25. How to Capture Java Heap dumps? – 7 options

Heap Dumps are vital artifacts to diagnose memory-related problems such as slow memory leaks, Garbage Collection problems, and java.lang.OutOfMemoryError.They are also vital artifacts to optimize the memory consumption.

There are great tools like Eclipse MAT and Heap Hero to analyze heap dumps. However, you need to provide these tools with heap dumps captured in the correct format and correct point in time.

This article gives you multiple options to capture heap dumps. However, in my opinion, first 3 are effective options to use and others are good options to be aware.

1. jmap

jmap print heap dumps into specified file location. This tool is packaged within JDK. It can be found in \bin folder.

Here is how you should invoke jmap:

```
jmap -dump:format=b,file=<file-path> <pid>

where
pid: is the Java Process Id, whose heap dump should be captured
file-path: is the file path where heap dump will be written in to.
```

Example

```
1 jmap -dump:format=b,file=/opt/tmp/heapdump.bin 37320
```

Note: It's quite important to pass "live" option. If this option is passed, then only live objects in the memory are written into the heap dump file. If this option is not passed, all the objects, even the ones which are ready to be garbage collected are printed in the heap dump file. It will increase the heap dump file size significantly. It will also make the analysis tedious. To troubleshoot memory problems or optimize memory, just "live" option should suffice the need.

2. HeapDumpOnOutOfMemoryError

When application experience java.lang.OutOfMemoryError, it's ideal to capture heap dump right at that point to diagnose the problem because you want to know what objects were sitting in memory and what percentage of memory they were occupying when java.lang.OutOfMemoryError occurred. However, due to the heat of the moment, most times, IT/Operations team forgets to capture heap dump. Not only that, they also restart the application. It's extremely hard to diagnose any memory problems without capturing heap dumps at right time.

That's where this option comes very handy. When you pass '-XX:+HeapDumpOnOutOfMemoryError' system property during application startup, JVM will capture heap dumps right at the point when JVM experiences OutOfMemoryError.

Sample Usage:

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/opt/tmp/heapdump.bin
```

Note: Captured heap dump will be printed at the location specified by '-XX:HeapDumpPath' system property.

Best Practice: Keep this property configured in all the applications at all the times, as you never know when OutOfMemoryError will happen.jcmd3.

3. jcmd

jcmd tool is used to send diagnostic command requests to the JVM. It's packaged as part of JDK. It can be found in \bin folder.

Here is how you should invoke jcmd:

```
jcmd <pid> GC.heap_dump <file-path>
where
pid: is the Java Process Id, whose heap dump should be captured
file-path: is the file path where heap dump will be written in to.
```

Example

```
1 jcmd 37320 GC.heap_dump /opt/tmp/heapdump.bin
```

4. JVisualVM

JVisualVM is a monitoring, troubleshooting tool that is packaged within the JDK. When you launch this tool, you can see all the Java processes that are running on the local machine. You can also connect to java process running on remote machine using this tool.

Steps:

1. Launch jvisualvm under \bin\ folder
2. Right click on one of the Java process
3. Click on the 'Heap Dump' option on the drop-down menu
4. Heap dump will be generated
5. File path where heap dump is generated will be specified in the Summary Tab > Basic Info > File section

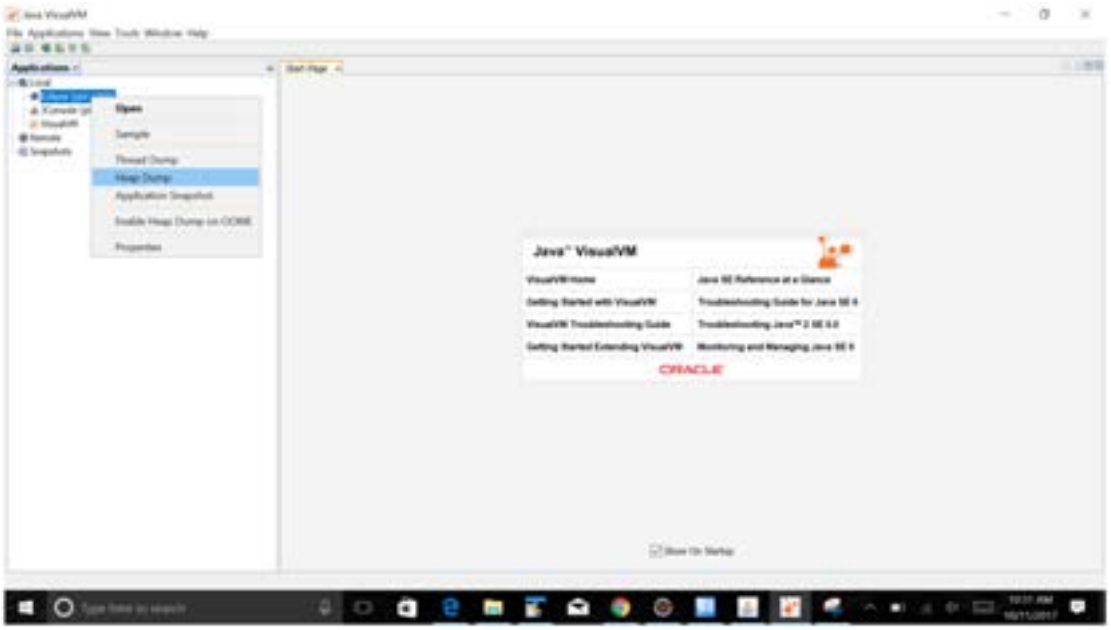


Fig 1: Capturing Heap Dump from JVisualVM

5. JMX

There is a com.sun.management:type=HotSpotDiagnostic MBean. This MBean has 'dumpHeap' operation. Invoking this operation will capture the heap dump. 'dumpHeap' operation takes two input parameters:

1. outputFile: File path where heap dump should be written
2. live: When 'true' is passed only live objects in heap are captured

You can use JMX clients such as JConsole, jmxsh, Java Mission Control to invoke this MBean operation.

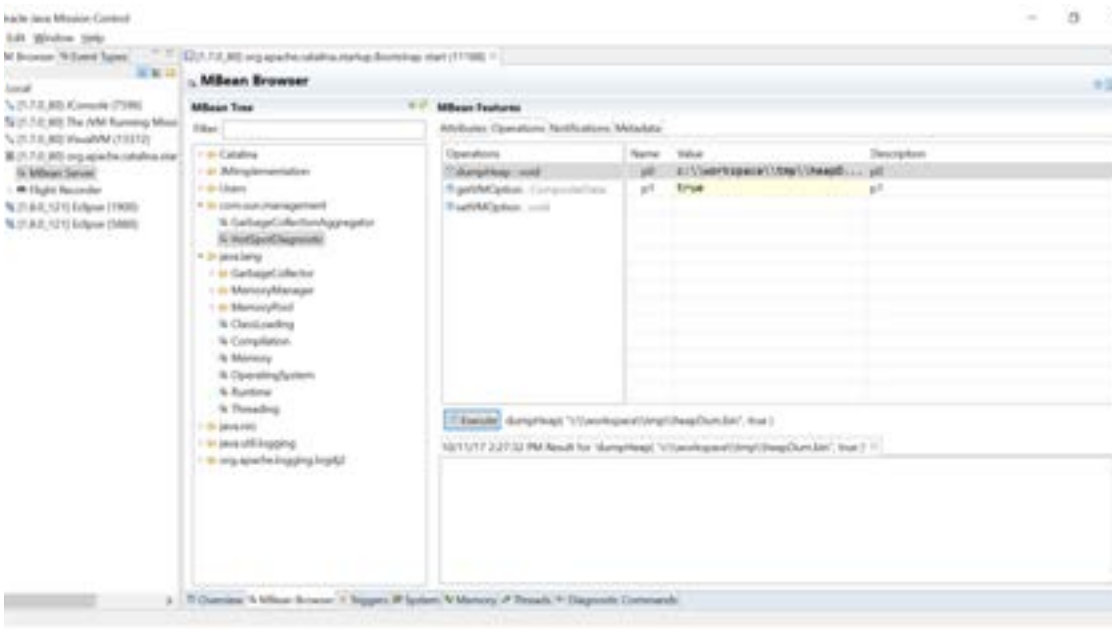


Fig 2: Using Java Mission Control as the JMX client to generate heap dump

6. Programmatic Approach

Instead of using tools, you can also programmatically capture heap dumps from the application. There might be cases where you want to capture heap dumps based on certain events in the application. Here is a good article from Oracle which gives the source code for capturing heap dumps from the application, by invoking the com.sun.management:type=HotSpotDiagnostic MBean JMX Bean, that we discussed in the above approach.

7. IBM Administrative Console

If your application is running on IBM Websphere Application Server, you can use the administrative console to generate heaps.

Steps:

1. Start administrative console
2. In the navigation pane, click Troubleshooting > Java dumps and cores
3. Select the server_name for which you want to generate the heap dump
4. Click Heap dump to generate the heap dump for your specified server

You can also use wsadmin to generate heap dumps.

Additional Knowledge

Java Heap Stack

26. Benefits of setting initial and maximum memory size to the same value

When we launch applications, we specify the initial memory size and maximum memory size. For the applications that run on JVM (Java Virtual Machine), initial and maximum memory size is specified through '-Xms' and '-Xmx' arguments. If Java applications are running on containers, it's specified through '-XX: InitialRAMPercentage' and '-XX: MaxRAMPercentage' arguments. Most enterprises set the initial memory size to a lower value than the maximum memory size. As opposed to this commonly accepted practice, setting the initial memory size the same as the maximum memory size has certain 'cool' advantages. Let's discuss them in this post.

Video: To see the visual walk-through of this post, click below:

Watch video

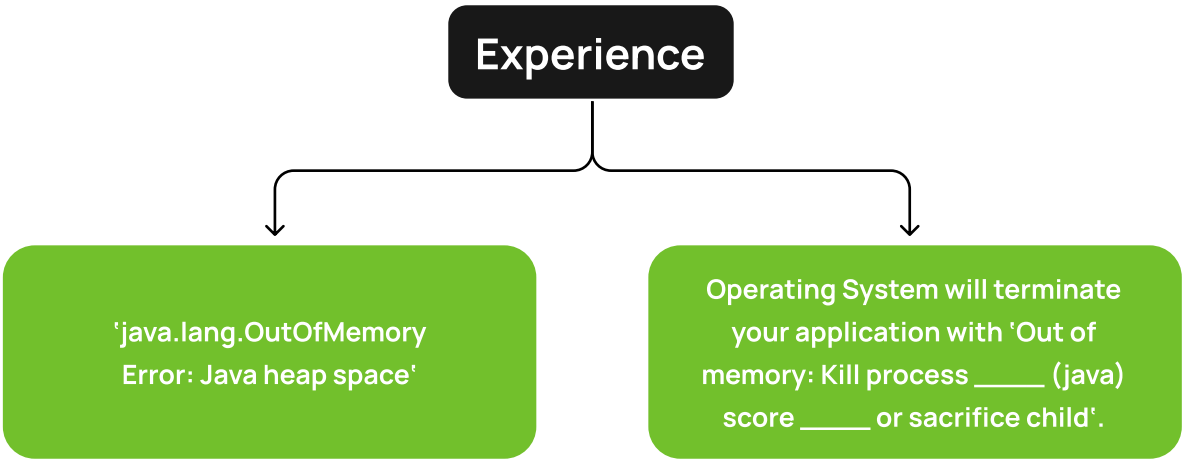
https://www.youtube.com/watch?v=TzSrz4cMWgU&t=1s

1. Availability

Say suppose you are launching your application with the initial heap size to be 2GB and maximum heap size to be 24GB. It means when the application starts, the operating system will allocate 2GB of memory to your application. From that point, as the application starts to process new requests, additional memory will be allocated until a maximum of 24GB is reached.

Say suppose while your application's memory consumption is in the process of growing from 2GB to 24GB, at that time, say some other process(es) gets launched in your device and it starts to consume memory. This situation is quite common in production/cloud environments, especially where your application runs with few other neighbors (such as custom scripts, cron jobs, monitoring agents…).

When this situation happens, your application will either experience:



It means, your application will crash in the middle of the transaction. If your application is launched with maximum memory during the startup time itself, then your application will be safe. The operating system will only terminate newly launched scripts/cron jobs whose memory consumption is growing and not your application whose memory was already fully allocated during the startup time. For more details about this issue, you may refer to this post.

2. Performance

We have also observed applications launched with the same initial heap size and maximum heap size tend to perform comparatively better than applications launched with lower initial heap sizes.

Here is a real-world case study: We took a memory-intensive application HeapHero for

our study. This application processes very large size binary heap dump files and generates analytics reports. In this HeapHero application, we repeatedly analyzed an 11GB binary file, so that it will put memory pressure on the operating system.

We conducted two test scenarios:

Scenario 1: We set the initial heap size to be 2GB and the maximum heap size to be 24GB.

Scenario 2: We set both initial and maximum heap size to be 24GB.

In Scenario #1, we observed the average response time to be 385.32 seconds, whereas in Scenario #2, we observed the average response time to be 366.55 seconds. There was a 5.11% improvement in the response time. This improvement in response time was happening due to 2 reasons:

a. Memory allocation and deallocation from OS

b. GC Pause time impact

Let's discuss them here:

a. **Memory allocation and deallocation from OS –** When you have set the different sizes for the initial and maximum heap size, then the JVM will have to negotiate with the Operating System to allocate memory when there is a need. Similarly, when an application's demand for memory goes down at the run time, Operating System will take away the allocated memory. This constant allocation and deallocation will add overhead to the application.

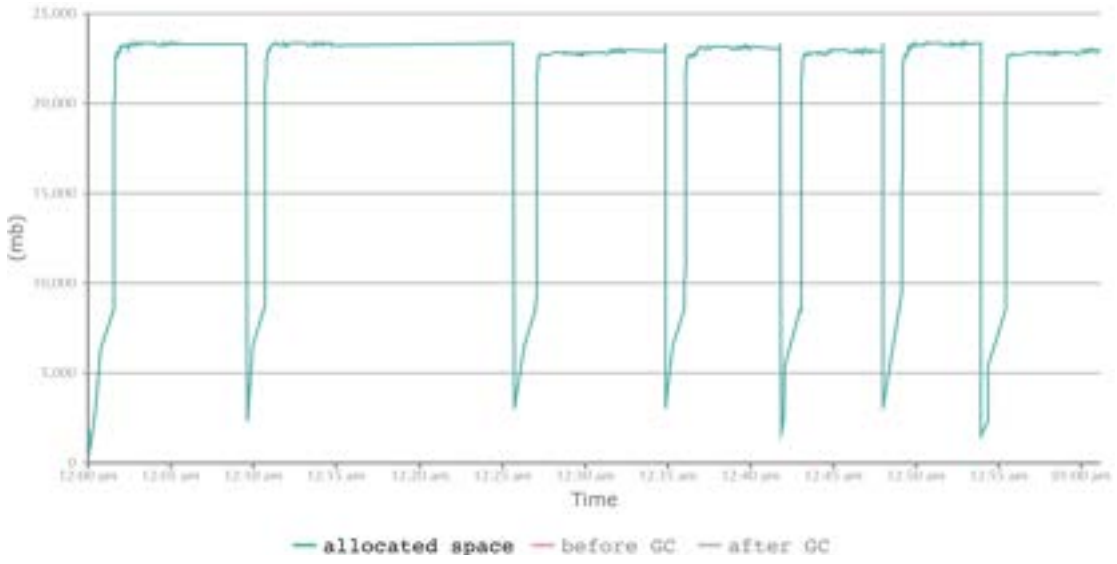


Fig 1: Scenario 1 memory allocation fluctuating (graph by GCeasy)

The above graph shows Scenario 1 JVM's allocated & deallocated memory. From the graph, you can notice that memory is constantly fluctuating. You can see heap size fluctuating between 2GB to 24GB. When the application processes heap dump, memory shoots up to 24GB. After processing, memory drops back to 2GB. Once again when it processes a new heap dump, memory shoots back to 24GB.

Fig 1.1: Scenario 2 memory allocation constant (graph by GCeasy)

The above graph shows Scenario 2 JVM's allocated memory during its lifetime. You can see that there is no fluctuation. Memory is reserved from the operating system during the startup time and from that point, there is no fluctuation. It always remains at 24GB irrespective of the activity in the application. This behavior has the potential to boost the application's performance to a certain degree.

b. **GC Pause time impact: –** When Garbage collection runs, it pauses your application which will have a negative customer impact. We studied the Garbage collection performance of both scenarios using the GCeasy tool. Below are the results:

Scenario	GC Throughput	Avg GC Pause time	Max GC Pause time
Scenario 1: initial heap size set lower than the max heap size	96.59%	99.2 ms	5.23 sec
Scenario 2: initial heap size set same as max heap size	97.83%	97.0 ms	1.65 sec

We noticed that there was minor degradation in the GC throughput and GC pause times. In scenario #1, GC throughput was 96.59%, whereas in scenario #2 GC throughput was slightly better – 97.83%. Similarly in scenario #1, the Max GC's pause time was 5.23 seconds, whereas in scenario #2 it was only 1.65 seconds. (If you would like to learn more about Garbage Collection KPIs and tuning, you may refer to this video clip).
If you would like to see the detailed metrics of the Garbage collection behavior between these two scenarios, you can refer to the reports here: Scenario 1 GC report, Scenario 2 GC report.

3. Application Startup Time

Your application's startup time will also be better if you set the initial heap size to be the same as the maximum heap size. Below is the excerpt from the Oracle documentation:

"If the initial heap is too small, the Java application startup becomes slow as the JVM is forced to perform garbage collection frequently until the heap grows to a more reasonable size. For optimal startup performance, set the initial heap size to be the same as the maximum heap size."

4. Cost

Irrespective of whether you set your initial heap size (-Xms) and maximum heap size (-Xmx) to the same value or different value, the computing cost ($) that you pay to your cloud hosting provider will not change. Say suppose you are using 2.2x. large 32G RHEL on-demand AWS ec2 – instances in US West (North California), then you will end up paying $ 0.5716/hour, irrespective of what value to set initial heap size and maximum heap size. Cloud providers don't charge you based on how much memory you use in that machine. They charge only based on the time you use the instances. Thus, there is no cost saving in setting the initial heap size lower than the maximum heap size.

Conclusion

Configuring the initial heap size to be less than the maximum heap size makes sense when you are configuring the thread pool or connection pool. In these resources over-allocating will have an unnecessary impact, however, it isn't the case with memory. Thus if you are building enterprise applications, you should strongly consider setting your initial heap size and maximum heap size to the same value.

27. Flavors of OutOfMemoryErrors

OutOfMemoryError is a runtime error in Java that occurs when the Java Virtual Machine (JVM) is unable to allocate an object due to insufficient space in the Java heap. The Java Garbage Collector (GC) cannot free up the space required for a new object, which causes the error.

There are different types of OutOfMemoryErrors. Each OutOfMemoryError has different causes. Each cause has its own fix. The above document summarizes all of this information in 1 page.

Of-course beauty is in the eye of the beholder, from my eye this document looks beautiful. If it looks beautiful to you as well, you can print it and post it on your desk. It might turn out to be a good (& purposeful) ornament to your office.

Here is a beautiful 1-page document on OutOfMemoryError.

No	OutOfMemoryError	Frequency	Cause	Solution	Comments
1	Java heap space	5 Stars ★★★★★	1. Object could not be allocated in the Java heap. 2. Increase in Traffic volume. 3. Application is unintentionally holding references to objects which prevents the objects from being garbage collected. 4. Application makes excessive use of finalizers. Finalizer objects aren't GCed immediately. Finalizers are executed by a daemon thread that services the finalization queue. Sometimes finalizer thread cannot keep up, with the finalization queue.	1. Increase Heap size '-Xmx'. 2. Fix memory leak in the application	GB –> G, g. MB –> M, m. KB –> K, k.

No	OutOfMemoryError	Frequency	Cause	Solution	Comments
2	GC overhead limit exceeded	5 Stars ★★★★★	1. Java process is spending more than 98% of its time doing garbage collection and recovering less than 2% of the heap and has been doing so far the last 5 (compile time constant) consecutive garbage collections	1. Increase heap size '-Xmx' 2. GC Overhead limit exceeded can be turned off with '-XX:-UseGCOverheadLimit' 3. Fix the memory leak in the application	
3	Requested array size exceeds VM limit	2 Stars ★★	1. Application attempted to allocate an array that is larger than the heap size	1. Increase heap size '-Xmx' 2. Fix bug in application. code attempting to create a huge array	
4	Permgen space	3 stars ★★★	1. Permgen space contains: a. Names,Fields, methods of the classes b. Object arrays and type arrays associated with a class c. Just In Time compiler optimizations When this space runs out of space this error is thrown	1. Increase Permgen size '-XX:MaxPermSize' 2. Application redeployment without restarting can cause this issues. So restart JVM.	
5	Metaspace	3 stars ★★★	1. From Java 8 Permgen replaced by Metaspace. Class metadata is allocated in native memory (referred as metaspace). If metaspace is exhausted then this error is thrown	1. If '-XX:MaxMetaSpaceSize', has been set on the command-line, increase its value. 2. Remove '-XX:MaxMetsSpaceSize' 3. Reducing the size of the Java heap will make more space available for MetaSpace. 4. Allocate more memory to the server 5. Could be bug in application. Fix it.	
6	Unable to create new native thread	5 stars ★★★★★	1. There isn't sufficient memory to create new threads. Threads are created in native memory. It indicates there isn't sufficient native memory space	1. Allocate more memory to the machine 2. Reduce Java Heap Space 3. Fix thread leak in the application. 4. Increase the limits at the OS level. ulimit -a max user processes (-u) 1800 5. Reduce thread stack size with -Xss parameter	
7	Kill process or sacrifice child	1 stars ★	1. Kernel Job – Out of Memory Killer. Will kill processes under extremely low memory conditions	1. Migrate process to different machine. 2. Add more memory to machine	Unlike all other OOM errors, this is not triggered by JVM. But by OS.
8	reason stack_trace_with_native_method	1 stars ★	1. Native method encountered allocation failure 2. a stack trace is printed in which the top frame is a native method	1. Use OS native utilities to diagnose	

28. Detect proactively whether application's memory is under-allocated

When the application's memory is under-allocated, it will result in the following side-effects:

Only when OutOfMemoryError is thrown, most of us start to look at our application's memory settings. This is like only when a patient goes to a critical condition; we begin to give treatment :-).

In this post, let's discuss how to detect whether your application's memory is under-allocated or not in a proactive manner.

Study the Garbage Collection behavior

Studying the garbage collection behavior of the application will clearly indicate whether your application's memory is under-allocated or over-allocated than the actual requirement. You can study the garbage collection behavior of your application by following these two simple steps:

1. To study the Garbage collection behavior, you first need to enable the Garbage collection log on your application. Garbage collection log can be enabled by passing the

JVM arguments mentioned in this post. Enabling garbage collection log doesn't add any observable overhead to your application. Thus you can consider enabling garbage collection log on all your production JVMs.

2. Once garbage collection logs are captured, you can use free GC log analysis tools such as GCeasy, IBM GC Visualizer, Google arbage Cat, HP Jmeter to study the Garbage Collection behaviour.

Let's discuss what metrics and patterns you need to look in the Garbage Collection report to determine whether your application's memory is under-allocated or not.

Healthy Normal application

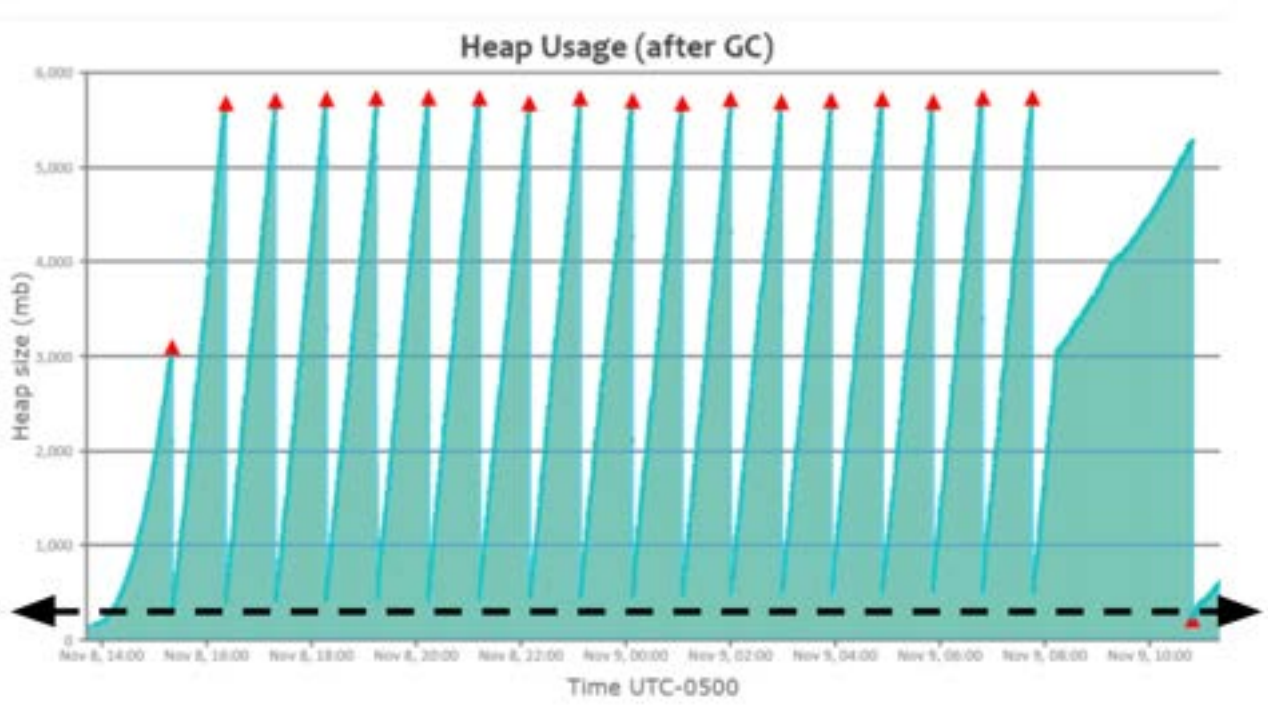


Fig 1: Healthy normal application GC pattern

Above is the heap usage graph generated by the GCeasy tool by parsing the garbage collection log file. As shown in the above graph, you can see a beautiful saw-tooth pattern. Heap usage will keep rising; once a 'Full GC' event is triggered, heap usage will drop all the way to the bottom. It indicates that the application is in a healthy condition.

In Fig 1, You can notice that 'Full GC' event (i.e., red triangle) runs approximately when the heap usage reaches ~5.8GB. When the 'Full GC' event runs, memory utilization drops all the way to the bottom i.e., ~200MB (Please refer to the dotted black arrow line in the graph). It indicates that the application is in a healthy state & not suffering from any sort

of memory problems.

Memory under-allocated application pattern-1

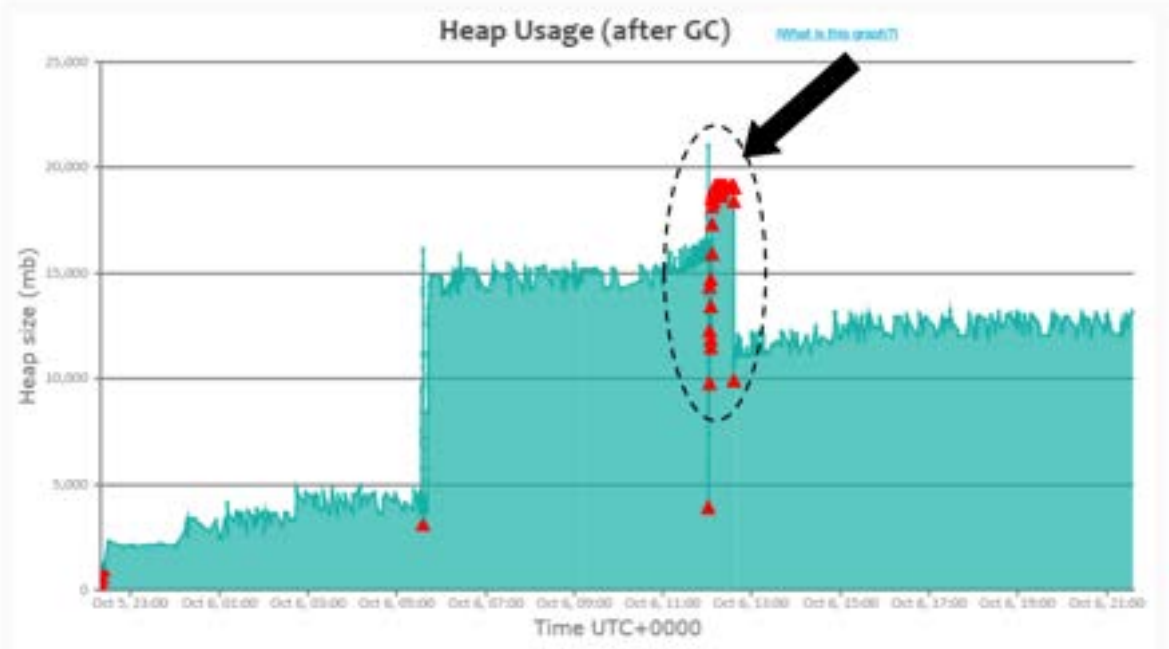


Fig 2: Consecutive Full GC pattern generated

In Fig 2. you can notice that from 12:02pm to 12:30 pm on Oct' 06, 'Full GC's (i.e., red triangle) are consecutively running (Please refer to the black arrow mark); however, heap usage isn't dropping during that time frame. It indicates that the application is creating a lot of objects during that time frame. Application was creating a lot of new objects because of the spike in traffic volume during that time frame. Since objects are created at a rapid phase, GC events also started to run consecutively.

Whenever a GC event runs, it has two side effects:

a. CPU consumption will go high (since GC does an enormous amount of computation).

b. Entire application will be paused; no customers will get response.

Thus, during this time frame, 12:02pm to 12:30pm on Oct' 06, application's CPU consumption would have been skyrocketing and customers wouldn't be getting back any response. Here is the real-world GC log analysis report which was suffering from this 'Consecutive Full GC' problem. When this kind of pattern surfaces, it's a clear indication that your application **needs more memory than what you have allocated**.

When this problematic pattern surfaces, you can also follow one of the approaches outlined in this post to resolve the problem.

Memory under-allocated application pattern-2

If you notice in Fig 2, after 12:30pm on Oct' 6, application recovered and started to function normally. It's because traffic volume died down after 12:30pm. Once traffic volume died down, application's object creation rate went down. Since the object creation rate went down, Full GCs also stopped running consecutively. However, in certain applications, you might see the GC pattern to appear like this as well:

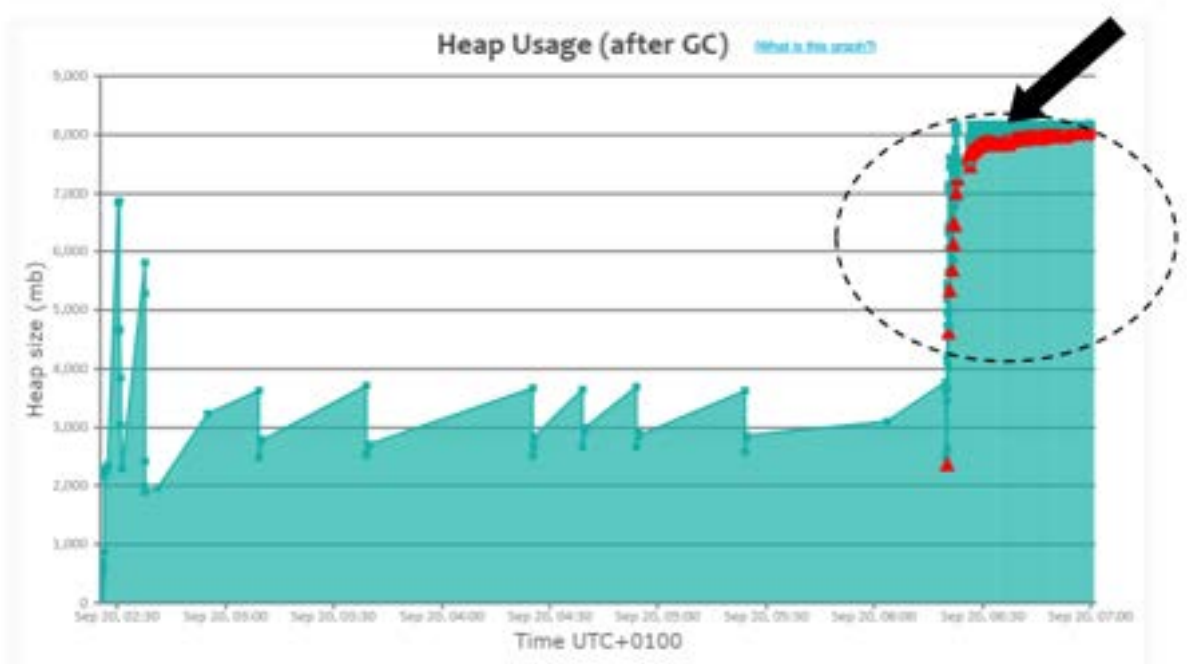


Fig 3: Memory leak GC pattern

Please refer to the black arrow in Fig 3. You can notice that 'Full GC' (i.e., red triangle) events are continuously running.

Here is the real-world GC log analysis report, which depicts this pattern. This pattern is similar to the previous pattern as highlighted in Fig 2, with one sharp difference. In the previous pattern, application recovered from consecutive Full GC runs and returned to a normal functioning state once traffic volume died down. However, in Fig 3, consecutive Full GCs never stopped running until the application got restarted. When you see Fig 3 type of pattern, there could be two possibilities:

a. Application needs more memory (due to spike in traffic volume)

b. Application is suffering from a memory leak

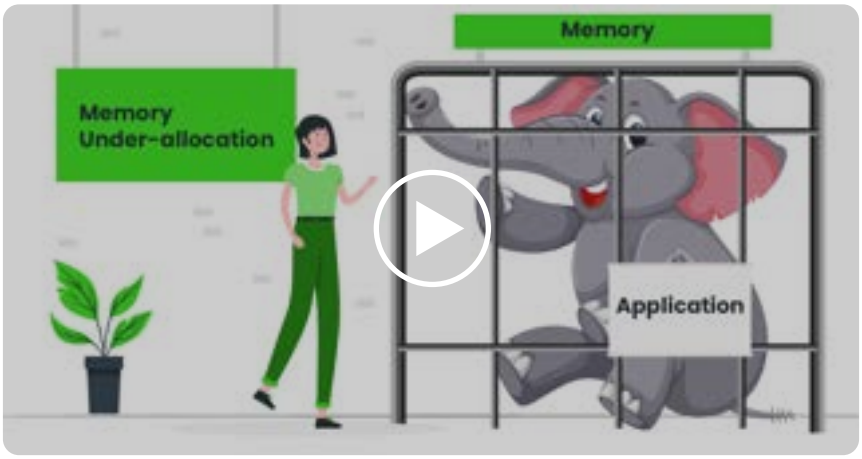
Now the question is: 'How to isolate whether it's #a or #b causing this problematic pattern?'. Try removing your JVM instance from the load balancer and stop the traffic. If the problem is triggered because of the spike in traffic volume, Full GCs will stop running, and heap usage will drop back to the normal level. However, Full GCs will continue to run consecutively if it's a memory leak, and heap usage will not drop back to the normal level.

Note: If the application is suffering from memory leak, you can use tools like **yCrash**, **HeapHero**, **Eclipse MAT** to diagnose memory leak. Here is a more detailed post on **how to diagnose Memory leak**.

Conclusion

Thus, by performing Garbage collection log analysis as outlined in this post, you would be able to determine whether your application's memory is under-allocated or not in a proactive manner.

Video



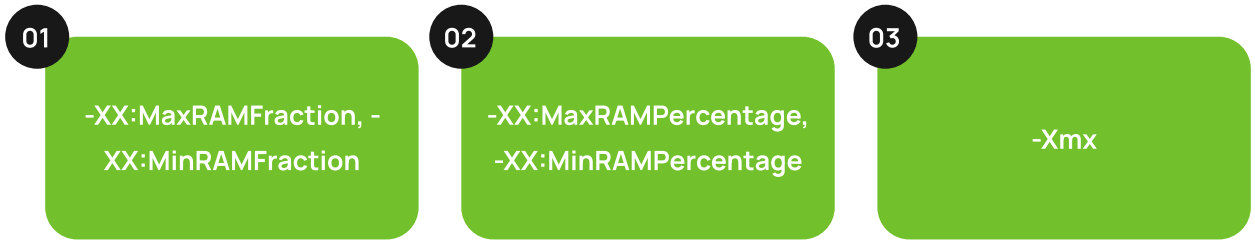
Watch video

https://www.youtube.com/watch?v=btnI7qRTb3U

29. Best practices: Java memory arguments for Containers

When you are running your Java application in physical servers, you would have been using '-Xmx' JVM argument to specify the Java heap size. If you are porting your application to Containers, you might be wondering how to configure Java heap size in the container's world? Are there any best practices? In this article, we will discuss the possible JVM arguments that can be used to specify the Java heap size and the best option to choose.

There are 3 different options to specify the maximum Java heap size in containers. They are:



Let us discuss these JVM arguments, their merits, and shortcomings.

1. -XX:MaxRAMFraction, -XX:MinRAMFraction

Supported Version

'-XX:MaxRAMFraction', '-XX:MinRAMFraction' JVM arguments are supported from only Java 8 update 131 to Java 8 update 190. So, if you are using any other version of JDK, you cannot use this option.

How does it work?

Say you have allocated 1 GB of memory to your container, then if you configure -XX:MaxRAMFraction=2, then approximately ~512MB (i.e. 1/2 of 1GB) will be allocated to your Java heap size.

If you are going to use '-XX:MaxRAMFraction' JVM argument, make sure to pass these two additional JVM arguments as well '-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap'. Only if you pass these two JVM arguments then JVM will derive the heap size value from the container's memory size, otherwise, it will derive the heap size value from the underlying host's memory size.

```
#  docker run -m 1GB openjdk:8u131 java -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -XX:MaxRAMFraction=2 -XshowSettings:vm -version VM settings:
   Max. Heap Size (Estimated): 494.94M
```

Here you can see when the docker container's memory is set to '1GB' (i.e., -m 1GB) and '-XX:MaxRAMFraction=2. Based on this setting, JVM is allocating Max heap size to be 494.9MB (approximately half the size of 1GB).

Note: Both '-XX:MaxRAMFraction' and '-XX:MinRAMFraction' are used to determine the maximum Java heap size. JDK development team could have given a better name than '-XX:MinRAMFraction'. This name makes us think, '-XX:MinRAMFraction' argument is used to configure minimum heap size. But it's not true. To learn more about their difference read this article.

What are its limitations?

Here are the drawbacks to this approach.

a. Say if you want to configure 40% of the docker's memory size, then we must set '-XX:MaxRAMFraction=2.5'. When you pass 2.5 as the value, JVM will not start. It is because '-XX:MaxRAMFraction' can take only integer values and not decimal values. See the below example where JVM is failing to start.

```
# docker run -m 1GB openjdk:8u131 java -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -XX:MaxRAMFraction=2.5 -XshowSettings:vm -version VM Improperly specified VM option 'MaxRAMFraction=2.5' Error: Could not create the Java Virtual Machine. Error: A fatal exception has occurred. Program will exit.
```

b. In this option your Java application's heap size will be derived from the container's memory size (because it is fraction basis). Say suppose if your application requires 1GB heap size for optimal performance and if container is configured to run with memory size that is less than 1GB then your application will still run but it will suffer from poor performance characteristics.

c. This argument has been deprecated in modern Java versions. It is only supported from Java 8 update 131 to Java 8 update 190.

2. -XX:MaxRAMPercentage, -XX: MinRAMPercentage

Supported Version

'-XX:MaxRAMPercentage', '-XX: MinRAMPercentage' JVM arguments are supported from Java 8 update 191 and above. So, if you are running on older JDK versions, you can't use this JVM argument.

How it works?

Say you have allocated 1 GB of memory to your container, then if you configure -XX:MaxRAMPercentage=50, then approximately 512MB (i.e. 1/2 of 1GB) will be allocated to your Java heap size.

```
# docker run -m 1GB openjdk:10 java -XX:MaxRAMPercentage=50 -XshowSettings:vm -version
VM settings:
   Max. Heap Size (Estimated): 494.94M
   Using VM: OpenJDK 64-Bit Server VM
```

Here you can see when docker container's memory is set to '-m 1GB' and '-XX:MaxRAMPercentage=50'. Based on this setting, JVM is allocating Max heap size to be 494.9MB (approximately half of 1GB).

Note: Both '-XX:MaxRAMPercentage' and '-XX:MinRAMPercentage' are used to determine the maximum Java heap size. JDK development team could have given a better name than '-XX:MinRAMPercentage'. This name makes us think, '-XX:MinRAMPercentage' argument is used to configure minimum heap size. But it's not true. To learn more about their difference read this article.

Note: In several articles in internet, it's mentioned that you need to pass '-XX:+UseContainerSupport' JVM argument when you are passing '-XX:MaxRAMPercentage', '-XX:InitialRAMPercentage', '-XX:MinRAMPercentage'. Actually, that's not true. '-XX:+UseContainerSupport' is passed by default argument in the JVM. So, you don't need to explicitly configure it.

What are the limitations?

Here are the limitations to this approach.

a. This argument is not supported in the older versions of Java. It is only supported from Java 8 update 191.

b. In this option your Java application's heap size will be derived by the container's memory size (because it is percentage basis). Say suppose if your application requires 1GB heap size for optimal performance and if container is configured to run with memory size that is less than 1GB then your application will still run but it will suffer from poor performance characteristics.

3. -Xmx

Supported Version

'-Xmx' is supported in all versions of Java

How it works?

Using '-Xmx' JVM argument you specify fine grained specific size such as 512MB, 1024MB.

Here you can see the -Xmx to supported in non-container (traditional Physical server world):

```
# java -Xmx512m -XshowSettings:vm -version
VM settings:
   Max. Heap Size: 512.00M
   Ergonomics Machine Class: client
   Using VM: OpenJDK 64-Bit Server VM
```

Here you can see the -Xmx to supported in java 8 update 131 version in the container world:

```
# docker run -m 1GB openjdk:8u131 java -Xmx512m -XshowSettings:vm -version VM
VM settings:
  Max. Heap Size: 512.00M
  Ergonomics Machine Class: client
  Using VM: OpenJDK 64-Bit Server VM
```

Here you can see the -Xmx to supported in java 10 version in the container world:

```
# docker run -m 1GB openjdk:10 java -Xmx512m -XshowSettings:vm -version
VM settings:
  Max. Heap Size: 512.00M
  Using VM: OpenJDK 64-Bit Server VM
```

What are the limitations?

a. If you are going to allocate '-Xmx' more than the container's memory size then your application will experience 'java.lang.OutOfMemoryError: kill process or sacrifice child'

Best Practices

1. Irrespective of what option you use for configuring heap size (i.e., -XX:MaxRAMFraction, -XX:MaxRAMPercentage, -Xmx), always make sure you allocate at least 25% more memory to your container (i.e. '-m') than your heap size value. Say you have configured -Xmx value to be 2GB, then configure container's memory size at least to be 2.5GB. Do this even if your Java application is the only process going to run on the container. Because lot of engineer thinks Java application will not consume more than -Xmx value. That is not true. Besides heap space your application needs space for Java threads, Garbage collection, metaspace, native memory, socket buffers. All these components require additional memory outside allocated heap size. Besides that, other small processes (like APM agents, splunk scripts, etc.) will also require memory. To learn more about them, watch this quick 'JVM Memory' video clip.
2. If you are running *only your Java application* within the container, then set initial heap size (i.e., using either one of '-XX:InitialRAMFraction', '-XX:InitialRAMPercentage', -Xms) to the same size as max heap size. Setting initial heap size and max heap has certain advantages. One of them is: you will incur lower Garbage Collection pause times. Because whenever heap size grows from the initial allocated size, it will pause the JVM. It can be circumvented when you set initial, and max heap sizes to be the same. Besides that, if you have under allocated container's memory size, then JVM will not even start (which is better than experiencing OutOfMemoryError when

transactions are in flight).

3. In my personnel opinion, I prefer to use -Xmx option than -XX:MaxRAMFraction, -XX:MaxRAMPercentage options to specify Java Heap size in the container world, for the following reasons:
    - I do not want the container's size to determine my java application's heap size. Your body's size should decide whether you are going to wear a 'small' or 'medium' or 'large' size T-shirts, not other way around. You do not want to fit-in a 6-foot man with a 'small' size T-shirt. Memory size plays a THE KEY ROLE in deciding your application's performance. It influences your garbage collection behavior and performance characteristics. You do not want that factor to be decided by your container's memory setting.
    - Using '-Xmx' I can set fine-grained/precision values like 512MB, 256MB.
    - -Xmx is supported on all java versions.
4. You should study whether your application's garbage collection, performance characteristics are impacted because of the new settings in the containers. To study the garbage collection behaviour, you can use free tools like **GCeasy**, **IBM GC & Memory Visualizer**, **HP jmeter**.

30. Large or Small memory size for my application?

Should I be running my application with few instances (i.e. machines) with large memory size or a lot of instances with small memory size? Which strategy is optimal? This question might be confronted often. After building applications for 2 decades, after building JVM performance engineering/troubleshooting tools (GCeasy, FastThread, HeapHero), I still don't know the right answer to this question. At the same time, I believe there is no binary answer to this question as well. In this article, I would like to share my observations and experiences on this topic.

Two multi-billion dollars enterprises story

Since our JVM performance engineering/troubleshooting tools has been widely used in major enterprises, I had an opportunity to see world-class enterprise applications implementations in action. Recently I had the chance to see two hyper-growth technology companies (If I say their name everyone reading this article will know them). Both companies are headquartered in Silicon Valley. Their business is technology, so they know what they are doing when it comes to engineering. They are wall-street darlings, enjoying great valuations. Their market cap is in the magnitude of several billions of dollars. They are the poster child of modern thriving enterprises. For our conversation let's call these two enterprises as company-A and company-B.

It **immensely surprises** me to see how both enterprises has adopted *two extremes* when it comes to memory size. Company-A has set its heap size (i.e. -Xmx) to be 250gb, whereas company-B has set its heap size to be 2gb. i.e. company-A's heap size is 125 times larger than Company-B's heap size. Both enterprises are confident about their memory size settings. As they say: 'Proof is in the pudding', both enterprises are scaling and handling billions of business-critical transactions.

This is a great experience to see both companies who are into the same business, having more or less same revenue/same market cap, located in the same geographical region, at same point in time adopting two extremes when it comes to memory size. Given this real-life experience, what is the right answer? Large size or small size memory? My conclusion is: You can succeed with either strategy if you have a good team in place.

Large memory size can be expensive

Large memory size with few instances (i.e. machines) tends to be expensive than with small memory size, a greater number of instances. Here is simple math, based on the cost of an AWS EC2 instances in US East (N. Virginia) region:

m4.16xlarge – 256GB RAM – Linux on-demand instance cost: $3.2/hour

T3a small – 2GB RAM – Linux on-demand instance cost: $0.0188/hour

So, to have capacity of 256GB RAM, we would have to get 128 'T3a small' instance (i.e. 128 instances x 2GB = 256GB).

128 x T3a small – 2GB RAM – Linux on-demand instance cost: $2.4064/hour (i.e. 128 x $0.0188/hour)

It means large memory size with few instances costs $0.793/hour (i.e. $3.2 – $2.4064) more than small memory size with a lot of instances. In other words, 'large memory size with few instances strategy is **33% more expensive**.

Of course, another counter-argument can be made is: you might need fewer engineers, less electricity, less real estate if you have a smaller number of machines. Patching, upgrading servers might be easier to do as well.

Business Demands

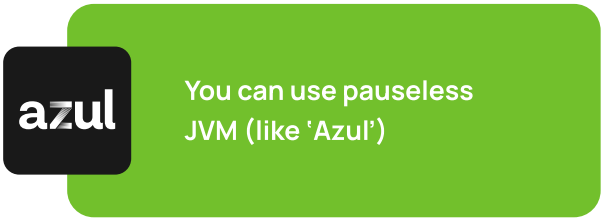
In some cases, the nature of your business itself dictates the memory size of your application. Here is a real-life incident that we faced: When we built HeapHero (Heap Dump analysis tool), our tool's memory size had to be larger than heap dump file it parses. Say suppose heap dump file size is 100gb, then HeapHero tool's memory size must be more than 100gb. There is no choice.

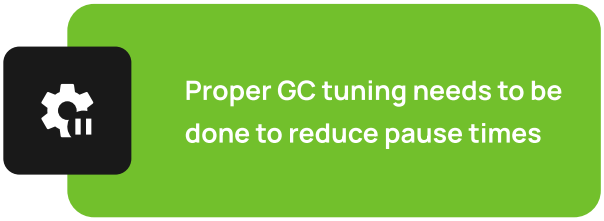
Say suppose you are caching large amount (say 200gb) of data for maximizing your application's performance, then your heap size must be more than 200gb. You will not

have a choice. Thus, in some cases, the business requirement will dictate your memory size.

Performance & Troubleshooting

If your memory size is large, then typically Garbage Collection pause times will also be high. Garbage collection is a process that runs in your application to clean-up unreferenced objects in memory. If your memory size is large, then the amount of garbage in the memory will also be large. Thus, amount of time taken to clean up garbage will also be high. When garbage collection runs, it pauses your application. But there are solutions to this problem:





Similarly, if you need to troubleshoot any memory problem, you will have to capture heap dumps from the application. A heap dump is basically a file which contains all information about your application's memory like what objects were present, what are their references, how much memory each object occupies, …. Heap dumps of large memory size application will also tend to be very large. Analyzing large size heap dumps are difficult as well. Even world's best heap dump tools like Eclipse MAT, HeapHero have challenges in parsing heap dumps that are more than 100gb. Reproducing these problems in test lab, storing these heap dump files, sharing these heap dump files are all challenges.

Emotions comes first, Rationale next

After reading books like 'How we decide' written by Jonah Lehrer – I am fairly convinced that your prior experience, emotions play a key role in deciding your application's memory size. I used to work for a major financial institution. Chief architect of this financial institution was suggesting us to run our JVMs with very large memory size, rationale he gave was: ***"We used to run mainframes with very large memory size".***

Conclusion

If you are working for very large corporations, then there is a 99.99% chance that you may not have to say on what should be the memory size for your application. Because that decision has already been made by elites/demi-gods who are sitting on the ivory tower. It might be to be hard to reverse or change that decision.

But if you have choice or option to make that decision, your decision for memory size can be most likely be influenced by your prior experience and emotions :). But either way, you can't go wrong (i.e. going with few instances with large memory size or lot of instances with small memory size), provided you have the right team in place.

Video

Watch video

https://www.youtube.com/watch?v=sOyWpGs6QtI&t=7s

31. OutOfMemoryError related JVM arguments

JVM has provided helpful arguments to deal with OutOfMemoryError. In this article, we would like to highlight those JVM arguments. It might come handy for you when you are troubleshooting OutOfMemoryError. Those JVM arguments are:

1. -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath
2. -XX:OnOutOfMemoryError
3. -XX:+ExitOnOutOfMemoryError
4. -XX:+CrashOnOutOfMemoryError

Let's discuss these JVM arguments in detail in this article.

1. -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath

Heap dump is basically a snapshot of memory. It contains details about objects that present in memory, actual data that is present within those objects, references originating of those objects. Heap dump is a vital artifact to troubleshoot memory problems.

In order to diagnose OutOfMemoryError or any memory related problem, one would have to capture heap dump right at the moment or few moments before the application starts to experience OutOfMemoryError. It's hard to do capture heap dump at the right moment manually because we will not know when OutOfMemoryError is going to be thrown. However, capturing heap dumps can be automated by passing following JVM arguments when you launch the application in the command line:

```
-XX:+HeapDumpOnOutOfMemoryError and -XX:HeapDumpPath={HEAP-DUMP-FILE-PATH}
```

Example

```
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/crashes/my-heap-dump.hprof
```

In '-XX:HeapDumpPath' you need to specify the filepath where heap dump should be stored.

When you pass these two JVM arguments, heap dumps will be automatically captured and written to the specified file path, when OutOfMemoryError is thrown.

Once heap dumps are captured, you can use tools like HeapHero, Eclipse MAT to analyze heap dumps.

2. -XX:OnOutOfMemoryError

You can configure JVM to invoke any script when OutOfMemoryError is thrown. Most of the time, OutOfMemoryError doesn't crash the application. However, it's better to restart the application, once OutOfMemoryError happens. Because OutOfMemoryError can potentially leave application in an unstable state. Requests served from an unstable application instance can lead to an erroneous result.

Example

```
-XX:OnOutOfMemoryError=/scripts/restart-myapp.sh
```

When you pass this argument, JVM will invoke "/scripts/restart-myapp.sh" script whenever OutOfMemoryError is thrown. In this script, you can write code to restart your application gracefully.

3. -XX:+CrashOnOutOfMemoryError

When you pass this argument JVM will exit right when it OutOfMemoryError is thrown. Besides exiting, JVM produces text and binary crash files (if core files are enabled). But personally, I wouldn't prefer configuring this argument, because we should always aim to achieve a graceful exit. Abrupt exit can/will jeopardize transactions that are in motion.

I ran an application which generates OutOfMemoryError with this '-XX:+CrashOnOutOfMemoryError' argument. I could see JVM exiting immediately when OutOfMemoryError was thrown. Below was the message in the standard output stream:

```
Aborting due to java.lang.OutOfMemoryError: GC overhead limit exceeded
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  Internal Error (debug.cpp:308), pid=26064, tid=0x0000000000004f4c
#  fatal error: OutOfMemory encountered: GC overhead limit exceeded
#
# JRE version: Java(TM) SE Runtime Environment (8.0_181-b13) (build 1.8.0_181-b13)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (25.181-b13 mixed mode windows-amd64 compressed oops)
# Failed to write core dump. Minidumps are not enabled by default on client versions of Windows
#
# An error report file with more information is saved as:
# C:\workspace\tier1app-svn\trunk\buggyapp\hs_err_pid26064.log
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.java.com/bugreport/crash.jsp
#
```

From the message, you could see hs_err_pid file to be generated in 'C:\workspace\tier1app-svn\trunk\buggyapp\hs_err_pid26064.log'. hs_err_pid file contains information about the crash. You can use tools like fastThread to analyze hs_err_pid file. But most of the time information present in hs_err_pid is very basic. It's not sufficient enough to troubleshoot OutOfMemoryError.

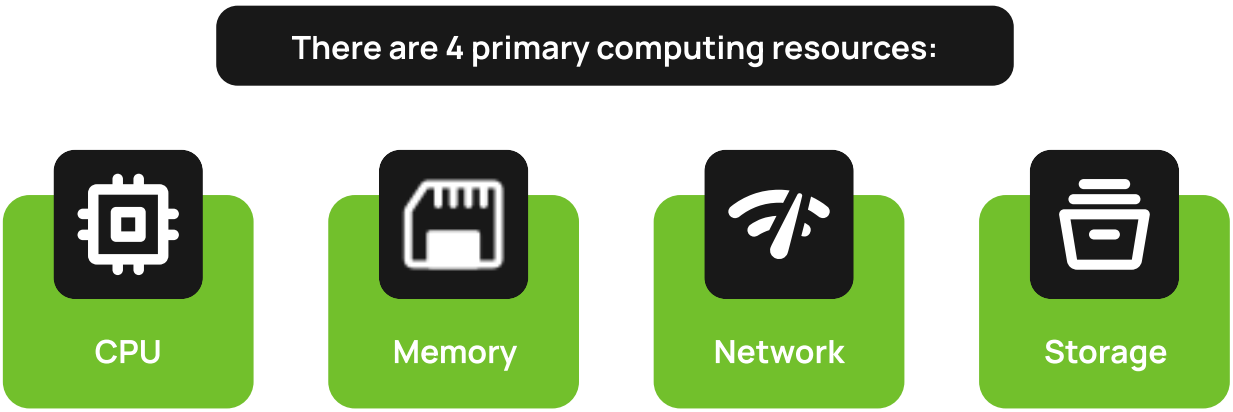
4. -XX:+ExitOnOutOfMemoryError

When you pass this argument, JVM will exit right when OutOfMemoryError is thrown. You may pass this argument if you would like to terminate the application. But personally, I wouldn't prefer configuring this argument, because we should always aim to achieve a graceful exit. Abrupt exit can/will jeopardize transactions that are in motion.

I ran the same memory leak program with this '-XX:+ExitOnOutOfMemoryError' JVM argument. Unlike '-XX:+CrashOnOutOfMemoryError', this JVM argument did not generate any text/binary file. JVM just exited.

32. How much memory is my application wasting?

In early 1970s 1 MB was costing 1 million $. Now 1 mb is costing fraction of that cost. There is no comparison. This is one of the reasons why engineers and enterprises don't worry about memory any more. 1 million $ in 1970s might be equivalent of several millions of dollars' worth today. Thus, back in the day's memory was treated so preciously. This preciousness has been vividly described in the book 'Idea Man' – autobiography of Paul Allen (Microsoft Co-founder). Paul Allen talks about the challenge he and Bill Gates faced in writing BASIC programming language (Microsoft's very first product) under 4 KB.



Your application might be running on tens, thousands of application server instances. In above mentioned 4 computing resources, which resource does your application instance saturates first? Pause for a moment here, before reading further. Give a thought to figure out which resource gets saturated first.

For most applications it is *memory*. CPU is always at 30 – 60%. There is always abundant storage. It's hard to saturate network (unless your application is streaming video content). Thus, for most applications it's the memory that is getting saturated first. Even though CPU, storage, network is underutilized, just because memory is getting saturated, you end up provisioning more and more application server instances increasing the

computing cost of organizations to millions/billions dollars.

On the other hand, without exception modern applications wastes anywhere from 30 – 90% of memory due to inefficient programming practices. Below are 9 different practices that is followed in the industry, that is causing memory wastage:

1. Duplicate Strings
2. Inefficient Collections
3. Duplicate Objects
4. Duplicate arrays
5. Inefficient arrays
6. Objects waiting for finalization
7. Boxed numbers
8. Overhead caused by Object Headers
9. Wrong memory size settings

If you can eliminate this 30 – 90% of memory wastage, it will bring 2 major advantages to your enterprise:

a. **Reduce computing cost:**
   As memory is the first resource to get saturated, if you can reduce memory consumption, you would be able to run your application on a smaller number of server instances. You might be able to reduce 30 – 40% of servers. It means your management can reduce 30 – 40% of the datacenter (or cloud hosting provider) cost, maintenance and support cost. It can account for several millions/billions of dollars in cost savings.

b. **Better customer experience:**
   If you are reducing number of objects that you are creating to service incoming request, your response time will get lot better. Since less objects are created, less CPU cycles will be spent in creating and garbage collecting them. Reduction in response time will give a lot better customer experience.

How much memory is my application wasting?

So now let's get back to original question of this article: 'How much memory is my application wasting?'. Sad story is: there aren't that many tools in the industry that can give you this answer. There are tools which can answer the question: 'How much memory is my application using?' like TOP, Application Performance Monitoring (APM) tools,

Nagios… 'Using' is different from 'Wasting'. But we will help you answer this question. You just need to follow these 2 steps to get to this answer:

Step 1: Capture Heap Dumps

In order to analyze, how memory is wasted, you first need to capture the heap dump. Heap Dump is basically a snapshot of your application's memory. It has information what all are the objects that are present in memory, who is referencing it.

There are 7 options to capture heap dumps from your Java application. https://blog.heaphero.io/2017/10/13/how-to-capture-java-heap-dumps-7-options/

There are 3 options to capture heap dumps from android application. https://blog.heaphero.io/2018/06/04/how-to-capture-heap-dump-from-android-app-3-options/

You can use the option that is more suited for you.

"It's recommended to capture heap dump when your application is taking traffic. When application is idle, new objects will not be created, thus you wouldn't be able to see how much memory is actually wasted".

Step 2: Analyze using HeapHero tool

Once you have captured heap dumps, you can upload the captured heap dumps to the free online heap dump analysis tool – HeapHero.

"Depending on your application, Heap dumps can contain PII data and other sensitive data. In such circumstance you can register here http://heaphero.io/heap-trial-registration.jsp to download and install the HeapHero tool locally in your machine".

Tool will give high level summary of total amount of memory that is wasted, actual data that is causing memory wastage, lines of code that is triggering this memory wastage and recommends solutions to fix the problem.

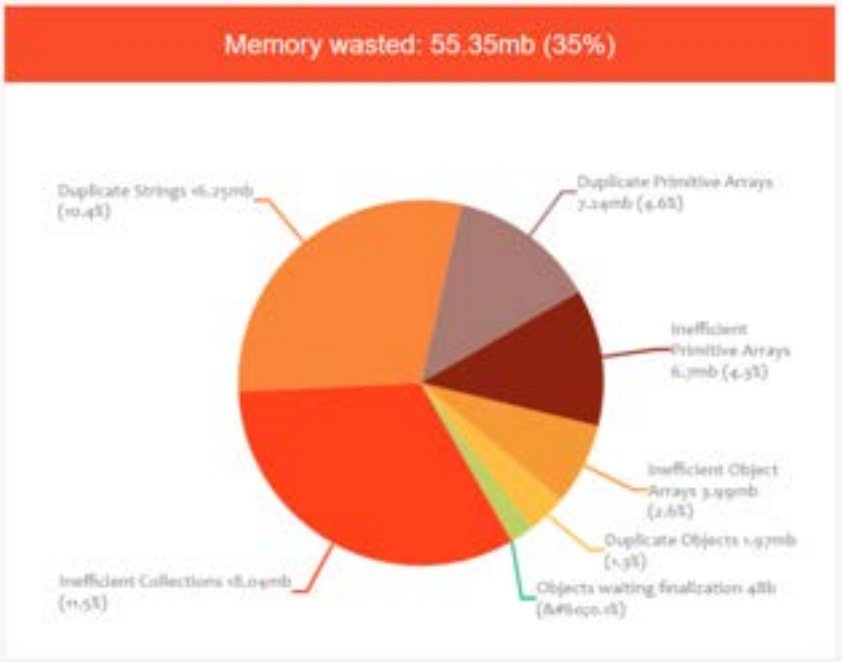


Fig 1: Summary showing how much memory is wasted in total

HeapHero prints a pie graph that summarizes how much memory is wasted due to each inefficient programming practice. Apparently, this application is wasting 35% of its memory. Top two reasons for this wastage are:

a. Inefficient collections (i.e. Data Structures) is causing 11.5% of memory wastage.

b. Duplication of strings is causing 10.4% of memory wastage.

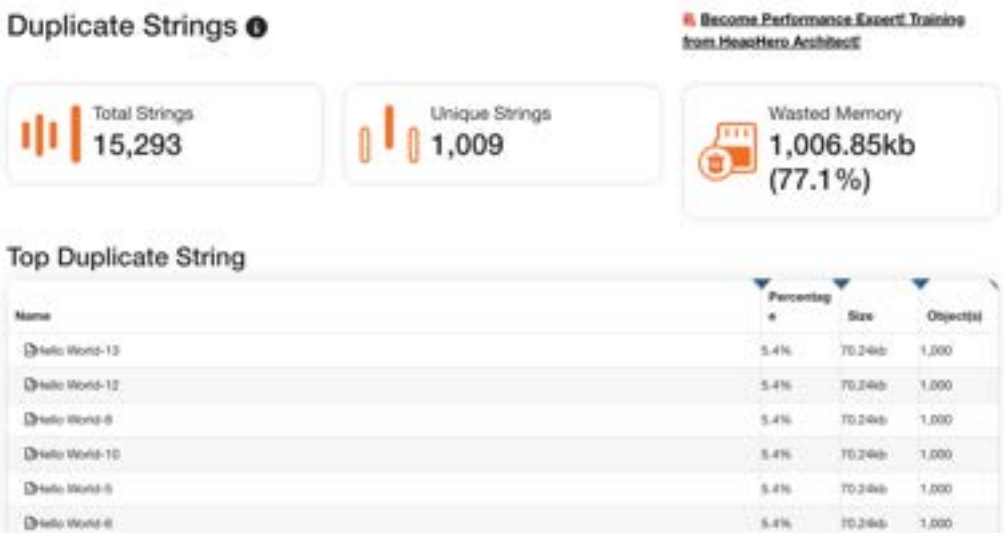


Fig 1.2: Memory wasted due to duplicate Strings

In this application there are 343,661 instances of string objects. Out of it, only 144,520 of them are unique. Apparently, there are 6,147 instances of “webservices.sabre.com” string objects. Why there as to be so many duplicates? Why can’t be cleaned and free-up

memory?

Fig 1.3: Memory wasted due to inefficient collections

In this application 34% of HashMap contains no elements. This brings question why so many HashMaps are created with without any elements. When you create a HashMap by default it creates 16 elements. Space allocated for those 16 elements gets wasted, when you don't insert any elements in to it. Even more interesting observation is 97% of Hashtable doesn't contain any elements.

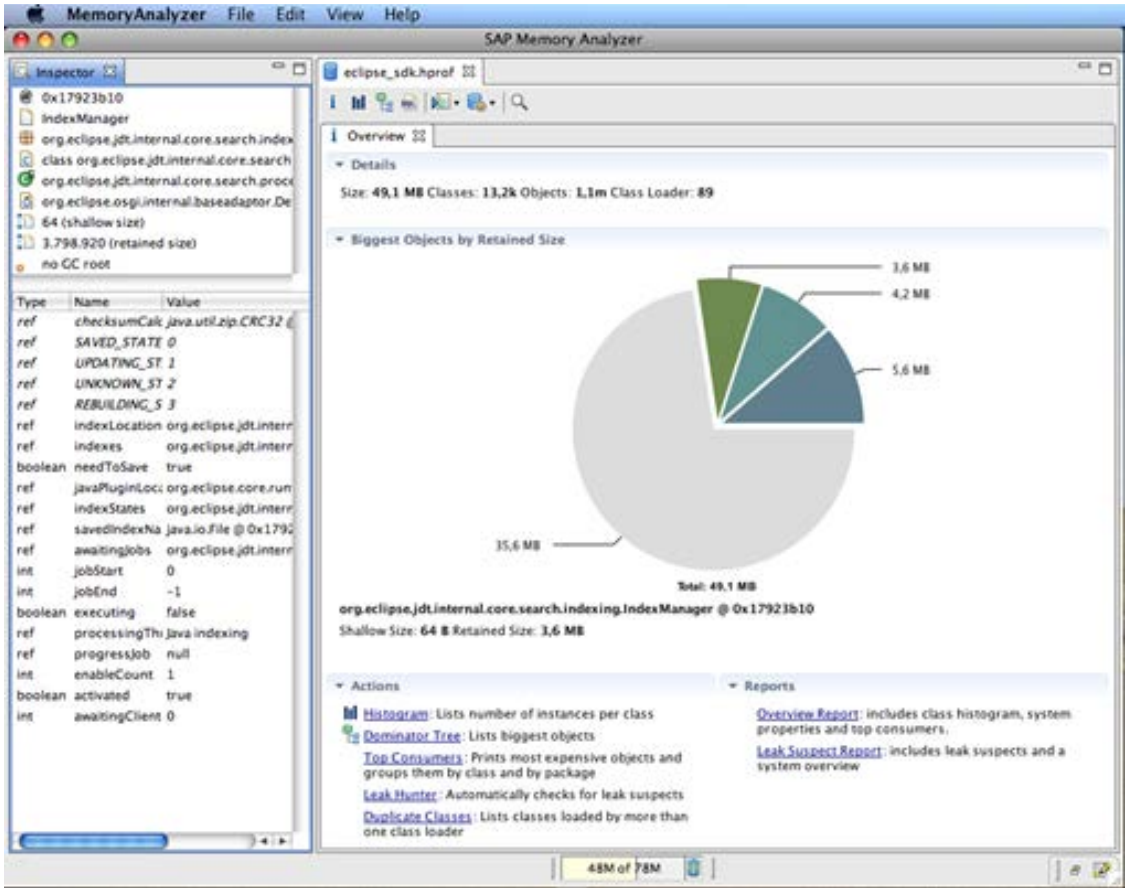
Fig 1.4: Memory wasted due to inefficient collections

Tool not only shows the objects that are wasting memory, but it also shows the code path that is causing triggering the memory wastage. This will facilitate the engineers to institute right fix in the right part of the application.

Happy hacking, happy coding!

33. Eclipse MAT – Tidbits

Eclipse MAT is a great JVM Memory Analysis tool. Here are few tidbits to use it effectively.



1. Use stand-alone version

Two versions of Eclipse MAT is available:

1. Stand-alone

2. Eclipse Plugin

Based on my personal experience, stand-alone version seems to works better and faster then plugin version. So I would highly recommend installing Stand-alone version.

2. Eclipse MAT – heap size

If you are analyzing a heap dump of size, say 2 GB, allocate at least 1 GB additional space for Eclipse MAT. If you can allocate more heap space, then it's more the merrier. You can allocate additional heap space for Eclipse MAT tool, by editing MemoryAnalyzer.ini file. This file is located in the same folder where MemoryAnalyzer.exe is present. To the MemoryAnalyzer.ini you will add -Xmx3g at the bottom.

Example

```
-startup
plugins/org.eclipse.equinox.launcher_1.3.0.v20140415-2008.jar
–launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.200.v20140603-1326
-vmargs
-Xmx3g
```

3. Enable 'keep unreachable objects'

From it's reporting Eclipse MAT removes the object which it thinks as 'unreachable.' As 'unreachable' objects are eligible for garbage collection, MAT doesn't display them in the report. Eclipse MAT classifies Local variables in a method as 'unreachable objects'. Once thread exits the method, objects in local variables will be eligible for garbage collection.

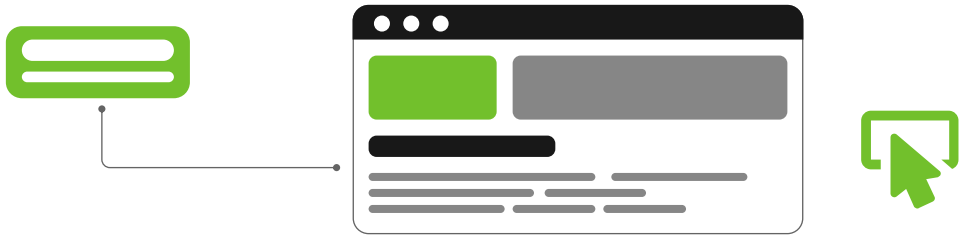
However, there are several cases where a thread will go into a 'BLOCKED' or prolonged 'WAITING', 'TIMED_WAITING' state. In such circumstances local variables will be still alive in memory, occupying space. Since Eclipse MAT default settings don't show the unreachable objects, you will not get visibility into these objects. You can change the default settings in Eclipse MAT 1.4.0 version by:

1. Go to Window > Preferences …

2. Click on 'Memory Analyzer'

3. Select 'Keep unreachable objects'

4. Click on 'OK' button



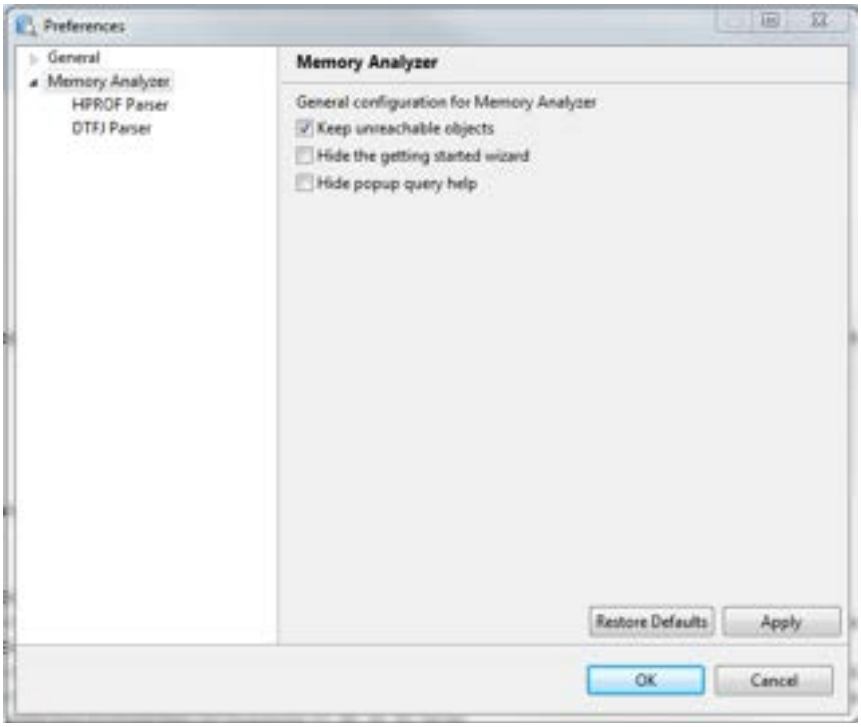


Fig 1: Eclipse MAT 1.4.0, showing how to enable 'Keep unreachable objects'

4. Smart Data Settings

Eclipse MAT by default displays data in bytes. It's difficult to read large object sizes in bytes and digest it. Example Eclipse MAT prints object size like this: "193,006,368". It's much easier if this data can be displayed in KB, MB, GB i.e. "184.07 MB".

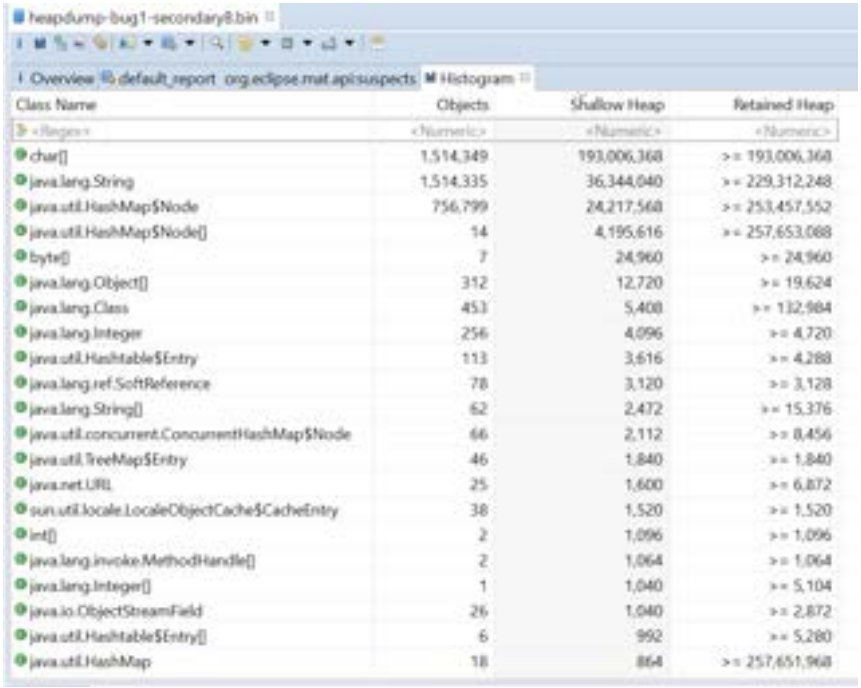


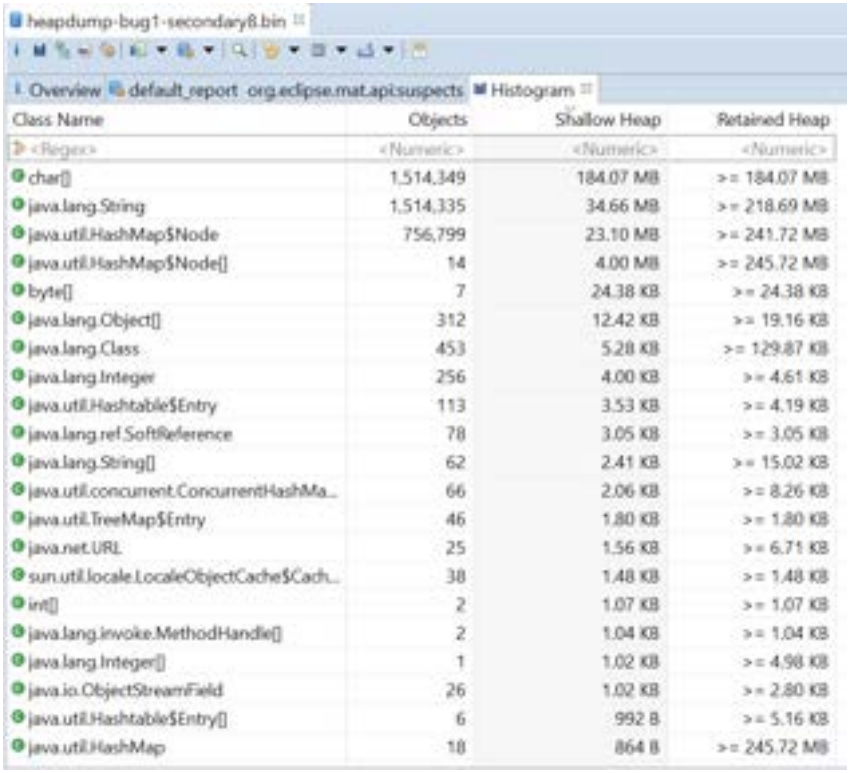
Fig 2: Eclipse MAT default option showing object size in bytes

Eclipse MAT provides an option to display object size in KB, MB, GB based on their appropriate size. It can be enabled by following below steps:

1. Go to Window > Preferences …
2. Click on 'Memory Analyzer'
3. In the 'Bytes Display' section select 'Smart: If the value is a gigabyte or …'
4. Click on 'OK' button

Fig 2.1: Settings to enable 'smart' data display

Once this setting change is made, all data will appear in much more readable KB, MB, GB format, as shown in below figure.



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	1,514,349	184.07 MB	>= 184.07 MB
java.lang.String	1,514,335	34.66 MB	>= 218.69 MB
java.util.HashMap$Node	756,799	23.10 MB	>= 241.72 MB
java.util.HashMap$Node[]	14	4.00 MB	>= 245.72 MB
byte[]	7	24.38 KB	>= 24.38 KB
java.lang.Object[]	312	12.42 KB	>= 19.16 KB
java.lang.Class	453	5.28 KB	>= 129.87 KB
java.lang.Integer	256	4.00 KB	>= 4.61 KB
java.util.Hashtable$Entry	113	3.53 KB	>= 4.19 KB
java.lang.ref.SoftReference	78	3.05 KB	>= 3.05 KB
java.lang.String[]	62	2.41 KB	>= 15.02 KB
java.util.concurrent.ConcurrentHashMa...	66	2.06 KB	>= 8.26 KB
java.util.TreeMap$Entry	46	1.80 KB	>= 1.80 KB
java.net.URL	25	1.56 KB	>= 6.71 KB
sun.util.locale.LocaleObjectCache$Cach...	38	1.48 KB	>= 1.48 KB
int[]	2	1.07 KB	>= 1.07 KB
java.lang.invoke.MethodHandle[]	2	1.04 KB	>= 1.04 KB
java.lang.Integer[]	1	1.02 KB	>= 4.98 KB
java.io.ObjectStreamField	26	1.02 KB	>= 2.80 KB
java.util.Hashtable$Entry[]	6	992 B	>= 5.16 KB
java.util.HashMap	18	864 B	>= 245.72 MB

Fig 2.2: Eclipse MAT displaying of object size in KB, MB, GB after enabling 'smart' settings.

34. Shallow Heap, Retained Heap

Eclipse MAT (Memory Analyzer Tool) is a powerful tool to analyze heap dumps. It comes quite handy when you are trying to debug memory related problems. In Eclipse MAT two types of object sizes are reported:

1. Shallow Heap
2. Retained Heap

In this article lets study the difference between them. Let's study how are they calculated?

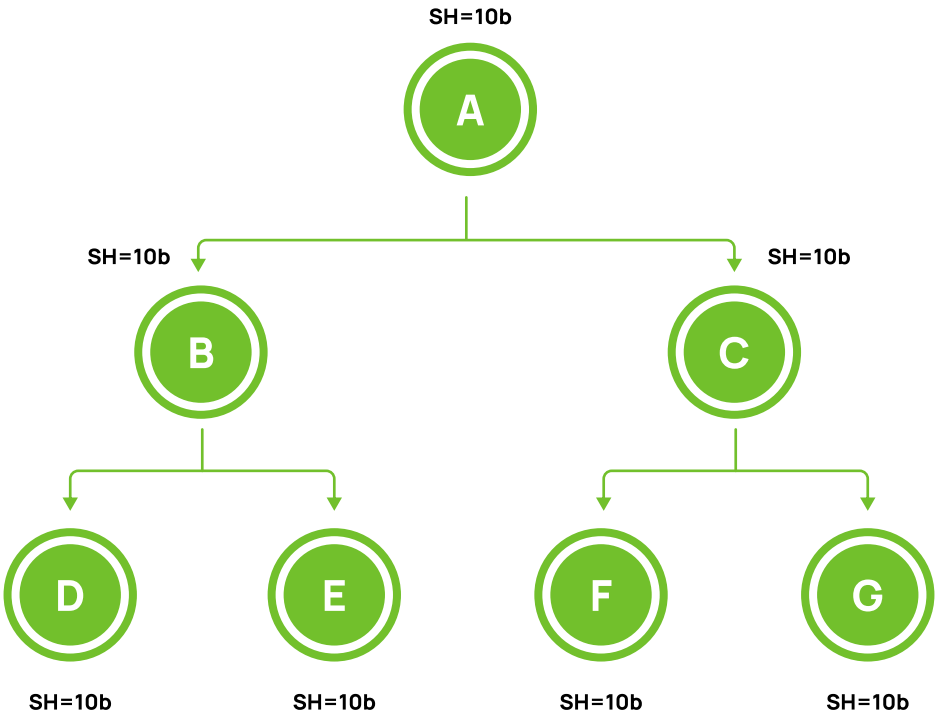


Fig 1: Objects in memory

It's easier to learn new concepts through example. Let's say your application's has object model as shown in Fig #1:

- Object A is holding reference to objects B and C.
- Object B is holding reference to objects D and E.
- Object C is holding reference to objects F and G.

Let's say each object occupies 10 bytes of memory. Now with this context let's begin our study.

Shallow Heap size

Shallow heap of an object is its size in the memory. Since in our example each object occupies 10 bytes, shallow heap size of each object is 10 bytes. Very simple.

Retained Heap size of B

From the Fig #1, you can notice that object B is holding reference to objects D and E. So, if object B is garbage collected from memory, there will be no more active references to object D and E. It means D & E can also be garbage collected. Retained heap is the amount of memory that will be freed when the particular object is garbage collected. Thus, retained heap size of B is:

= B's shallow heap size + D's shallow heap size + E's shallow heap size

= 10 bytes + 10 bytes + 10 bytes

= 30 bytes

Thus, retained heap size of B is 30 bytes.

Retained Heap size of C

Object C is holding reference to objects F and G. So, if object C is garbage collected from memory, there will be no more references to object F & G. It means F & G can also be garbage collected. Thus, retained heap size of C is:

= C's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes

= 30 bytes

Thus, retained heap size of C is 30 bytes as well

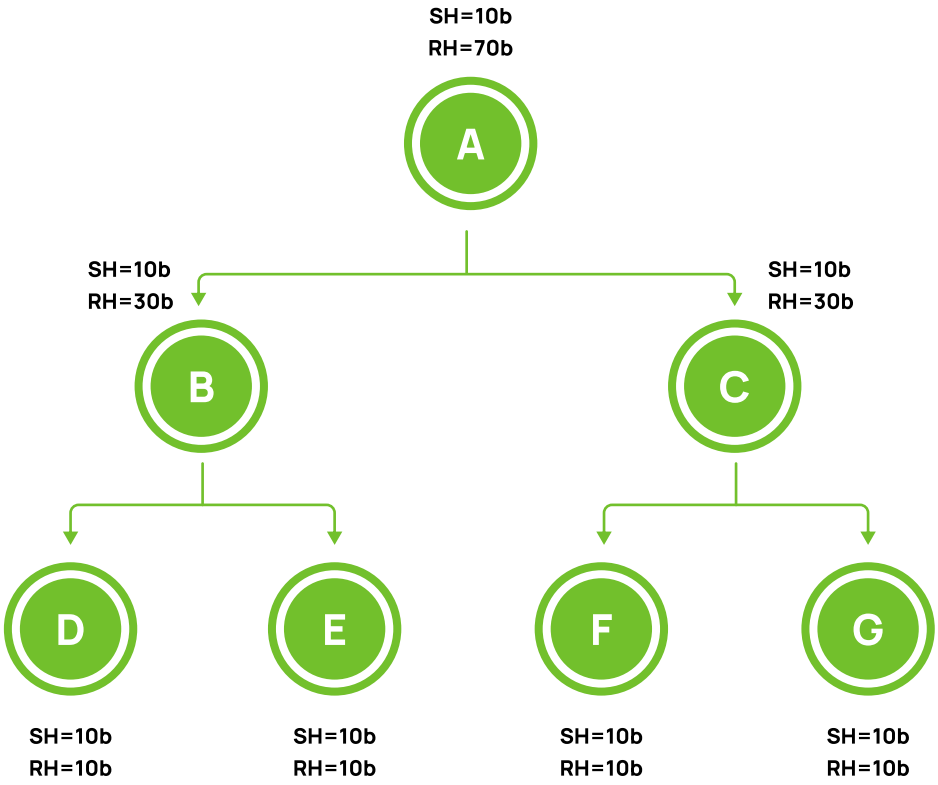


Fig 2: Objects Shallow and Retained Heap size

Retained Heap size of A

Object A is holding reference to objects B and C, which in turn are holding references to objects D, E, F, G. Thus, if object A is garbage collected from memory, there will be no more reference to object B, C, D, E, F and G. With this understanding let's do retained heap size calculation of A.

Thus, retained heap size of A is:

= A's shallow heap size + B's shallow heap size + C's shallow heap size + D's shallow heap size + E's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes + 10 bytes

= 70 bytes

Thus, retained heap size of A is 70 bytes.

Retained heap size of D, E, F and G

Retained heap size of D is 10 bytes only i.e. their shallow size only. Because D don't hold any active reference to any other objects. Thus, if D gets garbage collected no other objects will be removed from memory. As per the same explanation objects E, F and G's retained heap size are also 10 bytes only.

Let's make our study more interesting

Now let's make our study little bit more interesting, so that you will gain thorough understanding of shallow heap and retained heap size. Let's have object H starts to hold reference to B in the example. Note object B is already referenced by object A. Now two guys A and H are holding references to object B. In this circumstance lets study what will happen to our retained heap calculation.

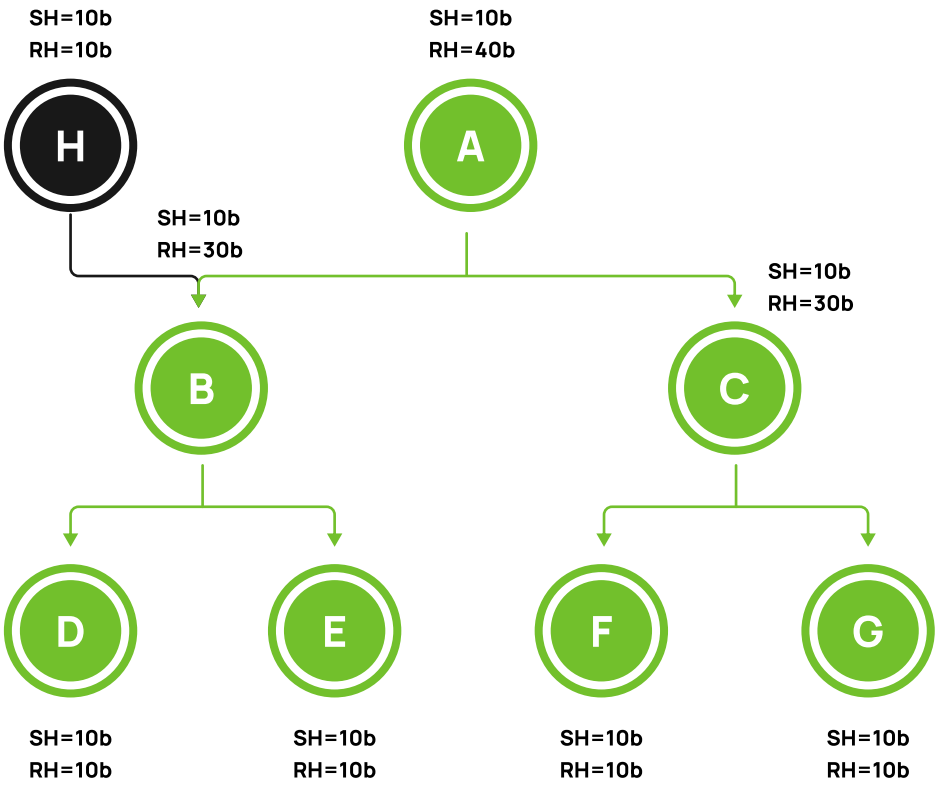


Fig 3: New reference to Object B

In this circumstance retained heap size of object A will go down to 40 bytes. Surprising? Puzzling? continue reading on. If object A gets garbage collected, then there will be no more reference to objects C, F and G only. Thus, only objects C, F and G will be garbage

collected. On the other hand, objects B, D and E will continue to live in memory, because H is holding active reference to B. Thus B, D and E will not be removed from memory even when A gets garbage collected.

Thus, retained heap size of A is:

= A's shallow heap size + C's shallow heap size + F's shallow heap size + G's shallow heap size

= 10 bytes + 10 bytes + 10 bytes + 10 bytes

= 40 bytes.

Thus, retained heap size of A will become 40 bytes. All other objects retained heap size will remain undisturbed, because there is no change in their references.

Hope this article helped to clarify Shallow heap size and Retained heap size calculation in Eclipse MAT. You might also consider exploring HeapHero– another powerful heap dump analysis tool, which shows the amount of memory wasted due to inefficient programming practices such as duplication of objects, overallocation and underutilization of data structures, suboptimal data type definitions,….

Video

Watch video

https://www.youtube.com/watch?v=Ns4Du2Qh91Y