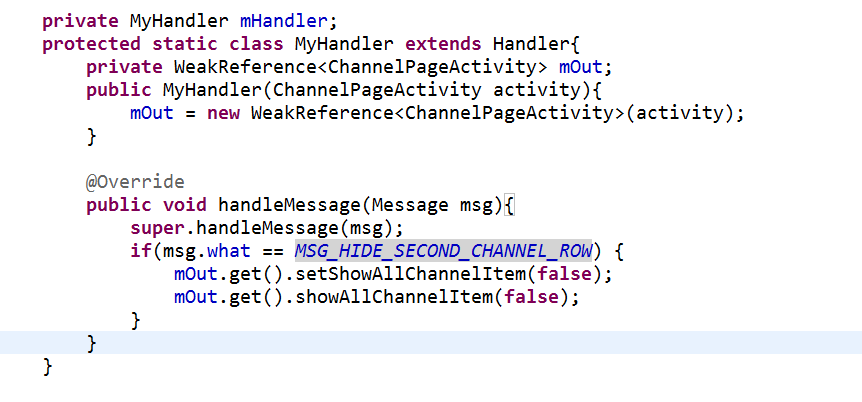
1. **防止内存泄露的一些实践**
2. 匿名内部Handler类导致的内存泄露：

在应用程序线程的MessageQueue中排队的Message对象还保留他们的目标Handler；如果Handler是一个内部类（注：无论是匿名还是非匿名，匿名是比较常见用法），它的外部类将被保留。为了避免泄漏外部类，声明一个Handler子类为静态内部类（注：这样就避免了Handler对象对外部类实例的自动引用），其内部持有一个对外部类对象的WeakReference。弱引用的方式就可以放心使用外部实例的变量了（比如Activity的mTextView等）



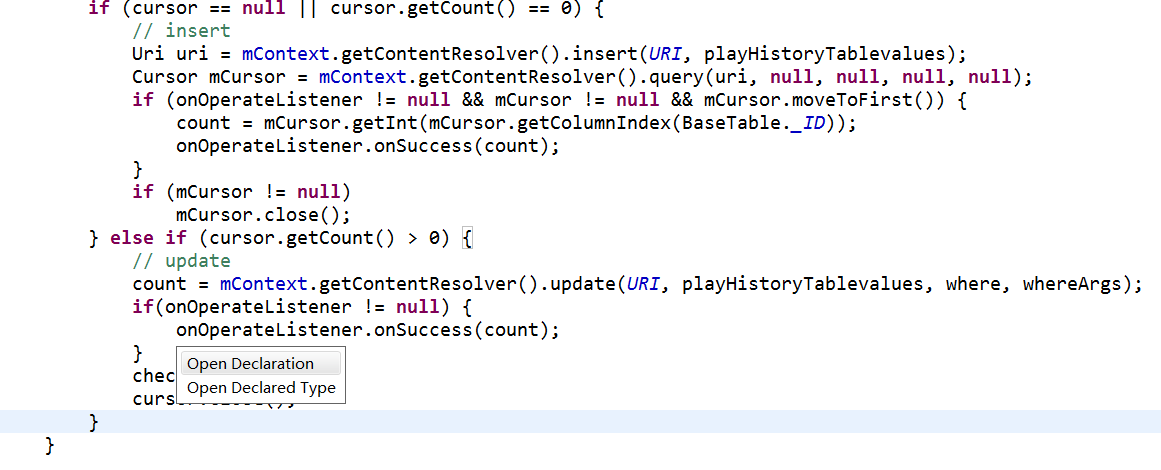
同时，在组件生命周期结束的时候，要执行handler.removeCallbacksAndMessages(null);

所以Handler使用的注意：

1）谨慎使用匿名内部类，要使用静态的子类实现，Activity等使用弱引用

2）生命周期的最后要remove掉handler的message和callback**s**

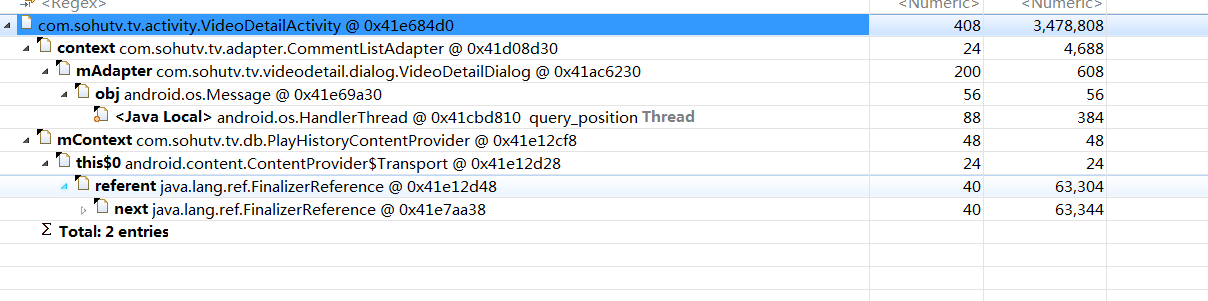
1. [数据库](http://www.2cto.com/database/)的cursor没有关闭（PlayHistoryContentProvider.java）

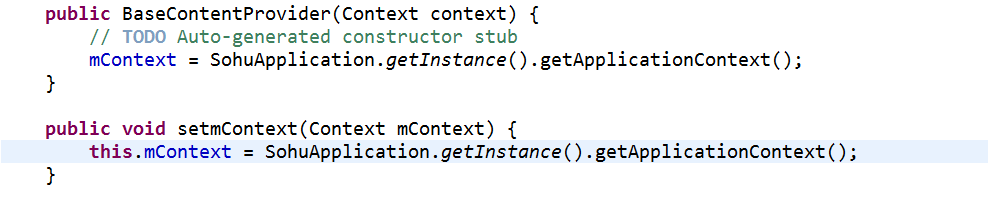


3、ContentProvider的context采用生命周期较长的Application的context

（BaseContentProvider.java）

一般与View不相关的context都可以使用ApplicationContext





4、注册到系统服务中的作为listener，自己生命周期完毕后，系统服务没有完毕，导致泄露。

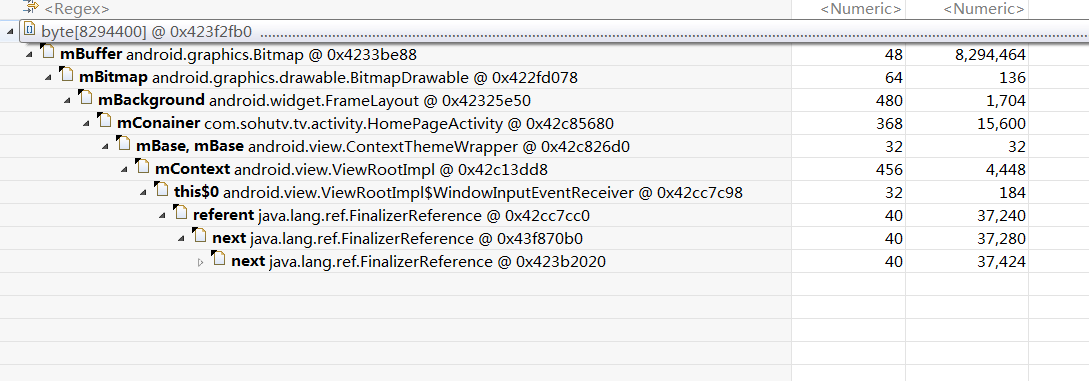
以及或者将自己作为listener注册到比自己生命周期长的里面，原理一样会泄露

5、背景图的解码方式的修改，降低内存的占用：

ImgUtil.decodeSampledBitmapFromResource(getResources(), R.drawable.new\_background, 2);

对于非9patch的本地资源，要采用UniversalImageLoader的方式加载，降低内存的占用。

切记使用setBackgroudResource(int rid)直接加载本地资源



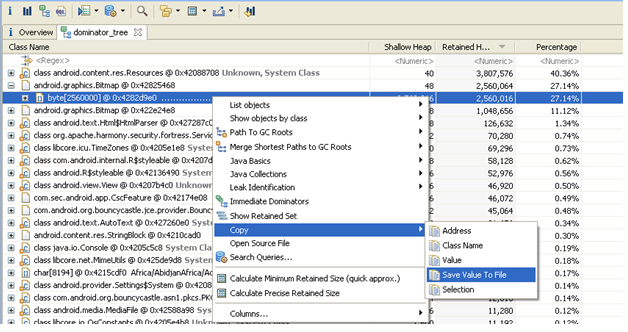
6、对于自定义View以及控件，由于其生命周期不受控，如果有业务流程（注册系统资源），需要在onDetachedFromWindow中释放资源，尽量使用android原生控件，android原生的控件都会在刷新内容时使用convertview，并且对使用过的view调用removeDetachedView。

7、不良代码导致内存泄露：比如集合类ArrayList<> HashMap<>没有及时清空，导致其越来越大；

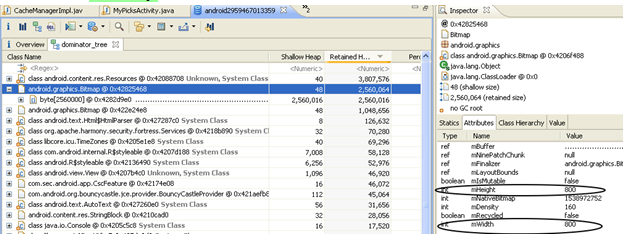
**二、内存泄露及Bitmap大内存定位方法**

1、作为内存占用的绝对大户，Bitmap所占的内存查看往往只能根据GC的路径来推测bitmap属于哪部分，然后进行进一步的分析。但是这种方式耗时，且有多个相同类的对象，则会出现定位不准的情况。下面的方法可以根据内存快照中的bitmap buffer，查看到具体是哪张图片和海报占内存。

1. 保存bitmap 的buffer



1. 利用eclipse的inspector来查看bitmap的高和宽



1. java MATBitmap buffer.bmp converted.bmp width height

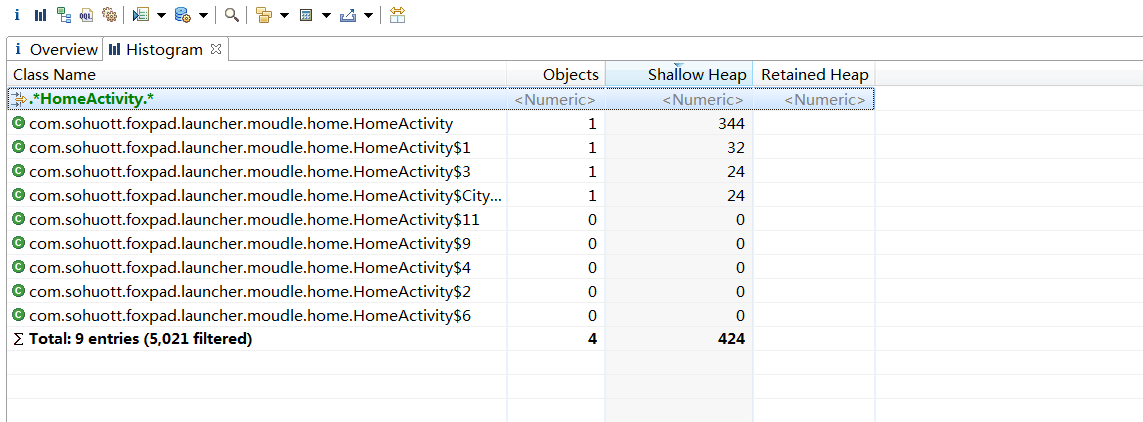
执行该行代码就可以，将原来的二进制文件，转化为可以直接查看的bitmap文件了。





2、对于各个模块的内存泄露问题，需要多次进入和退出功能模块，进行如此操作多次后，可以在MAT的Histogram中对类名进行搜索，如果类名下对应多个对象，那么是存在内存泄露的问题了。

3、Monkey测试后，导出内存快照，查看内存泄露。可以通过DominatorTree来查看大块内存占用者以及潜在内存泄露点。



4、内存分析的方法、次序：

固定占用大内存优化（大背景、浮层） --> 各模块内存泄露排查 --> bitmap（大内存）优化（控件重用机制是否释放、ImageLoader的Cache是否恰当）

注意：

内存oom问题的特点就是：发生oom的地方一般都不是导致oom的地方；

也就是说其他地方的大内存占用或者内存泄露的累积，才导致了这个界面在分配内存时候的oom，所以分析解决oom的问题需要全应用通盘考虑，降低整体应用的内存占用，才能根本解决oom的问题。内存问题是解决应用性能问题的基础，因为持续不断的GC会导致应用的卡顿。

5、Android的垃圾回收机制：

根搜索算法GC root，从根节点对heap进行遍历，如果没有直接或间接引用到GCroot的就是需要回收的。根搜索算法避免了引用计数法循环引用不能释放的问题（例如Activity引用了View，View的内部又含有Activity的实例，二者都有对方的引用计数，所以都释放不了）