转载自:<https://my.oschina.net/biezhi/blog/286223>

在工作中可能会遇到内存溢出这种灾难性的问题，那么程序肯定是存在问题，找出问题至关重要，上一篇文章讲了jmap命令的使用方法，当然用jmap导出的文件我们也看不懂啊，那就交给memory analyzer(mat)这个工具，让他帮助我们来观察程序的内存分布情况吧。

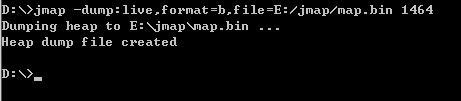
* [1. 用jmap生成堆信息](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h2_1)
* [2. 将堆信息导入到mat中分析](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h2_2)
* [3. 生成分析报告](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h2_3)
* [Histogram](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h4_4)
* [Dominator Tree](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h4_5)
* [Top consumers](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h4_6)
* [Leak Suspects](http://my.oschina.net/biezhi/blog/286223" \l "OSC_h4_7)

    MAT 不是一个万能工具，它并不能处理所有类型的堆存储文件。但是比较主流的厂家和格式，例如 Sun, HP, SAP 所采用的 HPROF 二进制堆存储文件，以及 IBM 的 PHD 堆存储文件等都能被很好的解析。下面来看看要怎么做呢，也许对你有用。官方文档：<http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.mat.ui.help/welcome.html>

造成OutOfMemoryError原因一般有2种：

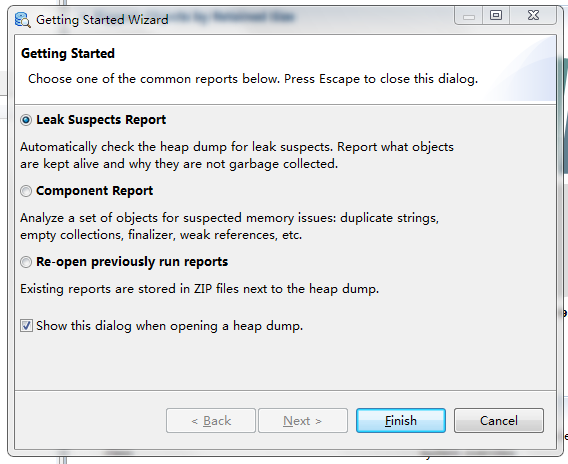
****1、内存泄露，对象已经死了，无法通过垃圾收集器进行自动回收，通过找出泄露的代码位置和原因，才好确定解决方案；  
2、内存溢出，内存中的对象都还必须存活着，这说明Java堆分配空间不足，检查堆设置大小（-Xmx与-Xms），检查代码是否存在对象生命周期太长、持有状态时间过长的情况。****

## **1. 用jmap生成堆信息**

[](http://static.oschina.net/uploads/space/2014/0702/100252_ot9E_1767531.png)

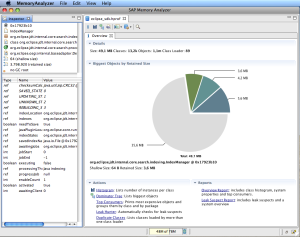
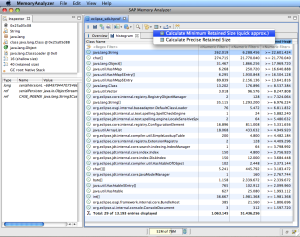
这样在E盘的jmap文件夹里会有一个map.bin的堆信息文件

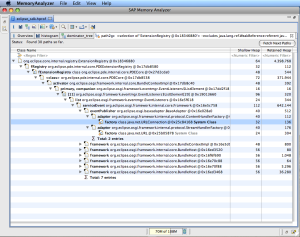
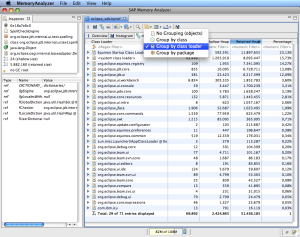
## **2. 将堆信息导入到mat中分析**

[](http://static.oschina.net/uploads/space/2014/0702/101421_bdMx_1767531.png)

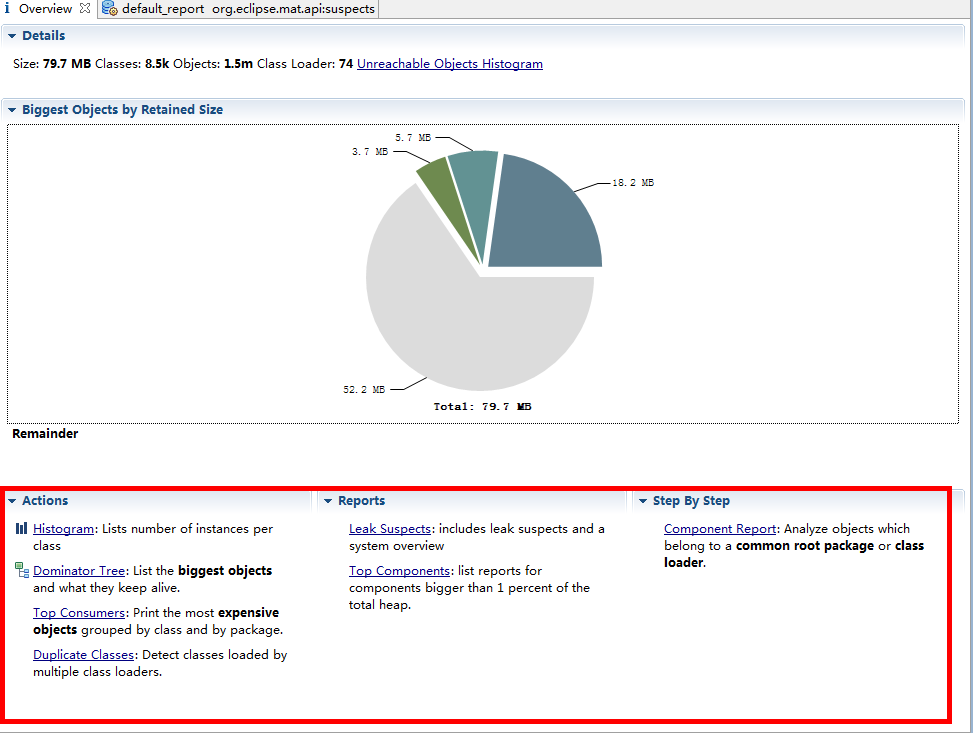
## **3. 生成分析报告**

    mat可以为我们生成多个报告：

[](http://static.oschina.net/uploads/space/2014/0702/140753_cUoj_1767531.png)    [](http://static.oschina.net/uploads/space/2014/0702/140754_Uq2P_1767531.png)

[](http://static.oschina.net/uploads/space/2014/0702/140755_JHbD_1767531.png)    [](http://static.oschina.net/uploads/space/2014/0702/140755_iAe6_1767531.png)

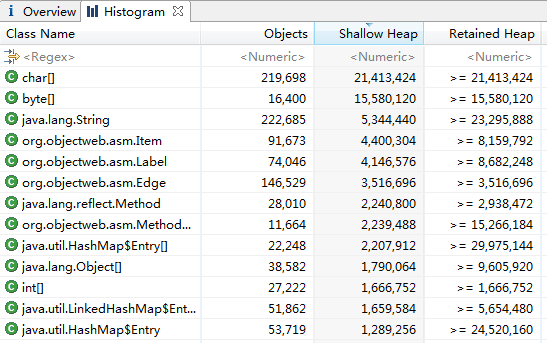
    下面来看看生成的这些数据对我们有什么帮助

[](http://static.oschina.net/uploads/space/2014/0702/114056_IgFu_1767531.png)

    从上图可以看到它的大部分功能，在饼图上,你会发现转储的大小和数量的类,对象和类加载器。  
正确的下面,饼图给出了一个印象最大的对象转储。移动你的鼠标一片看到对象中的对象的细节检查在左边。下面的Action标签中：

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****

**[](http://static.oschina.net/uploads/space/2014/0702/120039_qSi5_1767531.png)**

****Class Name ： 类名称，java类名****

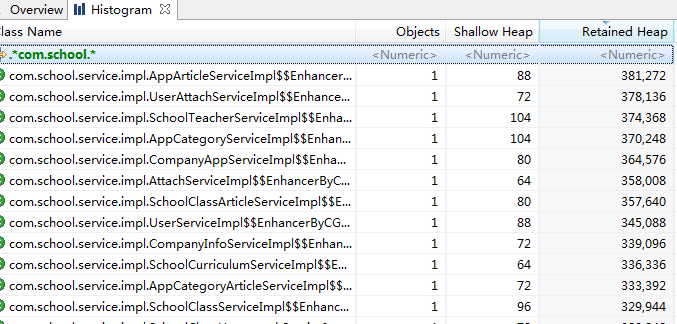
****Objects ： 类的对象的数量，这个对象被创建了多少个****

****Shallow Heap ：一个对象内存的消耗大小，不包含对其他对象的引用****

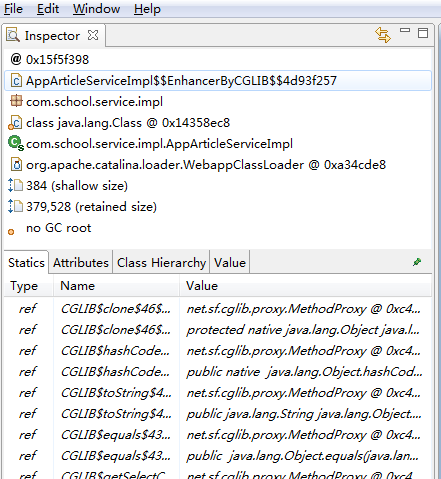
****Retained Heap ：是shallow Heap的总和，也就是该对象被GC之后所能回收到内存的总和****

一般来说,****Shallow Heap****堆中的对象是它的大小和保留内存大小相同的对象是堆内存的数量时,将释放对象被垃圾收集。  
保留设置一组主要的对象,例如一个特定类的所有对象,或所有对象的一个特定的类装入器装入的类或者只是一群任意对象,是释放的组对象如果所有对象的主要设置变得难以接近的。保留设置包括这些对象以及所有其他对象只能通过这些对象。保留大小是总堆大小中包含的所有对象的保留。*摘自eclipse*

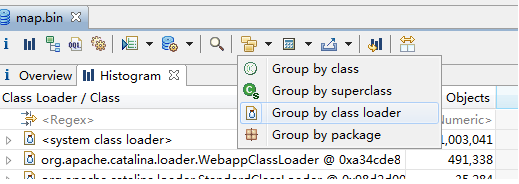
关于的详细讲解，建议大家查看[Shallow heap & Retained heap](http://bjyzxxds.iteye.com/blog/1532937)，这是个很重要的概念。

[](http://static.oschina.net/uploads/space/2014/0702/143746_3JnJ_1767531.png)

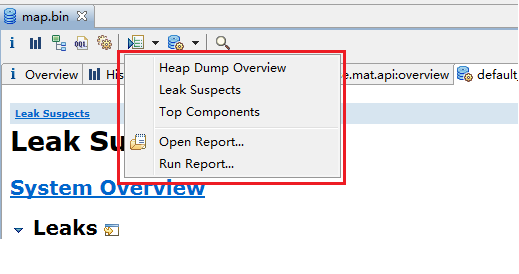
这儿借助工具提供的regex正则搜索一下我们自己的类，排序后看看哪些相对是占用比较大的。

[](http://static.oschina.net/uploads/space/2014/0702/144056_NLi7_1767531.png)

左边可以看到类的详细使用，比如所属包，父类是谁，所属的类加载器，内存地址，占用大小和回收情况等

[](http://static.oschina.net/uploads/space/2014/0702/144418_r62H_1767531.png)

这儿有个工具可以根据自己的需求分组查找，默认根据class分组，类似我们sql里的group by了~~

[](http://static.oschina.net/uploads/space/2014/0702/145027_tqXs_1767531.png)

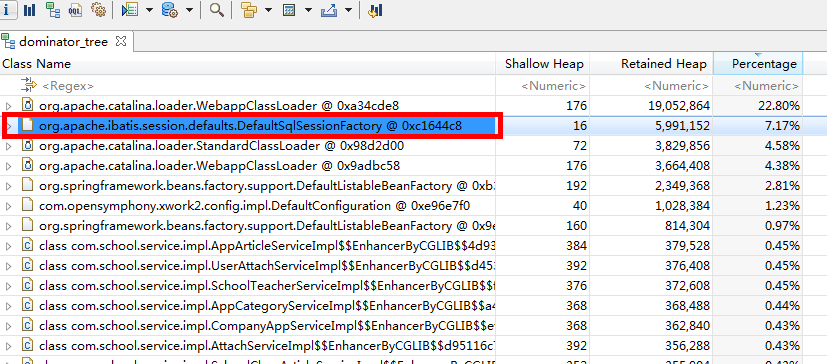
这里可以看到上面3个选项，分别生成overview、leak suspects、top components数据，但是这儿生成的不是图表，如果要看图表在[IMG_268](http://static.oschina.net/uploads/space/2014/0702/150452_97xc_1767531.png)（Overview）中的Action标签里点击查看。

这个是Overview中的 Heap Dump Overview视图，从工具栏中点开，这是一个全局的内存占用信息

|  |  |
| --- | --- |
| Used heap dump | 79.7 MB |
| Number of objects | 1,535,626 |
| Number of classes | 8,459 |
| Number of class loaders | 74 |
| Number of GC roots | 2,722 |
| Format | hprof |
| JVM version |  |
| Time | 格林尼治标准时间+0800上午9时20分37秒 |
| Date | 2014-7-2 |
| Identifier size | 32-bit |
| File path | E:\jmap\map.bin |
| File length | 108,102,005 |
| Total: 12 entries |  |

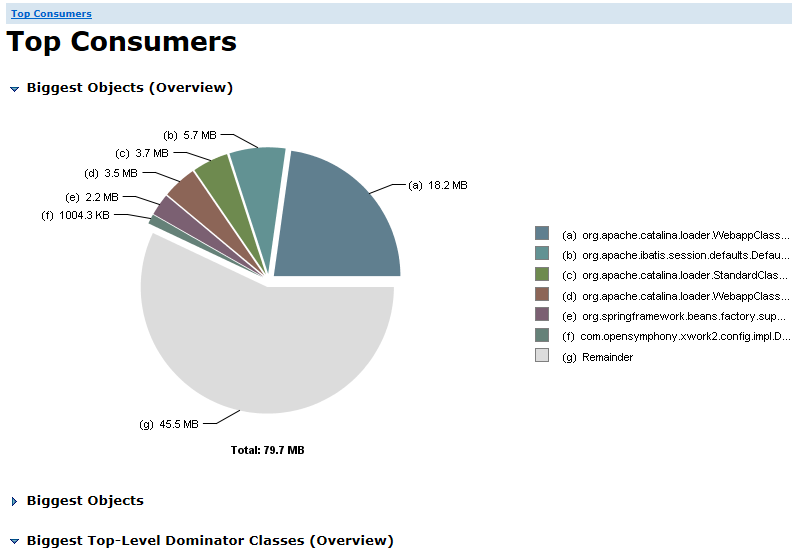
然后可以点开SystemProperties和Thread Overview进行查看，我这里就不贴了内容比较多。

****Dominator Tree****

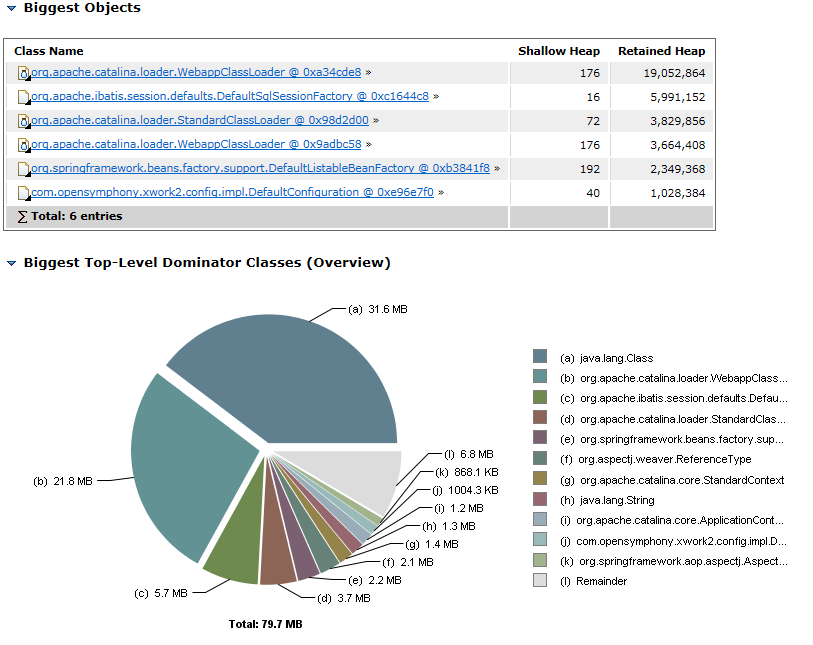
[](http://static.oschina.net/uploads/space/2014/0702/145926_K3ET_1767531.png)

我们可以看到ibatis占了较多内存

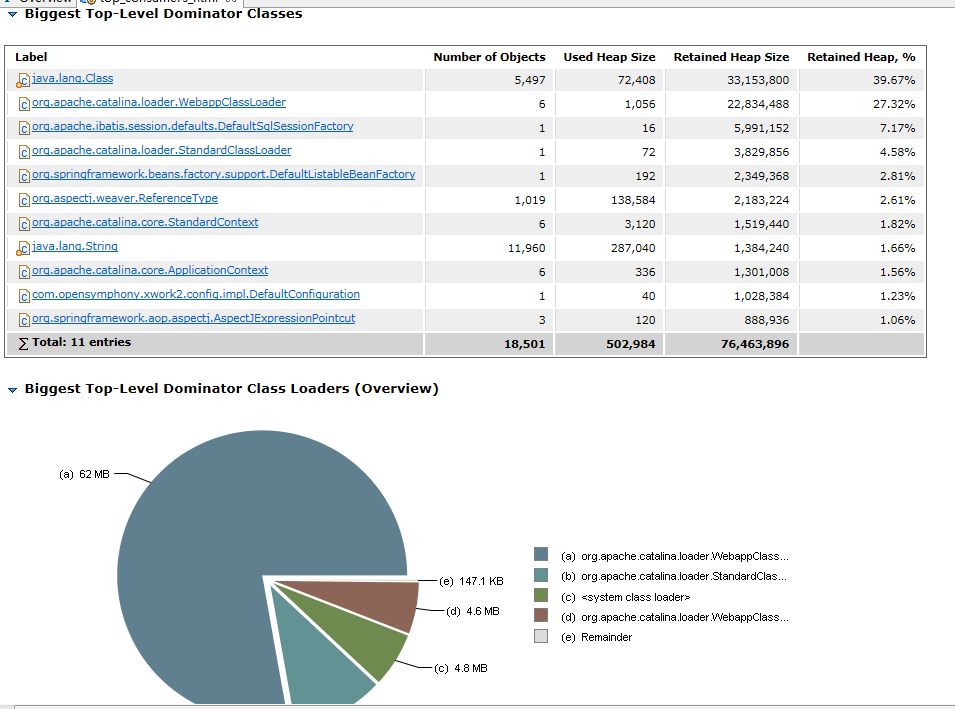
Top consumers

**[](http://static.oschina.net/uploads/space/2014/0702/150647_zCDB_1767531.png)**

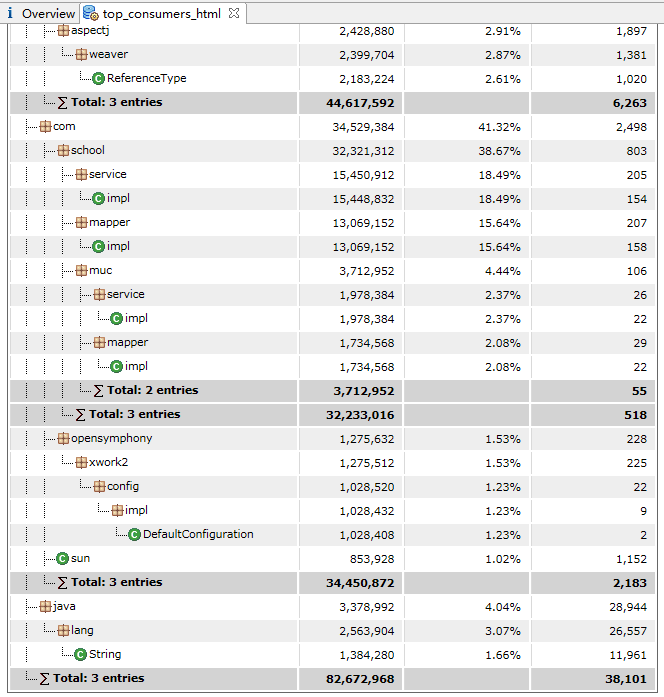
****这张图展示的是占用内存比较多的对象的分布，下面是具体的一些类和占用。****

**[](http://static.oschina.net/uploads/space/2014/0702/150945_hmjj_1767531.png)**

****按等级分布的类使用情况，其实也就是按使用次数查看，java.lang.Class被排在第一****

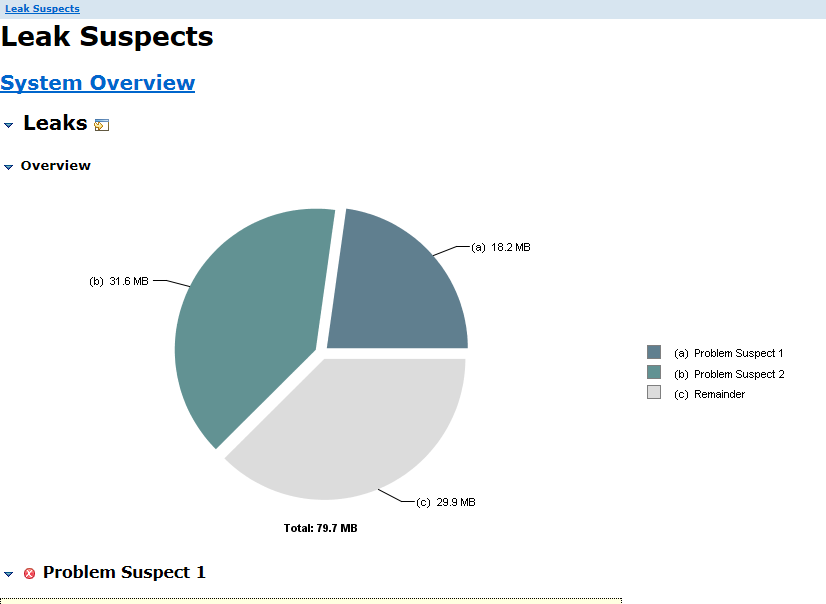
**[](http://static.oschina.net/uploads/space/2014/0702/151054_d2Q1_1767531.png)**

还有一张图是我们比较关心的，那就是按包名看占用，根据包我们知道哪些公共用的到jar或自己的包占用

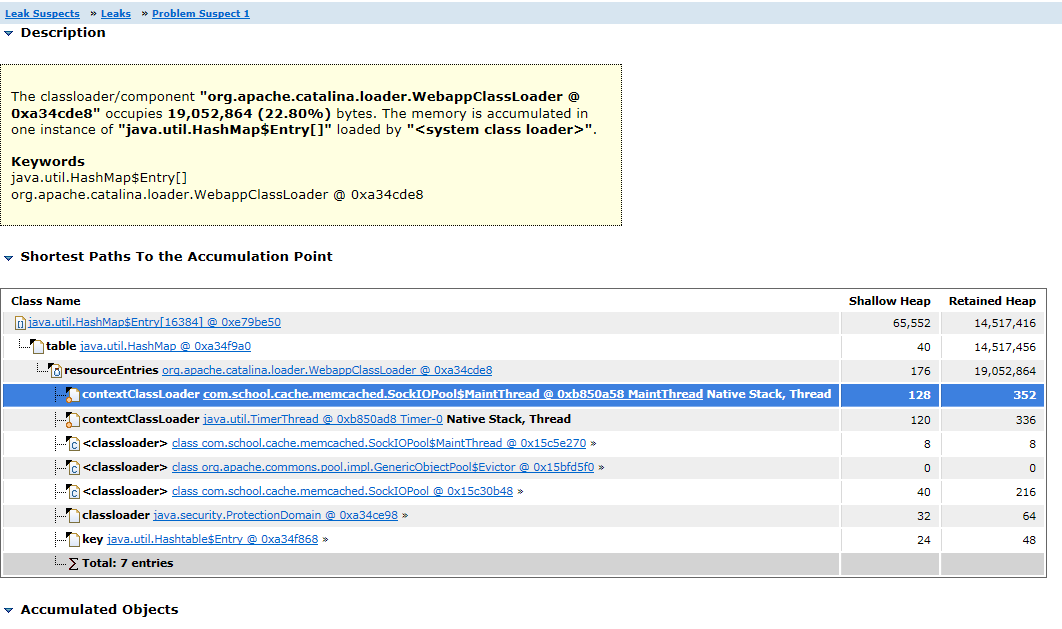
[](http://static.oschina.net/uploads/space/2014/0702/151609_GMMg_1767531.png)

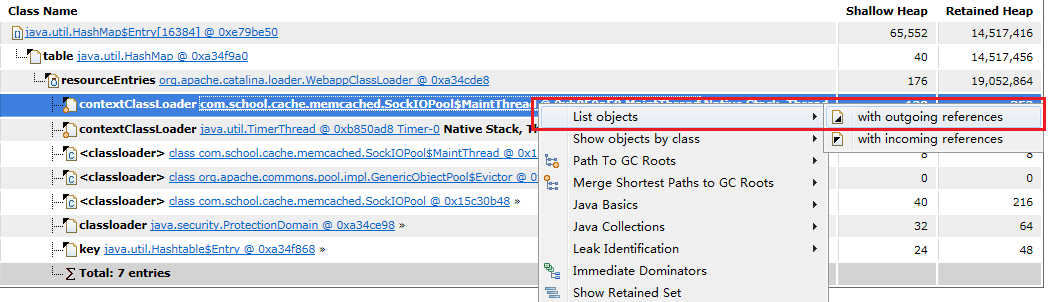
这样就可以看到包和包中哪些类的占用比较高。

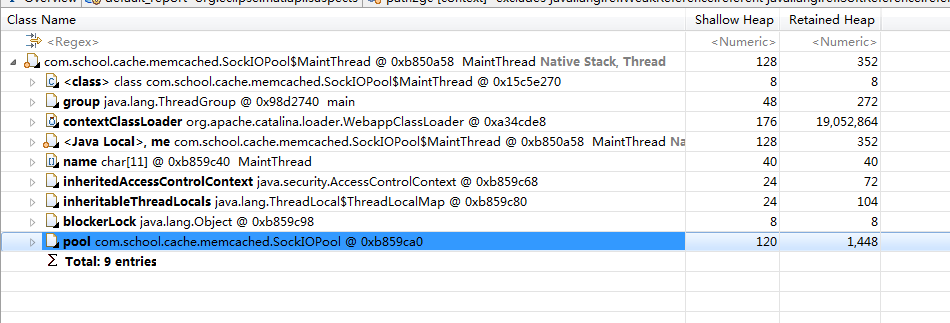
Leak Suspects

**[](http://static.oschina.net/uploads/space/2014/0702/151929_mYHX_1767531.png)**

从这份报告，看到该图深色区域被怀疑有内存泄漏，可以发现整个heap只有79.7M内存，深色区域就占了62%。所以，MAT通过简单的报告就说明了项目是有可疑代码的，具体点开详情来找到类，

[](http://static.oschina.net/uploads/space/2014/0702/152211_lLLc_1767531.png)

[](http://static.oschina.net/uploads/space/2014/0702/155719_7qTp_1767531.png)

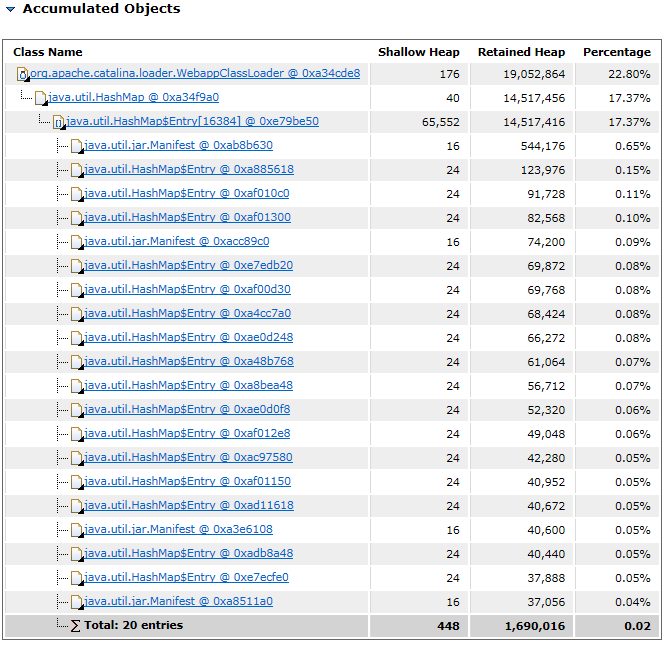
[](http://static.oschina.net/uploads/space/2014/0702/160132_F64s_1767531.png)

点击鼠标，在List Objects-> with outgoing references下可以查看该类都引用了什么对象，由此查看是否因为其他对象导致的内存问题。

下面继续查看pool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。

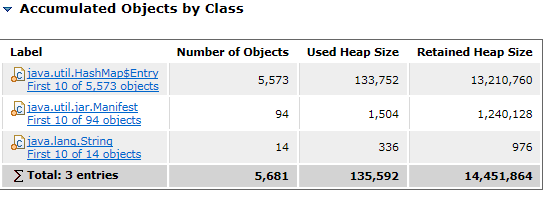
进入查看即可，我这儿的代码没有问题，就不用贴了。

[](http://static.oschina.net/uploads/space/2014/0702/155212_BekS_1767531.png)

The classloader/component ****"org.apache.catalina.loader.WebappClassLoader @ 0xa34cde8"**** occupies ****19,052,864 (22.80%)**** bytes. The memory is accumulated in one instance of ****"java.util.HashMap$Entry[]"**** loaded by ****"<system class loader>"****.  
  
****Keywords****  
java.util.HashMap$Entry[]  
org.apache.catalina.loader.WebappClassLoader @ 0xa34cde8

这段话是在工具中提示的，他告诉我们WebappClassLoader占了****19,052,864****字节的容量，这是tomcat的类加载器，JDK自带的系统类加载器中占用比较多的是HashMap。这个其实比较正常，大家经常用map作为存储容器。

除了在上一页看到的描述外，还有Shortest Paths To the Accumulation Point和Accumulated Objects部分，这里说明了从GC root到聚集点的最短路径，以及完整的reference chain。观察Accumulated Objects部分，java.util.HashMap的retained heap(size)最大，所以明显类实例都聚集在HashMap中了。

[](http://static.oschina.net/uploads/space/2014/0702/155256_saNb_1767531.png)

来看看Accumulated Objects by Class区域，这里能找到被聚集的对象实例的类名。java.util.HashMap类上头条了，被实例化了5573次，从这儿看出这个程序不存在什么问题，因为这个数字是比较正常的，但是当出问题的时候我们都会看到比较大的自定义类会在前面，而且占用是相当高。

当然，mat这个工具还有很多的用法，这里把我了解的分享给大家，不管如何，最终我们需要得出系统的内存占用，然后对其进行代码或架构，服务器的优化措施！