

Android性能调优之内存泄露

发件人：Steveyan<no-reply@yinxiang.com>

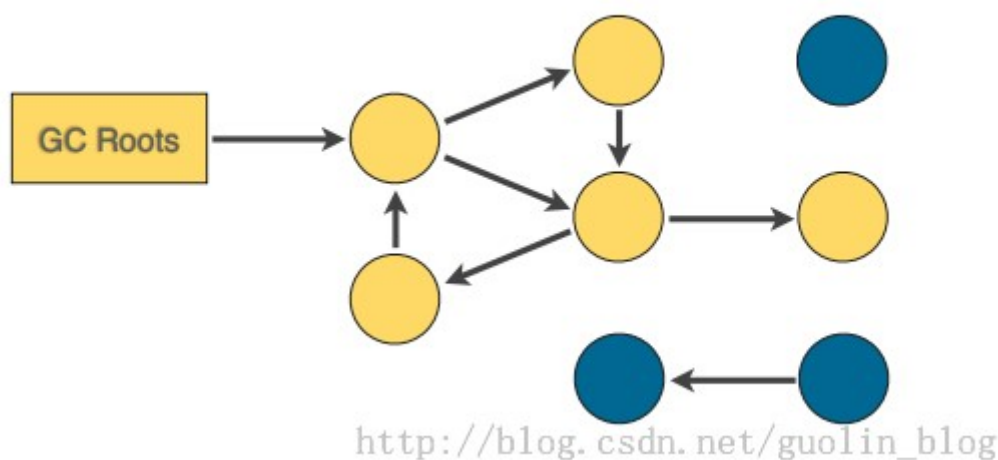
时间：2016年7月25日(星期一) 中午1:43

收件人：中流击水<719905193@qq.com>

基本概念：

垃圾回收是沿着GC Root不断的回收。能被访问到的都是不会被回收的。

Garbage Collection



系统每进行一次GC操作时，都会在LogCat中打印一条日志，我们只要去分析这条日志就可以了，日志的基本格式如下所示：

D/dalvikvm: <GC_Reason> <Amount_freed>, <Heap_stats>, <Pause_time>

首先第一部分**GC_Reason**，这个是触发这次GC操作的原因，一般情况下一共有以下几种触发GC操作的原因：

GC_CONCURRENT：当我们应用程序的堆内存快要满的时候，系统会自动触发GC操作来释放内存。

GC_FOR_MALLOC：当我们的应用程序需要分配更多内存，可是现有内存已经不足的时候，系统会进行GC操作来释放内存。

GC_HPROF_DUMP_HEAP：当生成HPROF文件的时候，系统会进行GC操作，关于HPROF文件我们下面会讲到。

GC_EXPLICIT：这种情况就是我们刚才提到过的，主动通知系统去进行GC操作，比如调用System.gc()方法来通知系统。或者在DDMS中，通过工具按钮也是可以显式地告诉系统进行GC操作的。

接下来第二部分**Amount_freed**，表示系统通过这次GC操作释放了多少内存。

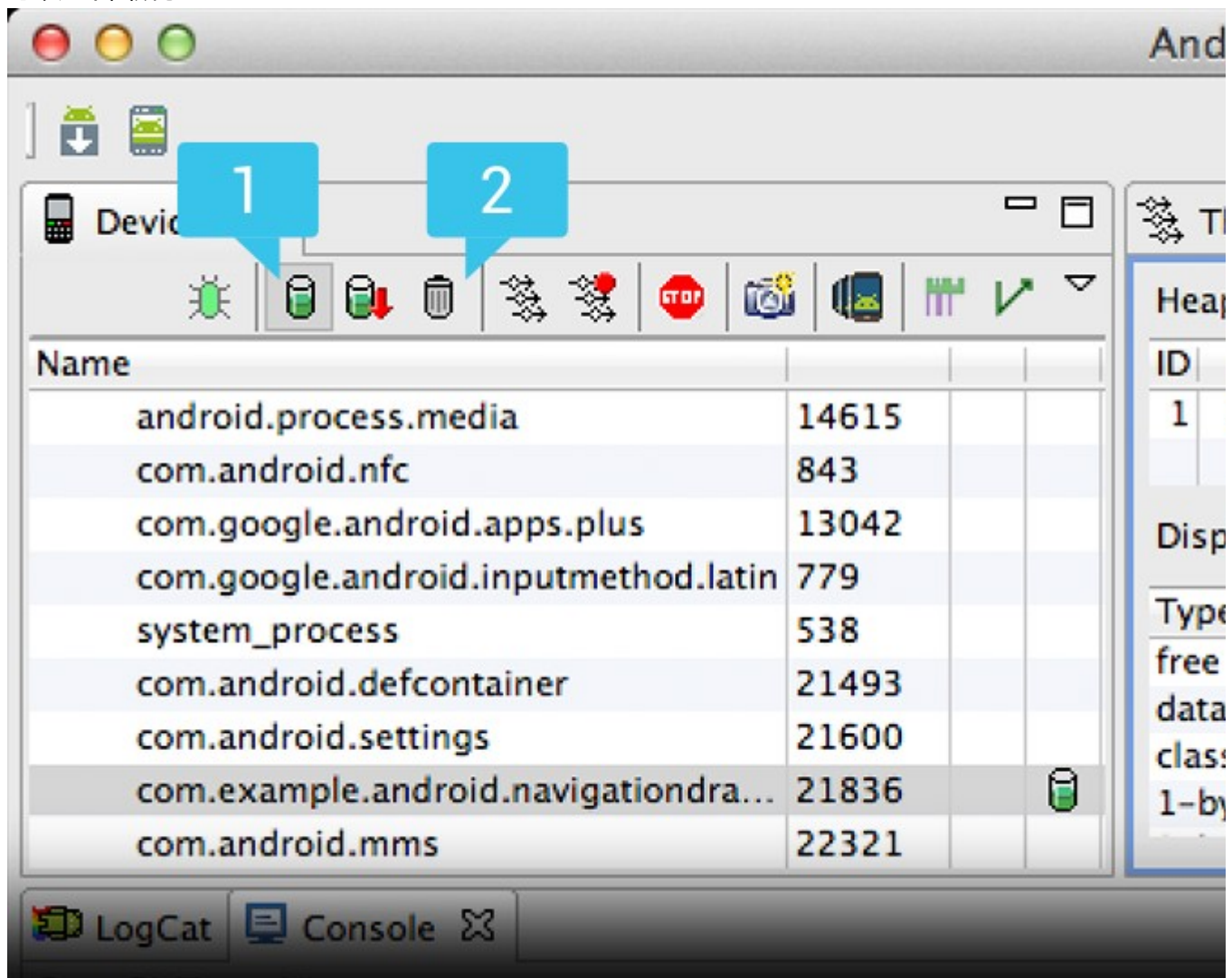
然后**Heap_stats**中会显示当前内存的空闲比例以及使用情况（活动对象所占内存 / 当前程序总内

存)。

最后**Pause_time**表示这次GC操作导致应用程序暂停的时间。关于这个暂停的时间，Android在2.3的版本当中进行过一次优化，在2.3之前GC操作是不能并发进行的，也就是系统正在进行GC，那么应用程序就只能阻塞住等待GC结束。虽说这个阻塞的过程并不会很长，也就是几百毫秒，但是用户在使用我们的程序时还是有可能感觉到略微的卡顿。而自2.3之后，GC操作改成了并发的方式进行，就是说GC的过程中不会影响到应用程序的正常运行，但是在GC操作的开始和结束的时候会短暂阻塞一段时间，不过优化到这种程度，用户已经是完全无法察觉到了。

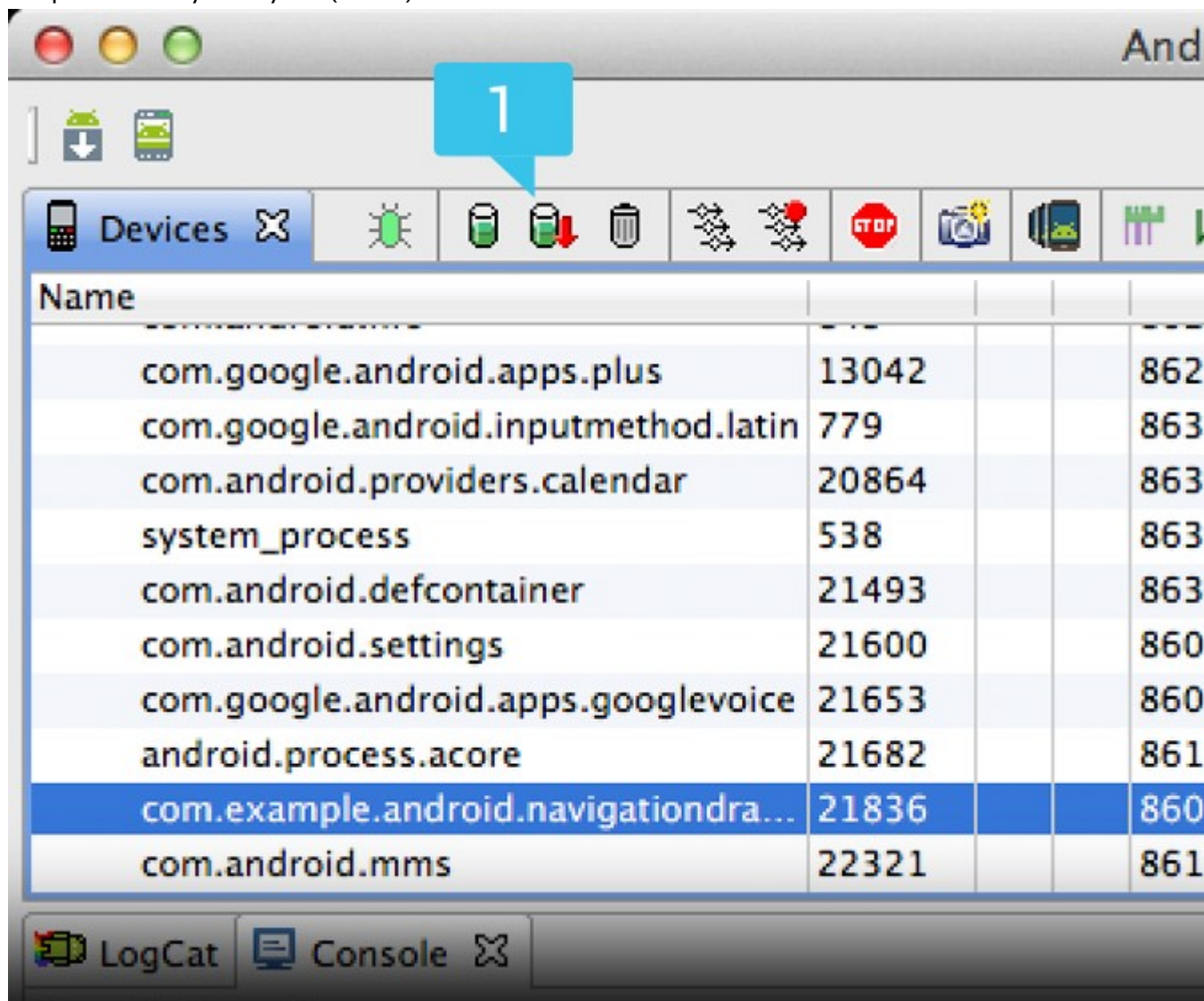
工具使用：

打开DDMS界面，在左侧面板中选择你要观察的应用程序进程，然后点击**Update Heap**按钮，接着在右侧面板中点击Heap标签，之后不停地点Click Cause GC按钮来实时地观察应用程序内存的使用情况即可，如下图所示：



接着继续操作我们的应用程序，然后继续点击**Cause GC**按钮，如果你发现反复操作某一功能会导致应用程序内存**持续增高而不会下降的话**，那么就说明这里很有可能发生内存泄漏了。

Eclipse Memory Analyzer (MAT) :



首先还是进入到DDMS界面，然后在左侧面板选中我们要观察的应用程序进程，接着点击**Dump HPROF file**按钮。

点击这个按钮之后需要等待一段时间，然后会生成一个HPROF文件，这个文件记录着我们应用程序内部的所有数据。但是目前MAT还是无法打开这个文件的，我们还需要将这个HPROF文件从Dalvik格式转换成J2SE格式，使用hprof-conv命令就可以完成转换工作，如下所示：

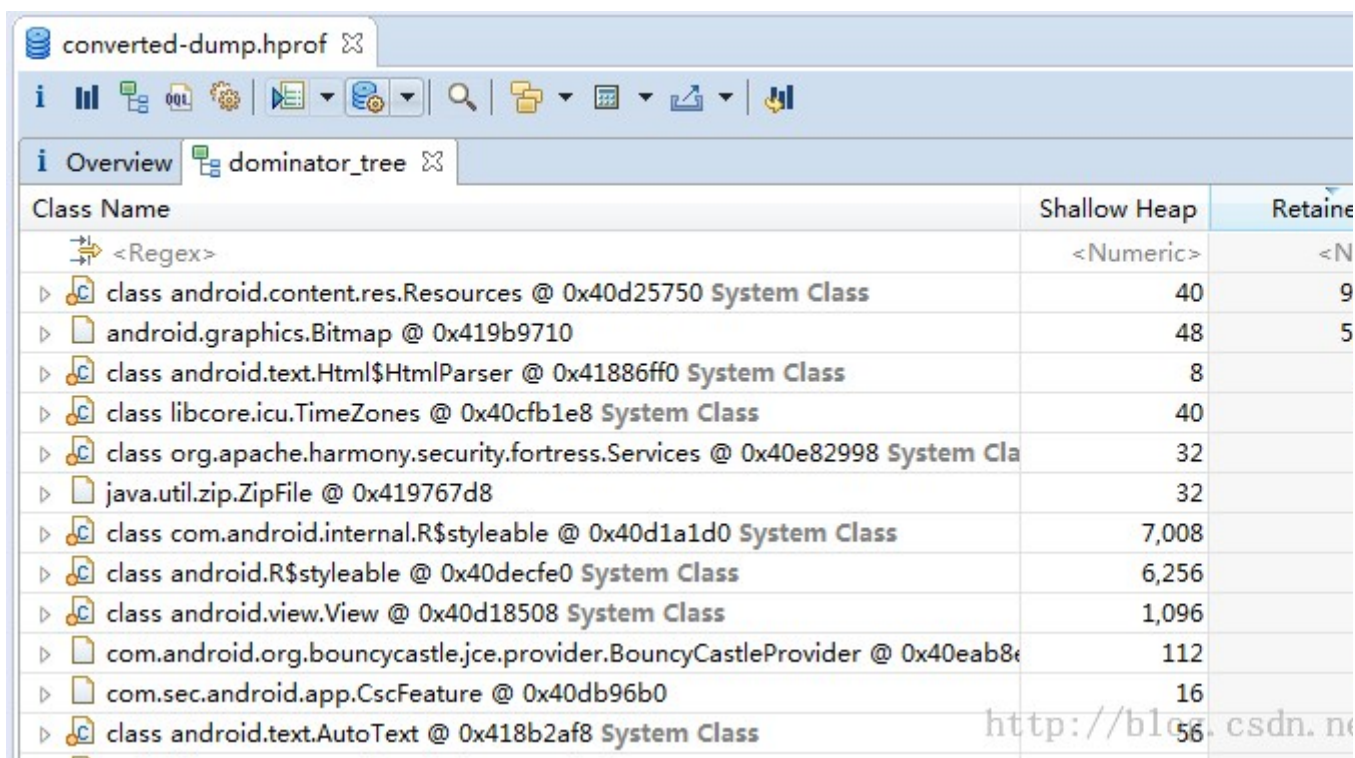
hprof-conv dump.hprof converted-dump.hprof （ hprof-conv命令文件存放于<Android Sdk>/platform-tools目录下面。）

运行MAT工具，然后选择打开转换过后的converted-dump.hprof文件。

Histogram可以列出内存中每个对象的名字、数量以及大小。

Dominator Tree会将所有内存中的对象按大小进行排序，并且我们可以分析对象之间的引用结构。

Dominator Tree的用法:



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<N
class android.content.res.Resources @ 0x40d25750 System Class	40	9
android.graphics.Bitmap @ 0x419b9710	48	5
class android.text.Html\$HtmlParser @ 0x41886ff0 System Class	8	
class libcore.icu.TimeZones @ 0x40cfb1e8 System Class	40	
class org.apache.harmony.security.fortress.Services @ 0x40e82998 System Class	32	
java.util.zip.ZipFile @ 0x419767d8	32	
class com.android.internal.R\$styleable @ 0x40d1a1d0 System Class	7,008	
class android.R\$styleable @ 0x40decfe0 System Class	6,256	
class android.view.View @ 0x40d18508 System Class	1,096	
com.android.org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x40eab8e	112	
com.sec.android.app.CscFeature @ 0x40db96b0	16	
class android.text.AutoText @ 0x418b2af8 System Class	56	

首先Retained Heap表示这个对象以及它所持有的其它引用（包括直接和间接）所占的总内存，因此从上图中看，前两行的Retained Heap是最大的，我们分析内存泄漏时，内存最大的对象也是最应该去怀疑的。

在每一行的最左边都有一个文件型的图标，这些图标有的左下角带有一个红色的点，有的则没有。带有红点的对象就表示是可以被GC Roots访问到的，根据上面的讲解，可以被GC Root访问到的对象都是无法被回收的。那么这就说明所有带红色的对象都是泄漏的对象吗？当然不是，因为有些对象系统需要一直使用，本来就不应该被回收。我们可以注意到，上图当中所有带红点的对象最右边都有写一个System Class，说明这是一个由系统管理的对象，并不是由我们自己创建并导致内存泄漏的对象。

上图当中，除了带有System Class的行之外，最大的就是第二行的Bitmap对象了，虽然Bitmap对象现在不能被GC Roots访问到，但不代表着Bitmap所持有的其它引用也不会被GC Roots访问到。现在我们可以对着第二行点击右键 -> Path to GC Roots -> exclude weak references，为什么选择exclude weak references呢？因为弱引用是不会阻止对象被垃圾回收器回收的，所以我们这里直接把它排除掉，结果如下图所示：

Class Name
<Regex>
android.graphics.Bitmap @ 0x419b9710
mBitmap android.graphics.drawable.BitmapDrawable @ 0x41976e20
mDrawable android.widget.ImageView @ 0x41976bd8
[0] android.view.View[1] @ 0x41967b10
mSortedVerticalChildren android.widget.RelativeLayout @ 0x419bc460
[0] android.view.View[12] @ 0x419bc858
mChildren android.support.v7.internal.widget.NativeActionModeAwareLayout @ 0x419c...
mNativeActionModeAwareLayout android.support.v7.app.ActionBarActivityDelegate
mDelegate com.example.tony.myapplication.MainActivity @ 0x419c7260
this\$0 com.example.tony.myapplication.MainActivity\$LeakClass @ 0x41976f48

Bitmap对象经过层层引用之后，到了MainActivity\$LeakClass这个对象，然后在图标的左下角有个红色的图标，就说明在这里可以被 GC Roots访问到了，并且这是由我们自己创建的Thread，并不是System Class了，那么由于MainActivity\$LeakClass能被GC Roots访问到导致不能被回收，导致它所持有的其它引用也无法被回收了，包括MainActivity，也包括MainActivity中所包含的图片。

这是Dominator Tree中比较常用的一种分析方式，即搜索大内存对象通向GC Roots的路径，因为内存占用越高的对象越值得怀疑。

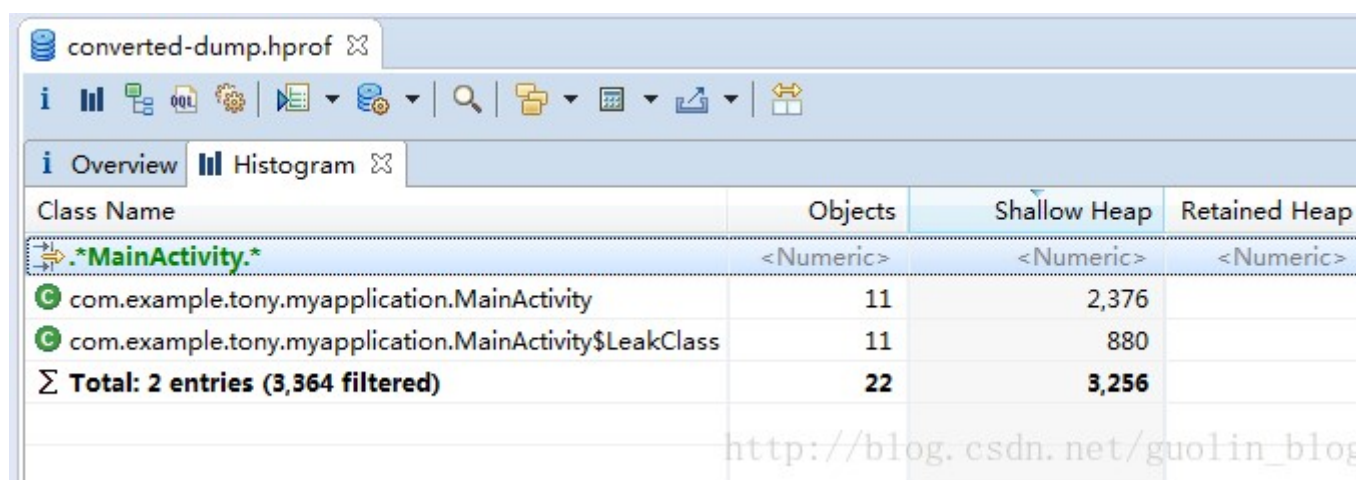
所以先看大内存对象，先看检查可疑对象。

Histogram的用法：

converted-dump.hprof			
i Overview Histogram			
Class Name	Objects	Shallow Heap	Retained H
<Regex>	<Numeric>	<Numeric>	<Numer
byte[]	1,826	15,500,360	
char[]	11,039	634,840	
java.lang.String	12,526	300,624	
int[]	2,406	126,144	
java.util.HashMap\$HashMapEntry	5,162	123,888	
java.lang.Class	3,366	85,704	
java.lang.String[]	1,955	82,608	
java.util.HashMap\$HashMapEntry[]	78	46,456	
java.lang.Integer	2,492	39,872	
java.lang.Object[]	580	35,912	
java.lang.ref.FinalizerReference	575	23,000	
java.util.Hashtable\$HashtableEntry	908	21,792	
android.graphics.drawable.NinePatchDrawable	271	19,512	
java.util.zip.ZipEntry	268	19,296	

这里是把当前应用程序中**所有的对象的名字**、数量和大小**全部都列出来了**，需要注意的是，这里的对象都是只有Shallow Heap而没有Retained Heap的，那么Shallow Heap又是什么意思呢？就是当前对象**自己所占内存的大小**，不包含引用关系的，比如说上图当中，byte[]对象的Shallow Heap最高，说明我们应用程序中用了很多byte[]类型的数据，比如说图片。可以通过**右键 -> List objects -> with incoming references**来查看具体是谁在使用这些byte[]。

那么通过Histogram又怎么去分析内存泄漏的原因呢？当然其实也可以用和Dominator Tree中比较相似的方式，即分析大内存的对象，比如上图中byte[]对象内存占用很高，我们通过分析byte[]，最终也是能找到内存泄漏所在的，但 是这里我准备使用另外一种更适合Histogram的方式。大家可以看到，Histogram中是可以显示对象的数量，那么比如说我们现在**怀疑 MainActivity中有可能存在内存泄漏**，就可以在**第一行的正则表达式框中搜索“MainActivity”**，如下所示：

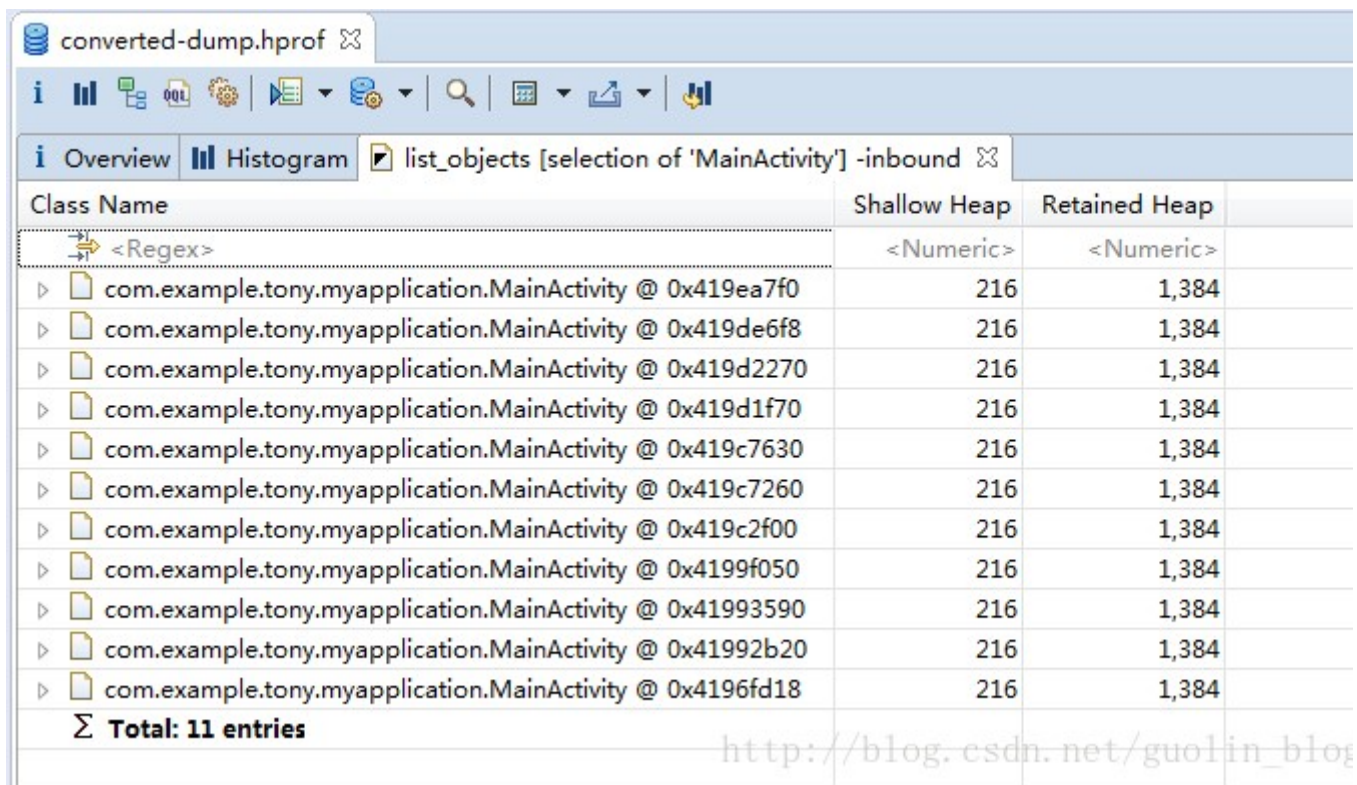


The screenshot shows the Android Studio interface with the Histogram tool open. The tool displays a table of memory usage for various classes. The first row is highlighted, showing the search results for MainActivity. The table has four columns: Class Name, Objects, Shallow Heap, and Retained Heap. The first row is a search filter: *MainActivity.*. The second row shows the results for MainActivity: 11 objects, 2,376 bytes of Shallow Heap, and 0 bytes of Retained Heap. The third row shows the results for MainActivity\$LeakClass: 11 objects, 880 bytes of Shallow Heap, and 0 bytes of Retained Heap. The total is 22 objects and 3,256 bytes of Shallow Heap.

Class Name	Objects	Shallow Heap	Retained Heap
MainActivity.	<Numeric>	<Numeric>	<Numeric>
com.example.tony.myapplication.MainActivity	11	2,376	
com.example.tony.myapplication.MainActivity\$LeakClass	11	880	
Σ Total: 2 entries (3,364 filtered)	22	3,256	

将包含“MainActivity”字样的所有对象全部列出了出来，其中第一行就是MainActivity的实例。但是大家有没有注意到，当前内存中是**有11个MainActivity的实例的**，这太不正常了，通常情况下一个Activity应该只有一个实例才对。

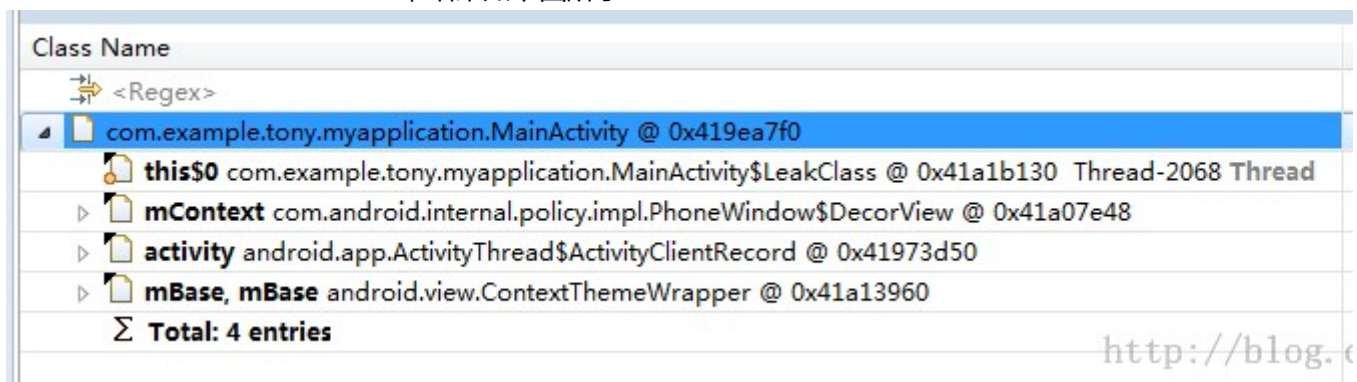
接下来对着**MainActivity右键 -> List objects -> with incoming references**查看**具体MainActivity实例**，如下图所示：



The screenshot shows the 'list_objects' view of the Memory Profiler. The table lists 11 instances of 'com.example.tony.myapplication.MainActivity'. Each instance has a 'Shallow Heap' of 216 bytes and a 'Retained Heap' of 1,384 bytes. The total for all 11 entries is 11 entries.

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.example.tony.myapplication.MainActivity @ 0x419ea7f0	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419de6f8	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419d2270	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419d1f70	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419c7630	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419c7260	216	1,384
com.example.tony.myapplication.MainActivity @ 0x419c2f00	216	1,384
com.example.tony.myapplication.MainActivity @ 0x4199f050	216	1,384
com.example.tony.myapplication.MainActivity @ 0x41993590	216	1,384
com.example.tony.myapplication.MainActivity @ 0x41992b20	216	1,384
com.example.tony.myapplication.MainActivity @ 0x4196fd18	216	1,384
Σ Total: 11 entries		

如果想要查看内存泄漏的具体原因，可以对着任意一个MainActivity的实例右键 -> Path to GC Roots -> exclude weak references，结果如下图所示：

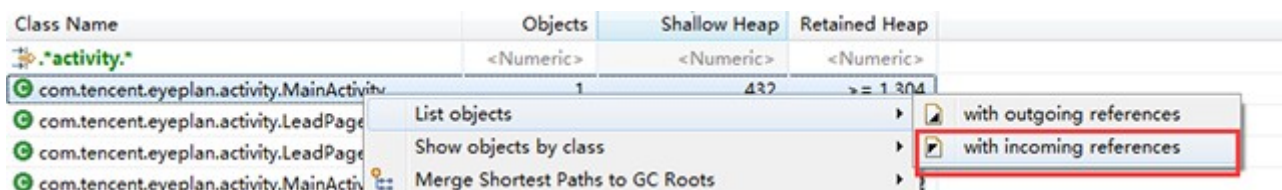


The screenshot shows the 'Path to GC Roots' view for a specific MainActivity instance. It lists 4 entries: 'this\$0' (com.example.tony.myapplication.MainActivity\$LeakClass @ 0x41a1b130), 'mContext' (com.android.internal.policy.impl.PhoneWindow\$DecorView @ 0x41a07e48), 'activity' (android.app.ActivityThread\$ActivityClientRecord @ 0x41973d50), and 'mBase' (android.view.ContextThemeWrapper @ 0x41a13960). The total is 4 entries.

Class Name
<Regex>
com.example.tony.myapplication.MainActivity @ 0x419ea7f0
this\$0 com.example.tony.myapplication.MainActivity\$LeakClass @ 0x41a1b130 Thread-2068 Thread
mContext com.android.internal.policy.impl.PhoneWindow\$DecorView @ 0x41a07e48
activity android.app.ActivityThread\$ActivityClientRecord @ 0x41973d50
mBase android.view.ContextThemeWrapper @ 0x41a13960
Σ Total: 4 entries

可以看到，我们再次找到了内存泄漏的原因，是因为MainActivity\$LeakClass对象所导致的。（左下角的图标）

查看持有该类引用的外部引用。



The screenshot shows the 'List objects' context menu for a MainActivity instance. The menu options are: 'List objects', 'Show objects by class', and 'Merge Shortest Paths to GC Roots'. The 'List objects' option is selected, and its sub-menu is shown, containing 'with outgoing references' and 'with incoming references'. The 'with incoming references' option is highlighted with a red box.

Class Name	Objects	Shallow Heap	Retained Heap
activity.	<Numeric>	<Numeric>	<Numeric>
com.tencent.eyepian.activity.MainActivity	1	432	1,304

list objects -- with outgoing references：查看这个对象持有的外部对象引用。

list objects -- with incoming references：查看这个对象被哪些外部对象引用。

总结：

找占用内存大的，然后找回收路径中书否有不能被回收的对象引用。