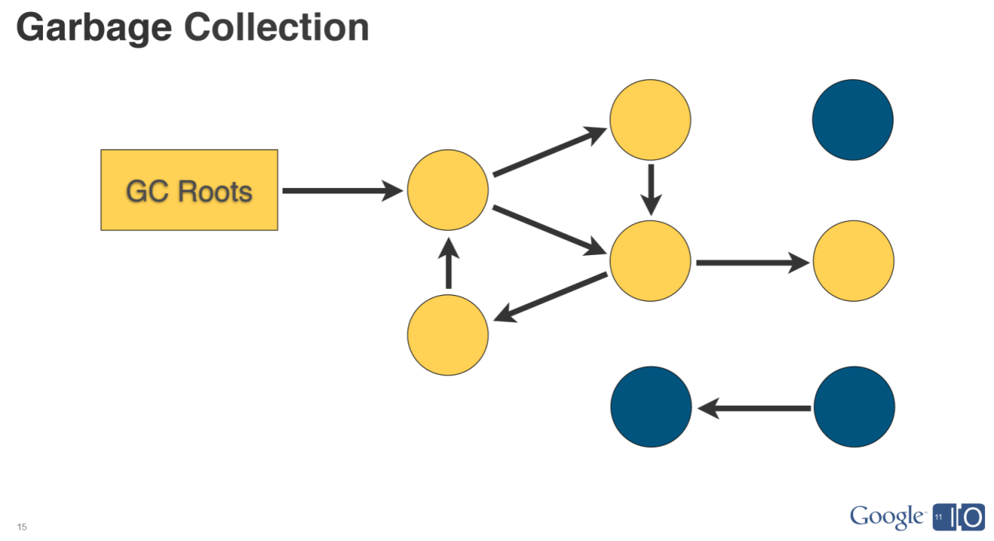
# **Android应用内存泄漏的定位、分析与解决策略**

## 什么是内存泄漏

对于不同的语言平台来说，进行标记回收内存的算法是不一样的，像 Android（Java）则采用 GC-Root 的标记回收算法。下面这张图就展示了 Android 内存的回收管理策略（图来自Google 2011的IO大会）



图中的每个圆节点代表对象的内存资源，箭头代表可达路径。当圆节点与 GC Roots 存在可达路径时，表示当前资源正被引用，虚拟机是无法对其进行回收的（如图中的黄色节点）。反过来，如果圆节点与 GC Roots 不存在可达路径，则意味着这块对象的内存资源不再被程序引用，系统虚拟机可以在 GC 过程中将其回收掉。

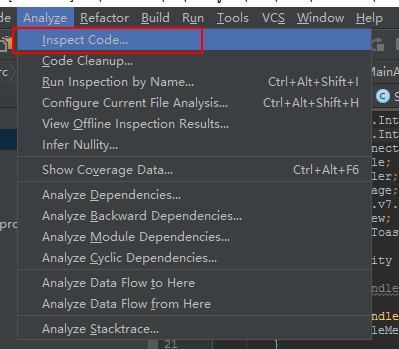
* 有了上面的内存回收的栗子，那么接下来就可以说说什么是内存泄漏了。从定义上讲，Android（Java）平台的内存泄漏是指**没有用的对象资源仍与GC-Root保持可达路径，导致系统无法进行回收**。举一个最简单的栗子，我们在 Activity 的 onCreate 函数中注册一个广播接收者，但是在 onDestory 函数中并没有执行反注册，当 Activity 被 finish 掉时，Activity 对象已经走完了自身的生命周期，应该被资源回收释放掉，但由于没有反注册， 此时 Activity 和 GC-Root 间任然有可达路径存在，导致 Activity 虽然被销毁，但是所占用的内存资源却无法被回收掉。

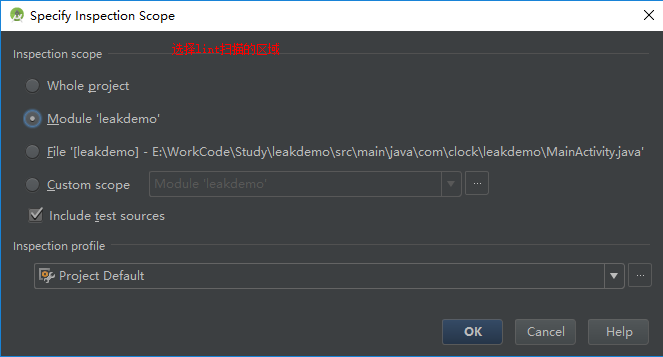
## 泄漏的定位

内存泄漏不像闪退的BUG，排查起来相对要比较困难些，比较极端的情况是当你的应用 OOM 了才发现存在内存泄漏问题，到了这种情况才去排查处理问题的话，对用户的影响就太大了。为此，我们能够在编码中尽早发现到问题就不要拖到上线之后才去填坑，下面介绍一些我比较常用排查内存泄漏的工具。

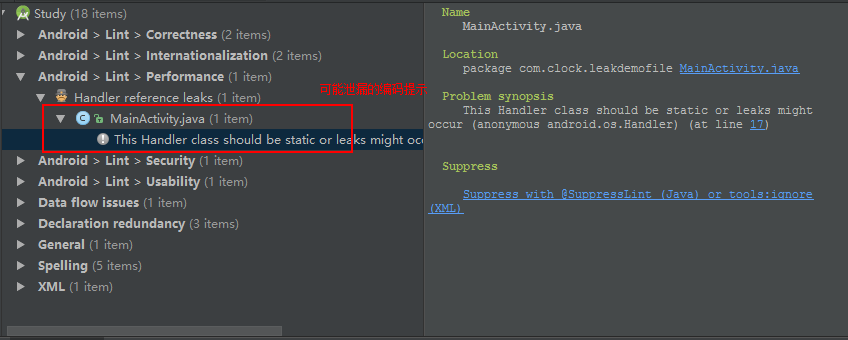
* 静态代码分析工具 —— Lint

Lint 是 Android Studio 自带的工具，使用姿势很简单 Analyze -> Inspect Code 然后选择想要扫面的区域即可





对可能引起泄漏的编码，Lint 都会进行温馨提示。



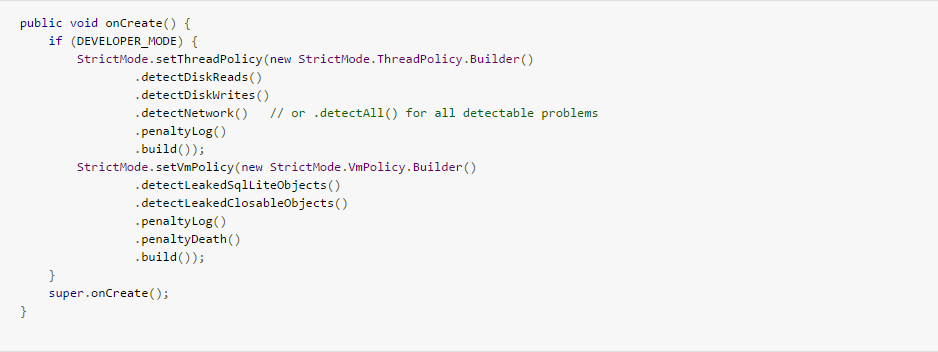
这里只是抛砖引玉的介绍 Lint ，实际上玩法还有很多，大家可以自行拓展学习。除了 Lint 外，还有像 FindBugs 、 Checkstyle 等静态代码分析工具也是很不错的。

* 严苛模式 —— StrictMode

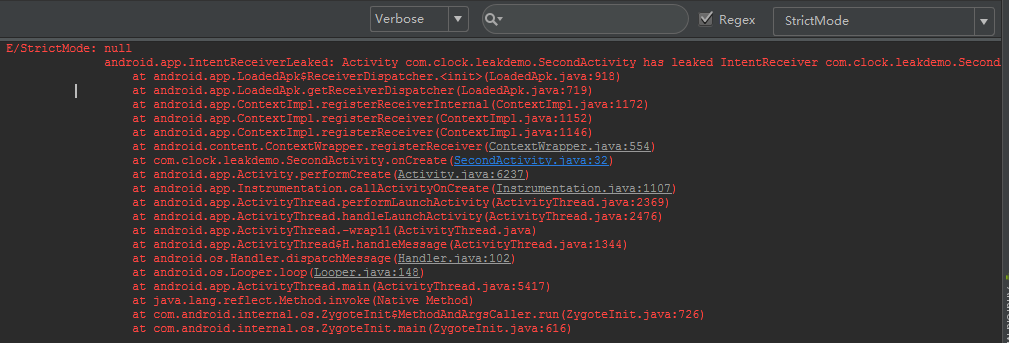
StrictMode 是 Android 系统提供的 API ，在开发环境下引入可以更早的暴露发现问题。官方文档链接在下面（需要科学上网）：

[https://developer.android.com/reference/android/os/StrictMode.html](https://developer.android.com/reference/android/os/StrictMode.html" \t "_blank)

以官网的示例代码为栗子，一般 StrictMode 只在测试环境下启用，到了生产环境就会进行关闭，通常我们都会借助 BuildConfig.DEBUG 来实现。



启用 StrictMode 后，在过滤日志的地方加上 StrictMode 的过滤 Tag ，如果手机连接着电脑进行开发，定期观察一下 StrictMode 这个 Tag 下的日志，一般你看到一大堆红色告警的 Log，就需要好好排查一下是否跟内存泄漏有关了。



* LeakCanary

Square 公司出品的内存分析工具，官方地址如下：

[https://github.com/square/leakcanary/](https://github.com/square/leakcanary/" \t "_blank)

LeakCanary 和 StrictMode 一样，需要在项目代码中集成，不过代码也非常简单，如下的官方示例。



build.gradle 引入，Application 中加入两三行代码，即可搞定。以上只是简单的引入，还有更多使用姿势建议详细阅读它的 Wiki 下 FAQ：

[https://github.com/square/leakcanary/wiki/FAQ](https://github.com/square/leakcanary/wiki/FAQ" \t "_blank)

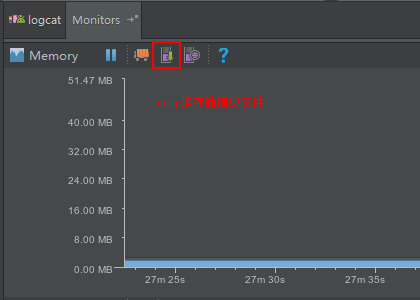
我对使用 LeakCanary 有以下两点感受：

当内存泄漏发生时，LeakCanary 会弹窗提示并生成对应的堆存储信息记录，这让我们对隐蔽的内存泄漏问题有了更加直观的感觉，但从实际使用来看，LeakCanary 的每个提示也并非是真正存在内存泄漏问题，要想确定是否存在问题我们还需要借助 MAT 来进行最后的确定。

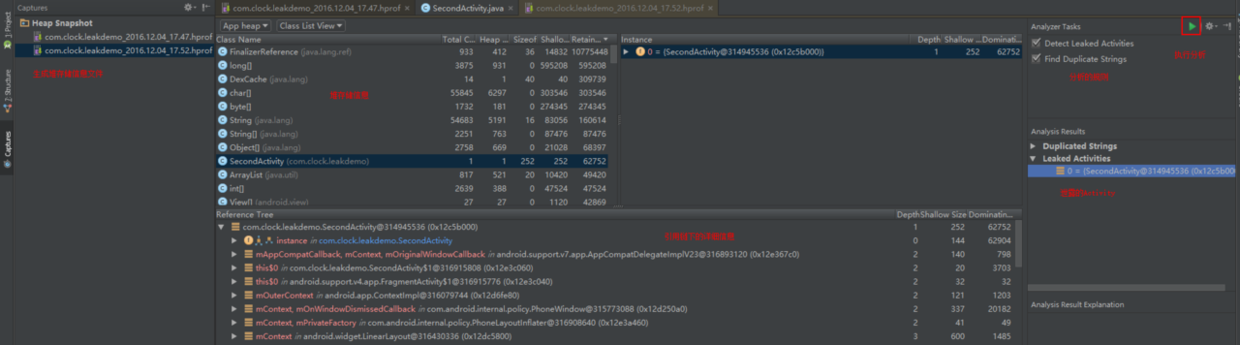
Android 系统本身就存在一些问题导致应用内存泄漏，LeakCanary 的 [AndroidExcludedRefs](https://github.com/square/leakcanary/blob/9e74a8529ca94287fe0c3b02b7a6b39d51ecd704/leakcanary-android/src/main/java/com/squareup/leakcanary/AndroidExcludedRefs.java" \t "_blank) 类帮助我们处理了不少这类问题。

Android Memory Monitor

AndroidStudio 提供的工具，用于监控应用的内存使用状态，在开发中也是非常实用的工具，可以用来打印出内存的状态信息。



打印获得的内存信息如下，可以通过右上角的绿色三角形按钮去分析泄漏的 Activity 和 一些重复的字符串，目前只支持这两个，希望 Google 后面能够加入更多可选分析规则

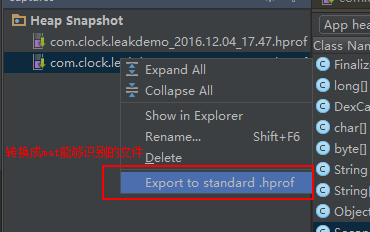


同样，这里也只是抛砖引玉的简单介绍，关于它的使用在官方文档已经说得很详细了，需要的童鞋自行查看下方链接（需科学上网）：

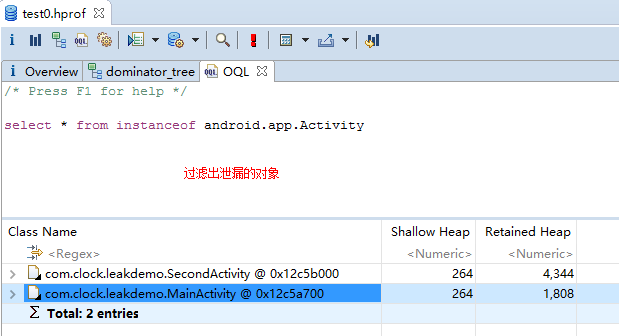
[https://developer.android.com/studio/profile/am-hprof.html](https://developer.android.com/studio/profile/am-hprof.html" \t "_blank)

* Memory Analyzer (MAT)

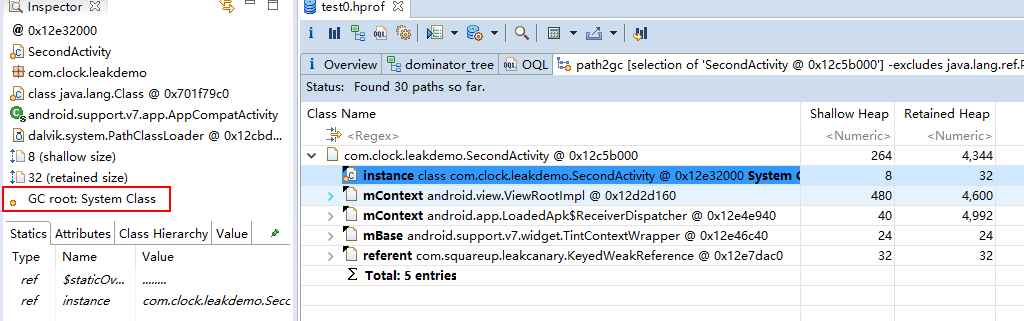
老牌子分析工具，可以从 [http://www.eclipse.org/mat/](http://www.eclipse.org/mat/" \t "_blank) 下载获得，网上关于 MAT 使用的文章好多，大家可以自行查找。上面的 Android Memory Monitor 生成的对储存信息文件可以配置 MAT 一起来分析使用，由于 Android Memory Monitor 生成的 hprof 文件不是标准格式，所以需要做一下转换，然后导入 MAT



然后通过 OQL 先定位出泄漏的对象



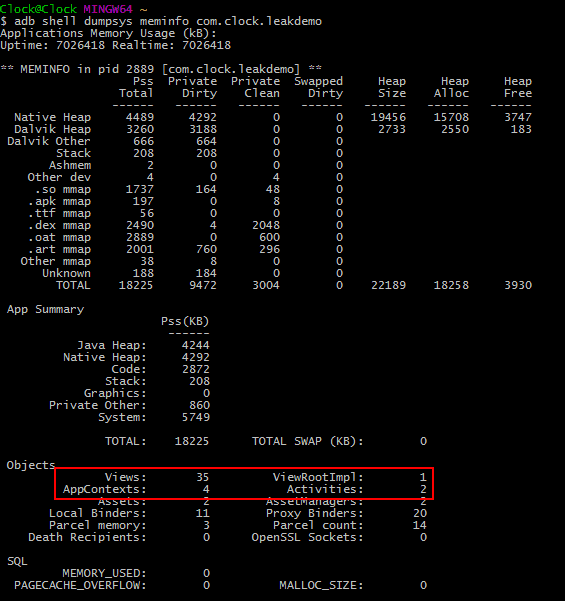
通过排除除了强引用之外的其他引用链，最后分析到 GC Root 的位置



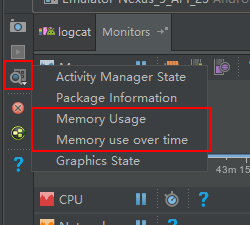
MAT 使用起来相对繁琐，但不失为定位根源问题的利器。

* adb shell 命令

使用 adb shell dumpsys meminfo [PackageName]，可以打印出指定包名的应用内存信息



使用该命令可以很直观的观察到 Activity 的泄漏问题，是我平常分析比较常用的一种方式。除了使用命令外，AndroidStudio 也提供了下面的功能，和使用命令是一样效果的。



如果对 adb shell 命令感兴趣，更多的信息可以看下面提供的资源：

* [http://adbshell.com/](http://adbshell.com/" \t "_blank)
* [https://github.com/mzlogin/awesome-adb](https://github.com/mzlogin/awesome-adb" \t "_blank)

以上就是我在做内存泄漏分析的时候会用到的工具，通常都是结合起来用，毕竟每个工具都有优缺点，通过使用多个工具互补分析问题可以极大的提高我们的效率和最终取得的效果。

## 泄漏的解决策略

聊完工具，最后来谈谈内存泄漏问题的解决策略。我把它总结为以下三点：

* 完成需求功能开发后，再去优化内存泄漏问题；
* 泄漏源有多处时，核心功能产生的泄漏优先处理，用户使用频繁的功能引起的泄漏优先处理；
* 处理泄漏避免影响原有的代码逻辑，优化过后最好能够让测试童鞋过一遍相关的功能，避免引入未知的BUG；

## 总结

对于如何在编码上去解决内存泄漏问题，网络上有提供了很多场景及其解决方案，大家可以自行借助搜索引擎。通过掌握分析方法和对泄漏场景及其解决方案的积累，相信大家处理内存泄漏问题是游刃有余的。当然，也并不是所有内存泄漏问题我们都能够进行处理，就例如第二章节提到的泄漏源头是由第三方代码引起时，我们就显得无能为力了。最近在排查的过程中就发现不少第三方 SDK 存在泄漏问题，遇上这种情况就得找找可替代的 SDK 进行更换了。以上就是我做内存泄漏分析的一些心得总结，如果有错误和不足，还请大家指出。