Android开发内存泄漏排查规范

|  |  |
| --- | --- |
|  | |
| 密级：内部使用 |  |

文档版本：I

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 主要内容 | 变更内容 | 作者 |
| I | 内存定位、相关示例 | 无 |  |
|  |  |  |  |

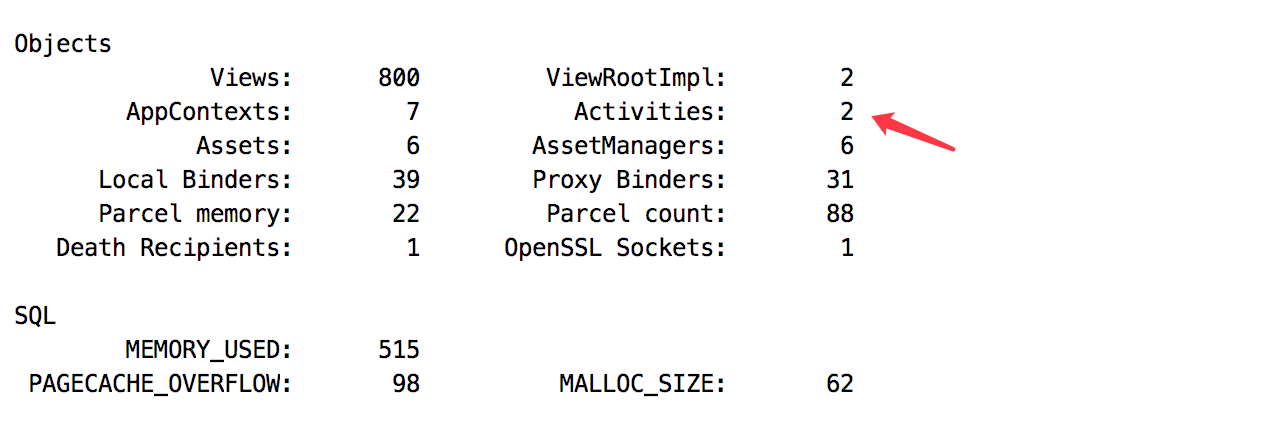
## 如何定位内存泄漏

## 1、Android Studio

* 通过命令行

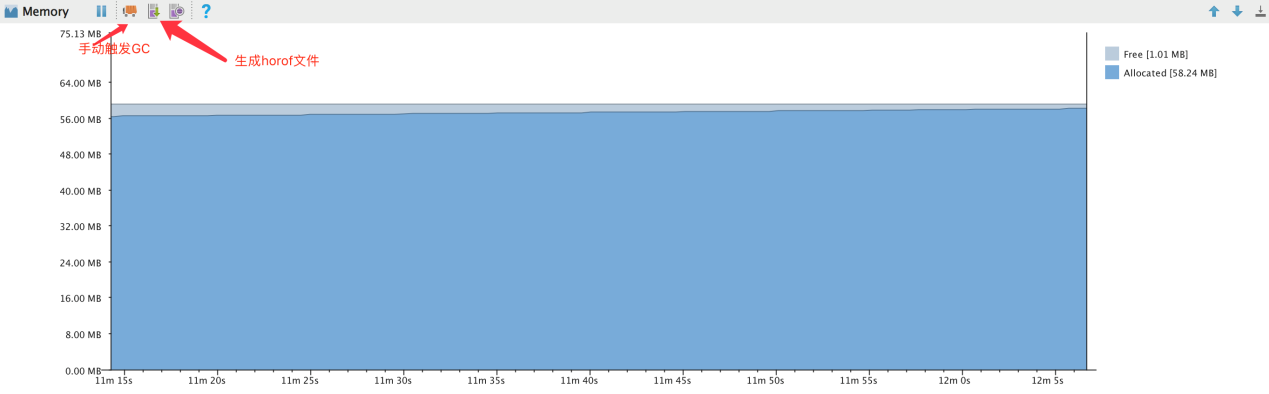
打开AndroidStudio，选中Terminal选项卡，运行项目到真机上，进入主界面操作各个界面，然后重新进入主界面，这个时候项目中应该只有一个MainActivity的实例，此时在Terminal中输入

adb shell dumpsys meminfo 进程名 -d ，就有目前栈中所有的Activity的实例，如果数量大于1，说明有内存泄漏的界面。

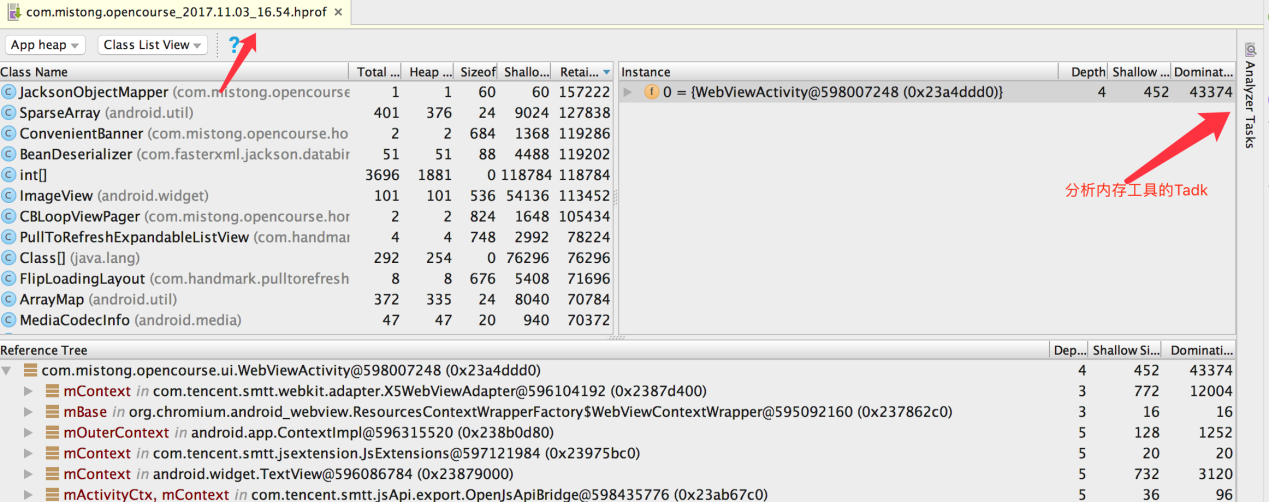


* 通过Monitors

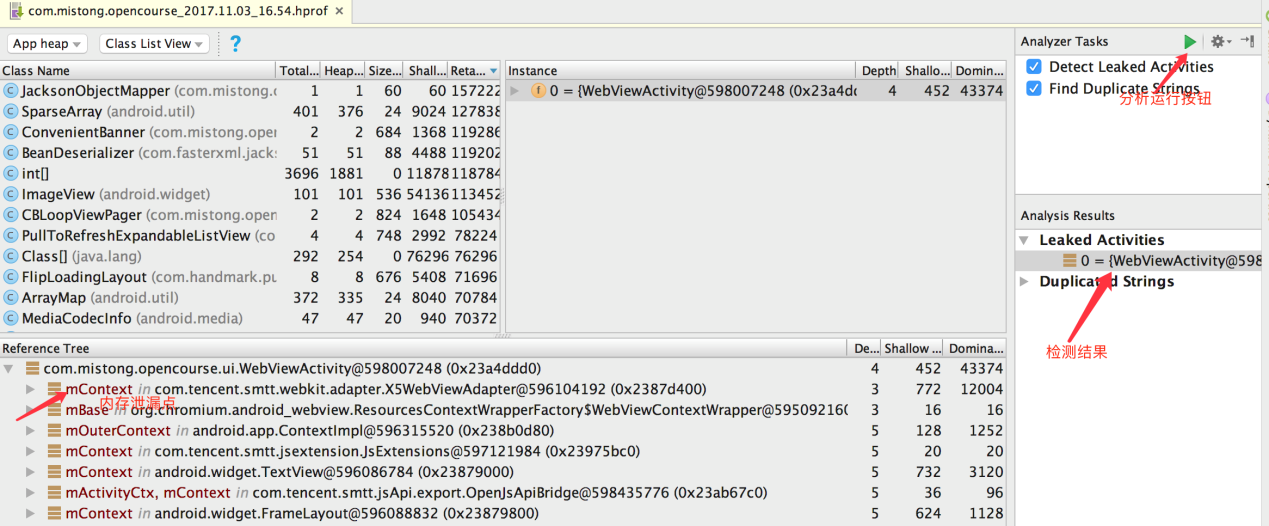
打开AndroidStudio，选中Monitors选项卡，这个时候选中Memory，会有一个可视化的视图。



操作App一遍，然后回退到MainActivity，先点击购物车（手动触发GC），再点击一下打点（生成horof文件），稍等片刻，会在AndroidStudio生成一个窗口。



点击分析内存的Task，会出现一个分析视图，然后点击运行，查看是否内存泄漏的Activity。



通过结果分析我们发现，WebViewActivity有内存泄漏，泄漏点是其中的WebView持有界面的Context。

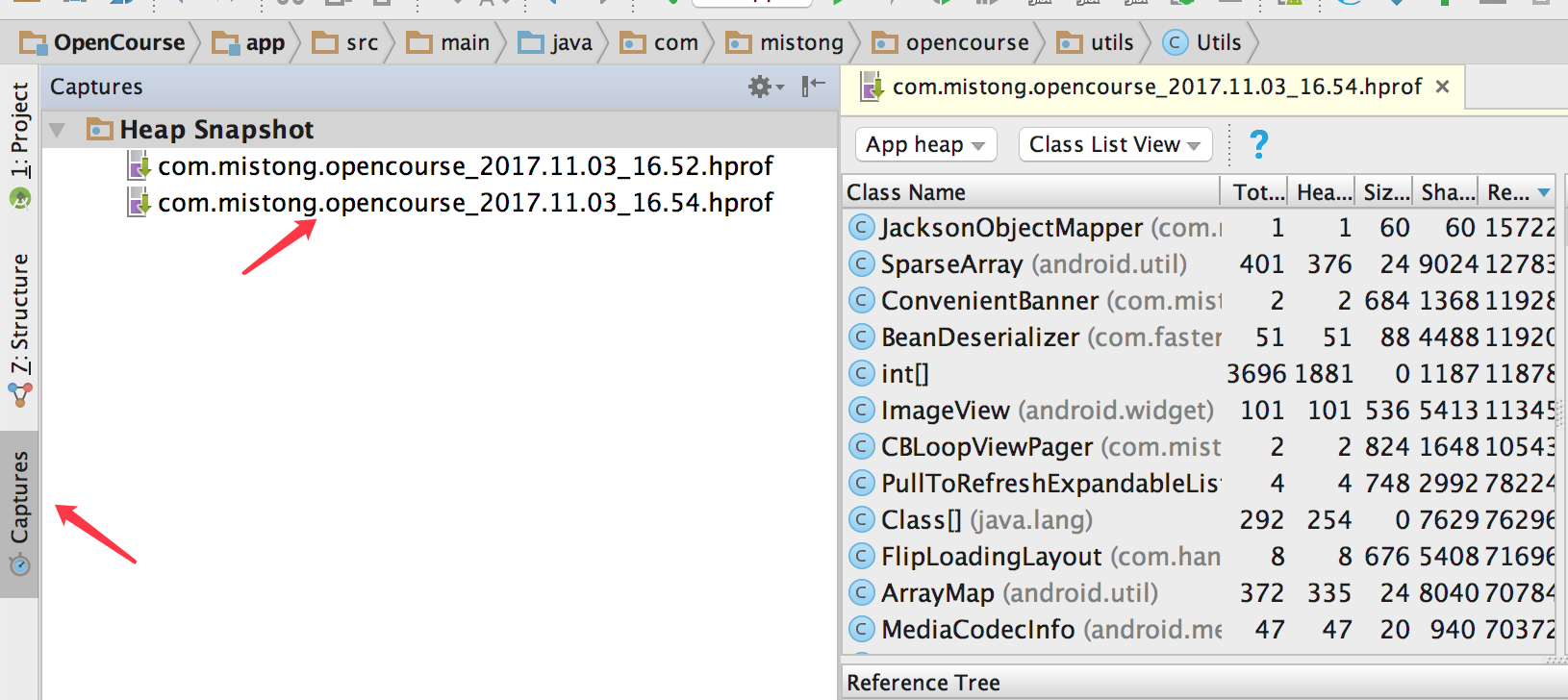
## 2、MAT

MAT早期是Eclipse中的插件，但可独立运行：

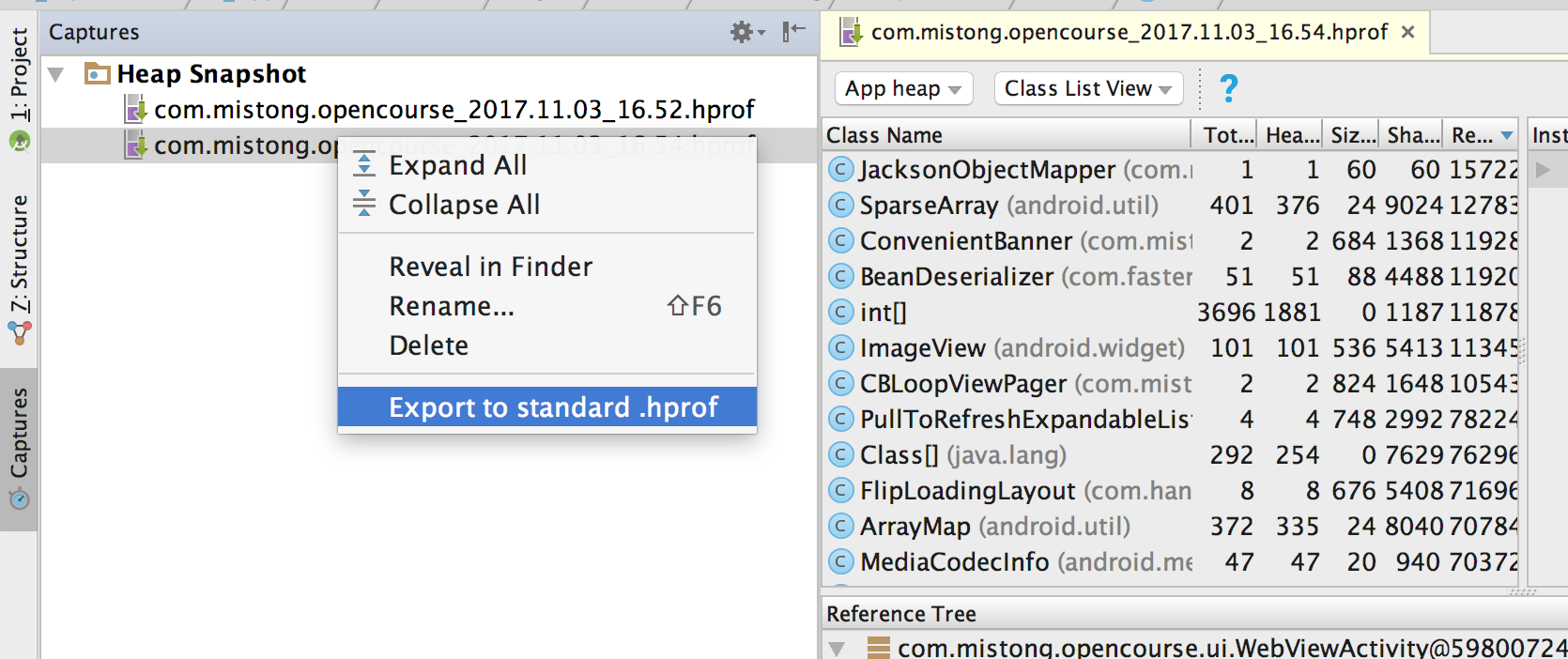
下载地址：<https://www.eclipse.org/mat/>

**第一步：生成horof对比文件**

**MAT**内存分析基本是靠手动来分析，所以要生成对比文件，**horof**文件的**生成可**参考Monitor一节，第一个文件生成：项目刚运行时。第二个文件的生成：项目操作运行一遍退回到第一个文件生成的界面，这时候点击AndroidStudio的Capture选项卡，会找到刚才我们生成的两个文件。

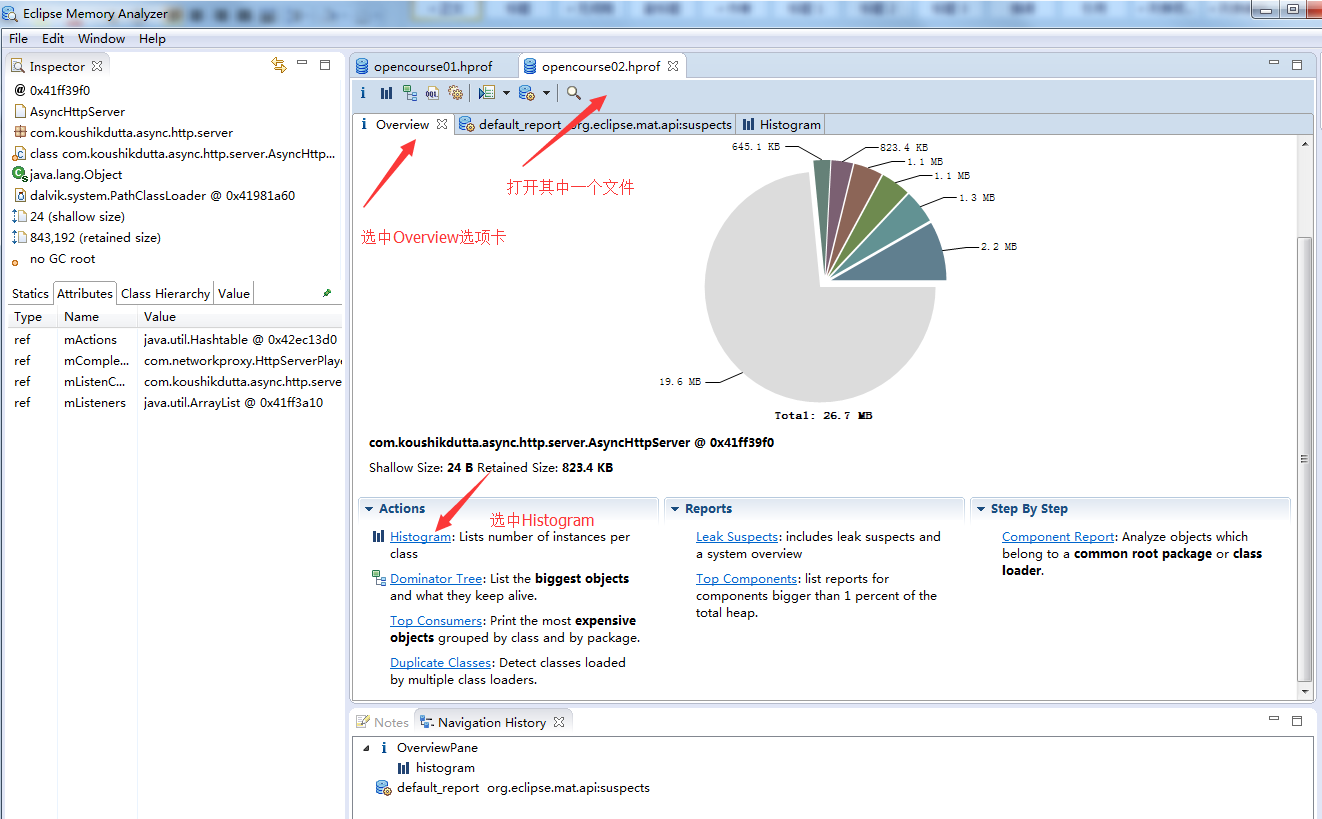


选中文件，导出到桌面，生成MAT可以读取的文件类型。

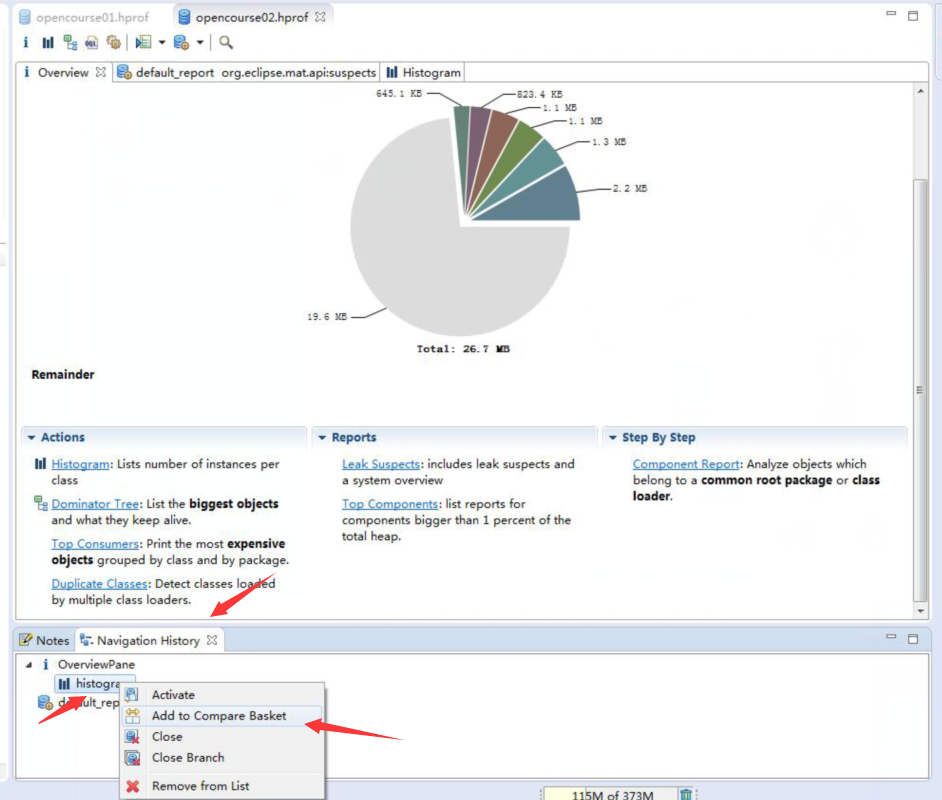
****

**第二步：对比horof文件**

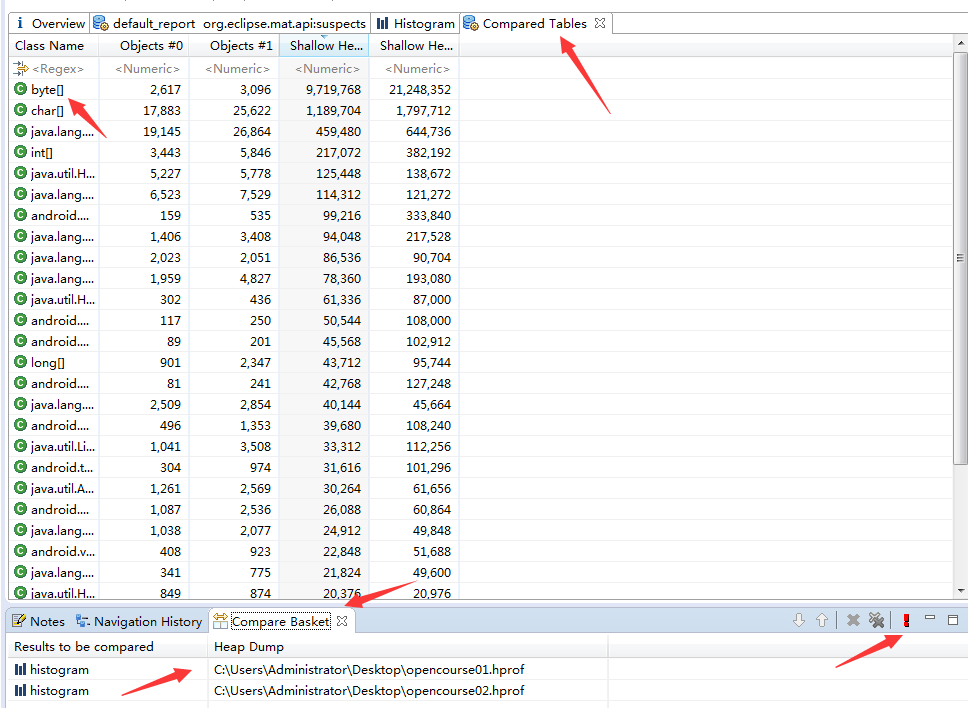
打开MAT导入刚才生成的两个文件，选中Overview选项卡，再选择下面的Histogram按钮，



再点击Nacigation History选项卡，然后右键点击histogram文件，把两个文件都加入对比。

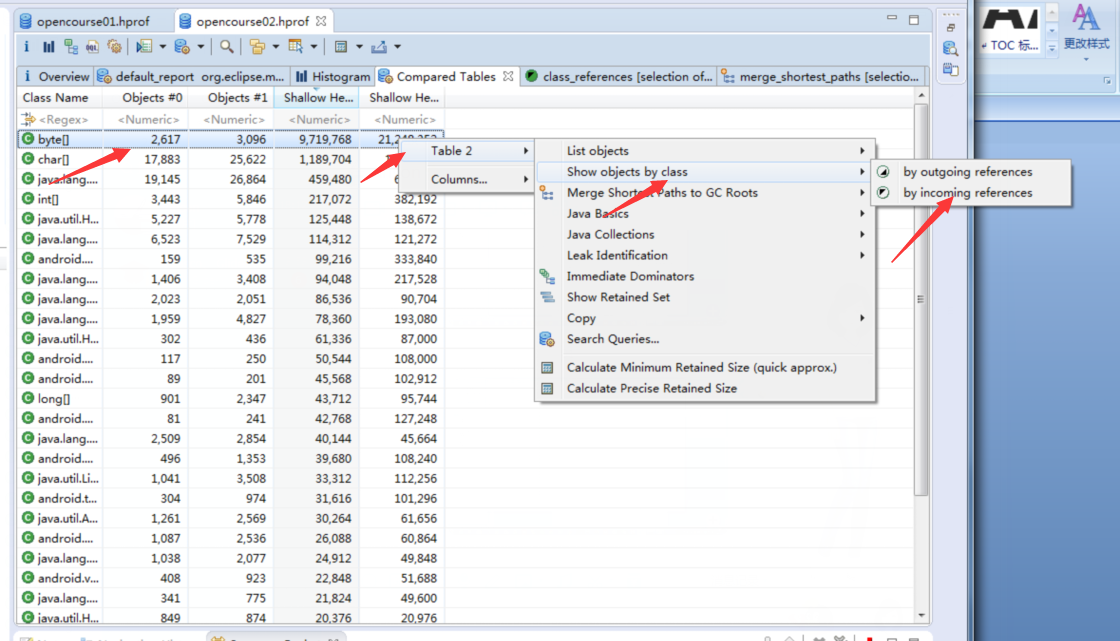


这个时候会产生一个对比选项卡，然后点击右边的红色感叹号，进行对比，会生成一个Compared Tables选项卡，通常我们分析的内容是byte【】。

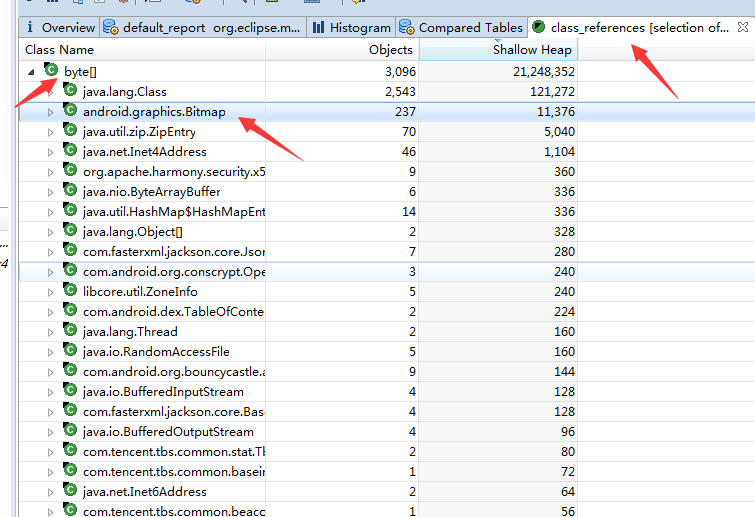


**第三步：分析疑似内存泄漏点**

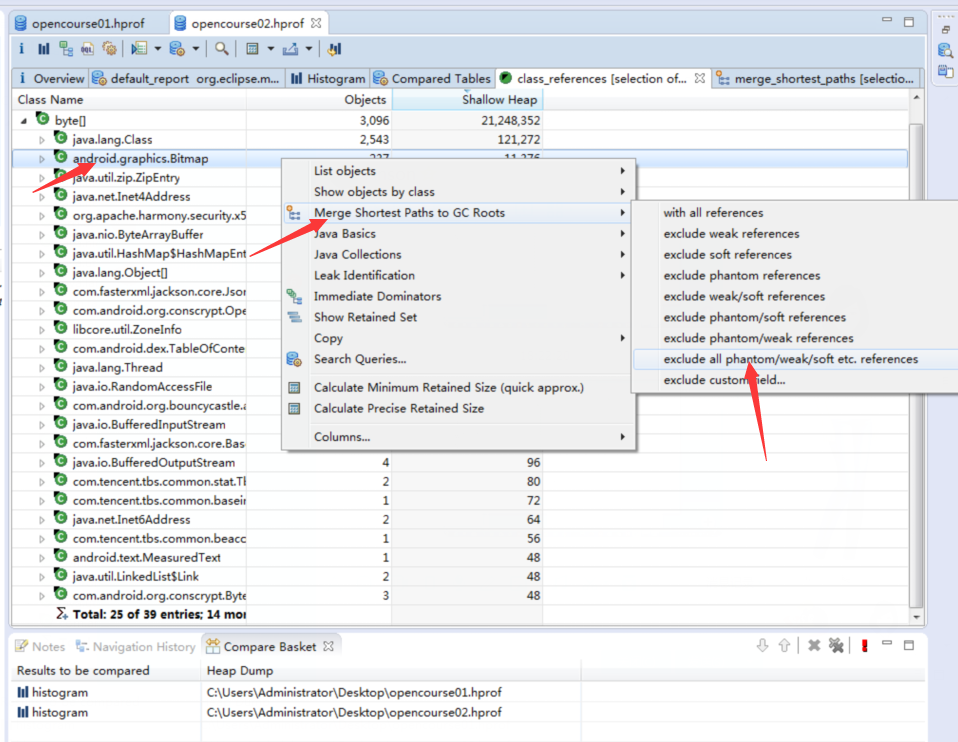
选中byte[]，右键按下图依次操作，这个操作主要看那些地方持有这些数组的引用。



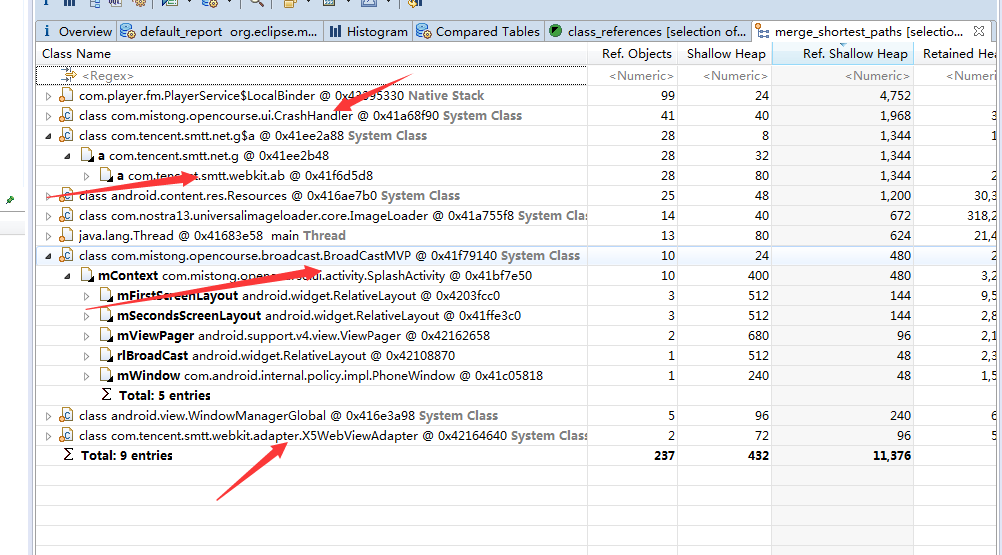
点击运行之后会生成一个新的选项卡，这个选项卡主要表示哪些类持有这些引用，通常我们主要分析BitMap。



选中BitMap 右键一路Next，找到该类的引用。

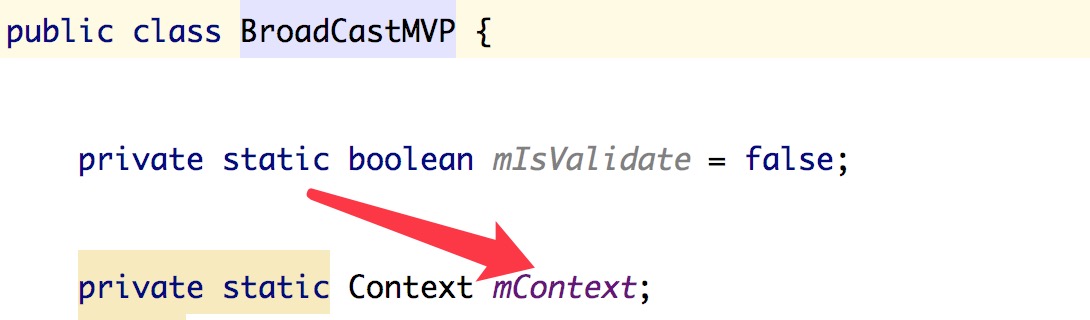


这时会生成一个更加详细的疑似内存泄漏点，这个列表会定位到我们项目中的类名，通过查看项目的源码来定位是否有内存泄漏。



可以看到定位的几个点，其中有CrashHandler、BroadCastMVP，和腾讯WebView。

腾讯WebView是已知问题，那么BradCastMvp是如何泄漏了呢？查看源码可以看到是这个类中静态持有了Context。



## 3、三方工具：如LeakCanary

|  |  |  |  |
| --- | --- | --- | --- |
| 方式 | 优点 | 缺点 | 备注 |
| AS | 快速 | 可能有遗漏点 |  |
| MAT | 全面 | 需要手动排查 |  |
| LeakCanary | 傻瓜式 | 可能有遗漏点 |  |

## 哪些操作可能造成长期的内存泄漏

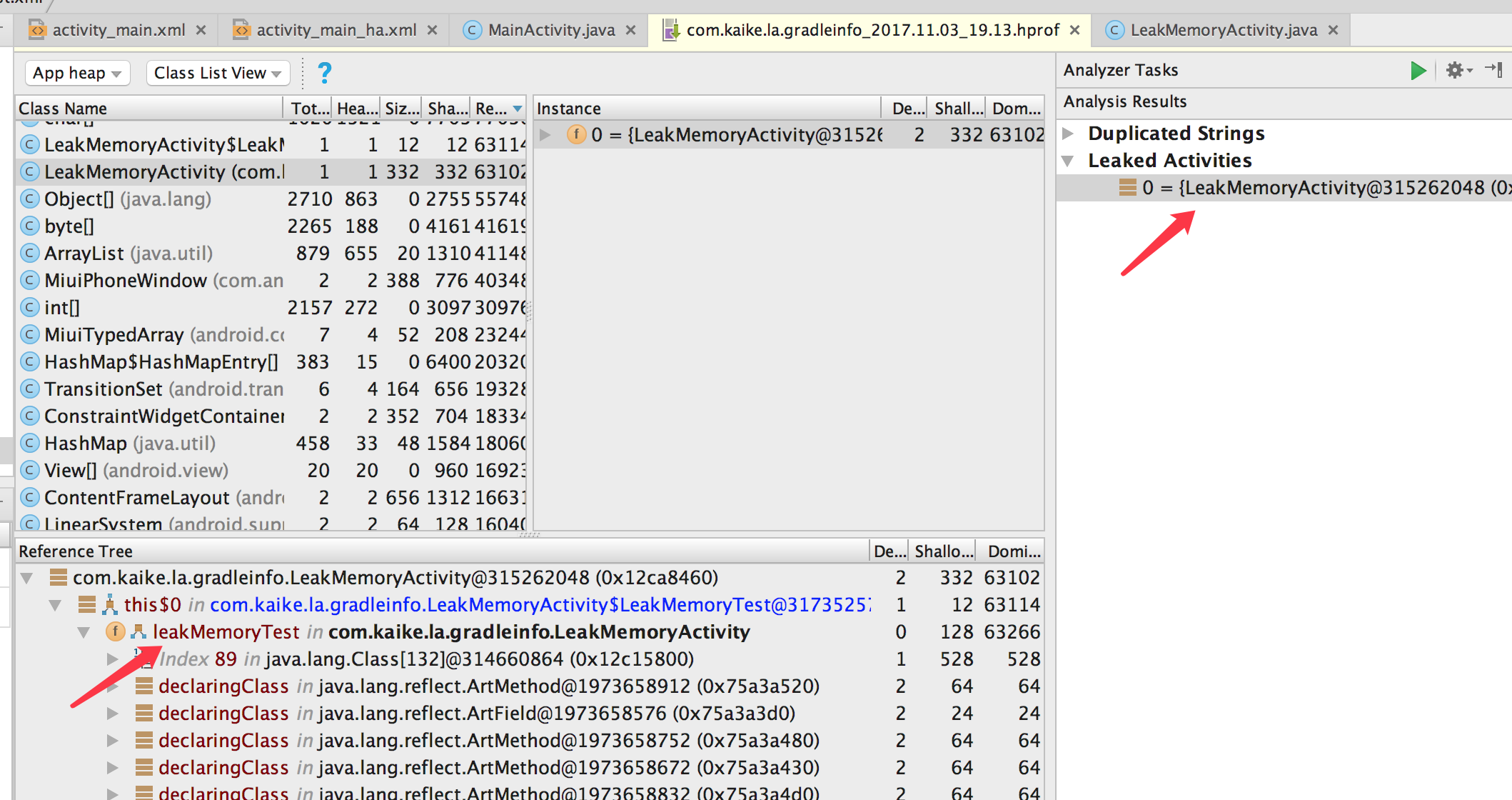
1. **静态变量-非静态内部类**

先上代码，请看下图。



我们启动之后再退出，然后检测一下内存，这个时候已经发现该界面内存泄漏了。

原理：静态变量一旦初始化之后，不会随着该界面的销毁会销毁，而是等该应用进程销毁掉才会销毁，那么静态变量的内部类为什么会持有该Activity？因为非静态的内部会持有外部对象的引用。



那该如何操作？

有两种方式，去除静态变量或者是把LeakMemoryTest 变成静态内部类即可。

**static class** LeakMemoryTest{  
 **public** LeakMemoryTest() {  
 }  
}

1. **线程操作**

先上代码，请看下图。



这里面是个线程操作，退出该界面再来检查一下Activity，发现已经泄露了。



原理：Java中的Thread有一个特点就是她们都是直接被GC Root所引用，也就是说Dalvik虚拟机对所有被激活状态的线程都是持有强引用，导致GC永远都无法回收掉这些线程对象，除非线程被手动停止并置为null或者用户直接kill进程操作。

参考的正确写法: 自定义的静态内部类：

**private void** leakMemoryFunction() {  
 MyStread myStread = **new** MyStread();  
 myStread.start();  
}  
  
**public static class** MyStread **extends** Thread{  
 @Override  
 **public void** run() {  
 **super**.run();  
 **while** (**true**) {  
 Log.*d*(**"haha"**, **"hehe"**);  
 }  
 }  
}

备注：匿名内部的Handler 也是同样的道理。

1. **Activity Fragment 被静态持有**

**参见 第一小节的 MAT BraodCastMVP 类。**

1. **单例模式传入的是Activity的Context。**

下面给出参考写法

*//有参构造方法***public** HttpServerPlayer(Context context) {  
 **mContext** = context.getApplicationContext(); *// 防止内存泄漏*}  
  
*//单例设计模式*@Deprecated  
**public static** HttpServerPlayer getInstance(Context context) {  
 **if** (*instance* == **null**) {  
 **synchronized** (HttpServerPlayer.**class**) {  
 **if** (*instance* == **null**) {  
 *instance* = **new** HttpServerPlayer(context);  
 }  
 }  
 }  
 **return** *instance*;  
}

1. **Android系统的内存泄漏**

当SDK版本在 19 到 23 之间时，LeakCanary 会检测出系统软键盘有内存泄漏。

可参见下面这篇文章。

<http://www.imooc.com/article/13913>

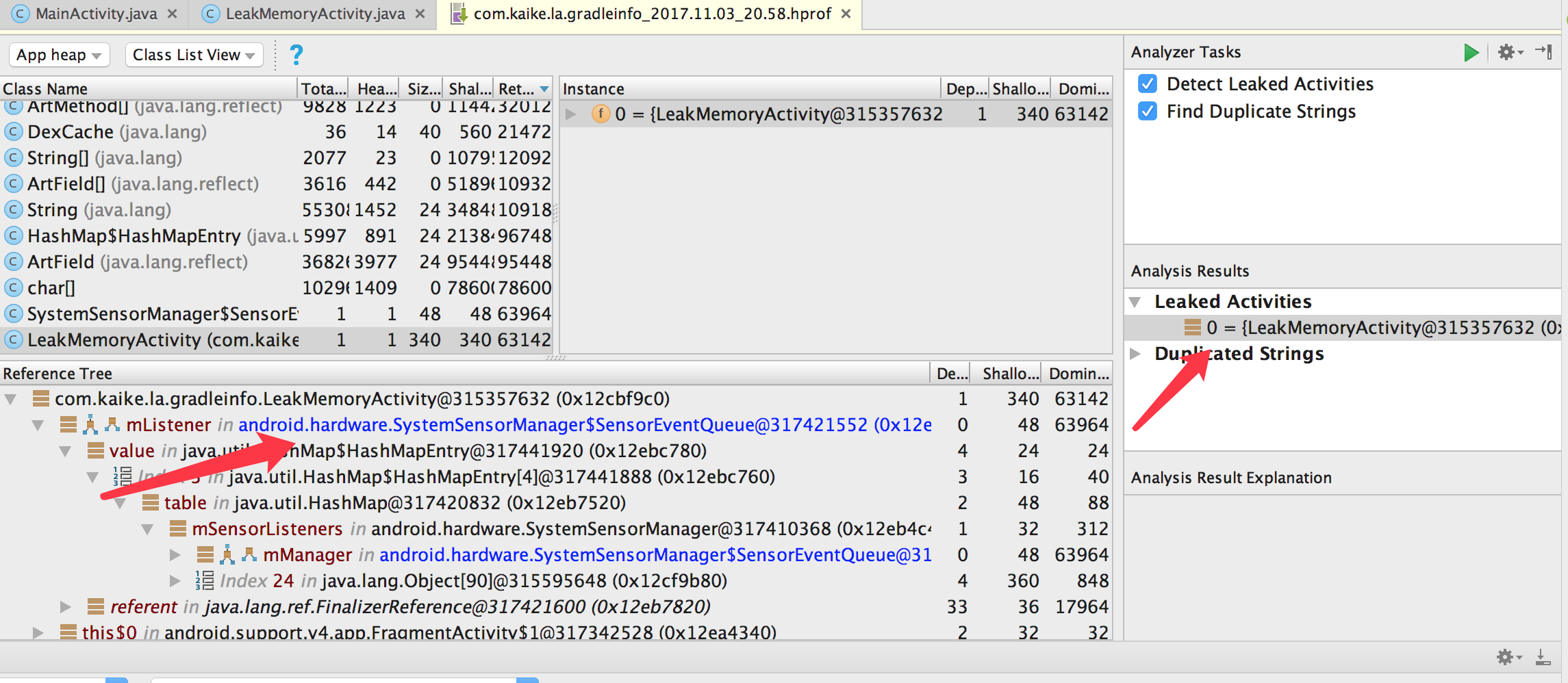
1. **Android 原生 WebView**

目前还没有找到有效的解决办法，除非多进程运行。

1. **使用系统服务没有反注册**

比如使用Android系统的传感器，要进行注册，但没有反注册有时也会造成内存泄漏，下图代码：

**private void** leakMemoryFunction() {  
  
 SensorManager sensorManager = (SensorManager) getSystemService(***SENSOR\_SERVICE***);  
 Sensor sensor = sensorManager.getDefaultSensor(Sensor.***TYPE\_ALL***);  
 sensorManager.registerListener(**this**, sensor, SensorManager.***SENSOR\_DELAY\_FASTEST***);  
  
}



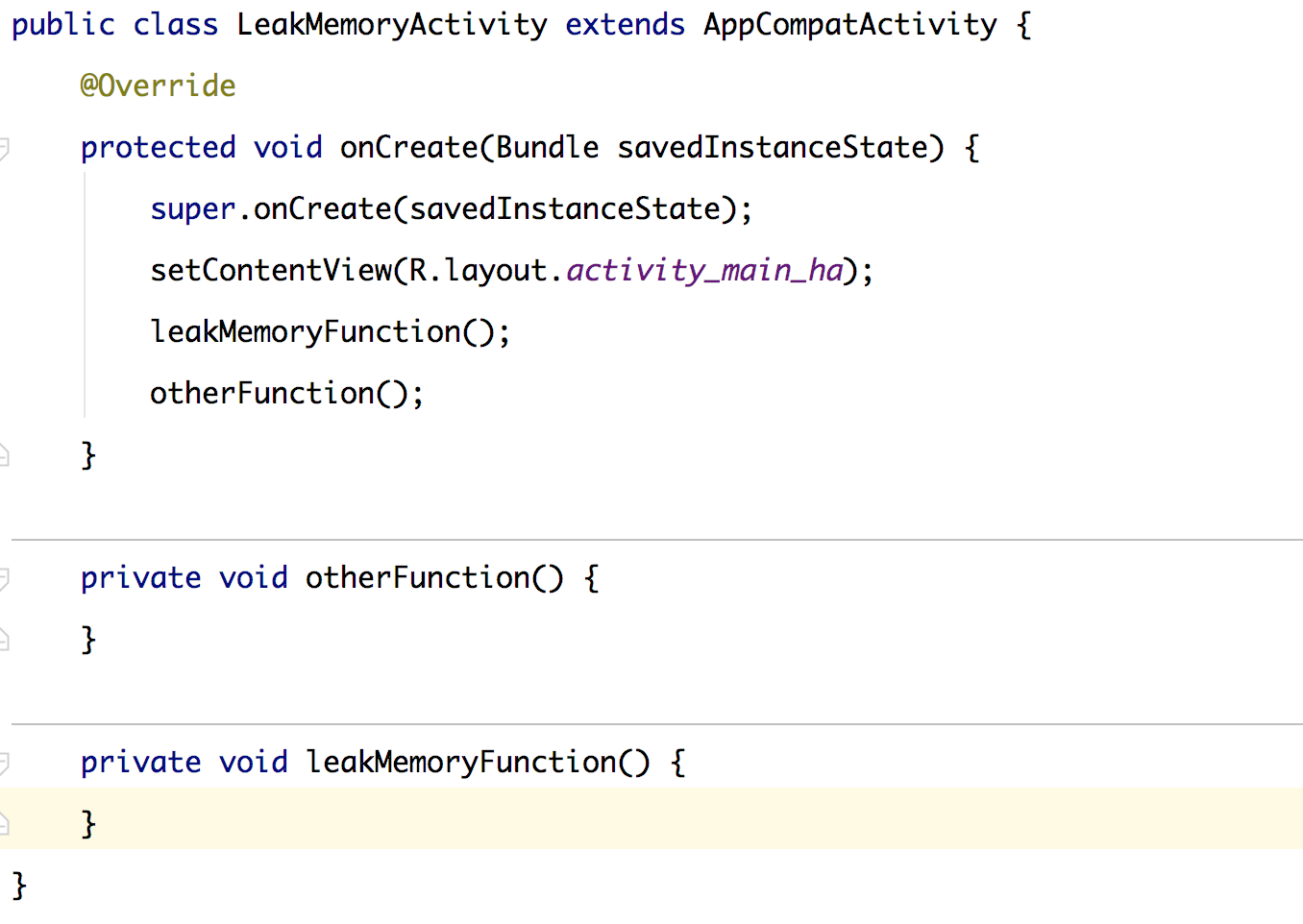
## 哪些操作可能造成短期的内存泄漏

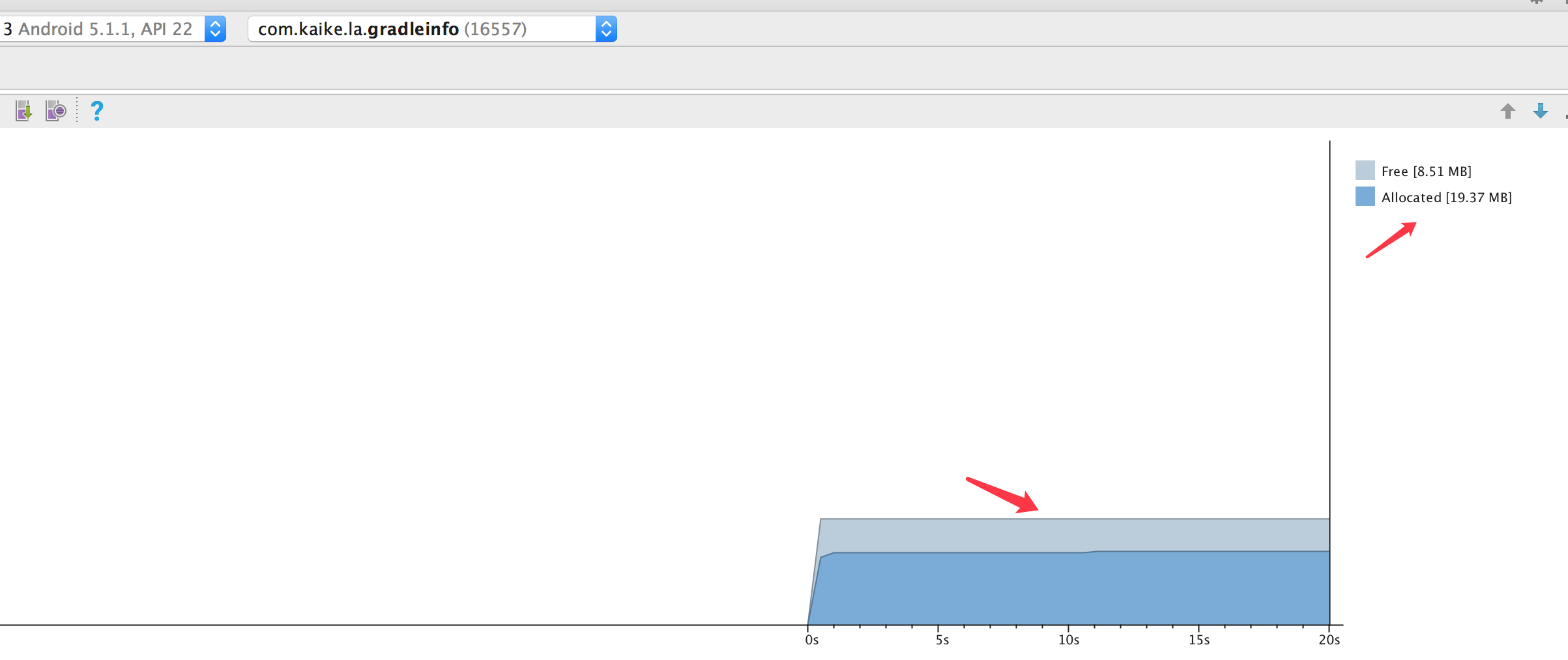
刚才基本是Activity层级的内存泄漏，这种泄漏是长期的，要优先解决，但Java层级的短期内存泄漏往往被Android开发者所忽略，OOM的产生就是长期的内存泄漏 + 短期的内存泄漏 + 高压力的内存使用造成的。

1. **多余的成员变量和临时变量**

在Java中 对象的引用是有作用域的，类中的成员变量会占用堆（heap）中一定的内存空间，如果该类持续被引用，那么该类中无效的成员变量也会因为强引用而无法释放。

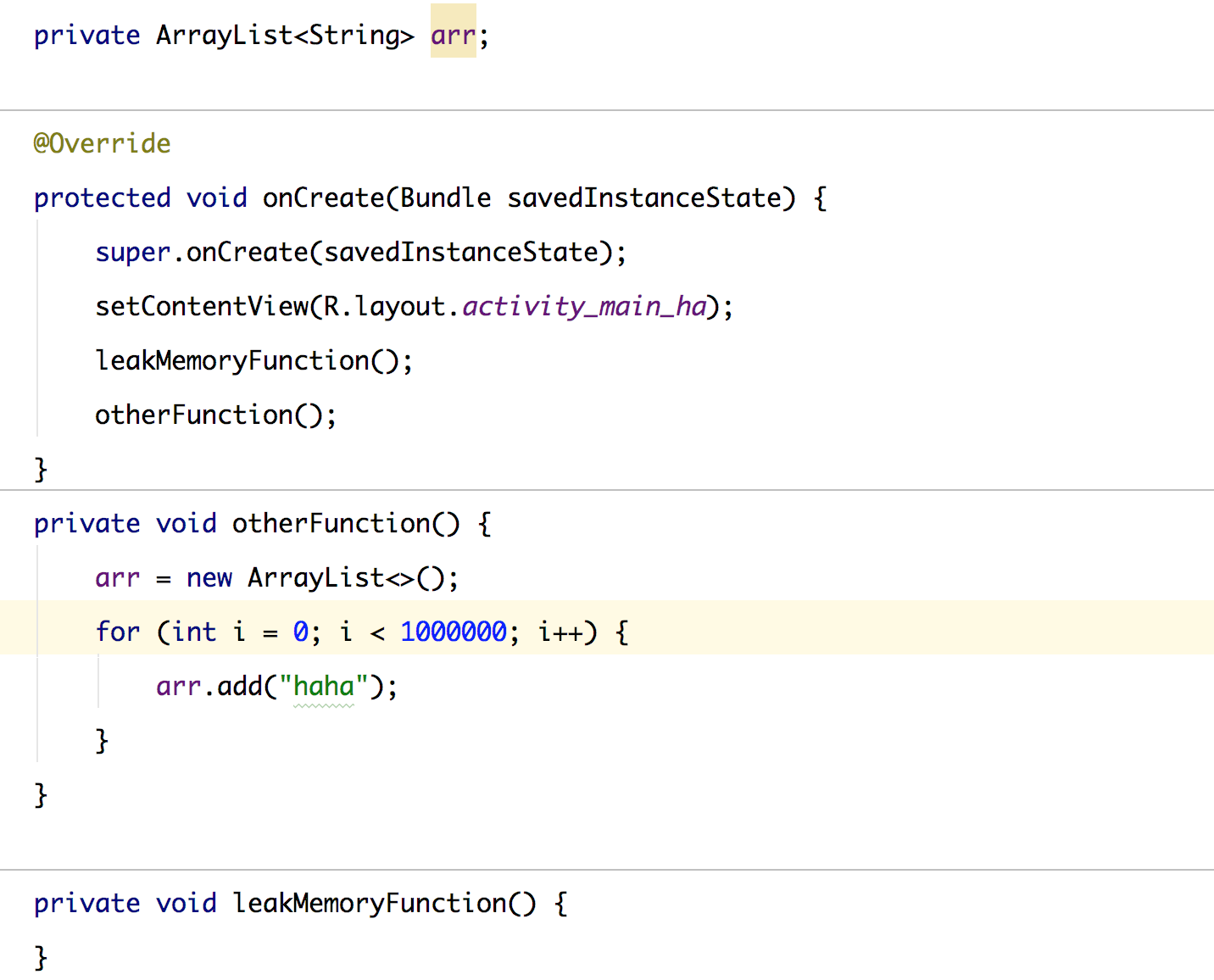
我们先来建个纯净的空页面：看下运行时候占用的内存空间



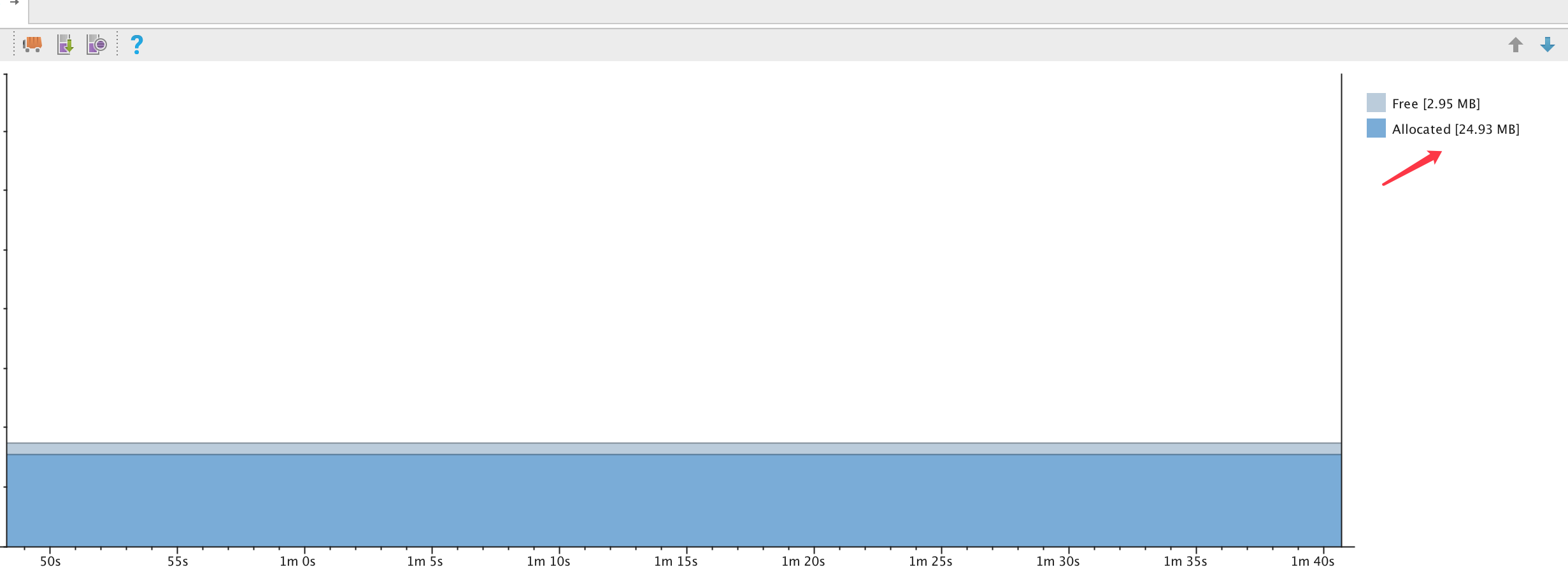


系统稳定在19.37MB。

下面我们在其中一个方法中创建成员变量。



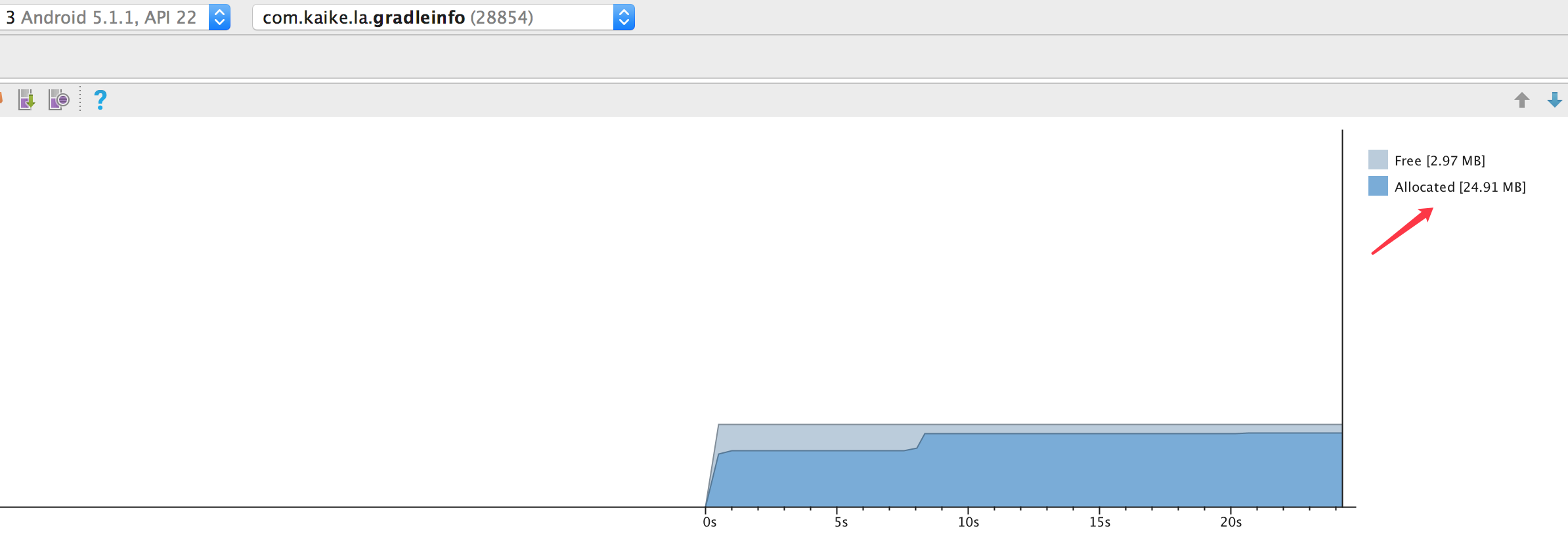
下面我们看下内存的使用，已经到达了24.93M：



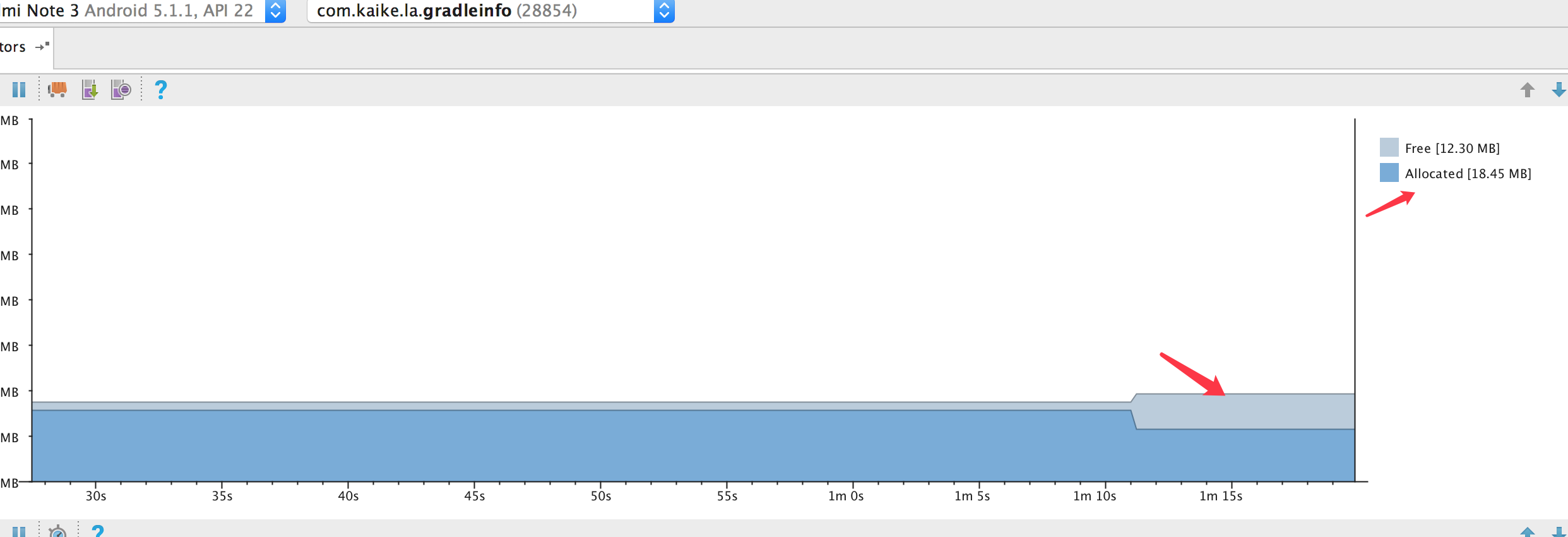
下面我们主动触发一下GC，系统到达了22.29M.



下面我们做下优化，由于该集合没有在其他方法中使用，那么我把成员变量转变成局部变量，看下使用的内存，跟使用成员变量差不多。



下面手动触发GC，可以发现竟然到达了空Activity的内存使用。



这样在低端机型上，当系统触发GC时可大大降低OOM的可能。

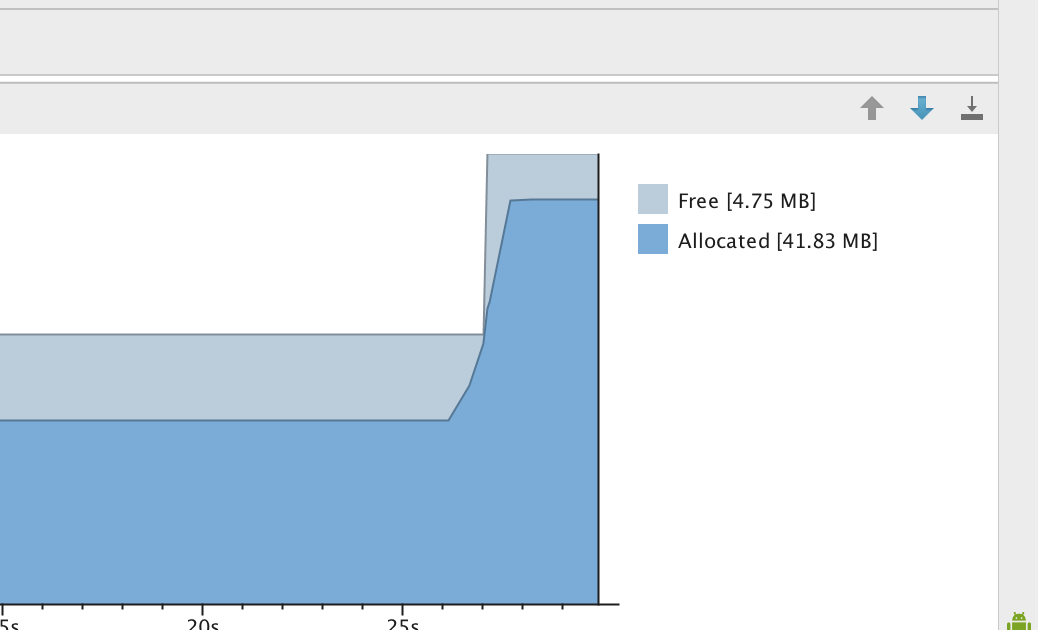
1. **没有在合适的时候释放成员变量**

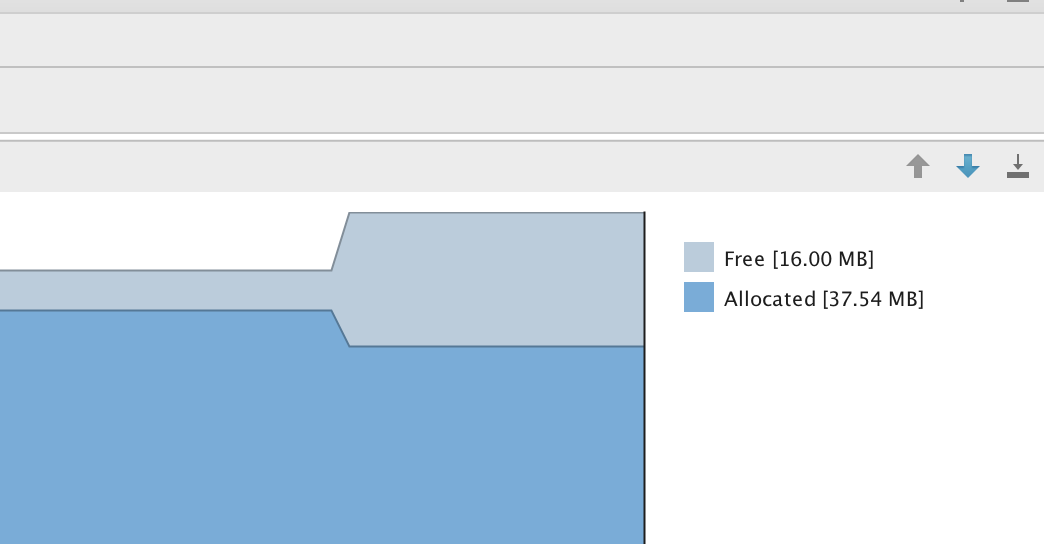
**相关场景：容器、Bitmap、资源文件、IO等。**

先展示下相关的耗时方法代码及大量产生的局部变量。



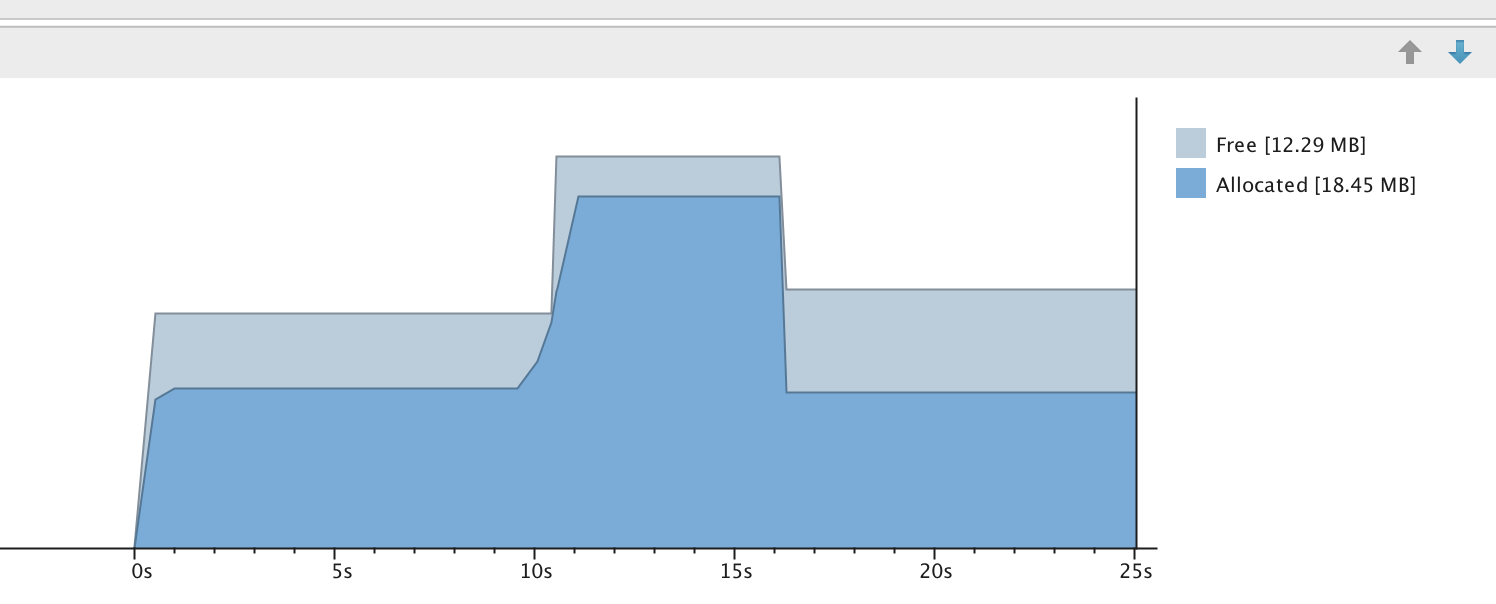
点击运行从19M上升到41，手动触发GC降到37M.





下面我们把代码改造一下，在合适的时候把无用的对象置为null，可以发现手动触发GC，应用使用的内存又回到了初始值

。

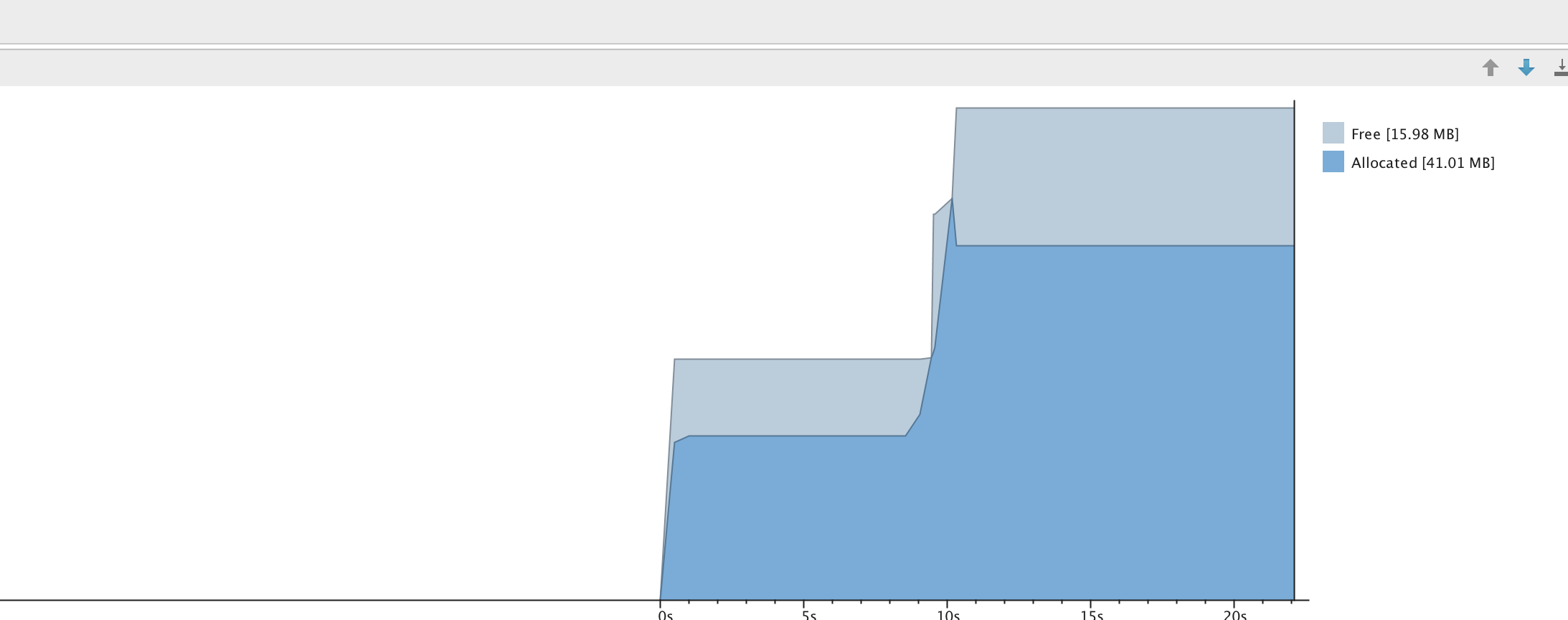


1. 比较大的容器，即使关闭界面也会造成短期内存问题，虽然不会造成长期的内存泄漏。

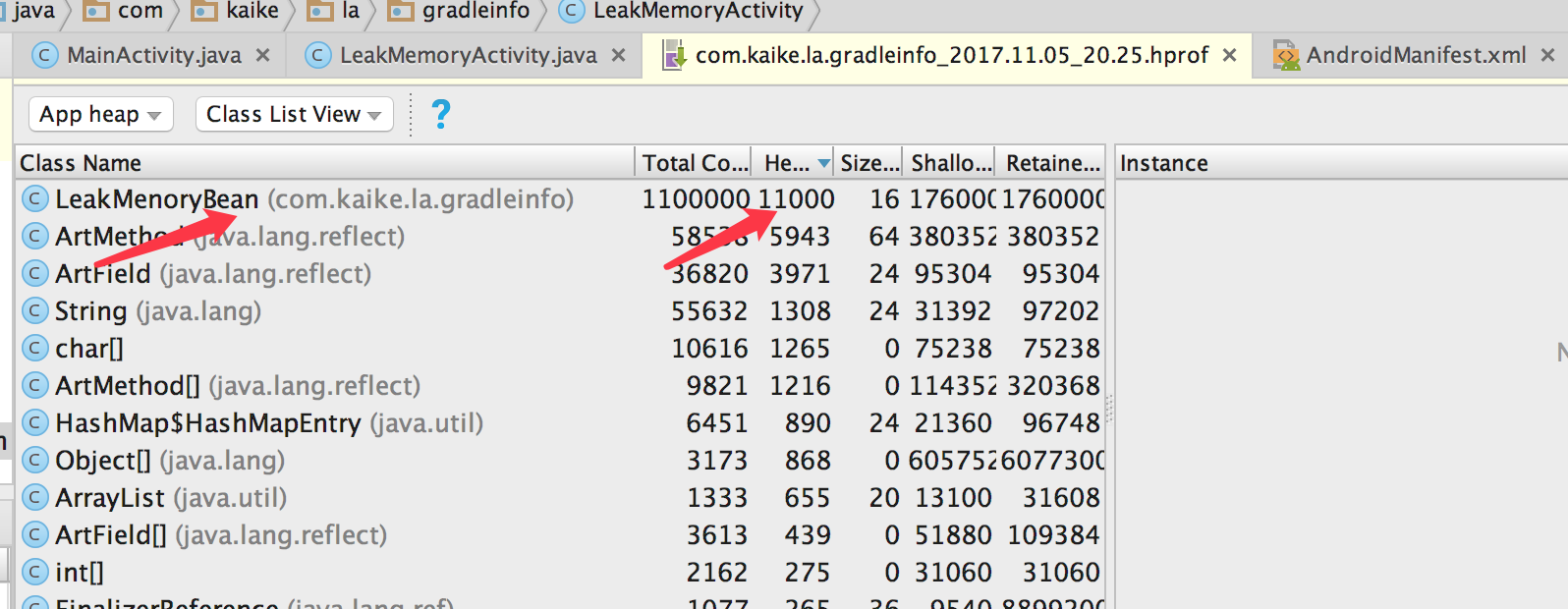
代码如下，这个界面有个较大的集合：



操作步骤如下：打开页面，再关闭页面，按理来说这个集合会从内存清除掉，但是页面关闭了，内存也没有降低下来。



我们来分析下此时的内存dump文件，看下为什么界面关闭了，内存还依然这么高。

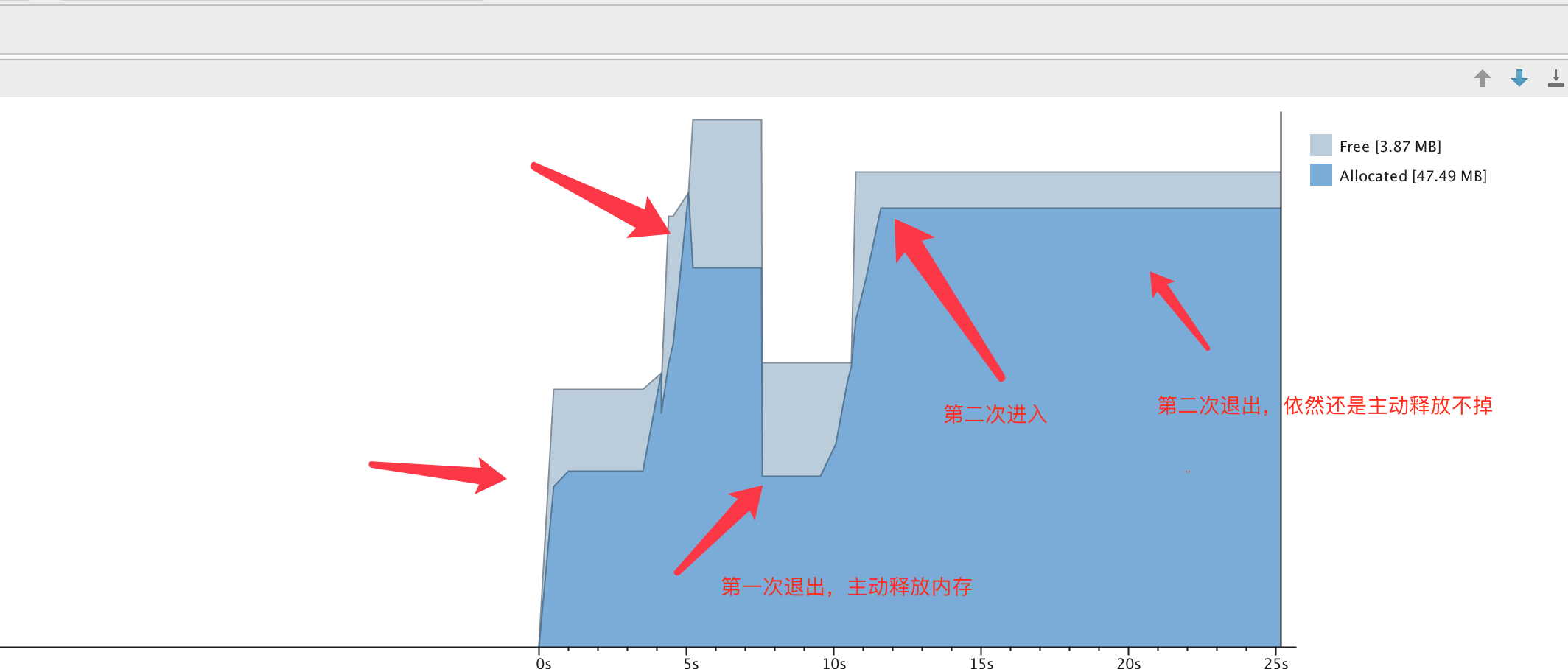


上图可看到这个对象还依然在内存里。

下面我们来加句代码。

@Override  
**protected void** onDestroy() {  
 **super**.onDestroy();  
 **arr**.clear();  
 **arr** = **null**;  
 System.*gc*();  
}

按理说这样总可以了吧，但是。。。。。 第一次打开有效，多次打开也会出现同样的问题。



那这样我们把这个集合改成软引用试试。

**private** SoftReference<ArrayList<LeakMenoryBean>> **soft**;

经过测试发现很容易空指针，在Activity里面做弱和软引用也不太现实。

目前来说这种如果想尽早释放，还是建议在 onDestroy里面做数据清空。

## 哪些操作可以减少内存使用压力

1. 较大的内存缓存类可使用弱引用和软引用，如BitMap。
2. 字符串拼接尽量使用StringBuffer和StringBudler。
3. 功能需要才去使用静态变量。
4. 大量的字符串存储建议使用文件存储，尽量不用sp。
5. 耗时操作要放入异步任务中。
6. 成员变量转化成局部变量，在适当的时机释放成员变量。
7. 尽量使用效率高的代码，比如增强for循环，减少使用枚举。
8. 三种常见的布局优化。
9. 方法中尽量减少多余的局部变量，比如 return 1 +2；而不是 int a = 1 +2：return a；
10. 尽量使用Android优化过的容器，如SparseArray,SparseBooleanArray, 与 LongSparseArray，但这种集合的使用场景并不一定适用多数场景。
11. 广播和service 的注册时机，并不一定是一进入App就去注册。
12. 三方库调研和对比，有些时候是某些库拖垮了我们的整个应用。
13. 多进程控制整个App的内存使用。
14. 三方图片加载框架在加载图片质量要求不高时，尽量不使用8888.
15. 适当使用ONtrimMemory 来处理资源的释放。