https://blog.csdn.net/lusa1314/article/details/84134458

**JVM参数-XX:+HeapDumpOnOutOfMemoryError使用方法**

2018年11月16日 11:07:29 [lusasky](https://me.csdn.net/lusa1314" \t "_blank) 阅读数 442

1、配置方法  
  
在JAVA\_OPTIONS变量中增加  
  
-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${目录}。  
例如：export JAVA\_OPTS="-Xms2048M -Xmx2048M -Xmn682M -XX:MaxPermSize=96M"  
  
2、参数说明  
  
（1）-XX:+HeapDumpOnOutOfMemoryError参数表示当JVM发生OOM时，自动生成DUMP文件。  
  
（2）-XX:HeapDumpPath=${目录}参数表示生成DUMP文件的路径，也可以指定文件名称，例如：-XX:HeapDumpPath=${目录}/java\_heapdump.hprof。如果不指定文件名，默认为：java\_<pid>\_<date>\_<time>\_heapDump.hprof。

**[java命令--jmap命令使用](https://www.cnblogs.com/kongzhongqijing/articles/3621163.html" \t "_blank)**

      jdk安装后会自带一些小工具，jmap命令(Java Memory Map)是其中之一。主要用于打印指定Java进程(或核心文件、远程调试服务器)的共享对象内存映射或堆内存细节。

   jmap命令可以获得运行中的jvm的堆的快照，从而可以离线分析堆，以检查内存泄漏，检查一些严重影响性能的大对象的创建，检查系统中什么对象最多，各种对象所占内存的大小等等。可以使用jmap生成Heap Dump。

java memory = direct memory（直接内存） + jvm memory(MaxPermSize +Xmx)

**1)直接内存跟堆**  
     直接内存则是一块由程序本身管理的一块内存空间，它的效率要比标准内存池要高，主要用于存放网络通信时数据缓冲和磁盘数据交换时的数据缓冲。   
    DirectMemory容量可以通过 -XX:MaxDirectMemorySize指定，如果不指定，则默认为与Java堆的最大值（-Xmx指定）一样。但是，在OSX上的最新版本的 JVM，对直接内存的默认大小进行修订，改为“在不指定直接内存大小的时默认分配的直接内存大小为64MB”，可以通过 -XX:MaxMemorySize来显示指定直接内存的大小。   
**2)堆(Heap)和非堆(Non-heap)内存**

按照官方的说法：“Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。  
可以看出JVM主要管理两种类型的内存：堆和非堆。简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

**3)栈与堆**

      栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；

      堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

    Java的堆是一个运行时数据区,类的(对象从中分配空间。这些对象通过new、newarray、anewarray和multianewarray等 指令建立，它们不需要程序代码来显式的释放。堆是由垃圾回收来负责的，堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时 动态分配内存的，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。 栈的优势是，存取速度比堆要快，仅次于寄存器，栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。栈中主要存放一些基本类 型的变量（,int, short, long, byte, float, double, boolean, char）和对象句柄。  
  
**线程占用大小在MaxPermSize中进行内存申请和分配**

**什么是堆Dump**

    堆Dump是反应Java堆使用情况的内存镜像，其中主要包括系统信息、虚拟机属性、完整的线程Dump、所有类和对象的状态等。 一般，在内存不足、GC异常等情况下，我们就会怀疑有[内存泄露](http://zh.wikipedia.org/zh-cn/%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F" \t "_blank)。这个时候我们就可以制作堆Dump来查看具体情况。分析原因。

**基础知识**

[Java虚拟机的内存组成以及堆内存介绍](http://www.hollischuang.com/archives/80) [Java GC工作原理](http://www.hollischuang.com/archives/76) 常见内存错误：

outOfMemoryError 年老代内存不足。  
outOfMemoryError:PermGen Space 永久代内存不足。  
outOfMemoryError:GC overhead limit exceed 垃圾回收时间占用系统运行时间的98%或以上。

**一、介绍**

      打印出某个java进程（使用pid）内存内的，所有‘对象’的情况（如：产生那些对象，及其数量）。它的用途是为了展示java进程的内存映射信息，或者堆内存详情。

可以输出所有内存中对象的工具，甚至可以将VM 中的heap，以二进制输出成文本。

jmap -heap:format=b pid   bin格式  javaversion 1.5

jmap -dump:format=b,file=filename pid javaversion >1.6

     jmap -dump:format=b,file=outfile 3024可以将3024进程的内存heap输出出来到outfile文件里;

     再配合MAT（内存分析工具(Memory Analysis Tool）或与jhat (Java Heap Analysis Tool)一起使用，能够以图像的形式直观的展示当前内存是否有问题。

64位机上使用需要使用如下方式：

jmap -J-d64 -heap pid

**二、命令格式**

SYNOPSIS  
jmap [ option ] pid  (to connect to remote debug server)     
jmap [ option ] executable core  (to connect to remote debug server)     
jmap [ option ] [server-id@]remote-hostname-or-IP  (to connect to remote debug server)

复制代码

where <option> is one of:

<none> to print same info as Solaris pmap

-heap to print java heap summary

-histo[:live] to print histogram of java object heap; if the "live" suboption is specified, only count live objects

-permstat to print permanent generation statistics

-finalizerinfo to print information on objects awaiting finalization

-dump:<dump-options> to dump java heap in hprof binary format

　　　　　　　　　　 dump-options:

live dump only live objects; if not specified,all objects in the heap are dumped.

　　　　　　　　　　 format=b binary format

file=<file> dump heap to <file>

Example: jmap -dump:live,format=b,file=heap.bin <pid>

-F force. Use with -dump:<dump-options> <pid> or -histo to force a heap dump or histogram when <pid> does not respond. The "live" suboption is not supported in this mode.

-h | -help to print this help message

-J<flag> to pass <flag> directly to the runtime system

参数如下：

-heap：打印jvm heap的情况

-histo：打印jvm heap的直方图。其输出信息包括类名，对象数量，对象占用大小。

-histo：live ：同上，但是只答应存活对象的情况

-permstat：打印permanent generation heap情况

复制代码

参数说明：

1)、options：

executable     Java executable from which the core dump was produced.(可能是**产生core dump的java可执行程序**)  
core **将被打印信息的core dump文件**  
remote-hostname-or-IP 远程debug服务的主机名或ip  
server-id 唯一id,假如一台主机上多个远程debug服务，用此选项参数标识服务器

2）基本参数：

<no option> 如果使用不带选项参数的jmap打印共享对象映射，将会打印目标虚拟机中加载的每个共享对象的起始地址、映射大小以及共享对象文件的路径全称。这与Solaris的pmap工具比较相似。  
**-dump:[live,]format=b,file=<filename>**使用hprof二进制形式,输出jvm的heap内容到文件, live子选项是可选的，假如指定live选项,那么只输出活的对象到文件.   
-finalizerinfo 打印正等候回收的对象的信息.  
**-heap** 打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况.  
**-histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”\*”. 如果live子参数加上后,只统计活的对象数量.**   
-permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来.   
-F 强迫.在pid没有响应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效.   
-h | -help 打印辅助信息   
-J<flag> 传递参数给jmap启动的jvm.   
pid 需要被打印配相信息的java进程id,可以用jps查问

      从安全点日志看，从Heap Dump开始，整个JVM都是停顿的，考虑到IO（虽是写到Page Cache，但或许会遇到background flush），几G的Heap可能产生几秒的停顿，在生产环境上执行时谨慎再谨慎。

live的选项，实际上是产生一次Full GC来保证只看还存活的对象。有时候也会故意不加live选项，看历史对象。

Dump出来的文件建议用JDK自带的VisualVM或Eclipse的MAT插件打开，对象的大小有两种统计方式：

* 本身大小(Shallow Size)：对象本来的大小。
* 保留大小(Retained Size)： 当前对象大小 + 当前对象直接或间接引用到的对象的大小总和。

看本身大小时，占大头的都是char[] ,byte[]之类的，没什么意思（用jmap -histo:live pid 看的也是本身大小）。所以需要关心的是保留大小比较大的对象，看谁在引用这些char[], byte[]。

(MAT能看的信息更多，但VisualVM胜在JVM自带，用法如下：命令行输入jvisualvm，文件->装入->堆Dump－>检查 -> 查找20保留大小最大的对象，就会触发保留大小的计算，然后就可以类视图里浏览，按保留大小排序了)

**三、使用实例**

1、jmap -heap pid 展示pid的整体堆信息

#ps -ef|grep tomcat  #获取tomcat的pid

复制代码

[root@localhost ~]# jmap -heap 27900

Attaching to process ID 27900, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 20.45-b01

using thread-local object allocation.

Mark Sweep Compact GC

Heap Configuration: #堆内存初始化配置

MinHeapFreeRatio = 40 #-XX:MinHeapFreeRatio设置JVM堆最小空闲比率

MaxHeapFreeRatio = 70 #-XX:MaxHeapFreeRatio设置JVM堆最大空闲比率

MaxHeapSize = 100663296 (96.0MB) #-XX:MaxHeapSize=设置JVM堆的最大大小

NewSize = 1048576 (1.0MB) #-XX:NewSize=设置JVM堆的‘新生代’的默认大小

MaxNewSize = 4294901760 (4095.9375MB) #-XX:MaxNewSize=设置JVM堆的‘新生代’的最大大小

OldSize = 4194304 (4.0MB) #-XX:OldSize=设置JVM堆的‘老生代’的大小

NewRatio = 2 #-XX:NewRatio=:‘新生代’和‘老生代’的大小比率

SurvivorRatio = 8 #-XX:SurvivorRatio=设置年轻代中Eden区与Survivor区的大小比值

PermSize = 12582912 (12.0MB) #-XX:PermSize=<value>:设置JVM堆的‘持久代’的初始大小

MaxPermSize = 67108864 (64.0MB) #-XX:MaxPermSize=<value>:设置JVM堆的‘持久代’的最大大小

Heap Usage:

New Generation (Eden + 1 Survivor Space): #新生代区内存分布，包含伊甸园区+1个Survivor区

capacity = 30212096 (28.8125MB)

used = 27103784 (25.848182678222656MB)

free = 3108312 (2.9643173217773438MB)

89.71169693092462% used

Eden Space: #Eden区内存分布

capacity = 26869760 (25.625MB)

used = 26869760 (25.625MB)

free = 0 (0.0MB)

100.0% used

From Space: #其中一个Survivor区的内存分布

capacity = 3342336 (3.1875MB)

used = 234024 (0.22318267822265625MB)

free = 3108312 (2.9643173217773438MB)

7.001809512867647% used

To Space: #另一个Survivor区的内存分布

capacity = 3342336 (3.1875MB)

used = 0 (0.0MB)

free = 3342336 (3.1875MB)

0.0% used

tenured generation: #当前的Old区内存分布

capacity = 67108864 (64.0MB)

used = 67108816 (63.99995422363281MB)

free = 48 (4.57763671875E-5MB)

99.99992847442627% used

Perm Generation: #当前的 “持久代” 内存分布

capacity = 14417920 (13.75MB)

used = 14339216 (13.674942016601562MB)

free = 78704 (0.0750579833984375MB)

99.45412375710227% used

复制代码

新生代的内存回收就是采用空间换时间的方式；  
如果from区使用率一直是100% 说明程序创建大量的短生命周期的实例，使用jstat统计一下jvm在内存回收中发生的频率耗时以及是否有full gc，使用这个数据来评估一内存配置参数、gc参数是否合理；

多次ygc后如果s区没变化，这种情况不出意外就是[担保](http://blog.csdn.net/u013295276/article/details/78468790?locationNum=1&fps=1" \t "_blank)了，可以jstat持续观察下；

复制代码

[root@localhost ~]# jmap -heap 4951

Attaching to process ID 4951, please wait...

Debugger attached successfully.

Client compiler detected.

JVM version is 20.45-b01

using thread-local object allocation.

Parallel GC with 1 thread(s)

Heap Configuration:

MinHeapFreeRatio = 40

MaxHeapFreeRatio = 70

MaxHeapSize = 734003200 (700.0MB)

NewSize = 314572800 (300.0MB)

MaxNewSize = 314572800 (300.0MB)

OldSize = 4194304 (4.0MB)

NewRatio = 2

SurvivorRatio = 8

PermSize = 104857600 (100.0MB)

MaxPermSize = 104857600 (100.0MB)

Heap Usage:

PS Young Generation

Eden Space:

capacity = 235929600 (225.0MB)

used = 169898184 (162.02753448486328MB)

free = 66031416 (62.97246551513672MB)

72.01223754882812% used

From Space:

capacity = 39321600 (37.5MB)

used = 0 (0.0MB)

free = 39321600 (37.5MB)

0.0% used

To Space:

capacity = 39321600 (37.5MB)

used = 0 (0.0MB)

free = 39321600 (37.5MB)

0.0% used

PS Old Generation

capacity = 419430400 (400.0MB)

used = 0 (0.0MB)

free = 419430400 (400.0MB)

0.0% used

PS Perm Generation

capacity = 104857600 (100.0MB)

used = 14140688 (13.485610961914062MB)

free = 90716912 (86.51438903808594MB)

13.485610961914062% used

复制代码

根据打印的结果：默认存活区与eden比率=2：8

1）查看eden区：225M

2）两个存活区大小：都为37.5M

3）年轻代大小：300M

4）老年代大小：400M

5）持久代大小：100M

6）最大堆内存大小：年轻代大小+老年代大小=700M

7）java应用程序占用内存大小：最大堆内存大小+持久代大小=700M+100M=800M

对应java参数（在tomcat的启动文件里）配置如下：

JAVA\_OPTS="-Xms700m -Xmx700m -Xmn300m -Xss1024K -XX:PermSize=100m -XX:MaxPermSize=100 -XX:+UseParallelGC -XX:ParallelGCThreads=1 -XX:+PrintGCTimeStamps

-XX:+PrintGCDetails -Xloggc:/opt/logs/gc.log -verbose:gc -XX:+DisableExplicitGC"

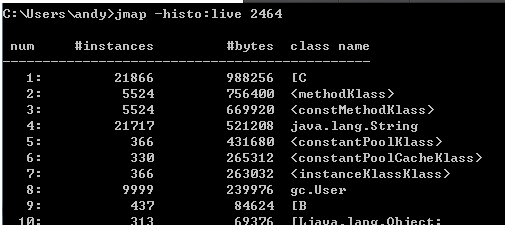
注意参数配置时，上述参数为一行。

各参数意义：

* Xms
* Xmx
* Xmn
* Xss
* -XX:PermSize
* -XX:MaxPermSize
* NewRatio = 2   表示新生代（e+2s）：老年代（不含永久区）=1：2，指新生代占整个堆的1/3
* SurvivorRatio = 8  表示2个S：eden=2：8，一个S占年轻代的1/10

2、jmap -histo pid 展示class的内存情况

说明：instances（实例数）、bytes（大小）、classs name（类名）。它基本是按照使用使用大小逆序排列的。



复制代码

#instance 是对象的实例个数

#bytes 是总占用的字节数

class name 对应的就是 Class 文件里的 class 的标识

B 代表 byte

C 代表 char

D 代表 double

F 代表 float

I 代表 int

J 代表 long

Z 代表 boolean

前边有 [ 代表数组， [I 就相当于 int[]

对象用 [L+ 类名表示

复制代码

 从打印结果可看出，类名中存在[C、[B等内容，

* 只知道它占用了那么大的内存，但不知道由什么对象创建的。下一步需要将其他dump出来，使用内存分析工具进一步明确它是由谁引用的、由什么对象。

3、 jmap -histo:live pid>a.log

可以观察heap中所有对象的情况（heap中所有生存的对象的情况）。包括对象数量和所占空间大小。 可以将其保存到文本中去，在一段时间后，使用文本对比工具，可以对比出GC回收了哪些对象。

jmap -histo:live 这个命令执行，JVM会先触发gc，然后再统计信息。

4、dump 将内存使用的详细情况输出到文件

jmap -dump:live,format=b,file=a.log pid

说明：内存信息dump到a.log文件中。

 这个命令执行，JVM会将整个heap的信息dump写入到一个文件，heap如果比较大的话，就会导致这个过程比较耗时，并且执行的过程中为了保证dump的信息是可靠的，所以会暂停应用。

该命令通常用来分析内存泄漏OOM，通常做法是：

1）首先配置JVM启动参数，让JVM在遇到OutOfMemoryError时自动生成Dump文件

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path

2）然后使用命令

jmap  -dump:format=b,file=/path/heap.bin 进程ID

如果只dump heap中的存活对象，则加上选项-live。

 3）然后使用MAT分析工具，如jhat命令，eclipse的mat插件。

最后在eclipse中安装MAT插件（http://www.eclipse.org/mat/），然后在eclipse中，file---->open，打开这个文件heap.bin，利用现成的OOM工具进行分析。

具体操作方法：

首先输入网址http://www.eclipse.org/mat/previousReleases.php，然后查看你自己的Eclipse版本，我的是Indigo的，所以点击链接“Previous Releases”，选择Indigo版本的URLhttp://download.eclipse.org/mat/1.1.0/update-site/

 用jhat命令可以参看 jhat -port 5000 heapDump 在浏览器中访问：http://localhost:5000/ 查看详细信息

**四、性能问题查找**

转自：http://www.cnblogs.com/gaojk/articles/3886503.html

**1、发现问题**

1）、使用uptime命令查看CPU的Load情况，Load越高说明问题越严重；

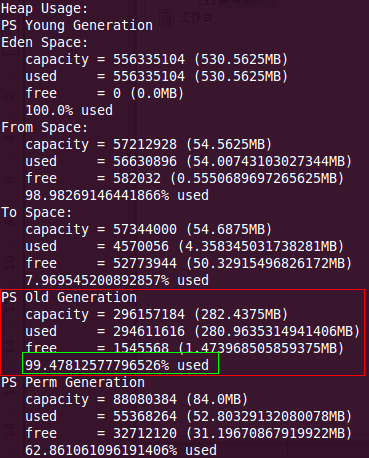
2）、使用jstat查看FGC发生的频率及FGC所花费的时间，FGC发生的频率越快、花费的时间越高，问题越严重；

**2、导出数据：在应用快要发生FGC的时候把堆导出来**

1）、查看快要发生FGC使用命令：

jmap -heap <pid>

会看到如下图结果：



    以上截图包括了新生代、老年代及持久代的当前使用情况，如果不停的重复上面的命令，会看到这些数字的变化，变化越大说明系统存在问题的可能性越大，特别是被红色圈起来的老年代的变化情况。现在看到的这个值为使用率为99%或才快接近的时候，就立即可以执行导出堆栈的操作了。

    注：这是因为我这里没有在jvm参数中使用"-server"参数，也没有指定FGC的阀值，在线上的应用中通过会指定CMSInitiatingOccupancyFraction这个参数来指定当老年代使用了百分之多少的时候，通过CMS进行FGC，当然这个参数需要和这些参数一起使用“-XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:+UseCMSInitiatingOccupancyOnly”，CMSInitiatingOccupancyFraction的默认值是68，现在中文站线上的应用都是70，也就是说当老年代使用率真达到或者超过70%时，就会进行FGC。

    2）、将数据导出：

jmap -dump:format=b,file=heap.bin <pid>

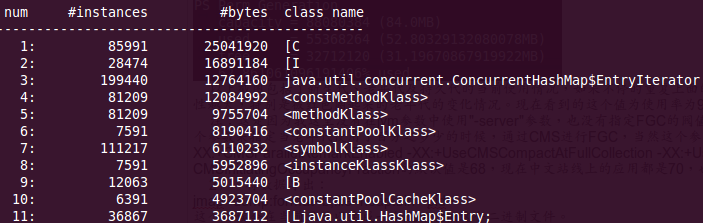
这个时候会在当前目录以生成一个heap.bin这个二进制文件。

**3、通过命令查看大对象**

也是使用jmap的命令，只不过参数使用-histo

使用：jmap -histo <pid>|less

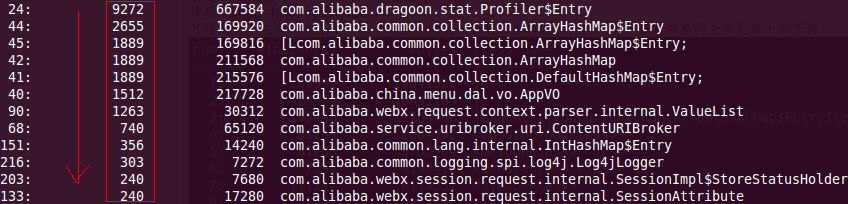
可得到如下包含对象序号、某个对象示例数、当前对象所占内存的大小、当前对象的全限定名，如下图：



查看对象数最多的对象，并按降序排序输出：

执行：jmap -histo <pid>|grep alibaba|sort -k 2 -g -r|less

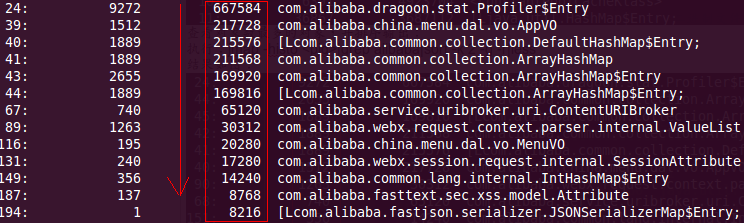
结果如图：



查看占用内存最多的最象，并按降序排序输出：

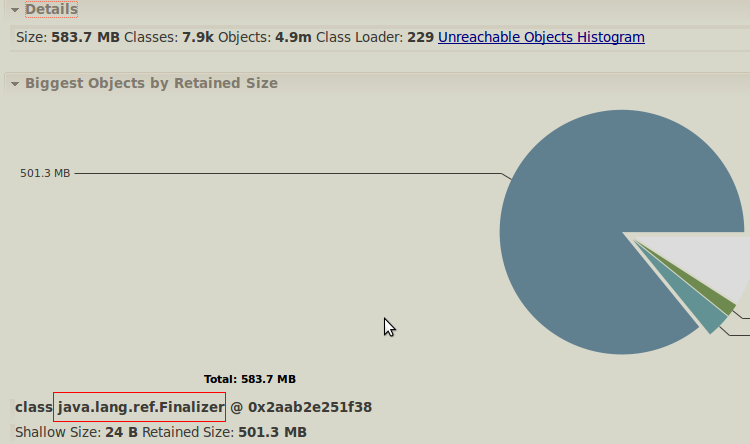
执行：jmap -histo <pid>|grep alibaba|sort -k 3 -g -r|less

结果如图：

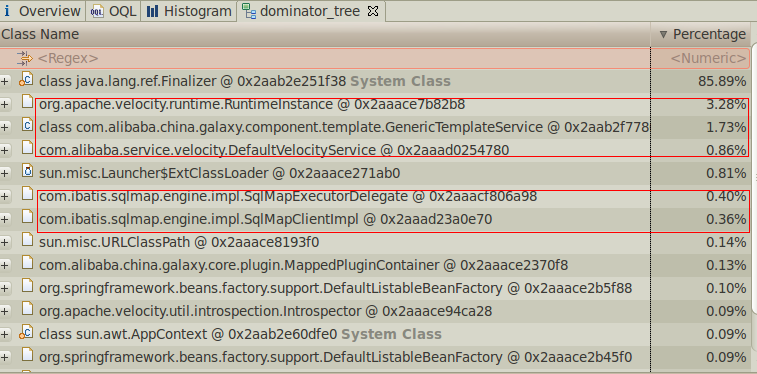


**4、数据分析**

这个时候将dump出的文件在ECLIPSE中打开，使用MAT进行分析（ECLIPSE需要先安装MAT插件），会展示如下截图：

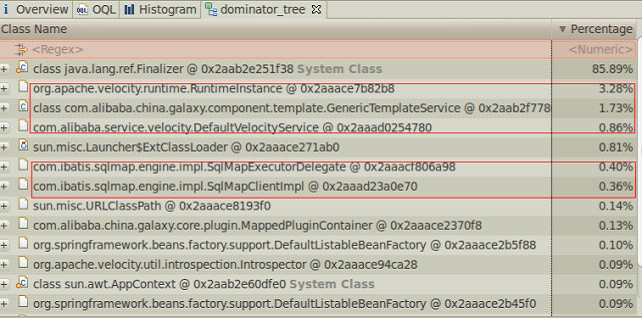


可以从这个图看出这个类java.lang.ref.Finalizer占用500多M，表示这其中很多不能够被回对象的对象，此时点开hisgogram视图，并通过Retained Heap进行排序，如下截图：



从图中可以看出，被线线框圈起来的三个对象占用量非常大，那说明这几个大的对象并没有被释放，那现在就可以有针对性的从代码中去找这几个对象为什么没有被释放了。

再切换到dominator\_tree视图：



这里可以看到velocity渲染也存在着问题，以及数据库的请求也比较多。

**5、优化**

优化的思路就是上面所列出来的问题，查看实现代码中所存在问题，具体问题具体分析。

**总结**

1.如果程序内存不足或者频繁GC，很有可能存在内存泄露情况，这时候就要借助Java堆Dump查看对象的情况。  
2.要制作堆Dump可以直接使用jvm自带的jmap命令  
3.可以先使用jmap -heap命令查看堆的使用情况，看一下各个堆空间的占用情况。  
4.使用jmap -histo:[live]查看堆内存中的对象的情况。如果有大量对象在持续被引用，并没有被释放掉，那就产生了内存泄露，就要结合代码，把不用的对象释放掉。  
5.也可以使用 jmap -dump:format=b,file=<fileName>命令将堆信息保存到一个文件中，再借助jhat命令查看详细内容  
6.在内存出现泄露、溢出或者其它前提条件下，建议多dump几次内存，把内存文件进行编号归档，便于后续内存整理分析。

7.在用cms gc的情况下，执行jmap -heap有些时候会导致进程变T，因此强烈建议别执行这个命令，如果想获取内存目前每个区域的使用状况，可通过jstat -gc或jstat -gccapacity来拿到。