Android性能专项测试之MAT

分类：

Android性能（13）

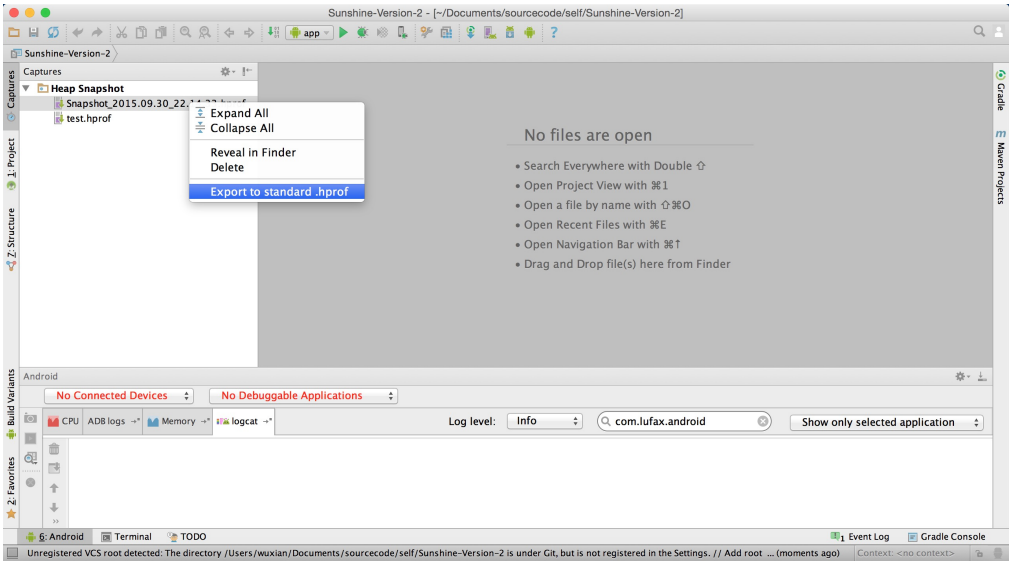
目录(?) [+]

参考文章:
Android内存优化之二：MAT使用进阶
Android内存优化之一：MAT使用入门
MAT中的Bitmap图像
10 Tips for using the Eclipse Memory Analyzer

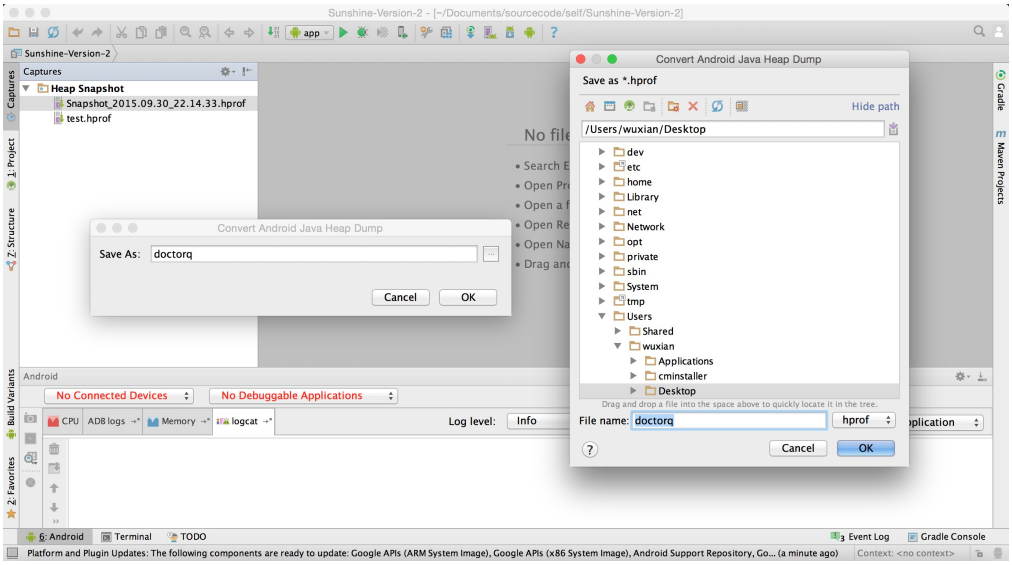
MAT使用

MAT工具全称为Memory Analyzer Tool，一款详细分析Java堆内存的工具，该工具非常强大，为了使用该工具，我们需要hprof文件，该文件我们在之前的Heap Snapshot工具的时候，我们就生成了该文件。但是该文件不能直接被MAT使用，需要进行一步转化，可以使用hprof-conv命令来转化，但是Android Studio可以直接转化,转化方法如下：

1.选择一个hprof文件，点击右键选择 `Export to standard .hprof` 选项。

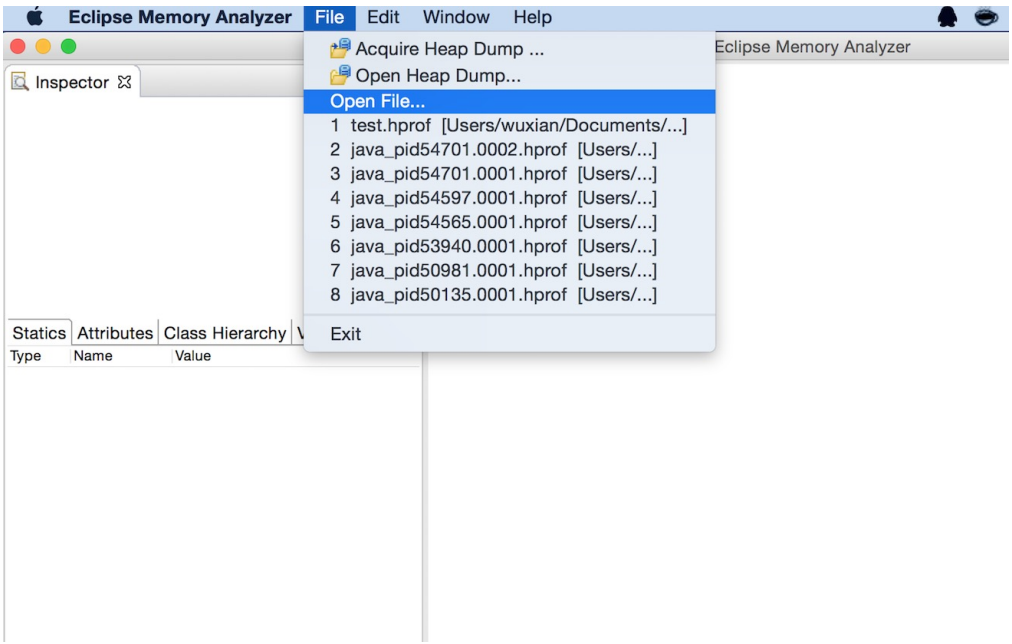


2.填写更改后的文件名和路径:

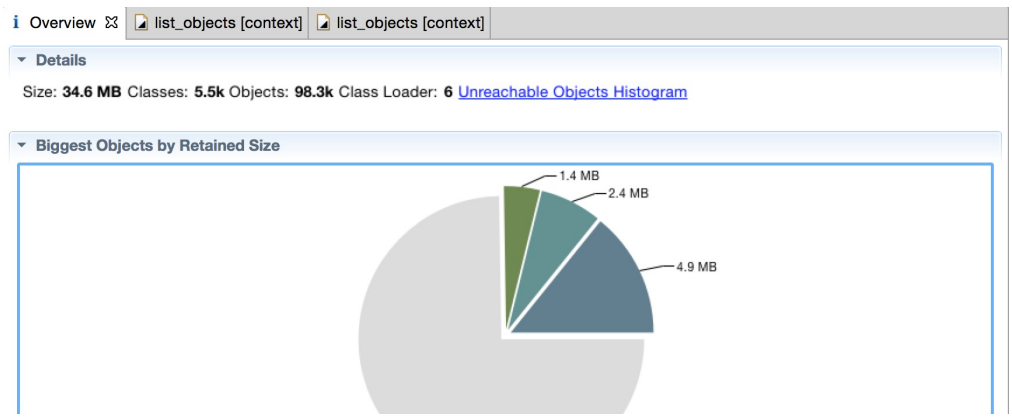


点击OK按钮后，MAT工具所需的文件就生成了，下面我们用MAT来打开该工具:

1.打开MAT后选择 `File->Open File` 选择我们刚才生成的doctorq.hprof文件



2.选择该文件后，MAT会有几秒种的时间解析该文件，有的hprof文件可能过大，会有更长的时间解析，解析后，展现在我们的面前的界面如下:



这是个总览界面，会大体给出一些分析后初步的结论

Overview视图

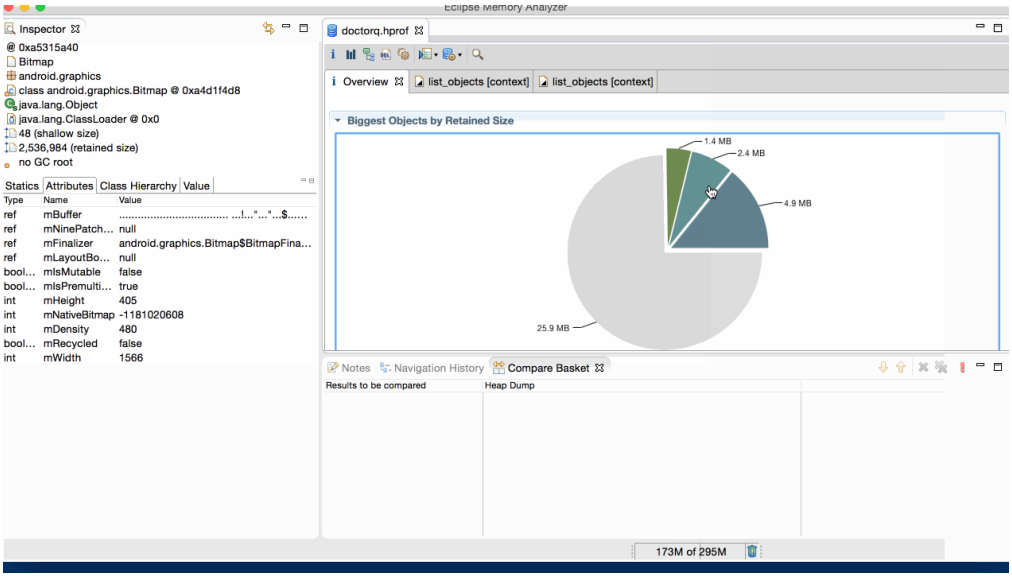
该视图会首页总结出当前这个Heap dump占用了多大的内存，其中涉及的类有多少，对象有多少，类加载器，如果有没有回收的对象，会有一个连接，可以直接参看(图中的Unreachable Objects Histogram)。

比如该例子中显示了Heap dump占用了41M的内存，5400个类，96700个对象，6个类加载器。

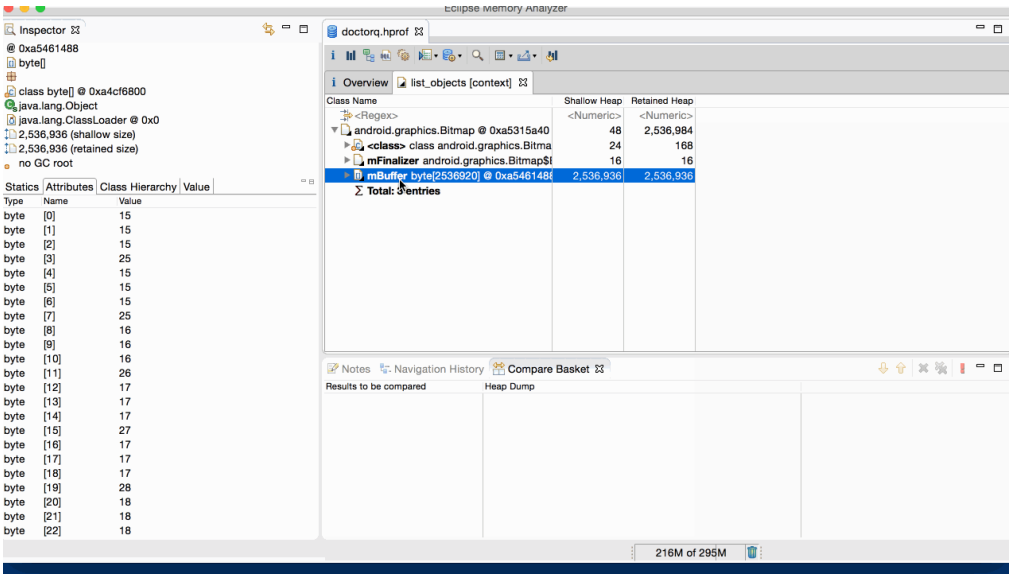
然后还会有各种分类信息:

Biggest Objects by Retained Size

会列举出Retained Size值最大的几个值，你可以将鼠标放到饼图中的扇叶上，可以在右侧看出详细信息:



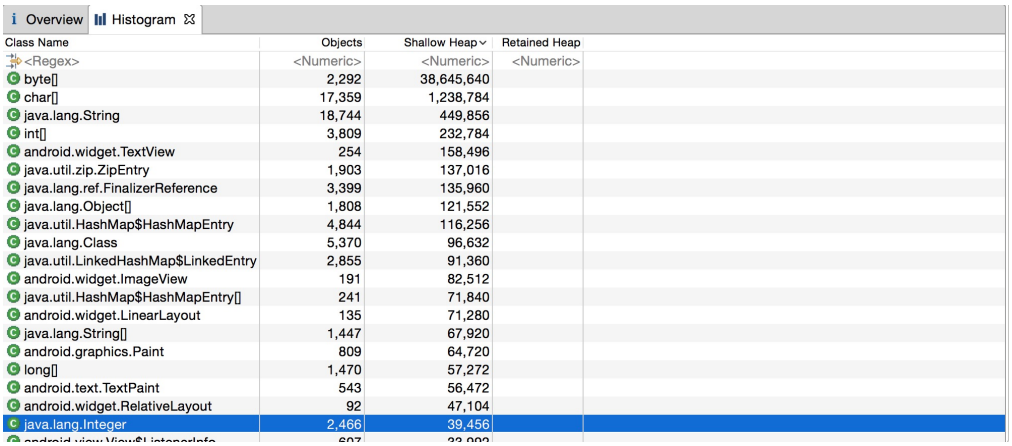
图中灰色区域，并不是我们需要关心的，他是除了大内存对象外的其他对象，我们需要关心的就是图中彩色区域，比如图中2.4M的对象，我们来看看该对象到底是啥:



该对象是一个Bitmap对象，你如果想知道该对象到底是什么图片，可以使用图片工具gimp工具浏览该对象.

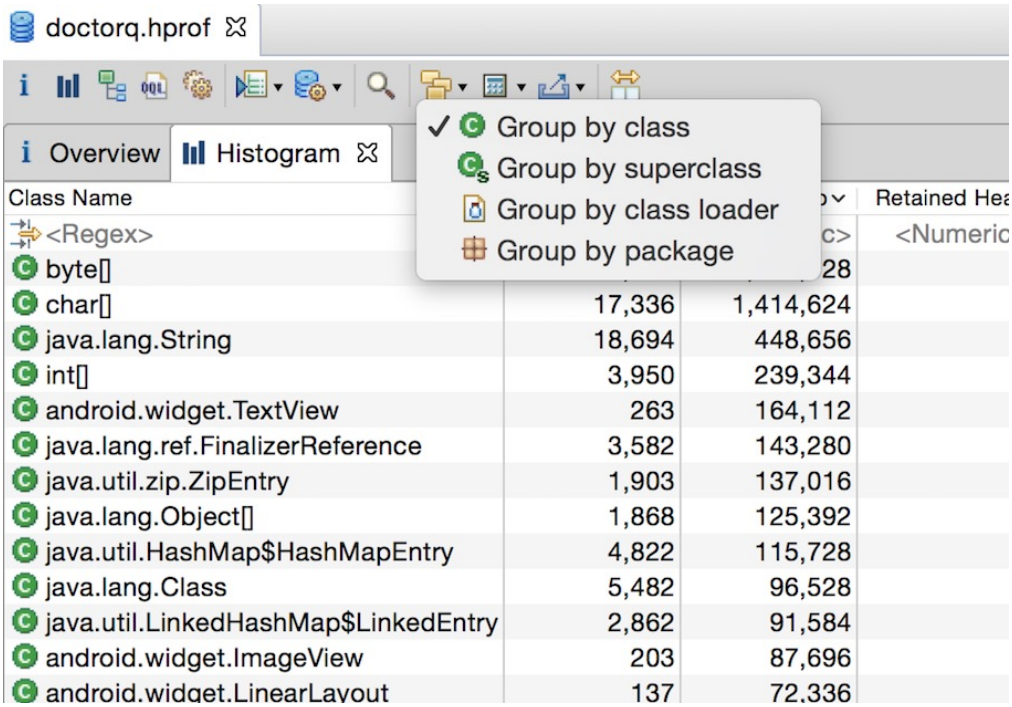
histogram视图

histogram视图主要是查看某个类的实例个数，比如我们在检查内存泄漏时候，要判断是否频繁创建了对象，就可以来看对象的个数来看。也可以通过排序看出占用内存大的对象:

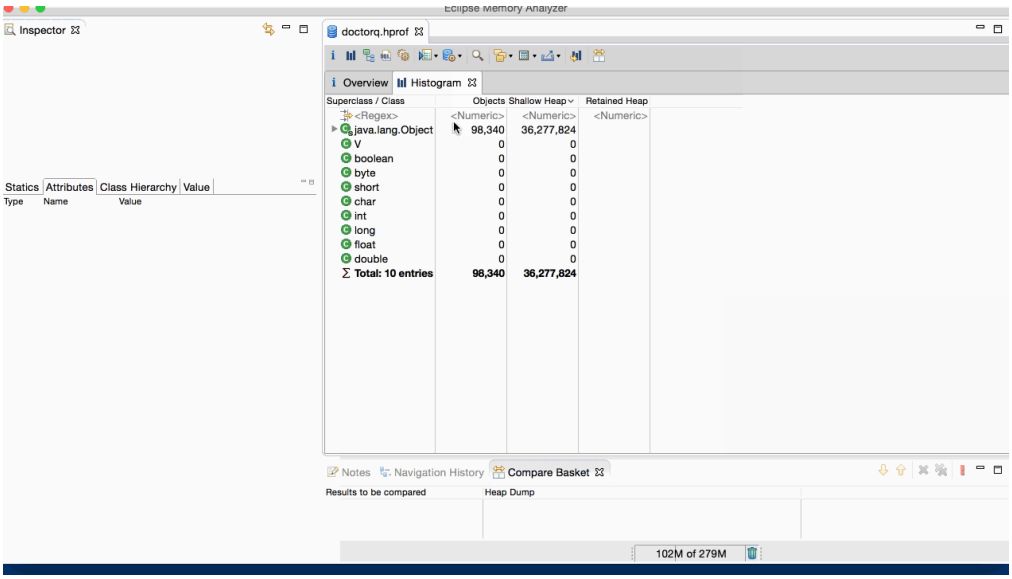


Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	2,292	38,645,640	
char[]	17,359	1,238,784	
java.lang.String	18,744	449,856	
int[]	3,809	232,784	
android.widget.TextView	254	158,496	
java.util.zip.ZipEntry	1,903	137,016	
java.lang.ref.FinalizerReference	3,399	135,960	
java.lang.Object[]	1,808	121,552	
java.util.HashMap$HashMapEntry	4,844	116,256	
java.lang.Class	5,370	96,632	
java.util.LinkedHashMap$LinkedEntry	2,855	91,360	
android.widget.ImageView	191	82,512	
java.util.HashMap$HashMapEntry[]	241	71,840	
android.widget.LinearLayout	135	71,280	
java.lang.String[]	1,447	67,920	
android.graphics.Paint	809	64,720	
long[]	1,470	57,272	
android.text.TextPaint	543	56,472	
android.widget.RelativeLayout	92	47,104	
java.lang.Integer	2,466	39,456	

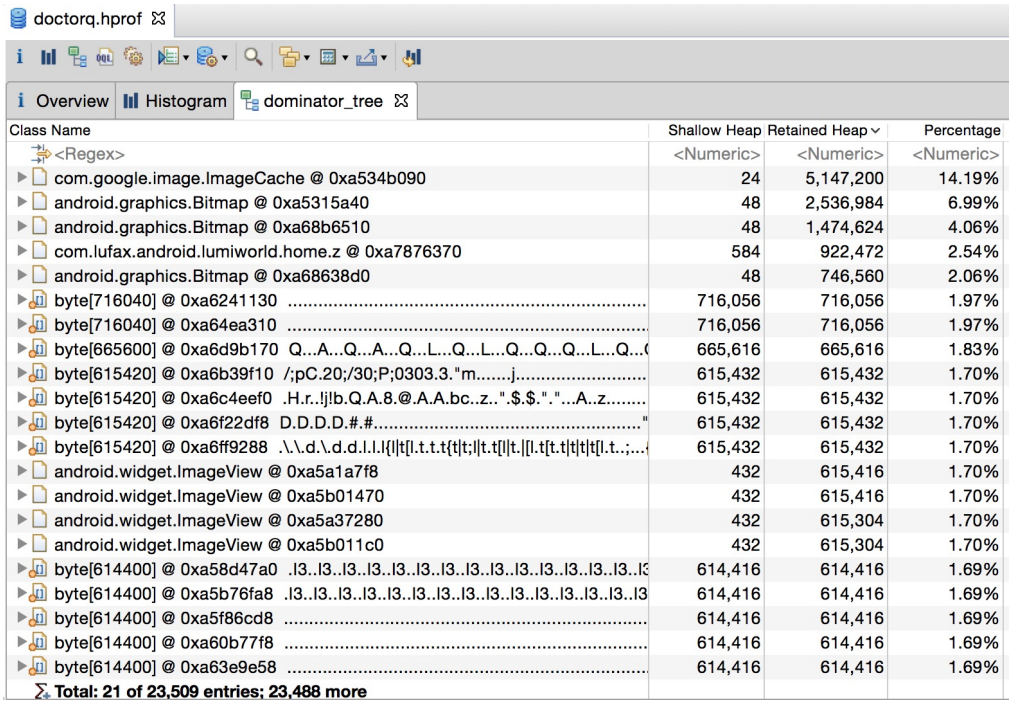
默认是类名形式展示，你也可以选择不同的显示方式，有以下四种方式:



下面来演示一下:



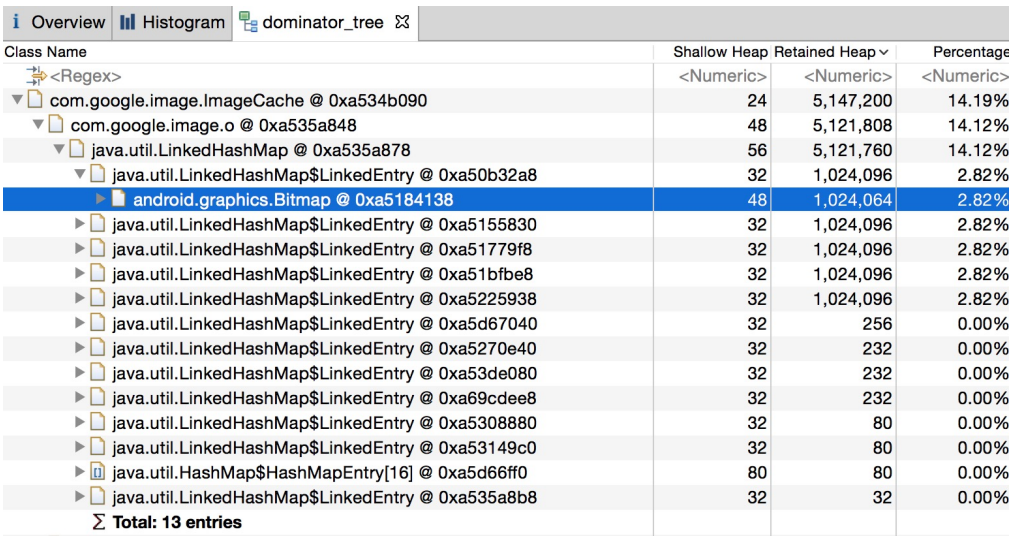
Dominator tree视图



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
com.google.image.ImageCache @ 0xa534b090	24	5,147,200	14.19%
android.graphics.Bitmap @ 0xa5315a40	48	2,536,984	6.99%
android.graphics.Bitmap @ 0xa68b6510	48	1,474,624	4.06%
com.lufax.android.lumiworld.home.z @ 0xa7876370	584	922,472	2.54%
android.graphics.Bitmap @ 0xa68638d0	48	746,560	2.06%
byte[716040] @ 0xa6241130	716,056	716,056	1.97%
byte[716040] @ 0xa64ea310	716,056	716,056	1.97%
byte[665600] @ 0xa6d9b170 Q...A...Q...A...Q...L...Q...L...Q...Q...Q...L...Q...	665,616	665,616	1.83%
byte[615420] @ 0xa6b39f10 /;pC.20;/30;P;0303.3."m.....j.....	615,432	615,432	1.70%
byte[615420] @ 0xa6c4eef0 .H.r..!j!b.Q.A.8.@.A.A.bc..z..".$.$."."...A..z.....	615,432	615,432	1.70%
byte[615420] @ 0xa6f22df8 D.D.D.D.#.#.....	615,432	615,432	1.70%
byte[615420] @ 0xa6ff9288 .\.\.d.\.d.d.l.l.l{|t[l.t.t.t{t|t;||t.t[l|t.||l.t[t.t|t|t|t[l.t..;...	615,432	615,432	1.70%
android.widget.ImageView @ 0xa5a1a7f8	432	615,416	1.70%
android.widget.ImageView @ 0xa5b01470	432	615,416	1.70%
android.widget.ImageView @ 0xa5a37280	432	615,304	1.70%
android.widget.ImageView @ 0xa5b011c0	432	615,304	1.70%
byte[614400] @ 0xa58d47a0 .l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3	614,416	614,416	1.69%
byte[614400] @ 0xa5b76fa8 .l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3..l3	614,416	614,416	1.69%
byte[614400] @ 0xa5f86cd8	614,416	614,416	1.69%
byte[614400] @ 0xa60b77f8	614,416	614,416	1.69%
byte[614400] @ 0xa63e9e58	614,416	614,416	1.69%
Total: 21 of 23,509 entries; 23,488 more			

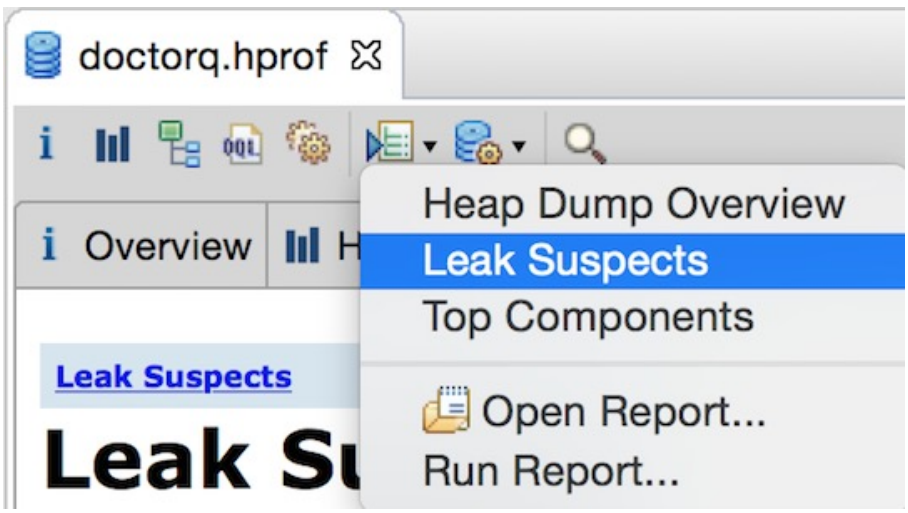
该视图会以占用总内存的百分比来列举所有实例对象，注意这个地方是对象而不是类了，这个视图是用来发现大内存对象的。这些对象都

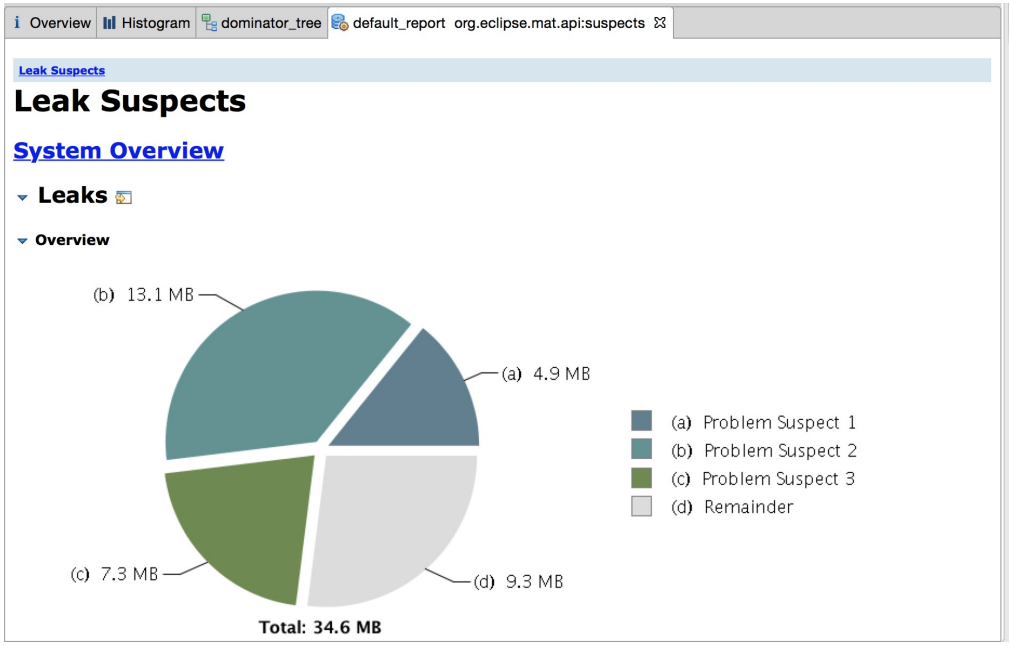
可以展开查看更详细的信息，可以看到该对象内部包含的对象：



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
com.google.image.ImageCache @ 0xa534b090	24	5,147,200	14.19%
com.google.image.o @ 0xa535a848	48	5,121,808	14.12%
java.util.LinkedHashMap @ 0xa535a878	56	5,121,760	14.12%
java.util.LinkedHashMap$LinkedEntry @ 0xa50b32a8	32	1,024,096	2.82%
android.graphics.Bitmap @ 0xa5184138	48	1,024,064	2.82%
java.util.LinkedHashMap$LinkedEntry @ 0xa5155830	32	1,024,096	2.82%
java.util.LinkedHashMap$LinkedEntry @ 0xa51779f8	32	1,024,096	2.82%
java.util.LinkedHashMap$LinkedEntry @ 0xa51bfbe8	32	1,024,096	2.82%
java.util.LinkedHashMap$LinkedEntry @ 0xa5225938	32	1,024,096	2.82%
java.util.LinkedHashMap$LinkedEntry @ 0xa5d67040	32	256	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa5270e40	32	232	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa53de080	32	232	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa69cdee8	32	232	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa5308880	32	80	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa53149c0	32	80	0.00%
java.util.HashMap$HashMapEntry[16] @ 0xa5d66ff0	80	80	0.00%
java.util.LinkedHashMap$LinkedEntry @ 0xa535a8b8	32	32	0.00%
Total: 13 entries			

Leaks suspects视图





这个视图会展示一些可能的内存泄漏的点，比如上图上图显示有3个内存泄漏可疑点，我们以 `Problem Suspect 1` 为例来理解该报告，首先我们来看该可疑点详细信息:

Problem Suspect 1

One instance of **"com.google.image.ImageCache"** loaded by **"dalvik.system.PathClassLoader @ 0xa50819f8"** occupies **5,147,200 (14.19%)** bytes. The memory is accumulated in one instance of **"java.util.LinkedHashMap"** loaded by **"<system class loader>"**.

Keywords
java.util.LinkedHashMap
com.google.image.ImageCache
dalvik.system.PathClassLoader @ 0xa50819f8

Details »

上面信息显示 `ImageCahe` 类的一个实例 `0xa50819f8` 占用了14.19%的内存，具体值为5147200字节(5147200/1024/1024=4.9M),并存放在LinkedHashMap这个集合中，然后我们点击Details跳转到更详细的页面:

Shortest Paths To the Accumulation Point

Class Name	Shallow Heap	Retained Heap
java.util.LinkedHashMap @ 0xa535a878	56	5,121,760
map com.google.image.o @ 0xa535a848	48	5,121,808
b com.google.image.ImageCache @ 0xa534b090	24	5,147,200
mImageCache com.lufax.android.LufaxApplication @ 0xa5083f10 Native Stack	48	464
a com.google.image.q @ 0xa534a698 »	40	144
∑ Total: 2 entries		

这样我们就能找到在我们的app源码中造成该泄漏可疑点的地方，很容易去定位问题.