensive list of garbage collection (GC) logging options. In order to determine where tuning can be applied, you need to understand the different steps involved in a collection, what those steps mean, and the impact they have on overall collection time. To do this, we're going to breakdown the available options into three distinct categories in order to look at what the logs mean and under what circumstances you should use them:

1. **Mandatory -** No one should be running in production without these settings.
2. **Advanced -** Depending on the maturity of the application and the growth rate of the load, these are considerations to run in production when additional tuning will be required.
3. **Debug -** These options are targeted towards solving a specific issue or performance problem and would not be used in production outside of cases where an issue cannot be reproduced anywhere else.

Beginning with the mandatory flags, these options provide a means of collection and the required

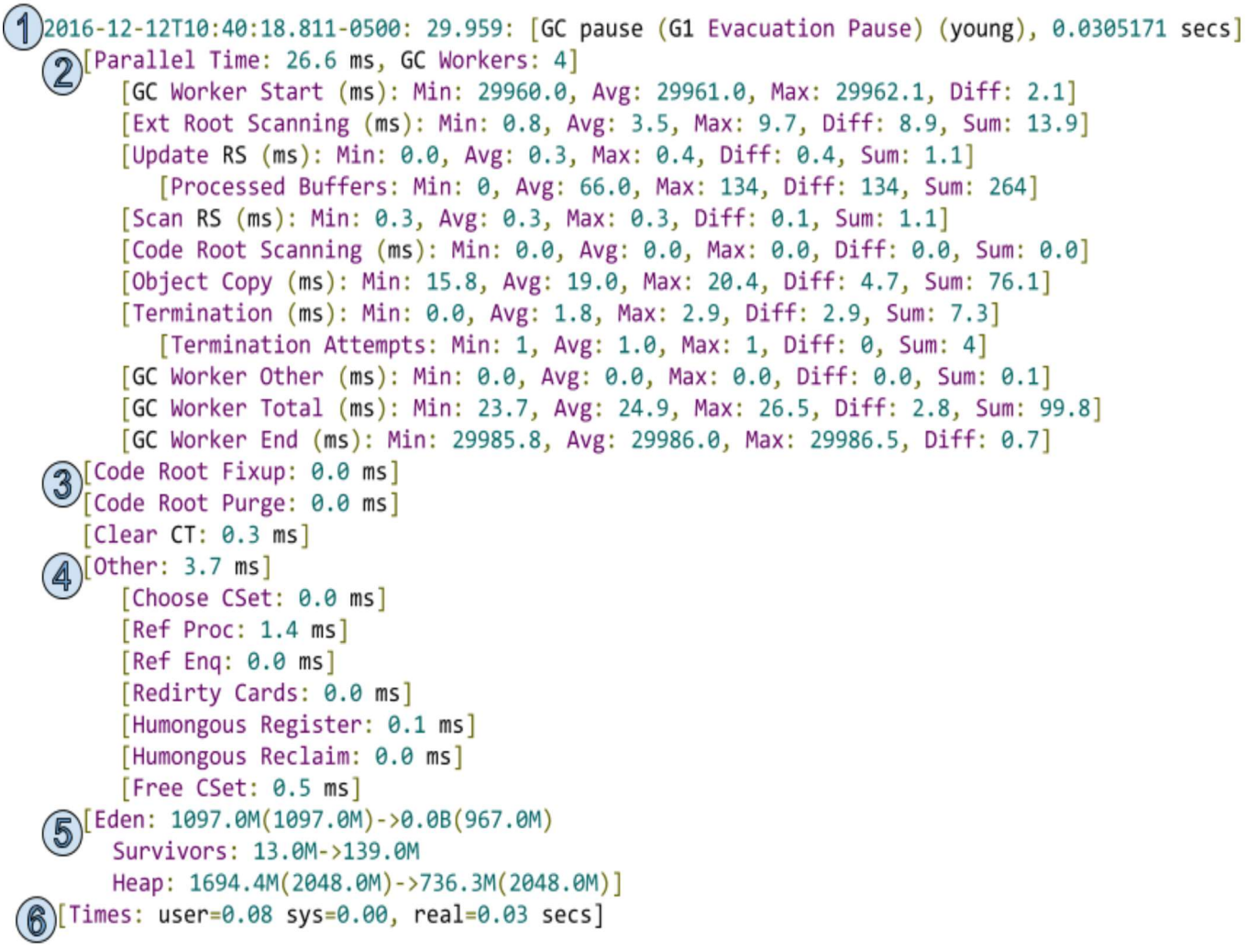
problem.

The flags are:

-Xloggc:/path/to/gc.log	Path where the GC logs are written
-XX:+UseGCLogFileRotation	Enable GC log file rotation
-XX:NumberOfGCLogFiles=<value>	Number of rotated GC logs files to retain
-XX:GCLogFileSize=<size>	Size of each GC logs file to initiate rotation
-XX:+PrintGCDetails	Detailed GC log
-XX:+PrintGCDateStamps	Actual date and timestamp of the collection
-XX:+PrintGCApplicationStoppedTime	Amount of time the application stopped during GC
-XX:+PrintGCApplicationConcurrentTime	Amount of time the application ran between GCs
-XX:-PrintCommandLineFlags	Prints all the command line flags in the GC log

Of these flags, selecting appropriate values for the number of log files and the size of each log file will ensure you maintain a suitable history of garbage collection logs. I always recommend keeping a minimum of one week of logs, as it gives you a baseline around how your application performs over the course of a given week.

Let's take a deeper look at the anatomy of a log generated through '-XX:+PrintGCDetails' broken down into six key points:



a. The actual date and time the event occurred, logged by setting '-XX:+PrintGCDateStamps' - **2016-12-12T10:40:18.811-0500**
b. The relative timestamp, since the start of the JVM - **29.959**
c. The type of collection - **G1 Evacuation Pause (young)** - identifies this as an evacuation pause and a young collection

    i. The next most popular cause is **G1 Humongous Allocation**

d. The time the collection took - **0.0305171 sec**

2. The second point outlines all of the parallel tasks:

a. **Parallel Time** - How much Stop The World (STW) time parallel tasks took, beginning at the start of the collection and finishing when the last GC worker ends - 26.6ms
b. **GC Workers** - The number of parallel GC workers, defined by -XX:ParallelGCThreads - **4**

    i. Defaults to the number of CPUs, up to 8. For 8+ CPUs, it defaults to a ⅝ thread to CPU ratio

c. **GC Worker Start** - The min / max timestamp since the start of the JVM when the GC workers began. The difference represents the time in milliseconds between the start of the first and last thread. Ideally, you want them to start quickly and at the same time
d. **Ext Root Scanning** - The time spent scanning external roots (thread stack roots, JNI, global variables, system dictionary, etc..) to find any that reach into the current collection set
e. **Update RS (Remembered Set or RSet)** - Each region has its own Remembered Set. It tracks the addresses of cards that hold references into a region. As writes occur, a post-write barrier manages changes in inter-region references by marking the newly referred card dirty and placing in on the log buffer or dirty card queue. Once full, concurrent refinement threads process these queues in parallel to running application threads. **Update RS** comes in to enable the GC Workers to process any outstanding buffers which were not handled prior to the start of the collection. This ensures that each RSet is up-to-date

    i. **Processed Buffers** - This shows how many Update Buffers were processed during Update RS

f. **Scan RS** - The Remembered Set of each region is scanned to look for references that point to the regions in the Collection Set
g. **Code Root Scanning** - The time spent scanning the roots of compiled source code for references into the Collection Set
h. **Object Copy** - During an evacuation pause, all regions in the Collection Set must be evacuated. Object copy is responsible for copying all remaining live objects to new regions
i. **Termination** - When a GC worker finishes, it enters a termination routine where it synchronizes with the other workers and tries to steal outstanding tasks. The time represents

how long it took from when a worker first tried to terminate and when it actually terminated

   i. **Termination Attempts** - If a worker successfully steals tasks, it re-enters the termination routine and tries to steal more work or terminate. Every time tasks are stolen and termination is re-entered, the number of termination attempts will be incremented

j. **GC Worker Other -** This represents time spent on tasks not accounted to the previous tasks

k. **GC Worker Total -** This shows the min, max, average, diff and sum for the time spent by each of the parallel worker threads

l. **GC Worker End -** The min / max timestamp since the start of the JVM when the GC workers ended. The diff represents the time in milliseconds between the end of the first and last thread. Ideally, you want them to end quickly and at the same time

3. The third point outlines serial tasks:

   a. **Code Root Fixup -** Walking marked methods that point into the CSet to fix any pointers that may have moved during the GC

   b. **Code Root Purge -** Purge entries in the code root table

   c. **Clear CT -** The card table is cleared of dirtied cards

4. The fourth point outlines other tasks not previously accounted for. They are also serial.

   a. **Choose CSet -** Selects the regions for the Collection Set

   b. **Ref Proc -** Processes any soft/weak/final/phantom/JNI references discovered by the STW reference processor

   c. **Ref Enq -** Loops over references and enqueues them to the pending list

   d. **Reditry Cards -** Cards modified through the collection process are remarked as dirty

   e. **Humongous Register -** With 'G1ReclaimDeadHumongousObjectsAtYoungGC' enabled (default true / feature added in JDK 8u60) G1 will try to eagerly collect Humongous regions during Young GC. This represents how long it took to evaluate if the Humongous regions are candidates for eager reclaim and to record them. Valid candidates will have no existing strong code roots and only sparse entries in the Remembered Set. Each candidate will have its Remembered Set flushed to the dirty card queue and, if emptied, the region will be added to the current Collection Set

   f. **Humongous Reclaim -** The time spent ensuring the humongous object is dead and cleaned up, freeing the regions, resetting the region type and returning the regions to the free list and accounting for the freed space

   g. **Free CSet -** The now evacuated regions are added back to the free list

5. The fifth point outlines how the generations have changed and how they have been adapted as a result of the current collection:

   a. **Eden: 1097.0M(1097.0M)->0.0B(967.0M)**

i. This shows that the current Young Collection was triggered because the Eden space was full - 1097.0M of the allocated (1097.0M)

ii. It shows that all Eden regions were evacuated and the usage was reduced to 0.0B by the collection

iii. It also shows that the total allocation of Eden space has been reduced to 967.0M for the next collection

b. **Survivors: 13.0M->139.0M**

i. As a result of the young evacuation, the survivor space grew from 13.0M to 139.0M

c. **Heap: 1694.4M(2048.0M)->736.3M(2048.0M)**

i. At the time of the collection, the overall heap allocation was 1694.4M of the max (2048.0M)

ii. After the collection, the overall heap allocation was reduced to 736.3M and the max heap was unchanged at (2048.0M)

6. The sixth point represents the time taken for the collection:

a. **user=0.08**

i. Amount of CPU time spent in the user code within the process during the collection. This accounts for all threads across all CPUs. This does not account for time spent outside the process or time spent waiting. Depending on the number of parallel threads, user time will be quite a bit higher than the real time

b. **sys=0.00**

i. Amount of CPU time spent in the kernel within the process. This accounts for all threads across all CPUs. This does not account for time spent outside the process or time spent waiting

c. **real=0.03**

i. This is the real wall clock time from the start to the end of the collection. This also includes time spent in other process and time spent waiting

The next event you're likely to encounter is Concurrent Marking. As discussed in Part 1, Concurrent Marking can be triggered in a few of different ways, but it always performs the same work.

```
1 2016-12-12T10:40:08.363-0500: 19.510: [GC pause (G1 Evacuation Pause) (young)
      (initial-mark), 0.0387872 secs]
  <standard Young Collection as above occurs>
2 2016-12-12T10:40:08.402-0500: 19.549: [GC concurrent-root-region-scan-start]
  2016-12-12T10:40:08.405-0500: 19.552: [GC concurrent-root-region-scan-end, 0.0030613 secs]
3 2016-12-12T10:40:08.405-0500: 19.553: [GC concurrent-mark-start]
  2016-12-12T10:40:08.711-0500: 19.858: [GC concurrent-mark-end, 0.3055438 secs]
4 2016-12-12T10:40:08.713-0500: 19.861: [GC remark
      2016-12-12T10:40:08.713-0500: 19.861: [Finalize Marking, 0.0014099 secs]
      2016-12-12T10:40:08.715-0500: 19.862: [GC ref-proc, 0.0000480 secs]
      2016-12-12T10:40:08.715-0500: 19.862: [Unloading, 0.0025840 secs], 0.0055136 secs]
      [Times: user=0.01 sys=0.00, real=0.00 secs]
5 2016-12-12T10:40:08.724-0500: 19.872: [GC cleanup 1757M->914M(2048M), 0.0023579 secs]
      [Times: user=0.01 sys=0.00, real=0.00 secs]
6 2016-12-12T10:40:08.727-0500: 19.875: [GC concurrent-cleanup-start]
  2016-12-12T10:40:08.729-0500: 19.876: [GC concurrent-cleanup-end, 0.0012954 secs]
```

1. The first point denotes the start of marking:
    a. GC pause (G1 Evacuation Pause) (young) (initial-mark)
        i. To take advantage of the STW pause and trace all reachable objects, initial-mark is done as part of a Young Collection. The initial-mark sets two top-at-mark-start (TAMS) variables to distinguish existing objects and objects allocated during concurrent marking. Any objects above the top TAMS are implicitly considered live for this cycle
2. The second point is the first concurrent event:
    a. GC concurrent-root-region-scan-start
        i. Region Root Scanning takes the new Survivor regions from the initial-mark and scans them for references. Any references found from these 'root' regions are subsequently marked
    b. GC concurrent-root-region-scan-end
3. The third point denotes concurrent marking:
    a. GC concurrent-mark-start
        i. It runs concurrently alongside the application threads. The number of concurrent threads are, by default, 25% of the number of Parallel Threads. It can also be set explicitly through -XX:ConcGCThreads
        ii. Tracing the heap and marking of all live objects in a bitmap. Because all objects above the top TAMS are implicitly live, we only need to mark what's below that threshold
        iii. Accounting for the concurrent changes during marking. In order for SATB to work, it must be able to trace objects via the pointer when it took the initial snapshot. In order for that to occur, a pre-write barrier logs the original value to a SATB buffer that, when full, is

added to a global list that is periodically processed by the concurrent marking threads

iv. Live data accounting happens concurrently alongside the marking process. The consumed space of each region is tabulated as live objects are marked in order to establish a liveness ratio

b. GC concurrent-mark-end

4. The fourth point is a STW Phase:

a. GC remark / Finalize Marking / GC ref-proc / Unloading

i. This phase is STW so that a final picture can be painted. The remaining SATB buffers are processed and any lingering live objects are marked

5. The fifth point is also a STW phase:

a. GC cleanup

i. A final live object count is completed. This is done per regions using the standard set of parallel threads

1. It marks the card bitmap for all objects allocated since the initial-mark (everything above the TAMS)
2. It also marks the regions bitmap for any region with at least one live object

ii. In preparation for the next marking, the previous and next bitmaps are swapped

iii. Dead Old and Humongous regions with zero live bytes are freed and cleared

iv. Scrubs the Remembered Sets' of regions with no live objects

v. To prepare for the next cycle, the old regions are sorted for Collection Set selection, based on their liveness

vi. The concurrent unloading of dead classes from the metascape

6. The sixth point is once again a concurrent phase:

a. GC concurrent-cleanup-start

i. It performs a final cleanup of the empty regions processed in step five

1. The Remembered Set of each region is cleaned: this includes sparse and coarse entries, the from card cache and the code root tables

ii. As the regions are fully cleaned, they are added to a temporary list. Once cleanup is finished, the temporary list is merged into the secondary free region list where they wait to be added back to the master free list

b. GC concurrent-cleanup-end

Once Concurrent Marking is complete, you will see a Young GC immediately followed by a Mixed GC (this assumes the correct criteria is met, as discussed in Part 1). As you can see in the log below, the tasks of the 'Mixed Collection' are identical to the previously explained Young Collection. There are only two differences between the Young and Mixed Collections:

1. The first point identifies this collection as Mixed.

    a. GC pause (G1 Evacuation Pause) (mixed)

2. The collection set will contain Old regions, as determined through Concurrent Marking

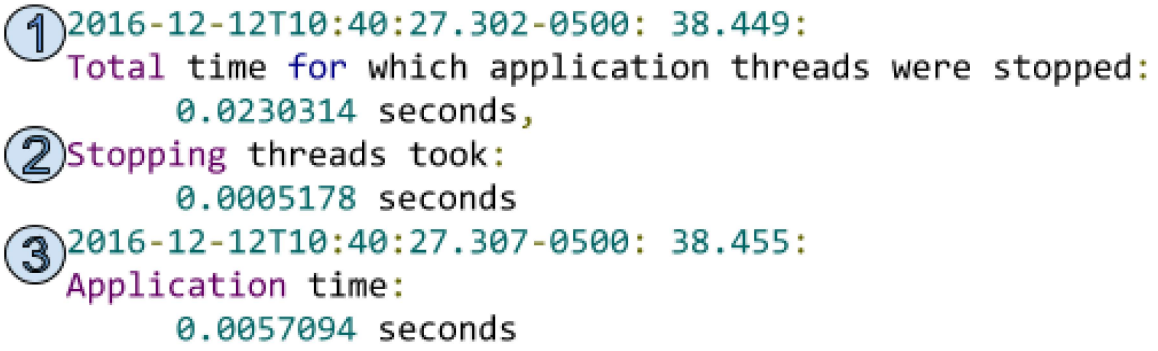
```
1 2016-12-12T10:40:08.773-0500: 19.921: [GC pause (G1 Evacuation Pause) (mixed), 0.0129474 secs]
  2 [Parallel Time: 10.9 ms, GC Workers: 4]
      [GC Worker Start (ms): Min: 19921.9, Avg: 19923.5, Max: 19926.0, Diff: 4.1]
      [Ext Root Scanning (ms): Min: 0.5, Avg: 1.4, Max: 2.4, Diff: 1.8, Sum: 5.8]
      [Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.5, Diff: 0.5, Sum: 1.1]
         [Processed Buffers: Min: 0, Avg: 1.2, Max: 2, Diff: 2, Sum: 5]
      [Scan RS (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.3]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Object Copy (ms): Min: 6.0, Avg: 7.4, Max: 8.0, Diff: 1.9, Sum: 29.5]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.3]
         [Termination Attempts: Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 4]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.1]
      [GC Worker Total (ms): Min: 6.7, Avg: 9.3, Max: 10.8, Diff: 4.1, Sum: 37.1]
      [GC Worker End (ms): Min: 19932.7, Avg: 19932.7, Max: 19932.8, Diff: 0.0]
  3 [Code Root Fixup: 0.0 ms]
    [Code Root Purge: 0.0 ms]
    [Clear CT: 0.1 ms]
  4 [Other: 2.0 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.1 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.1 ms]
      [Humongous Register: 0.1 ms]
      [Humongous Reclaim: 0.0 ms]
      [Free CSet: 0.1 ms]
  5 [Eden: 89.0M(89.0M)->0.0B(649.0M)
     Survivors: 13.0M->13.0M
     Heap: 1034.5M(2048.0M)->955.0M(2048.0M)]
6 [Times: user=0.04 sys=0.00, real=0.01 secs]
```

The third type of collection you may encounter, and the one we're working to avoid, is the Full GC. In G1, a Full GC is a single threaded Stop The World (STW) pause which will evacuate and compact all regions. There are three important pieces of information you can get from the Full GC logs.

```
1 2016-12-12T10:40:35.982-0500: 47.130:
  [Full GC (Allocation Failure)  2047M->330M(1103M), 0.1467370 secs]
        [Eden: 59.0M(102.0M)->0.0B(523.0M)
        Survivors: 26.0M->0.0B
        Heap: 2047.6M(2048.0M)->330.7M(1103.0M)],
        [Metaspace: 2840K->2840K(1056768K)]
    [Times: user=0.10 sys=0.06, real=0.15 secs]
```

1. The GC Cause, (Allocation Failure), tells you what triggered the Full GC. This will aid in how you approach tuning. Another popular cause is Metadata GC Threshold
2. How frequently they are occurring. A Full GC every few days may not be a relevant problem, whereas a Full GC every hour would be
3. How long the Full GC took. Depending on your requirements, a Full GC may not be a critical issue if it's not taking significant time to complete

The last log I want to touch on is generated from '-XX:+PrintGCApplicationStoppedTime' and '-XX:+PrintGCApplicationConcurrentTime'. These flags provide three useful data points:



1. The first point tells us the amount of time the application threads were stopped during a safepoint
2. The second point tells us the amount of time it took to bring all threads to a safepoint and suspend them
3. The third point tells us how long the application threads ran between safepoints

Moving on to the advanced flags, these options provide valuable insights into the values and thresholds.

The flags are:

-XX:+PrintAdaptiveSizePolicy	Details about the collector ergonomics
-XX:+PrintTenuringDistribution	Survivor space usage and distribution
-XX:+PrintReferenceGC	Time spent processing references

Beginning with '-XX:+PrintAdaptiveSizePolicy', this flag adds details about the G1 ergonomics to each of the previously discussed events. This provides an insight into Collection Set selection and pause time estimates.

Looking at a Young Collection, there are five new pieces of information added inside the original detailed log:

```
2016-12-30T13:28:18.343-0500: 130.629: [GC pause (G1 Evacuation Pause) (young)
(1) 130.629: [G1Ergonomics (CSet Construction) start choosing CSet, _pending_cards: 1792, predicted
base time: 2.98 ms, remaining time: 197.02 ms, target pause time: 200.00 ms]
(2) 130.629: [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 664 regions,
survivors: 112 regions, predicted young region time: 90.15 ms]
(3) 130.629: [G1Ergonomics (CSet Construction) finish choosing CSet, eden: 664 regions, survivors:
112 regions, old: 0 regions, predicted pause time: 93.13 ms, target pause time: 200.00 ms]
(4) 130.655: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: recent GC overhead higher
than threshold after GC, recent GC overhead: 28.75 %, threshold: 10.00 %, uncommitted: 0 bytes,
calculated expansion amount: 0 bytes (20.00 %)]
(5) 130.655: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation, reason:
occupancy higher than threshold, occupancy: 1013972992 bytes, allocation request: 0 bytes,
threshold: 966367620 bytes (45.00 %), source: end of GC], 0.0266860 secs]
   [Parallel Time: 25.5 ms, GC Workers: 4]
```

1. The first point tells us the number of cards in the dirty card queue which still need to be processed. It also provides a prediction on how long that processing will take. The prediction includes time for Update RS and Scan RS.
2. The second point tells us the number of regions that will be included in this collection. The time prediction includes an estimate for object copy.
3. The third point provides a final CSet and time prediction for the collection. The prediction is a combination of the base time and the young region CSet time.
4. The fourth point is logged under certain conditions, so you may only see it occasionally. G1 will try expand the heap if the amount of time you spend doing GC work versus application work is greater than a specific threshold. Note: If your min/max heap are the same, expansion cannot occur.
5. The fifth point is only logged when concurrent marking is requested. There are a few variations of this log around the GC causes, like humongous allocations or if marking tried to initiate while it was already running. The most common form we see is that the heap occupancy is greater than our IHOP and thus marking is started.

Immediately following a Young Collection where a concurrent cycle is requested, you will see a Concurrent Cycle ergonomic with a Young Collection initial-mark. The remaining details of this Young Collection remain the same.

```
(1) 130.726: [G1Ergonomics (Concurrent Cycles) initiate concurrent cycle,reason: concurrent cycle
initiation requested]
2016-12-30T13:28:18.441-0500: 130.726: [GC pause (G1 Evacuation Pause) (young) (initial-mark)
 130.726: [G1Ergonomics (CSet Construction) start choosing CSet, _pending_cards: 1282, predicted
base time: 1.75 ms, remaining time: 198.25 ms, target pause time: 200.00 ms]
```

Once marking completes, you will see a Young Collection with an ergonomic log about mixed collections.

```
2016-12-30T13:28:18.745-0500: 131.030: [GC pause (G1 Evacuation Pause) (young)
...
(1) 131.051: [G1Ergonomics (Mixed GCs) start mixed GCs, reason: candidate old regions available,
candidate old regions: 740 regions, reclaimable: 485716240 bytes (22.62 %), threshold: 5.00 %]
```

1. The first point tells us we're going to start mixed GCs, because the reclaimable percentage (22.62%) is higher than our default G1HeapWastePercent (5%).

    a. Note: If the reclaimable percentage is below 5%, you would see a similar log: do not start mixed GCs, reason: reclaimable percentage not over threshold

Now Mixed Collections will begin:

```
(1) 2016-12-30T13:28:18.777-0500: 131.063: [GC pause (G1 Evacuation Pause) (mixed)
 131.063: [G1Ergonomics (CSet Construction) start choosing CSet, _pending_cards: 1061, predicted
base time: 2.66 ms, remaining time: 197.34 ms, target pause time: 200.00 ms]
 131.063: [G1Ergonomics (CSet Construction) add young regions to CSet, eden: 89 regions,
survivors: 13 regions, predicted young region time: 11.28 ms]
(2) 131.063: [G1Ergonomics (CSet Construction) finish adding old regions to CSet, reason: old CSet
region num reached max, old: 205 regions, max: 205 regions]
(3) 131.063: [G1Ergonomics (CSet Construction) finish choosing CSet, eden: 89 regions, survivors: 13
regions, old: 205 regions, predicted pause time: 19.04 ms, target pause time: 200.00 ms]
(4) 131.073: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions available,
candidate old regions: 535 regions, reclaimable: 305363768 bytes (14.22 %), threshold: 5.00 %],
0.0141132 secs]
```

1. The first point, including the selection of the CSet and addition of young regions remain the same as a Young Collection.
2. The second point outlines the Old regions being added to the CSet for the Mixed Collection. In this case, CSet selection ended because the max old region threshold was hit. By default, G1 will only add up to 10% of the old regions to the CSet, defined by -XX:G1OldCSetRegionThresholdPercent=X.
3. The third point provides the final CSet and pause time prediction.
4. The fourth point provides us with details on the state of the Mixed GC cycle. In this case, we still have 535 old regions we could collect, amounting to 305363768 bytes or 14.22% of the heap. Given this is still higher than our waste percentage, the next collection will again be Mixed.

The following mixed collection will look identical to the previous, but may also choose to end Mixed GCs:

```
2016-12-30T13:28:18.877-0500: 131.163: [GC pause (G1 Evacuation Pause) (mixed)
...
(1) 131.187: [G1Ergonomics (Mixed GCs) do not continue mixed GCs, reason: reclaimable percentage not
over threshold, candidate old regions: 254 regions, reclaimable: 107174304 bytes (4.99 %),
threshold: 5.00 %], 0.0172178 secs]
```

1. The first point of this Mixed collection shows that the reclaimable percentage has now fallen below the 5% threshold and Mixed collections will no longer continue. The following collection will once again be Young.

Finally, let's look at the ergonomics of the Full GC:

```
1 106.445: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: allocation request failed,
allocation request: 24 bytes]
2 106.445: [G1Ergonomics (Heap Sizing) expand the heap, requested expansion amount: 1048576 bytes,
attempted expansion amount: 1048576 bytes]
3 106.445: [G1Ergonomics (Heap Sizing) did not expand the heap, reason: heap already fully expanded]
2016-12-30T13:27:54.160-0500: 106.445: [Full GC (Allocation Failure)
4 106.539: [G1Ergonomics (Heap Sizing) attempt heap shrinking, reason: capacity higher than max
desired capacity after Full GC, capacity: 2147483648 bytes, occupancy: 391145472 bytes, max desired
capacity: 1303818239 bytes (70.00 %)]
5 106.570: [G1Ergonomics (Heap Sizing) shrink the heap, requested shrinking amount: 843665409 bytes,
aligned shrinking amount: 843055104 bytes, attempted shrinking amount: 843055104 bytes]
 2047M->373M(1244M), 0.1278200 secs]
```

1. The first point tells us that there are no free regions in the primary or secondary free list, and as such, an allocation request failed and heap expansion has been requested.
2. The second point logs the amount of the expansion request. It's important to note that in both points 1 and 2, no expansion has actually been attempted yet.
3. The third point tells us that heap expansion will not be attempted. This short circuits the expansion logic when the number of available uncommitted regions is 0. As a result of the allocation failure, we end up doing a Full GC.
4. The fourth point comes up in situations where the min heap is less than the max heap. In this case, G1 will attempt to shrink the heap to 70% after a Full GC.
5. The fifth point tell us that the heap is being shrunk and by how much.

The PrintTenuringDistribution flag provides information about the Survivor space layout and threshold during each collection. This is useful because it enables you to see how objects are aging.

```
2016-12-31T10:57:07.780-0500: 10.995: [GC pause (G1 Evacuation Pause) (young)
1 Desired survivor size 79167488 bytes, new threshold 13 (max 15)
- age   1:   71477752 bytes,   71477752 total
- age   2:     857616 bytes,   72335368 total
- age   3:     807168 bytes,   73142536 total
- age   4:     580152 bytes,   73722688 total
- age   5:     529704 bytes,   74252392 total
- age   6:     756720 bytes,   75009112 total
- age   7:     605376 bytes,   75614488 total
- age   8:     908064 bytes,   76522552 total
- age   9:     529704 bytes,   77052256 total
- age  10:     681048 bytes,   77733304 total
- age  11:     605376 bytes,   78338680 total
- age  12:     630600 bytes,   78969280 total
```

1. The Tenuring Distribution data shows us three important aspects of the Survivor space:

    a. The desired survivor size, equal to the survivor size multiplied by the TargetSurvivorRatio (default 50%)

b. The target threshold, also known as the age or the number of Young GCs an object may remain. This is calculated by adding up the size of the objects in each age until the size is greater than the desired survivor size

c. The age distribution, including the size of all objects at each age and an incremental total of the Survivor space usage

```
2016-12-31T11:12:48.813-0500: 18.885: [SoftReference, 10 refs, 0.0000467 secs]
2016-12-31T11:12:48.813-0500: 18.885: [WeakReference, 67 refs, 0.0000093 secs]
(1)2016-12-31T11:12:48.813-0500: 18.885: [FinalReference, 166 refs, 0.0000306 secs]
2016-12-31T11:12:48.813-0500: 18.885: [PhantomReference, 0 refs, 0.0000087 secs]
2016-12-31T11:12:48.813-0500: 18.885: [JNI Weak Reference, 0.0000077 secs]
```

Finally, we have the diagnostic and experimental flags. These flags can add a significant amount of logging and should be used only when necessary and when you're trying to debug a specific problem.

-XX:+UnlockDiagnosticVMOptions	
-XX:+G1SummarizeConcMark	Summarizes Concurrent Mark at JVM exit
-XX:+G1PrintHeapRegions	Print the heap regions selected for allocation, cleanup, reuse, compact, cset, commit, failure, etc...
-XX:+G1PrintRegionLivenessInfo	Prints previous and next liveness data per Old region before and after every concurrent mark cycle
-XX:+G1SummarizeRSetStats -XX:G1SummarizeRSetStatsPeriod=1	Print RSet processing information every X, where X is measured in GC cycles
-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1 -XX:+LogVMOutput -XX:LogFile=/path/to/gc.log	Prints the reason and some details about safepoint synchronization. Can control how many events to collect before printing. By default, logs to STDOut - LogVMOutput can push it to a file
-XX:+UnlockExperimentalVMOptions	
-XX:G1LogLevel=fine, finer, finest	Increases logging verbosity on collections
-XX:+G1TraceEagerReclaimHumongousObjects	Prints details about live and dead Humongous objects during each collection
-XX:+G1ConcRegionFreeingVerbose	Debug JVM

The output from '-XX:+G1PrintHeapRegions' adds a LOT of logging. The image below represents a compacted set of events to illustrate the various types that can occur. Certain events occur under Young GC whereas others occur during Full GC.

```
G1HR #StartGC 921
G1HR #StartFullGC 1081
 G1HR COMMIT [0x00000000d3400000,0x00000000d3500000]
 G1HR ALLOC(Eden) 0x00000000d3400000
 G1HR CSET 0x00000000d3400000
 G1HR CLEANUP 0x00000000d3400000
 G1HR UNCOMMIT [0x00000000d3400000,0x00000000d3500000]
 G1HR ALLOC(Old) 0x00000000bed00000
 G1HR RETIRE 0x00000000bed00000 0x00000000bed71e00
 G1HR REUSE 0x00000000bed00000
 G1HR ALLOC(Survivor) 0x00000000d3300000
 G1HR EVAC-FAILURE 0x00000000fab00000
 G1HR POST-COMPACTION(Old) 0x0000000080000000 0x00000000800fff80
 G1HR ALLOC(SingleH) 0x00000000b2c00000 0x00000000b2cfffb8
 G1HR ALLOC(StartsH) 0x00000000b2d00000 0x00000000b2e00000
 G1HR ALLOC(ContinuesH) 0x00000000b2e00000 0x00000000b2e00038
G1HR #EndGC 921
G1HR #EndFullGC 1081
```

Printing of the heap region events is only necessary when trying to debug very specific problems such as:

- Debugging evacuation failures and the number of failed regions
- Determining the size and frequency of Humongous objects
- Tracking and evaluating the number of Eden, Survivor and Old regions being allocated and collected as part of the CSet

1. COMMIT
    - The heap is being initialized or expanded, defines the top and bottom of the region being modified
2. ALLOC(Eden)
    - A region, defined by the bottom address, is allocated as an Eden region
3. CSET
    - Region selected for the CSET, all of these regions will be reclaimed
4. CLEANUP
    - A region, defined by the bottom address, is completely empty and cleaned up during concurrent marking
5. UNCOMMIT
    - After a Full GC, if the heap is shrunk, you will see a set of regions uncommitted

- A region, defined by the bottom address, is allocated as an Old region

7. RETIRE
   - At the end of a collection, the last allocated Old region is marked as retired
8. REUSE
   - At the start of the next GC, the previously retired Old region is reused as the starting point
9. ALLOC(Survivor)
   - A region, defined by the bottom address, is allocated as a Survivor region
10. EVAC-FAILURE
    - If an evacuation failure happens during a collection, this event will outline each of the regions which failed
11. POST-COMPACTION(Old)
    - After a Full GC, a post-compaction event is generated for Old and Humongous regions which contain the remaining live data
12. ALLOC(SingleH)
    - A region, defined by the bottom and top address, is allocated as a Single Humongous region where the object fits into a single region
13. ALLOC(StartsH)
    - A region, defined by the bottom and top address, is allocated as a Starts Humongous region where the object is large enough to span more than one region
14. ALLOC(ContinuesH)
    - A region, defined by the bottom and top address, is allocated as a Continues Humongous region where this is the continuation of an object spanning more than one region

The output from '-XX:+G1PrintRegionLivenessInfo' also produces a lot of additional logging, but it's limited to the frequency of concurrent marking cycles. It's very useful if you want to analyze what your Old regions look like after Concurrent Marking, as well as after they're sorted for efficiency for CSet selection.

```
1 ### PHASE Post-Marking @ 73.555
  ### HEAP  reserved: 0x0000000080000000-0x0000000100000000  region-size: 1048576
  ### type address-range                          used prev-live next-live     gc-eff remset code-roots
  ### OLD  0x0000000080000000-0x0000000080100000 1048240    741024    713920    20331.7  27128        1296
  ### OLD  0x0000000080100000-0x0000000080200000 1034000    960520    956632        0.0  26624         792
  ### OLD  0x0000000080300000-0x0000000080400000 1044952    593744    569136  1267409.7   4176          16
  ### OLD  0x0000000080400000-0x0000000080500000 1037944    837368    762392   588331.7   4176          16
  ### OLD  0x0000000082300000-0x0000000082400000 1039720   1039720   1039720   456255.5   3832          16
  ### OLD  0x0000000084900000-0x0000000084a00000 1048576   1048576    926536  5111832.7   3144          16
  ### OLD  0x0000000084a00000-0x0000000084b00000 1048576   1048576         0 14000207.8   3144          16
  ### OLD  0x0000000084d00000-0x0000000084e00000 1048576   1048576    784296 12047617.2   3832          16
  ### OLD  0x0000000084e00000-0x0000000084f00000 1048576   1048576         0 27699644.2   3144          16
  ### OLD  0x0000000085000000-0x0000000085100000 1048576   1048576    540256  2165897.6   3144          16
  ### OLD  0x0000000085200000-0x0000000085300000 1048576   1048576         0 23474578.3   3488          16
  ### OLD  0x000000008b300000-0x000000008b400000 1048576   1048576    156752  8969162.3   3144          16
  ### SUMMARY  capacity: 2048.00 MB  used: 1872.46 MB / 91.43 %  prev-live: 1760.23 MB / 85.95 %  next-live:
  1170.63 MB / 57.16 %  remset: 6.54 MB  code-roots: 0.03 MB

2 ### PHASE Post-Sorting @ 73.560
  ### HEAP  reserved: 0x0000000080000000-0x0000000100000000  region-size: 1048576
  ### type address-range                          used prev-live next-live      gc-eff remset code-roots
  ### OLD  0x00000000aa800000-0x00000000aa900000 1048576       176         0 216132901.1   3144          16
  ### OLD  0x00000000a4700000-0x00000000a4800000 1048576       192         0  38258547.5   3144          16
  ### OLD  0x00000000bd400000-0x00000000bd500000 1048576    130528         0  37362281.9   3144          16
  ### OLD  0x0000000091a00000-0x0000000091b00000 1048576    135976         0  35935160.9   3144          16
  ### OLD  0x000000008c500000-0x000000008c600000 1048576    136416         0  35823933.8   3144          16
  ### OLD  0x00000000adc00000-0x00000000add00000 1048576    141560         0  34565451.6   3144          16
  ### OLD  0x00000000bff00000-0x00000000c0000000 1048576    147056         0  33300802.9   3144          16
  ### OLD  0x00000000ac200000-0x00000000ac300000 1048576    100656         0  30258927.1   3144          16
  ### OLD  0x00000000aa200000-0x00000000aa300000 1048576    109456         0  28755940.9   3144          16
  ### OLD  0x00000000c0300000-0x00000000c0400000 1048576    110664         0  28559134.0   3144          16
  ### OLD  0x00000000bd500000-0x00000000bd600000 1048576    110768         0  28542292.7   3144          16
  ### SUMMARY  capacity: 319.00 MB  used: 318.89 MB / 99.97 %  prev-live: 139.60 MB / 43.76 %  next-live:
  0.00 MB / 0.00 %  remset: 1.22 MB  code-roots: 0.01 MB
```

The Post-Marking output, which can also include EDEN, FREE, SURV, HUMS and HUMC region types, outlines a detailed per region breakdown of the previous and next top-at-mark-start (TAMS) levels as it pertains to object liveliness for the marking cycle. There are eight pieces of information defined in the output:

- Region Type: Can be Old, Eden, Survivor, Humongous Start, Humongous Continues and Free
- Address Range:The bottom and end value for the region
- Used: Total number of used bytes in the region, measured as the total between the bottom and current top of the region
- Prev-Live: The number of live bytes from the point of view of the previous marking cycle, measured as the total between the previous TAMS and the current top plus the known live objects (previously marked) from the previous marking cycle
- Next-Live: The number of live bytes from the point of view of the current marking cycle, measured as the total between the the next TAMS and the current top plus the known live objects (marked) from the current marking cycle
- GC-Eff: This is measured by taking the reclaimable bytes (known live bytes minus total region capacity) divided by the estimated time to collect the region (RS Scan, Object Copy and Other Time). The higher the efficiency, the better the region is as a candidate

- Remset: Size of the remembered set for the region, measured by adding the size of the region table to the size of the bitmap multiplied by the heap word size
- Code-Roots: The amount of memory, in bytes, the region is consuming for strong code roots.

The verbosity of '-XX:+G1SummarizeRSetStats' can be controlled by setting '-XX:G1SummarizeRSetStatsPeriod=XX', which defines the number of GC cycles between summaries. This setting is useful when debugging RSet specific issues, such as excessive time spent during Scan and or Update RS activity.

```
1 2017-01-01T14:04:50.094-0500: 75.103: [GC pause (G1 Evacuation Pause) (young) Before GC RS summary
   Recent concurrent refinement statistics
    Processed 506 cards
    Of 94 completed buffers:
              94 (100.0%) by concurrent RS threads.
               0 (  0.0%) by mutator threads.
    Did 0 coarsenings.
    Concurrent RS threads times (s)
           0.00     0.00     0.00     0.00
    Concurrent sampling threads times (s)
           0.00
2  Current rem set statistics
    Total per region rem sets sizes = 6433K. Max = 26K.
          3137K ( 48.8%) by 1022 Young regions
             0K (  0.0%) by 0 Humonguous regions
          1074K ( 16.7%) by 350 Free regions
          2220K ( 34.5%) by 676 Old regions
     Static structures = 64K, free_lists = 295K.
     17612 occupied cards represented.
              0 (  0.0%) entries by 1022 Young regions
              0 (  0.0%) entries by 0 Humonguous regions
              0 (  0.0%) entries by 350 Free regions
          17612 (100.0%) entries by 676 Old regions
     Region with largest rem set =
  0:(O)[0x0000000080000000,0x00000000800ffff0,0x0000000080100000],size = 26K, occupied = 3K.
3  Total heap region code root sets sizes = 34K.  Max = 1K.
            15K ( 46.4%) by 1022 Young regions
             0K (  0.0%) by 0 Humonguous regions
             5K ( 15.9%) by 350 Free regions
            12K ( 37.7%) by 676 Old regions
     85 code roots represented.
              0 (  0.0%) elements by 1022 Young regions
              0 (  0.0%) elements by 0 Humonguous regions
              0 (  0.0%) elements by 350 Free regions
             85 (100.0%) elements by 676 Old regions
     Region with largest amount of code roots =
  0:(O)[0x0000000080000000,0x00000000800ffff0,0x0000000080100000], size = 1K, num_elems = 0.
```

This log defines 3 key sections of data:

1. The first has to do with concurrent refinement statistics. In this case, 506 cards were processed from 94 buffers and 100% of the work was completed by the concurrent threads. The last piece, coarsenings, tells us how many RSets have been coarsened. Essentially, when many regions reference an object in one region, you may not have enough space for a new region bitmap. In this case, the referencing regions bitmap is marked as coarsened. This is expensive because the

entire referencing region must be scanned for incoming references;

2. The second section defines the overall remember set statistics on a per region type basis
3. The third section defines the overall code root set statistics on a per region type basis

When it comes to a standard collection, many of the line items are comprised of several underlying tasks. If one task is taking longer than anticipated, you can enable '-XX:G1LogLevel=finest' to provide a detailed breakdown of each operation. Additionally, it adds detailed execution time each individual worker spent on each task.

```
   [Parallel Time: 26.2 ms, GC Workers: 4]
      [GC Worker Start (ms):  141584.0  141584.0  141584.0  141584.0
       Min: 141584.0, Avg: 141584.0, Max: 141584.0, Diff: 0.1]
      [Ext Root Scanning (ms):  0.1  0.1  0.1  0.1
       Min: 0.1, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.4]
         [Thread Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [StringTable Roots (ms):  0.1  0.1  0.1  0.1
          Min: 0.1, Avg: 0.1, Max: 0.1, Diff: 0.0, Sum: 0.3]
         [Universe Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [JNI Handles Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [ObjectSynchronizer Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [FlatProfiler Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Management Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [SystemDictionary Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [CLDG Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [JVMTI Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [CodeCache Roots (ms):  25.9  23.5  23.5  23.5
Min: 23.5, Avg: 24.1, Max: 25.9, Diff: 2.4, Sum: 96.5]
         [CM RefProcessor Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Wait For Strong CLD (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Weak CLD Roots (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
[SATB Filtering (ms):  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Update RS (ms):  25.9  23.5  23.5  23.5
       Min: 23.5, Avg: 24.1, Max: 25.9, Diff: 2.4, Sum: 96.4]
         [Processed Buffers:  215  329  341  343
          Min: 215, Avg: 307.0, Max: 343, Diff: 128, Sum: 1228]
```

```
   Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Code Root Scanning (ms):  0.0  0.0  0.0  0.0
       Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [Object Copy (ms):  0.1  0.1  0.0  0.0
       Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.1]
      [Termination (ms):  0.0  2.4  2.4  2.4
       Min: 0.0, Avg: 1.8, Max: 2.4, Diff: 2.4, Sum: 7.2]
   [Termination Attempts:  1  1  1  1
         Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 4]
      [GC Worker Other (ms):  0.0  0.0  0.0  0.0
       Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
      [GC Worker Total (ms):  26.1  26.1  26.0  26.0
       Min: 26.0, Avg: 26.1, Max: 26.1, Diff: 0.0, Sum: 104.2]
      [GC Worker End (ms):  141610.0  141610.1  141610.0  141610.1
       Min: 141610.0, Avg: 141610.0, Max: 141610.1, Diff: 0.0]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.3 ms]
   [Other: 4.1 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.2 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 0.3 ms]
         [Parallel Redirty:  0.0  0.0  0.0  0.0
          Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]
         [Redirtied Cards:  1  0  0  0
          Min: 0, Avg: 0.2, Max: 1, Diff: 1, Sum: 1]
      [Humongous Register: 0.1 ms]
         [Humongous Total: 511]
         [Humongous Candidate: 511]
      [Humongous Reclaim: 2.5 ms]
         [Humongous Reclaimed: 510]
      [Free CSet: 0.0 ms]
         [Young Free CSet: 0.0 ms]
         [Non-Young Free CSet: 0.0 ms]
   [Eden: 1024.0K(1227.0M)->0.0B(1227.0M) Survivors:
1024.0K->1024.0K Heap: 1858.1M(2048.0M)->4105.1K(2048.0M)]
 [Times: user=0.09 sys=0.00, real=0.03 secs]
```

The final setting we're going to touch on is '-XX:+G1TraceEagerReclaimHumongousObjects'. If you're experiencing a large number of Humongous allocations, you can enable this option to provide detailed information on which Humongous objects the collector considers to be Dead or Live for eager reclamation. Eager reclaim of Humongous regions was introduced in JDK8u60.