## Eclipse Memory Analyzer

### 10 Tips for using the Eclipse Memory Analyzer « EclipseSource Blog

http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/

Analyzing and understanding the memory use of an application is challenging. A subtle logic error can result in listeners never being disposed, ultimately leading to the dreaded ***OutOfMemory***error. Even if your application is properly disposing of all unused objects, it may still be requiring 10 or 100 times more memory than necessary.

Lucky for us, the Eclipse Memory Analyzer (**MAT**) can help provide details of an application’s memory use. The tool is useful for both tracking memory leaks and for periodically reviewing the state of your system. In this tutorial I’ll outline 10 tips to help you use the MAT more effectively. If you’re a Java developer, the Eclipse Memory Analyzer Tool  should certainly be in your [debugging toolbox](http://eclipsesource.com/blogs/2013/01/08/effective-java-debugging-with-eclipse/).

**[ Looking for more useful tools? See our**[**Eclipse Tools**](http://eclipsesource.com/technology/eclipse-tools/)**page. | Managing your Eclipse workspace is easier with**[**Yoxos**](https://yoxos.eclipsesource.com/)**.**[**Create a free profile now**](https://yoxos.eclipsesource.com/share.html)**. ]**

The Memory Analyzer can be installed using the Install New Software dialog or through the Eclipse *MarketPlace Client*. You can also include it in your own custom Eclipse install using [Yoxos](http://eclipsesource.com/en/products/yoxos/).

**In this example, we are using a very simple program that allocates 100,000 Listeners, and stores them in 4 different lists. The application then sleeps without removing or clearing these lists.**

1. Acquiring a memory snapshot (Heap Dump)

There are several ways you can get started with MAT. You can:

1. Configure an application to dump the contents of its memory when an out of memory error occurs,
2. Connect the MAT to an existing java process, or
3. Manually acquire a heap dump and load it into the MAT

In all cases it’s important to keep in mind that this is a ***snapshot of memory at a point in time***. The MAT cannot tell you why an object was created, nor can it show you objects that have already been garbage collected. However, if you use the MAT alongside [other debugging tools and techniques](http://eclipsesource.com/blogs/2013/01/08/effective-java-debugging-with-eclipse/), you can often conquer memory leaks pretty quickly.

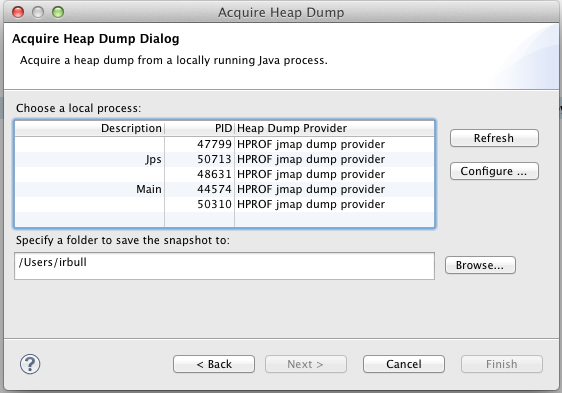
To configure your application to produce a heap dump whenever an OutOfMemory Error is thrown, add the following vm argument:

-XX:+HeapDumpOnOutOfMemoryError

Alternatively, you can use **jstack** to acquire a Heap dump from a currently running java process.

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

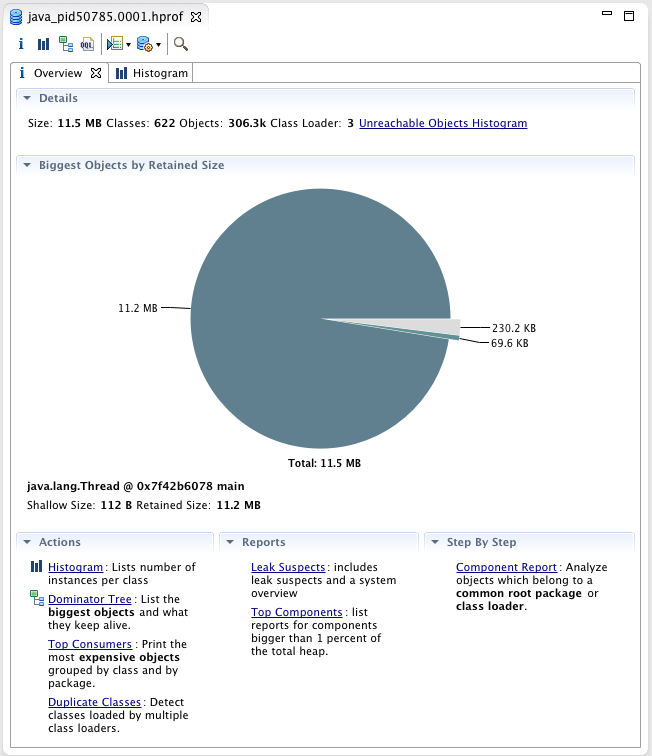
Finally, you can use the MAT’s **Acquire Heap Dump** action to choose an existing Java process on your local machine.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-12-08-57-pm/)

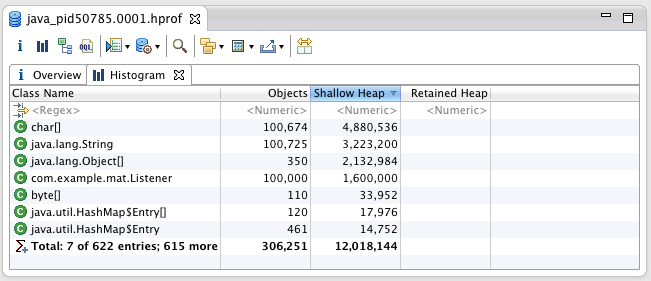
When you load a Heap dump for the first time, the MAT will index it. This make take a few minutes, but the results will be persisted so subsequent loads will be quick.

**2. Understanding the Histogram**

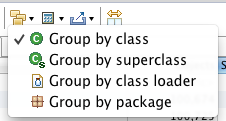
When you first acquire your heap dump, the MAT will show you an overview of the applications memory use.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-12-57-28-pm/)

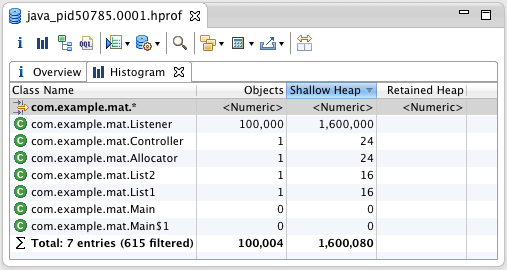
The pie chart in the middle shows you the biggest objects by **retained size.** That means if we could dispose a particular instance ofjava.lang.Thread we would save 11.2Mb, and over 90% of the memory used in this application. While that might look interesting, java.lang.Thread is unlikely the real problem here. To get a better sense of what objects currently exist, you can use the **Histogram**.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-12-59-18-pm/)

The histogram shows the number of instances of a particular class and how much memory each one uses. Of course, char[], String and Object[] are unlikely the problem. To help better organize this view, you can **group by** classloader or package. This will allow you to focus on your Objects.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-12-27-34-pm/)

The histogram can also be filtered using a regular expression. For example, we can show only classes that match the pattern **com.example.mat.\***

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-01-48-pm/)

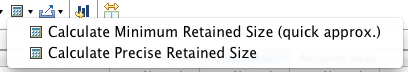
With this view, we can now start to see that 100,000 **Listener** Objects are alive in the system. We can also see the amount of memory each Object is using. There are two calculations, **Shallow Heap** and **Retained Heap.**  Shallow heap is the amount of memory consumed by one object. An Object requires 32 (or 64 bits, depending on the architecture) for each reference. Primitives such as integers and longs require 4 or 8 bytes, etc… While this can be interesting, the more useful metric is the **Retained Heap.**

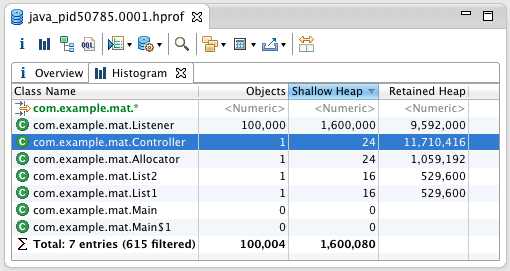
**3. Understanding the Retained Heap**

The **retained heap** shows the sum of the shallow heap size of all objects that would be removed when this object is garbage collected. For example, if an ArrayList held 100,000 items, and each item required 16 bytes, then removing the ArrayList would free 16 x 100,000 + X, where X is the shallow size of the ArrayList. (**Note:**this assumes that these objects are only being referenced by the ArrayList, and not elsewhere).

The **retained heap** is computed by adding the size all the objects in the **retained set.**A retained set of Xis the set of objects which would be removed by the GC when X is collected.

The retained heap can be calculated in two different ways, using the **quick approximation**or the **precise retained size.**

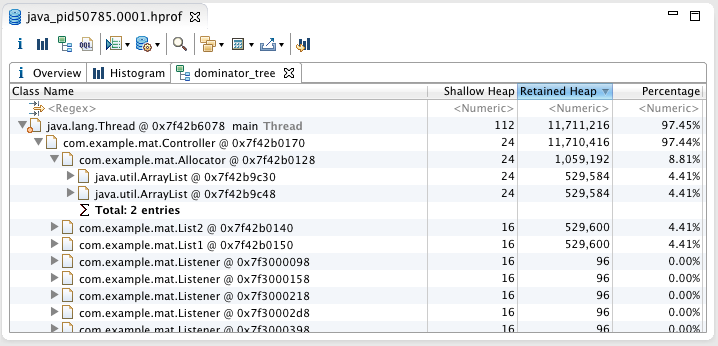
**[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-12-38-46-pm/)**

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-06-24-pm/)

By calculating the **Retained Heap** we can now see that com.example.mat.Controller is holding the majority of the memory, even though it’s only 24 bytes itself. By finding a way to free up the Controller, we can certainly get our memory problem under control.

**4. The Dominator Tree**

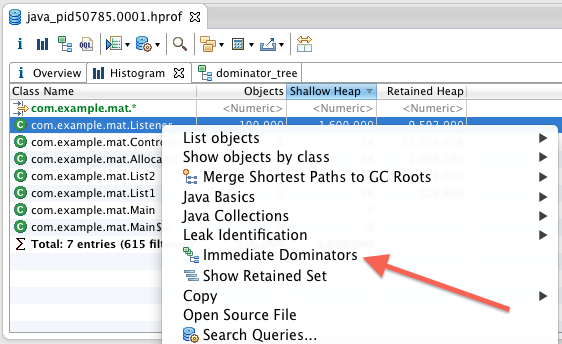
The key to understanding your retained heap, is looking at the**dominator tree.** The dominator tree is a **tree** produced by the complex object **graph** in your system. The dominator tree allows you to identify the largest memory graphs. An Object X is said to **dominate**an Object Y if every path from the Root to Y must pass through X. Looking at the dominator tree for our example, we can start to see where the bulk of our memory is leaking.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-22-02-pm/)

By looking at the dominator tree, we can easily see that it’s not thejava.lang.Thread that’s the problem, but rather the **Controller** and the **Allocator** that hold the memory. All 100,000 Listeners are being retained by the Controller. By either removing freeing these objects, or freeing the lists that they contain, we can likely improve our situation. There are a few useful properties of the dominator tree:

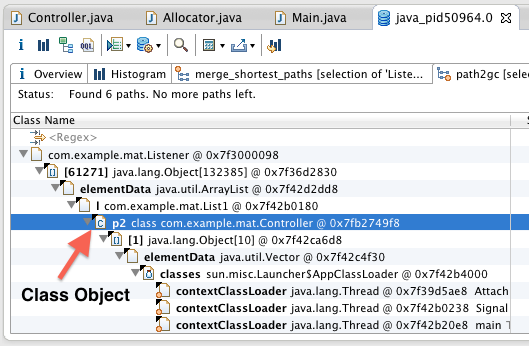
* All Objects that belongs to a subtree of X (com.example.mat.Controller in this case) are said to be in the retained set of X
* If X is the **immediate** **dominator**of Y (Controller is the immediate dominator of the Allocator), then the immediate dominator of X (which is java.lang.Thread in our example) also dominates Y.
* The parent child relationship in the tree do not necessarily correspond to the relationships in the Object graph

From the Histogram you can also choose a particular class and find all the objects that **dominate** the instances of this class.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-32-47-pm-2/)

**5. Exploring Paths to the GC Roots**

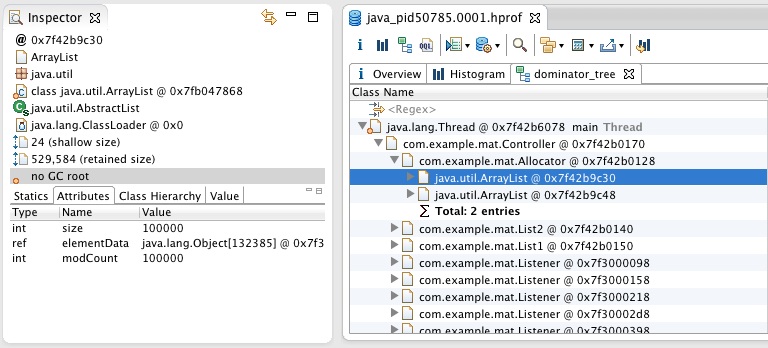
Sometimes you have a large collection of Objects that you’re certain your disposing of. Finding the the dominator may help, but often you want the exact path from that Object to the roots. For example, if I now properly dispose of my Controller, then certainly my memory problem should go away,  unfortunately they didn’t.  If I now choose an instance of the Listener, and look at **Paths To GC Roots,**I can see that the Controller class (**Note:**the class, not an Object) reference to a list of Listeners. This is because one of the list was declared **static**.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-54-55-pm/)

You can also explore all the incoming and outgoing references to an Object. This is very useful if you want to see all the references to a particular Object in the Object graph.

**6. The ‘Inspector’**

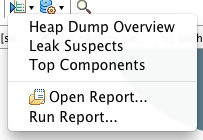
The **Inspector** provides detailed information about the currently selected Class or Object. In this example we can see that the currently selected ArrayList contains 100,000 elements and references an object array at memory location 0x7f354ea68.

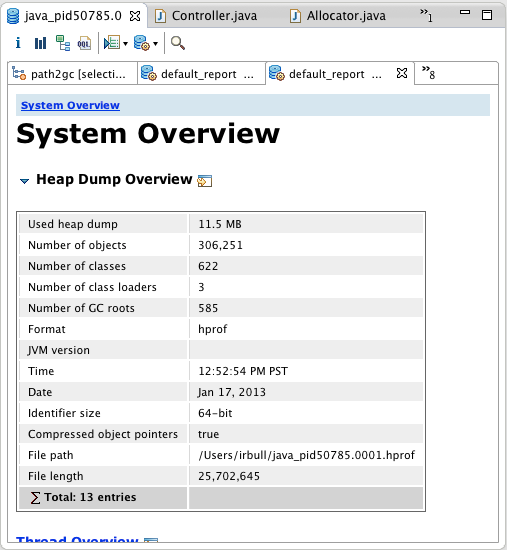
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-1-37-53-pm/)

Keeping the Inspector and the Snapshot linked will provide you with important statistics about all your selections.

**7. Common Memory Anti-Patterns**

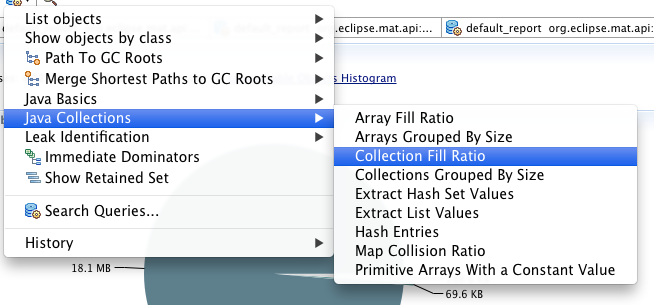
The MAT provides reports for common memory use anti-patterns. These can be used to get an idea of where memory leaks are occurring, or by looking for some low hanging fruit which can be cleaned up to help improve performance.

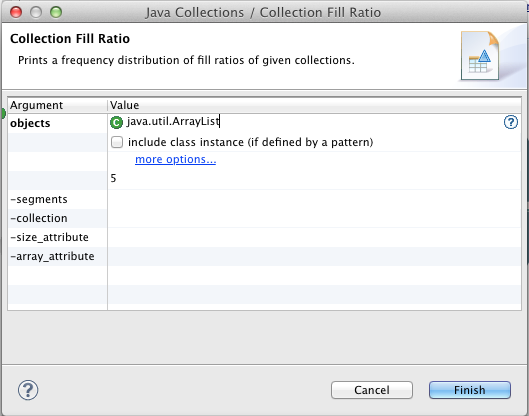
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-13-51-pm/)The **Heap Dump Overview** will show you detailed information about your Heap Dump and provide links to common tools (like the Histogram). Information such as the threads which were running, total number of Objects in the system, size of the heap, and are also shown.

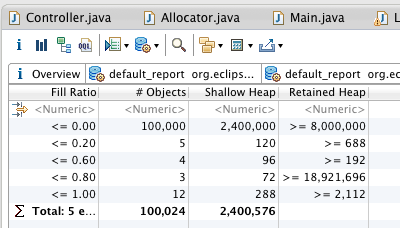
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-16-30-pm/)

The **Leak Suspects** report displays possible memory leaks, and provides links to the tools and charts to analyze these findings.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-18-29-pm/)Another common **anti-pattern** is the use of a large number of collections, with very few entries in each one. For example, if our listeners each had an array of notifiers (items that need to be notified of certain events), but these notifiers were only used occasionally, we would end up wasting a lot of space. The **Java Collections** tools can help with these problems.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-27-19-pm/)

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-31-25-pm/)

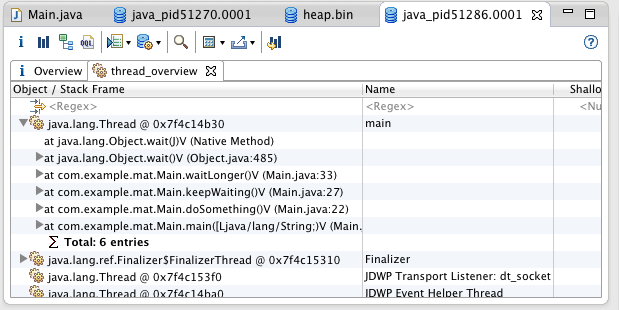
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-2-28-41-pm/)

By running a **Collection -> Fill Ratio Report** we can see that there are 100,000 ArrayLists that are empty. If were allocated these in a lazy manner (only when needed), we would save almost 8Mb.

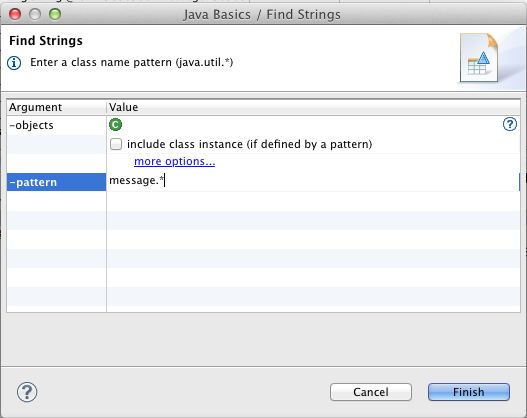
We can also use Collection Analysis to see **array fill ratios**, **collection size statistics** and **map collision ratios**.

**8. Java Tools**

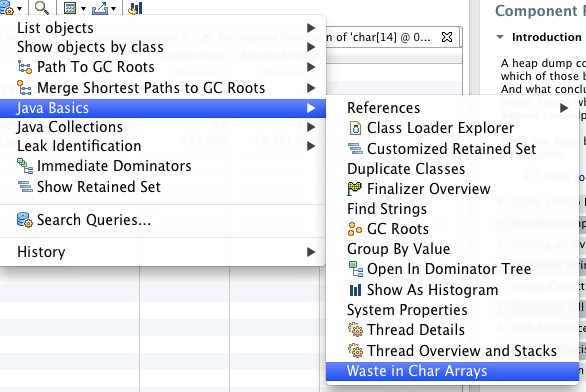
The MAT has a number of built in tools to generate reports tailored to the details of the Java runtime. For example, the **Threads and Stacks** report will show details about all the treads in the system. You can see the local variables which are currently kept alive on each stack.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-3-26-48-pm/)

You can **Find all the Strings** in the system that match a particular pattern:

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-3-30-49-pm-2/)

Or even find Strings in the system which contain wasted space in their character arrays (often due to repeated use of substring).

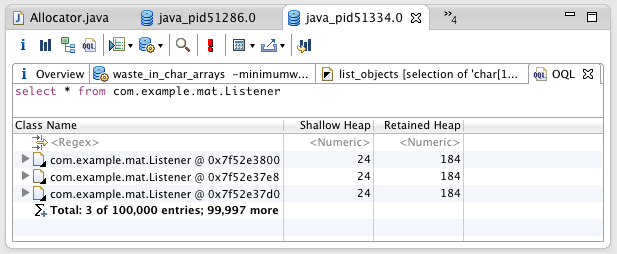
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-4-18-14-pm/)

**9. Object Query Language**

As we’ve shown, the Eclipse Memory Analyzer has a lot of tools to help track both memory leaks and excessive memory use. While most memory problems can likely be addressed using the techniques described above, a Heap Dump contains much more information. The Object Query Language  (OQL) allows you to build your own reports based on the results of a Heap Dump.

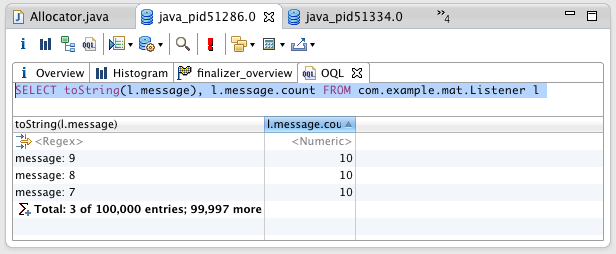
The OQL is an SQL-like language. Just think of **Classes** as tables,**Objects** as rows and **Fields** as columns. For example. to show all Objects of type com.example.mat.Listener, you would simply write:

select \* from com.example.mat.Listener

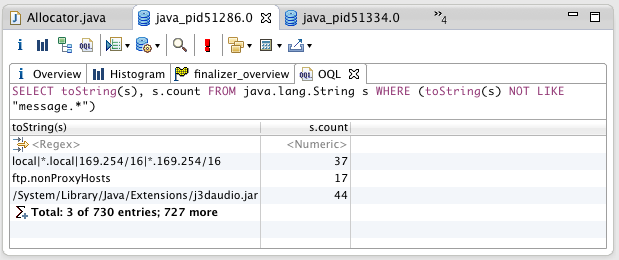
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-4-23-57-pm/)

Columns can be configured using different fields, such as:

SELECT toString(l.message), l.message.count FROM com.example.mat.Listener l

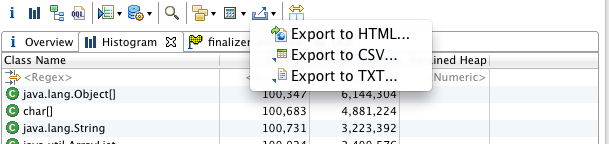
[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-4-28-07-pm/)And finally, the WHERE clause can be used to specify particular criteria, such as all the Strings in the system which are not of the format “message:.\*”

SELECT toString(s), s.count FROM java.lang.String s WHERE (toString(s) NOT LIKE "message.\*")

**[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-4-34-53-pm/)**

**10. Exporting your results**

The memory analyzer is a great tool for creating reports about the state of an application’s memory. A Heap Dump contains valuable information about the state of your system, and the MAT provides the tools needed to access this data. However, like with many open tools, if something is missing you’re not locked in, nor are you out of luck. With the MAT you can export your results to several different formats including HTML, CSV or even Plain Text. You can then use your favorite spreadsheet program (or your own tool) to continue your analysis.

[](http://eclipsesource.com/blogs/2013/01/21/10-tips-for-using-the-eclipse-memory-analyzer/screen-shot-2013-01-17-at-4-39-34-pm/)

The **Eclipse Memory Analyzer** is a powerful tool, one all Java Developers should be familiar with. Tracking memory leaks and other memory related problems is often challenging, but hopefully with the MAT you can get to the root of your problems relatively quickly.

**You may also like...**

* [**Creating Tomcat heap dumps on Windows**](http://eclipsesource.com/blogs/2013/01/31/creating-tomcat-heap-dumps-on-windows/)
* [**Effective Java Debugging with Eclipse**](http://eclipsesource.com/blogs/2013/01/08/effective-java-debugging-with-eclipse/)

### Eclipse Memory Analyzer 分析内存泄露 - 21004129 - 博客园

http://www.cnblogs.com/walson/p/3913055.html

### 一次使用Eclipse Memory Analyzer分析Tomcat内存溢出 - - ITeye技术网站

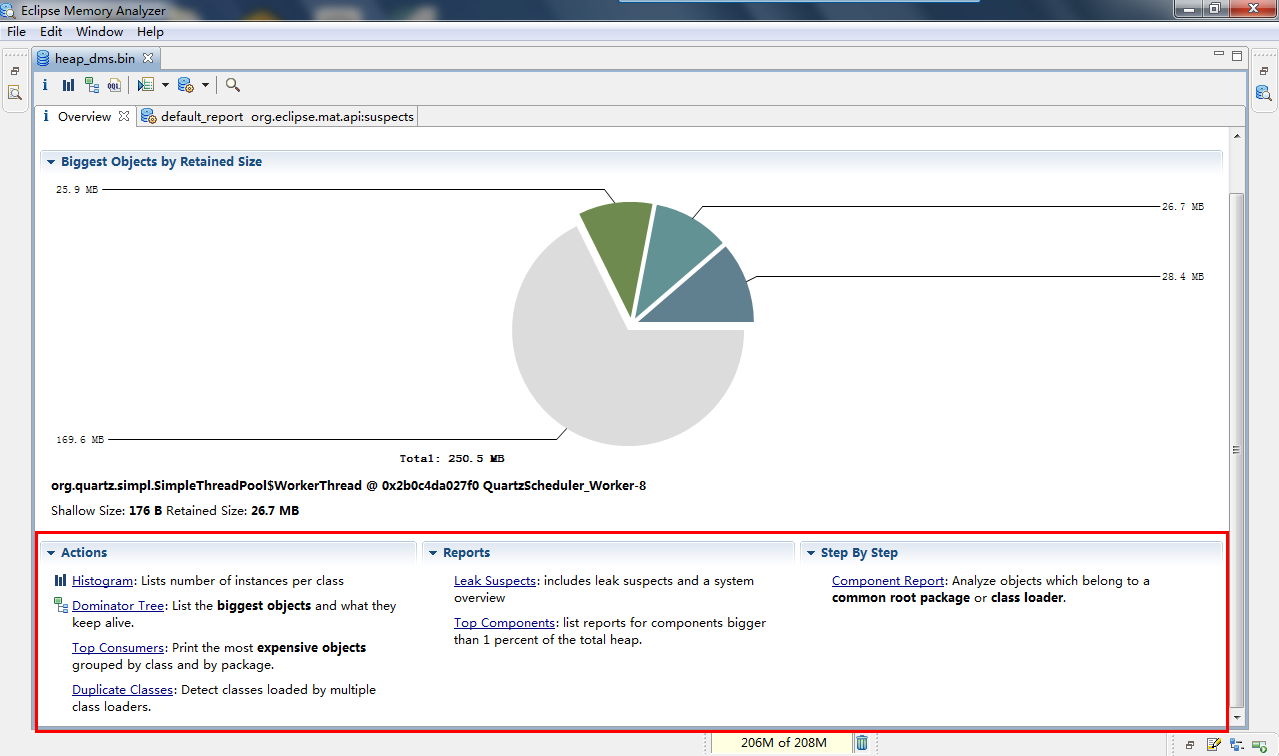
http://tivan.iteye.com/blog/1487855

在平时开发、测试过程中、甚至是生产环境中，有时会遇到OutOfMemoryError，Java堆溢出了，这表明程序有严重的问题。我们需要找造成OutOfMemoryError原因。一般有两种情况：

1、内存泄露，对象已经死了，无法通过垃圾收集器进行自动回收，通过找出泄露的代码位置和原因，才好确定解决方案；  
2、内存溢出，内存中的对象都还必须存活着，这说明Java堆分配空间不足，检查堆设置大小（-Xmx与-Xms），检查代码是否存在对象生命周期太长、持有状态时间过长的情况。  
以上是处理Java堆问题的思路，具体是怎么进行分析，这里介绍的是使用Eclipse Memory Analyzer tool(MAT)工具分析的过程。

生成dump文件  
  
     通过jvm参数--XX:-HeapDumpOnOutOfMemoryError可以让JVM在出现内存溢出是Dump出当前的内存转储快照；  
     或者，用jmap生产dump文件，win通过任务管理器查看tomcat的进程pid，linux用ps命令查看进程pid,然后用jmap命令（Java5：jmap -heap:format=b <pid>；Java6：jmap -dump:format=b,file=HeapDump.bin <pid>）。  
      
     我这里使用的是，我一生产环境项目，运行一段时间大概3周的样子，就会报OutOfMemoryError。（ps：这个项目出现这种情况已经有好长一段时间了，我们之前的做法是定期的重启tomcat，没有去分析它的原因。）JDK64位主要参数：-Xmx3078M -Xms3078M -XX:PermSize=1024M -XX:MaxPermSize=1024M，内存还是蛮大的。

MAT安装与介绍  
     下载地址：http://www.eclipse.org/mat/downloads.php。  
     通过MAT打开dump出来的内存文件，打开后如下图：



     从上图可以看到它的大部分功能。  
     1. Histogram可以列出内存中的对象，对象的个数以及大小。  
     2. Dominator Tree可以列出那个线程，以及线程下面的那些对象占用的空间。  
     3.Top consumers通过图形列出最大的object。  
     4.Leak Suspects通过MA自动分析泄漏的原因。

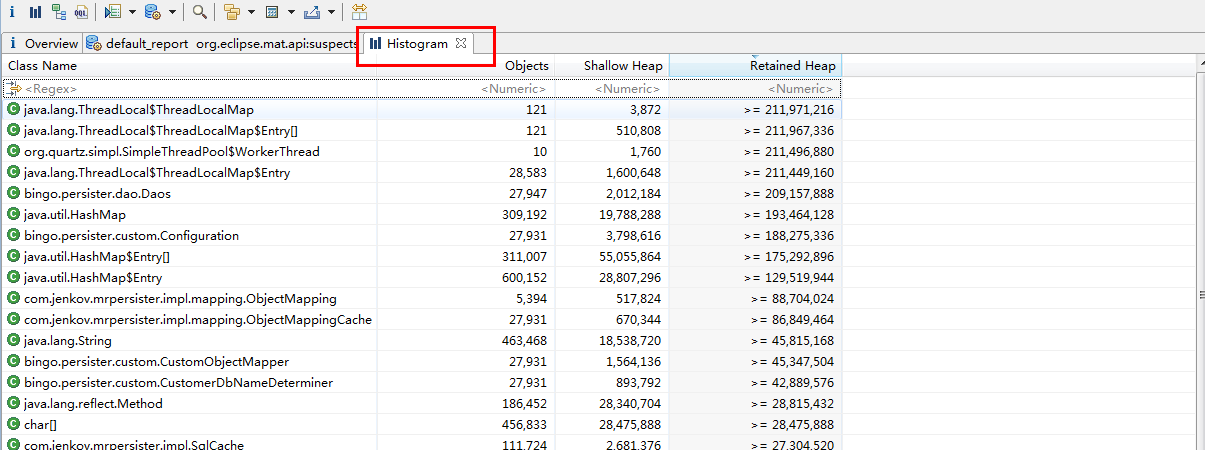
     Histogram如下图：

     Objects:类的对象的数量。

     Shallow size：就是对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量（不是成员变量的值）的总和。

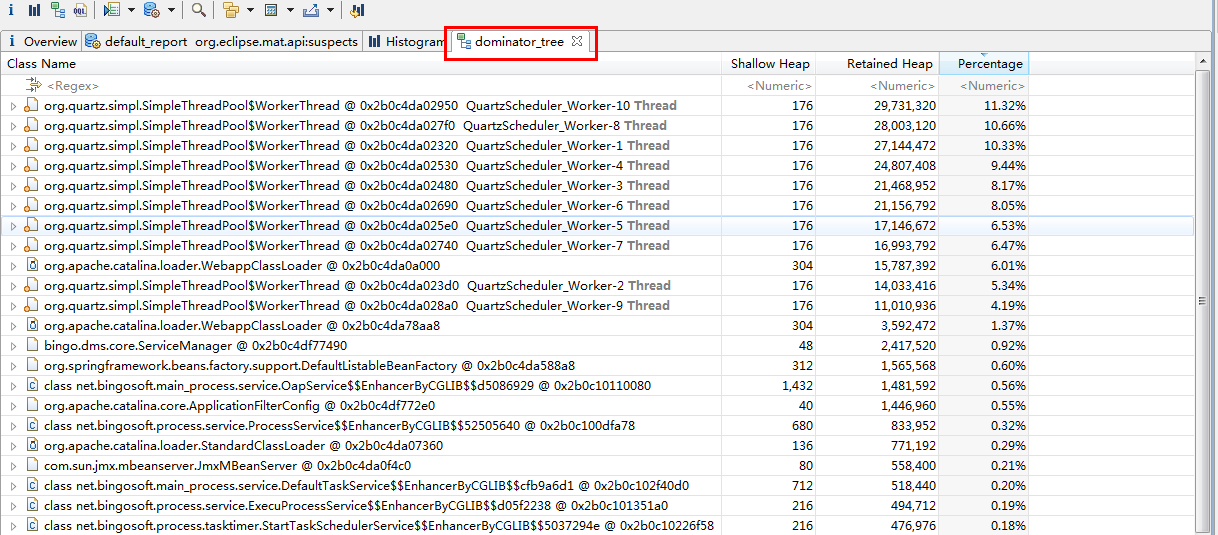
     Retained size：是该对象自己的shallow size，加上从该对象能直接或间接访问到对象的shallow size之和。换句话说，retained size是该对象被GC之后所能回收到内存的总和。

     我们发现ThreadLocal和bingo.persister.dao.Daos类的对象占用了很多空间。



     Dominator Tree如下图：

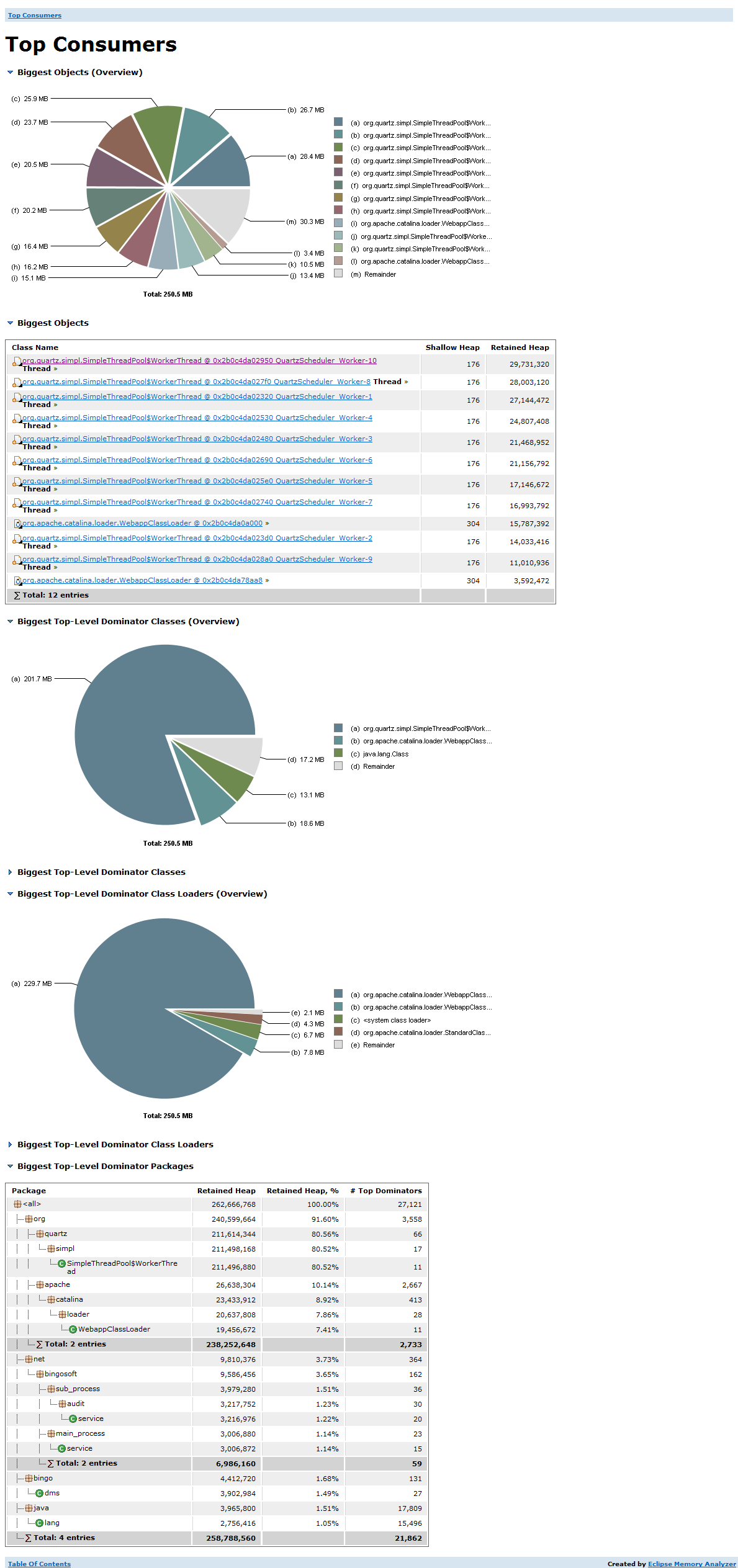
     我们发现quartz的定时器的工作线程（10个）占了很多的内存空间



     Top consumers如下图：

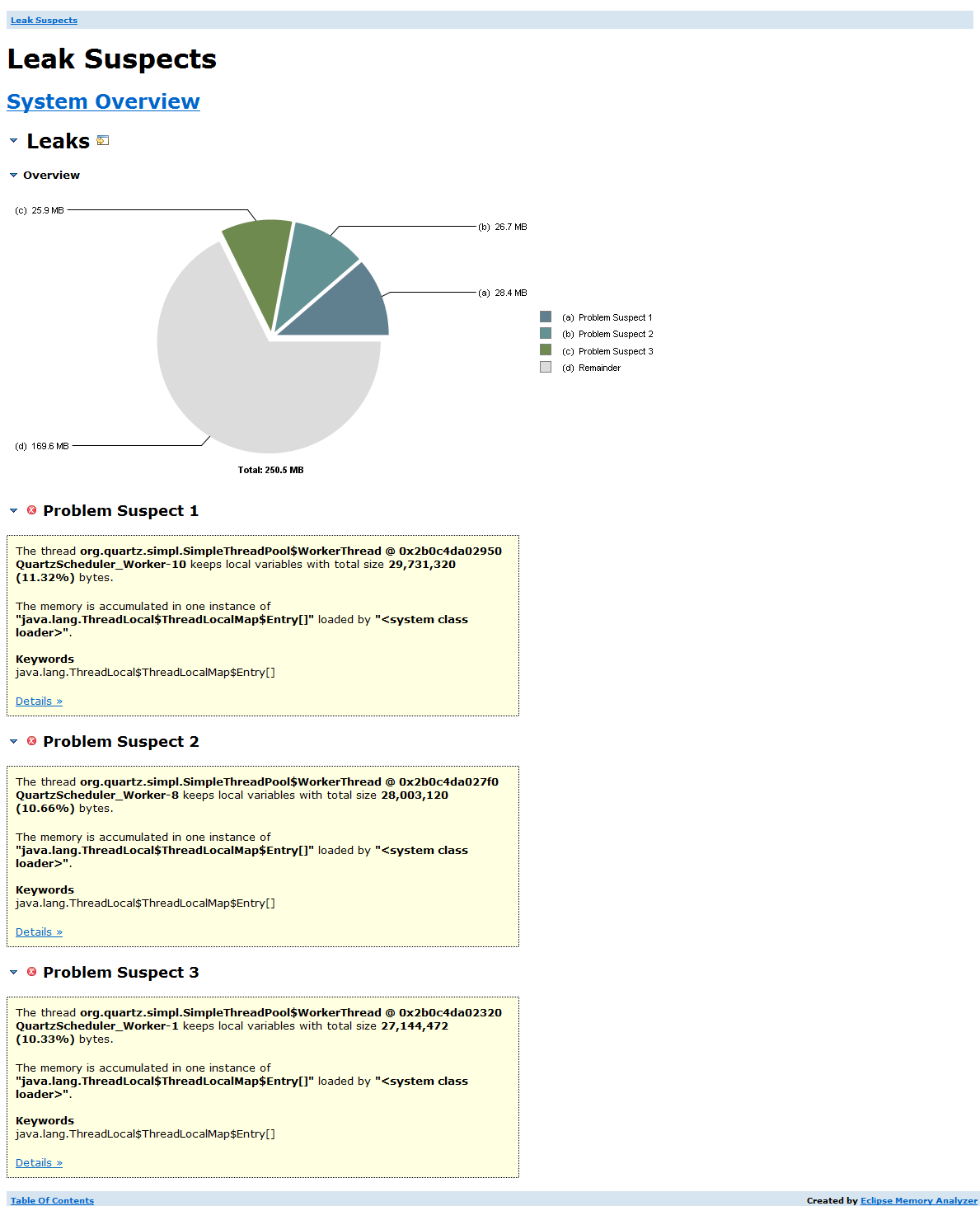
     这里显示了内存中最大的对象有哪些，他们对应的类是哪些，类加载器classloader是哪些。

     有些时候，我们在这里就可以看到代码泄露的位置。

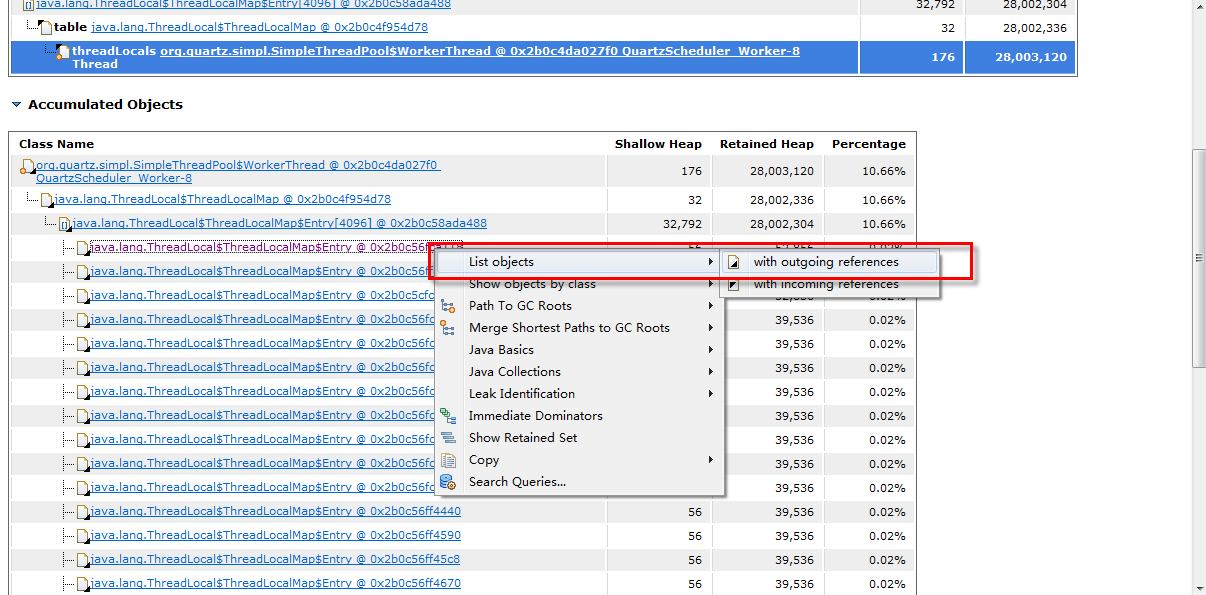
     Leak Suspects如下图：

     从那个饼图，该图深色区域被怀疑有内存泄漏，可以发现整个heap才250M内存，深色区域就占了34%。后面的描述，告诉我们quartz线程占用了大量内存，并指出system class loader加载的"java.lang.ThreadLocal"实例的内存中聚集（消耗空间），并建议用关键字"java.lang.ThreadLocal$ThreadLocalMap$Entry[]"进行检查。所以，MAT通过简单的报告就说明了问题所在。

通过Leak Suspects的Problem Suspect 1点击【Details »】，

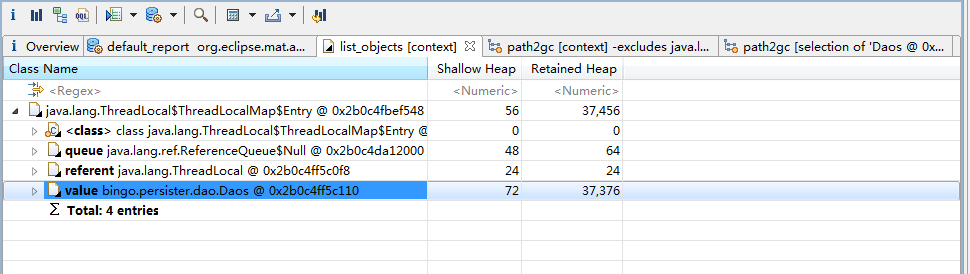
如下图如下图所示的上下文菜单中选择 List objects -> with outgoning references, 查看ThreadLocal都应用了些什么对象。

现在看到ThreadLocal中引用的对象如下图：

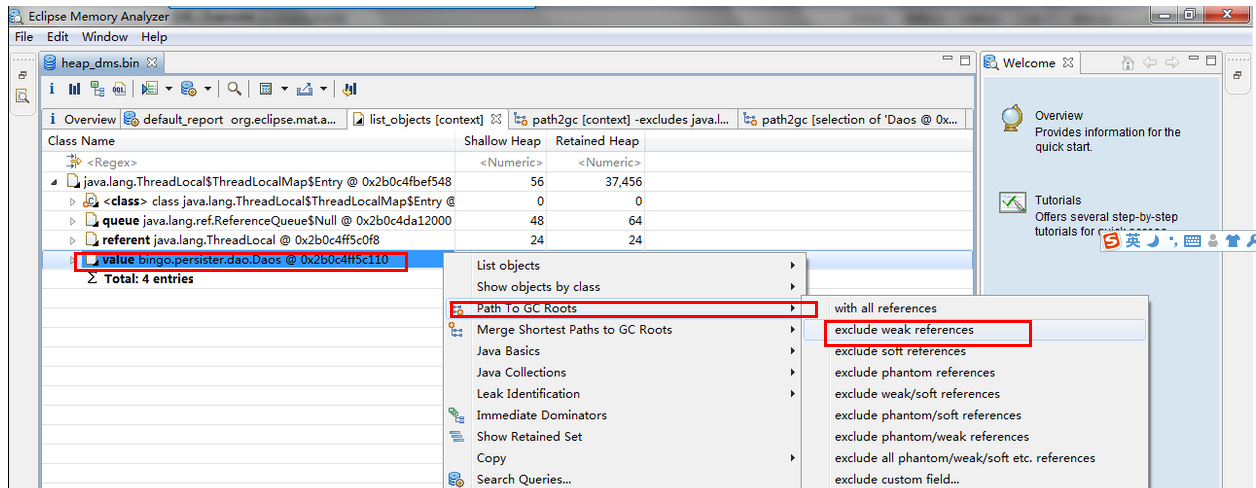
是dao对象

*ps：该dao对象包含一个轻量级的ORM关系内容，所以Retained size比较大*。

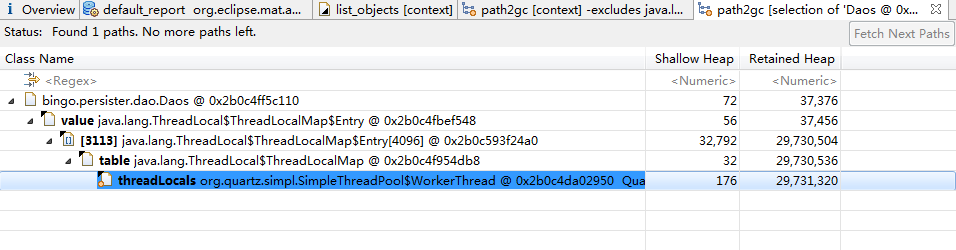
  
 

下面继续查看dao的gc ROOT

如下图所示的上下文菜单中选择 Path To GC Roots -> exclude weak references, 过滤掉弱引用，因为在这里弱引用不是引起问题的关键。



从下图中，可以看到在org.quartz.simpl.SimpleThreadPool中保存了daos的引用。所以可以得出是是因为定时器在运行的过程中持有大量的Daos对象应起了内存泄露。为什么会有那么多的Daos呢,Daos不是一个无状态的单例的、可以重用的吗？继续查看spring配置文件发现Daos的bean配置成scope="prototype"，导致定时任务又是每次调用都生产新的Daos实例。由于是Daos是无状态的，修改为单例的，问题解决。

以上是通过MAT分析Tomcat应用程序，找到内存泄露的原因，并解决。