**-: Garbage Collection:-**

In this post, we will review how garbage collector logs look like and what useful information one can obtain from them. For this purpose, we have turned on GC logging for a JVM running with *-XX:+UseSerialGC* by using the following startup parameters:

-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -Xloggc:/<PATH>/<Log\_file\_name>

This resulted in the output similar to following:

2015-05-26T14:45:37.987-0200: 151.126: [GC (Allocation Failure) 151.126: [DefNew: 629119K->69888K(629120K), 0.0584157 secs] 1619346K->1273247K(2027264K), 0.0585007 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]

2015-05-26T14:45:59.690-0200: 172.829: [GC (Allocation Failure) 172.829: [DefNew: 629120K->629120K(629120K), 0.0000372 secs]172.829: [Tenured: 1203359K->755802K(1398144K), 0.1855567 secs] 1832479K->755802K(2027264K), [**Metaspace**: 6741K->6741K(1056768K)], 0.1856954 secs] [Times: user=0.18 sys=0.00, real=0.18 secs]

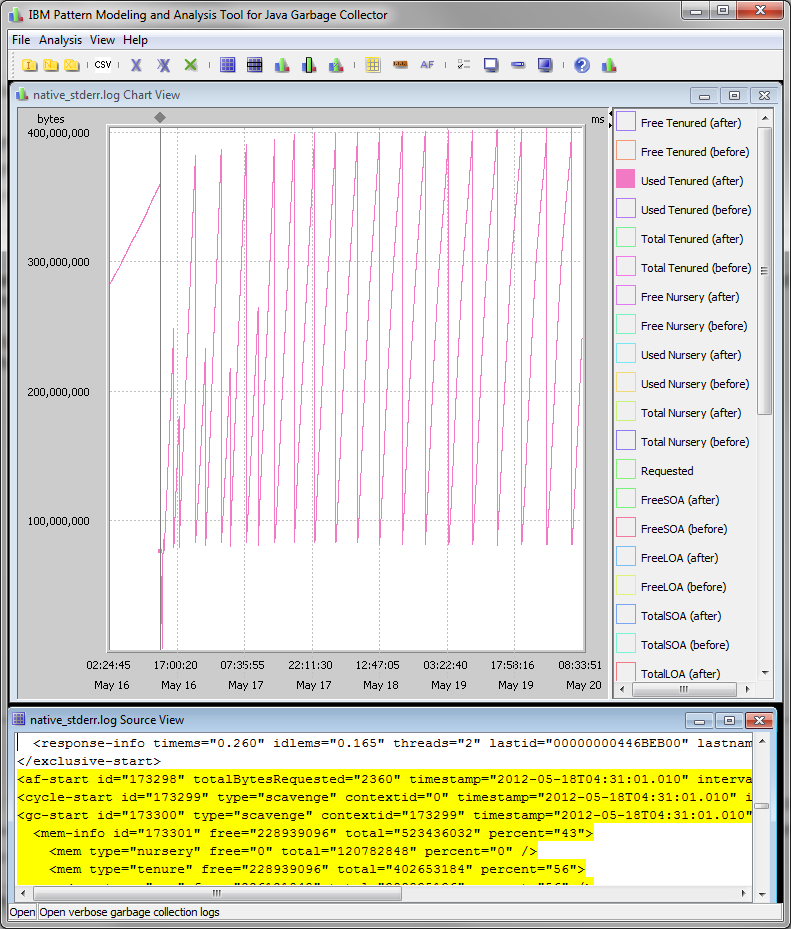
The above short snippet from the GC logs exposes a lot of information about what is going on inside the JVM. As a matter of fact, in this snippet there were two Garbage Collection events taking place, one of them cleaning Young generation and the other one taking care of entire heap. Lets look at the first one of these events, a Minor GC, taking place in Young generation:

2015-05-26T14:45:37.987-02001:151.1262:[GC3(Allocation Failure4) 151.126: [DefNew5:629119K->69888K6(629120K)7, 0.0584157 secs]1619346K->1273247K8(2027264K)9,0.0585007 secs10][Times: user=0.06 sys=0.00, real=0.06 secs]11

1. 2015-05-26T14:45:37.987-0200 – Time when the GC event started.
2. 151.126 – Time when the GC event started, relative to the JVM startup time. Measured in seconds.
3. GC – Flag to distinguish between Minor & Full GC. This time it is indicating that this was a Minor GC.
4. Allocation Failure – Cause of the collection. In this case, the GC is triggered due to a data structure not fitting into any region in Young Generation.
5. DefNew – Name of the garbage collector used. This cryptic name stands for the single-threaded mark-copy stop-the-world garbage collector used to clean Young generation.
6. 629119K->69888K – Usage of Young generation before and after collection.
7. (629120K) – Total size of the Young generation.
8. 1619346K->1273247K – Total used heap before and after collection.
9. (2027264K) – Total available heap.
10. 0.0585007 secs – Duration of the GC event in seconds.
11. [Times: user=0.06 sys=0.00, real=0.06 secs] – Duration of the GC event, measured in different categories:
    * user – Total CPU time that was consumed by Garbage Collector threads during this collection
    * sys – Time spent in OS calls or waiting for system event
    * real – Clock time for which your application was stopped. As Serial Garbage Collector always uses just a single thread, real time is thus equal to the sum of user and system times.
12. From the above snippet we can understand exactly what was happening with memory consumption inside the JVM during the GC event. Before this collection heap usage totaled at 1,619,346K. Out of this amount, the Young generation consumed 629,119K. From this we can calculate the Old generation usage being equal to 990,227K.
13. A more important conclusion is hidden in the next batch of numbers, indicating that after the collection young generation usage decreased by 559,231K, but total heap usage decreased only by 346,099K. From this we can again derive that 213,132K of objects had been promoted from Young generation to Old.

Once you have gathered the verbose GC you can use IBM Pattern Modeling and Analysis Tool for Java Garbage Collector from the below link:

<ftp://public.dhe.ibm.com/software/websphere/appserv/support/tools/pmat/ga456.jar>



**AWS-2 GC Log**

Java HotSpot(TM) 64-Bit Server VM (25.66-b17) for linux-amd64 JRE (1.8.0\_66-b17), built on Oct 6 2015 17:28:34 by "java\_re" with gcc 4.3.0 20080428 (Red Hat 4.3.0-8)

Memory: 4k page, physical 1016292k(669028k free), swap 0k(0k free)

CommandLine flags: -XX:InitialHeapSize=16260672 -XX:MaxHeapSize=1048576000 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops

0.171: [GC (Allocation Failure) 0.171: [DefNew: 4369K->512K(4928K), 0.0018554 secs] 4369K->1198K(15872K), 0.0019024 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

0.297: [GC (Allocation Failure) 0.297: [DefNew: 4928K->512K(4928K), 0.0024565 secs] 5614K->1940K(15872K), 0.0024935 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

2.110: [GC (Allocation Failure) 2.110: [DefNew: 1885K->146K(4928K), 0.0014455 secs]2.111: [Tenured: 6829K->5072K(10944K), 0.0215997 secs] 8657K->5072K(15872K), [Metaspace: 16858K->16858K(1064960K)], 0.0231096 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]

**AWS-1-GCLog**

Java HotSpot(TM) 64-Bit Server VM (24.80-b11) for linux-amd64 JRE (1.7.0\_80-b15), built on Apr 10 2015 19:53:14 by "java\_re" with gcc 4.3.0 20080428 (Red Hat 4.3.0-8)

Memory: 4k page, physical 1016292k(266192k free), swap 0k(0k free)

CommandLine flags: -XX:InitialHeapSize=16260672 -XX:MaxHeapSize=1048576000 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedOops

0.170: [GC0.170: [DefNew: 4239K->512K(4800K), 0.0027390 secs] 4239K->1155K(15424K), 0.0027950 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

0.258: [GC0.258: [DefNew: 4800K->512K(4800K), 0.0029420 secs] 5443K->1862K(15424K), 0.0029720 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

0.412: [GC0.412: [DefNew: 4800K->359K(4800K), 0.0023150 secs] 6150K->1913K(15424K), 0.0023570 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

2.563: [GC2.563: [DefNew: 1361K->96K(4800K), 0.0009370 secs]2.564: [Tenured: 5844K->4791K(10624K), 0.0238120 secs] 7163K->4791K(15424K), [Perm : 16787K->16787K(21248K)], 0.0248230 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

MetaSpace is based on native memory, so it keeps the java objects on the disks rather than on VM?

We can achieve the famed OOM error by setting the MaxMetaspaceSize argument to JVM and running the sample program provided.

Even the PermGen is GC collected by adding the args like -XX:+CMSClassUnloadingEnabled