Memory Analysis for large heaps with Memory Analyzer Tool (MAT)

Table of Contents

About this Document

Product	Memory Analyzer
Module	All
Version	1.6.0.20160531
Contributors	Tomas Mac Carthy

Often in production environments we have a large heap size (>12GB). When there are memory issues (e.g. memory leaks) we need to be able to analyze the heap and we need more memory than the heap size to be able to do this with the available tools.

Example:

- Customer is using a 16GB heap in production and the developer has 8GB of memory on his laptop.
- JProfiler 9 can handle this but it is an expensive commercial tool.
- Eclipse Memory Analyzer Tool (MAT) is a good tool for analyzing the heap. There is an option to run the memory intensive indexing part in headless mode (e.g. on a server with enough memory).

This article explores alternative tools that perform a similar task and shows how to run MAT tool in a more efficient way.

Generating a heap dump

Most of Hybris clients run on the Oracle JVM, so I will try to keep this post focussed on the instructions for the Oracle JVM. In case your application server runs out of memory you can instruct the JVM to generate a heap dump when an OutOfMemory Exception (OOME) occurs. This heap dump will be generated in the HPROF binary format.

You can do this in by:

- Manually: by using 'jmap', which is available since JDK 1.5.
  e.g. jmap -dump:format=b,file=java_pid<pid>.hprof <pid>
- Automatically by providing the following JVM command line parameters: -XX:+HeapDumpOnOutOfMemoryError
  By default the heap dump is created in a file called java_*pid*.hprof in the working directory of the VM, as in the example above. You can specify an alternative file name or directory with the `-XX:HeapDumpPath=` option. For example `-XX:HeapDumpPath=/disk2/dumps` will cause the heap dump to be generated in the `/disk2/dumps` directory.

In theory, the size of the heap dump should be around the same size as the configured max heap size JVM parameter. So if you have set your max heap size to -Xmx4G , your heap dump will be around that size. However, in the test performed for the purpose of this article the generated d ump size was bigger that the configure maximum heap size: for a heap of Xmx24G the heap dump size was 36Gb.

Analyzing the heap dump

Now you have the heap dump and want to figure out what was inside the heap at the moment the OOME occurred. As mention before there are several Java heap dump analyzers out there, where most of them can do more then just heap analysis. The products range from commercial to open source and these are the ones that I tried:

- Eclipse Memory Analyzer (MAT)
- jhat (Java Heap Analysis Tool)
- Visual VM

Most of the above applications were unable to handle large dump files in reasonable amount of time or either present the heap data in a clear and understandable way. Eclipse Memory Analyzer was actually the only heap dump analyzer that was able handle a 36Gb dump. All the other analyzers were unable to handle a file of this size. This was perform on a production-like server, of course, on a laptop this will not fit. Eclipse MAT was able to analyze this file within approximately **30 minutes**.

About Eclipse Memory Analyzer

One of the most important things about Eclipse Memory Analyzer is that it indexes the heap dumps only once, with the capability of processing of the heap dump independent of the analysis. Once you've parsed the entire heap dump, reopening it is very fast, because it does not have to process it all over again.

Other features are:

- Headless indexing: the process of parsing and indexing can be performed separately through command line to make it more efficient –JVM parameters can be tuned when the resources are limited (like in our own laptops) or to make it faster.
- Analyzes and reporting: Apart from the capability to browse the instances a and classes in the dump, MAT also provides algorithmic analyzes to find suspect memory leaks and top components.
- Portability: The creation of the index files represent the most resource demanding activity and has to be done only once. When heap dump size is large its recommended to do this on the production server and then transfer the data to the consultant laptop for manual analyzis through the UI and report generation.

Once you have the heap dump on your screen the dominator tree view is the most useful view and can give you a very good insight on what was loaded when the server ran out of memory.

Next to the statistical views there is also an automatic leak hunter available to help you figure out the problem as fast as possible.

Eclipse Memory Analyzer is portable, easy to install and can be tuned to run with limeted resources.

Download and Install MAT Standalone

MAT can be installed into Eclipse as a plugin or it can be run in standalone mode. The standalone provides many benefits and its can also be executed in a headless mode.

A standalone version can be found from https://eclipse.org/mat/downloads.php

Running MAT in headless mode

When you unpack MAT on your machine you will find a batch file and a shell script ParseHeapDump.sh (MemoryAnalyzer for MAC users) in the start directory, use it to process your heap dump.

```
/MemoryAnalyzer -consolelog -application org.eclipse.mat.api.parse
../today_heap_dump/jvm.hprof
```

It will populate the directory with all kind of index files. The whole directory can then be copied to a smaller desktop machine and then used to generate some reports.

The shell script needs less resources than parsing the heap from the GUI, plus you can run it on a server with more resources (you can allocate more resources by adding something like `-vmargs -Xmx40g -XX:-UseGCOverheadLimit` to the end of the last line of the script. For instance, it might look like this after modification.

```
./MemoryAnalyzer -consolelog -application org.eclipse.mat.api.parse
../today_heap_dump/jvm.hprof -vmargs -Xmx40g -XX:-UseGCOverheadLimit
```

The size of the memory Xmx for the headless mode can be defined in the in the command line or the MemoryAnalizer.ini file. Here you can also set garbage collection alogoritms to keep the MAT getting OOME while running it in command line mode instead of GUI as it takes more memory to analyse.

```
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled
-XX:+CMSClassUnloadingEnabled -XX:+UseCMSInitiatingOccupancyOnly
```

After the headless parsing is finished you can either use the created reports only (they are HTML inside ZIP), or start the GUI and just open the heap with "open". It will detect that the file is already parsed and use the existing index files. (For this to work they have to be in the same dir as the hprof file).

The headless mode does take a bit less memory than the GUI mode.

For the Xmx, you can go risky and specify RAM size or even a few percentage more. Chances are the parsing does not use all of it
(but completes before OOME).

Generating the report

After getting the indices, try to generate reports from that as well and copying those to your local machines to try to find the memory issue.

It takes a while to execute and generates indices files. Then use the indices created in the previous step and run a "Leak suspects" report or other analysis on the heap dump.

```
./MemoryAnalyzer ../today_heap_dump/jvm.hprof org.eclipse.mat.api:suspects
```

The output is a small and easy to download jvm_Leak_Suspects.zip. This has HTML files just like the MAT Eclipse UI and it can be easily SCP'ed/emailed around.

Other report types possible.

- org.eclipse.mat.api:suspects
- org.eclipse.mat.api:overview
- org.eclipse.mat.api:top_components

More details - http://wiki.eclipse.org/index.php/MemoryAnalyzer/FAQ.

If those reports are not enough and more digging is needed copy the indices files as well as hprof file to my local machine, and then open the heap dump (with the indices in the same directory as the heap dump) with Eclipse MAT plugin or standalone version. From there, it does not need too much memory to run.

Important points:

- Only the generation of the indices is the memory intensive part of Eclipse MAT. After you have the indices, most of your processing from Eclipse MAT would not need that much memory.
- Doing this on a shell script means I can do it on a headless server (and I normally do it on a headless server as well, because they're normally the most powerful ones). And if you have a server that can generate a heap dump of that size, chances are, you have another server out there that can process that much of a heap dump as well (twice disk space is recommend when running the MAT in headless mode).

Analysis simulation

For the purpose of this article a large heap dump has been generated on a production-like server and then analyzed using MAT headless mode.

Server Specs

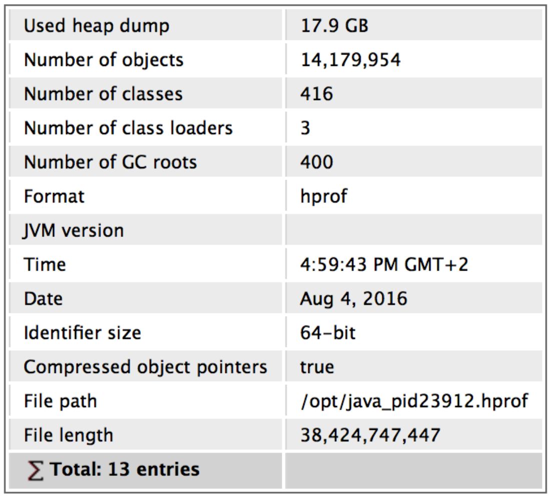
```
hybris@extservices11:~/mat$ lscpu Architecture: x86_64 CPU op-mode(s): 32-bit, 64-bit
CPU(s): 4 hybris@extservices11:~/mat$ cat /proc/meminfo MemTotal: 33021980 kB
```

A simple java code was written to fill the heap space, Finalizer.jar

The code was executed through command line with a maximum heap space of 24G.

```
java -Xmx24G -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/var/tmp -jar
Finalizer.jar
```

After a couple of minutes the OOME was thrown and the a heap dump of 36Gb is created in the specified path /var/tmp.



Used heap dump	17.9 GB
Number of objects	14,179,954
Number of classes	416
Number of class loaders	3
Number of GC roots	400
Format	hprof
JVM version	
Time	4:59:43 PM GMT+2
Date	Aug 4, 2016
Identifier size	64-bit
Compressed object pointers	true
File path	/opt/java_pid23912.hprof
File length	38,424,747,447
∑ Total: 13 entries	

We then perform the indexing running the MAT in headless mode.

```
./MemoryAnalyzer -consolelog -application org.eclipse.mat.api.parse
/opt/java_pid23912.hprof -vmargs -Xmx40G
```

The parsing took **30 minutes** and .index files were generated in the same directory of the heap dump (~650M).

```
hybris@extservices11:/opt$ du -sh *
15M java_pid23912.a2s.index
40M java_pid23912.domIn.index
120M java_pid23912.domOut.index
36G java_pid23912.hprof
8.0K java_pid23912.i2sv2.index
103M java_pid23912.idx.index
149M java_pid23912.inbound.index
1.9M java_pid23912.index
56K java_pid23912_Leak_Suspects.zip
31M java_pid23912.o2c.index
109M java_pid23912.o2hprof.index
81M java_pid23912.o2ret.index
147M java_pid23912.outbound.index
40K java_pid23912_System_Overview.zip
4.0K java_pid23912.threads
56K java_pid23912_Top_Components.zip
```

During the parsing its been observed that some temporary file were created. The size of these files was not measured but is recommended to keep a considerable amount of space in disk when performing this operation (more than 650M).

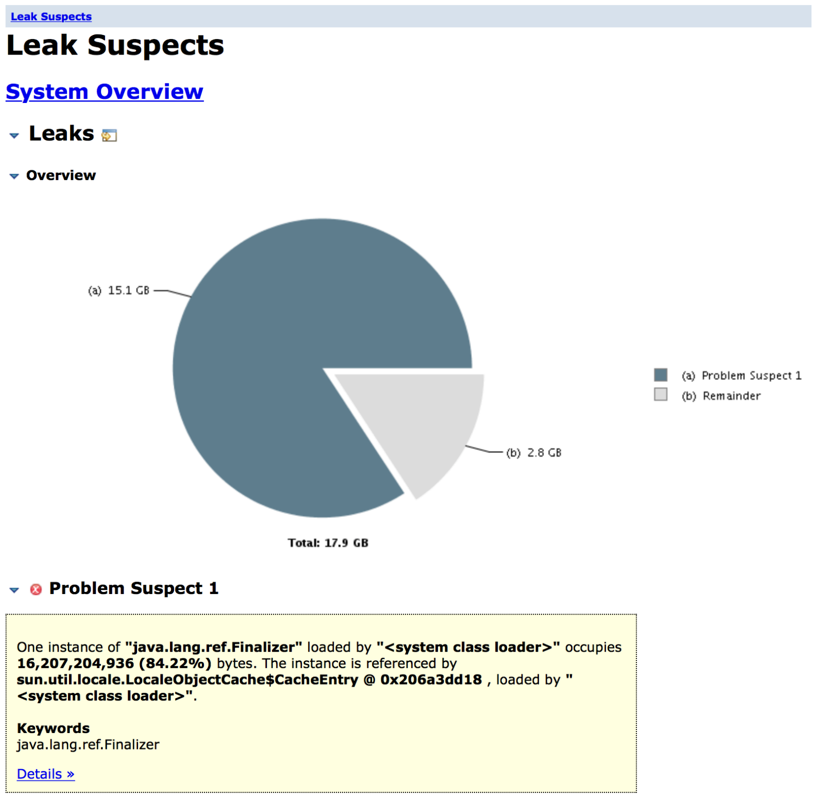
Keep in mind that there are some differences between the heap dump generated in this exercise and one taken from a productive system, e.g. Total classes.

Once the parsing is complete we run the analysis and generate the reports. Below are the three most common analysis.

```
./ParseHeapDump.sh /var/tmp/java_pid23912.hprof org.eclipse.mat.api:suspects
./ParseHeapDump.sh /var/tmp/java_pid23912.hprof org.eclipse.mat.api:overview
./ParseHeapDump.sh /var/tmp/java_pid23912.hprof org.eclipse.mat.api:top_components
```

The analysis and report generation took between 2 and 10 minutes depending on the type of analysis.

From the reports generates we can clearly see the an instances of "java.lang.ref.Finalizer" present in most of the heap space and the responsible for the OOME.



Leak Suspects

Leak Suspects

System Overview

Leaks

Overview

Problem Suspect 1

One instance of **"java.lang.ref.Finalizer"** loaded by **"<system class loader>"** occupies **16,207,204,936 (84.22%)** bytes. The instance is referenced by **sun.util.locale.LocaleObjectCache$CacheEntry @ 0x206a3dd18** , loaded by **"<system class loader>"**.

Keywords
java.lang.ref.Finalizer

Details »

Summary

If you ever have to analyze a large heap dump I would recommend to use Memory Analyzer Tool as a main tool and run it in Headless mode. It's fast, portable, easy to use and free.