**内存泄漏**

**一、为什么要解决内存泄漏**

**什么是内存泄露？**

如果一个无用对象（不需要再使用的对象）仍然被其他对象持有引用，**造成该对象无法被系统及时回收**，以致该对象在堆中所占用的内存单元无法被释放而造成内存空间浪费，这种情况就是内存泄露。

**为什么要解决内存泄露？**

内存泄漏并不会导致程序功能异常，但是它会导致Android程序的内存占用过大。系统给每一个应用分配的内存是固定的，一旦发生了内存泄露，就会导致该应用可用内存越来越小，严重时会发生内存溢出OOM导致 Force Close。（如要更多内存可分多个进程）

**扩展：内存溢出？图片压缩？三级缓存？**

1. **Java垃圾回收机制**
2. **Java内存分配策略**

Java程序运行时的内存分配策略有三种，分别是静态分配，栈式分配和堆式分配。对应的，三种存储策略使用的内存空间主要分别是静态存储区、栈区和堆区。

* 静态存储区：编译时就分配好，在程序整个运行期间都存在。主要存放静态数据和常量。
* 栈区：当方法执行时，会在栈去内存中创建方法体内部的局部变量，方法结束后自动释放内存。
* **堆区（heap）**：通常存放new出来的对象，由Java垃圾回收器管理内存的回收。

内存泄露主要关注的是堆区；

**2、java垃圾回收机制（GC）检测策略**

**1）引用计数（缺点）**

首先介绍一种用于说明垃圾收集工作方式的策略，\*\* 引用计数 \*\*：

每个对象都含有一个引用计数器，当有引用连接至对象时，引用计数加1。当引用离开作用域或者被置为null时，引用计数减1。垃圾回收器在遍历所有对象时**发现引用计数为0便释放其内存**。这种策略很难处理循环引用的情况。不过我们无需过多的考虑此策略有何优缺点，这仅仅是用来让你了解一些垃圾回收的工作方式。而且**现在JVM大多也不用这种策略来进行垃圾回收。**

**2）可达性分析算法（根搜索算法）**

既然引用计数有缺点，那么可以采用其他的策略，Java采用了一种新的算法：可达性分析算法。

对象引用遍历从一组对象开始（GC Roots），沿着整个对象图上的每条链接，递归确定可到达（reachable）对象并生成一棵引用树，树的节点视为可达对象，反之视为不可达。之后垃圾回收器在进行垃圾回收的时候便可以回收那些**不可达**的对象。

**3、Java的引用类型**

**1）强引用（Strong Reference）**

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器**绝不会回收**它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**2）软引用（Soft Reference）**

如果一个对象只具有软引用，那么如果内存**空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存**。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中

**作用：**SoftReference多用作来实现缓存机制(cache)，如图片加载框架ImagLoader等;

**3）弱引用（Weak Reference）**

如果一个对象只具有弱引用，那么在垃圾回收器线程扫描的过程中，一旦发现了只具有弱引用的对象，**不管当前内存空间足够与否，都会回收它的内存**。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。弱引用也可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

**作用：**WeakReference一般用来防止内存泄漏

**4）虚引用（Phantom Reference）**

略

参考文献：

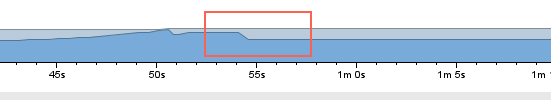
<https://www.jianshu.com/p/96149e01486f>

<https://www.jianshu.com/p/8488079a939b>

**三、内存检测工具**

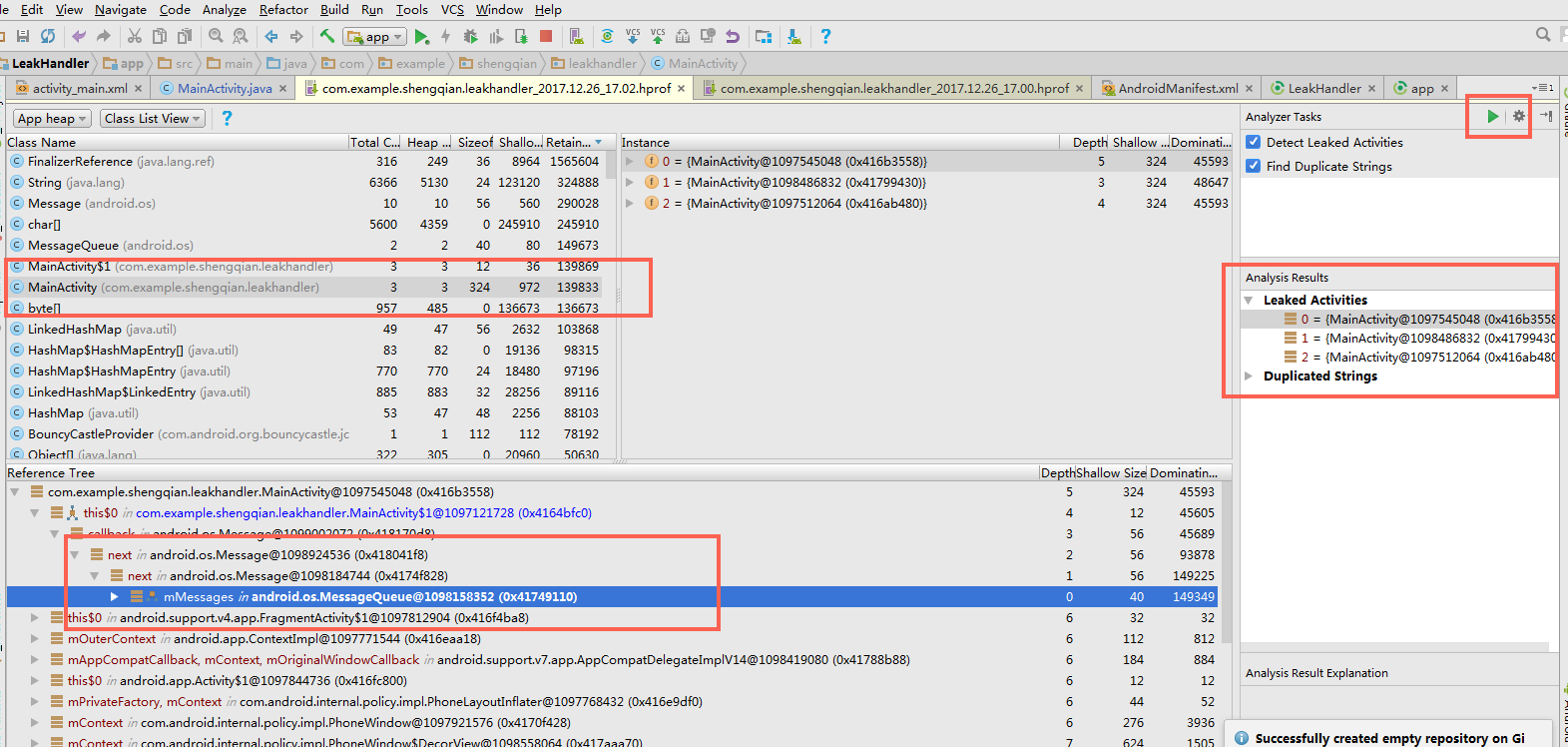
**1、使用Android studio检测泄露**

1）通过GC释放内存，查看是否有内存泄露



当内存泄露时，反复操作内存会增加，而由于导致泄露的引用为强引用，则GC不能有效清理

2）通过hprof文件，分析泄露的Activity及其引用树



参考文献：<https://www.cnblogs.com/taoweiji/p/5760537.html>

**2、使用MAT检测泄露**

MAT全称Eclipse Memory Analyzer

**MAT的安装：**

1）Eclipse Mat插件安装

2）单独安装Mat

Mat工具地址：<https://www.eclipse.org/mat/>

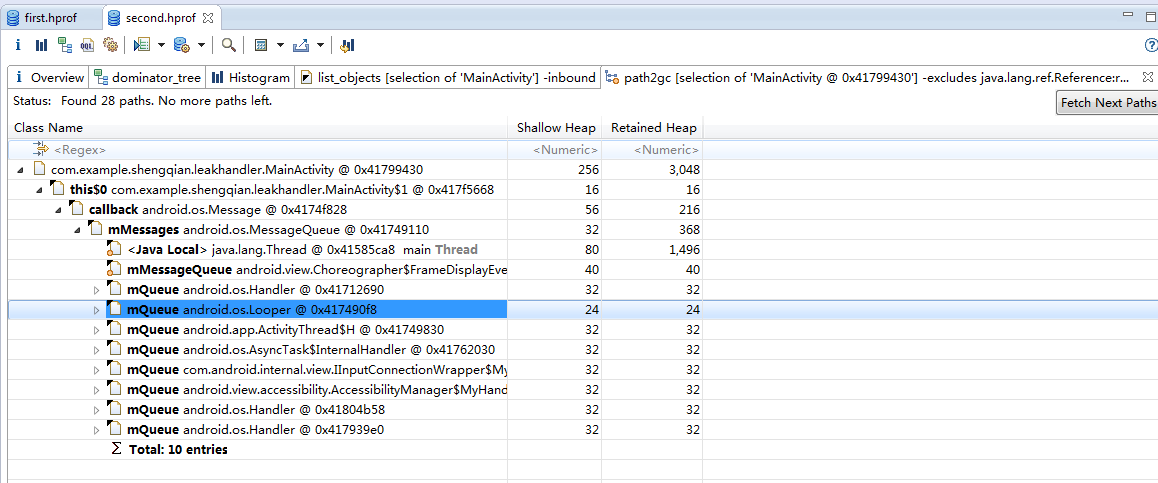
**MAT检测步骤：**

1）获取 hprof文件

2）转换 hprof文件（**如果是用 MAT Eclipse 插件获取的 Dump文件，则不需要经过转换**）

3）Historgram查询泄露的Activity

4）Path to GC Roots分析该Activity的引用树



参考文献：<https://www.cnblogs.com/larack/p/6071209.html>

**3、使用Leakcanary检测泄露**

1. **Leakcanary原理**

Github地址：<https://github.com/square/leakcanary>

基本就是新建一个WeakReference对象指向要关注的Activity，Activity被回收的时候**WeakReference对象会加到ReferenceQueue队列中，检测ReferenceQueue队列是否有WeakReference对象就知道有没有泄露了**，有的话就没有泄露，没有的话就泄露！

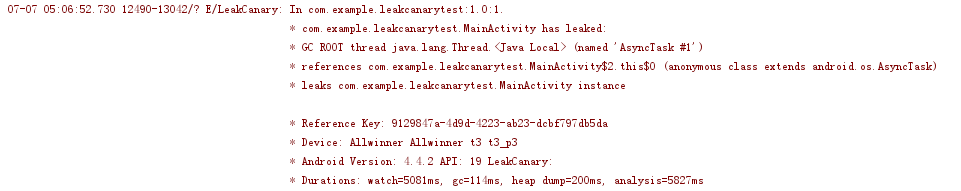
参考文献：

<https://www.jianshu.com/p/51395d8e512f>

<https://www.jianshu.com/p/0448dab89625>

1. **集成soling Leakcanary jar包的问题**

说明：Leakcanary jar包中，已去除Leakcanary通知栏显示、快捷键Activity显示、资源文件，最终内存泄露结果只以log显示！（检测Tag为LeakCanary）



固件如果集成LeakCanary，则可以使用动态引用jar包，应用Realease版本注释掉相关调用；

集成步骤：

1）确保sdk下已编译leakcanarylib生成jar包

2）在App项目的Android.mk中对该jar包进行引用

3）在项目AndroidMainfiest.xml中注册组件HeapAnalyzerService、DisplayLeakService（已经不需要声明sd卡读写权限和注册Activity）

具体集成文档请参考gitlab仓库的ReadMe文件；

<http://gitlab.slcisv.com/shengqian/leakcanarylib.git>

**四、常见内存泄漏的原因及解决方案**

**1、静态变量导致的内存泄漏**

因为static变量的生命周期是在**类加载时开始、类卸载时结束**，也就是说static变量是在程序进程死亡时才释放。

如果在static变量中引用了Activity，那么这个Activity由于被引用，便会随static变量的生命周期一样，一直无法被释放，造成内存泄漏！

**解决办法：**   
1）在Activity被静态变量引用时，使用getApplicationContext；因为Application生命周期从程序开始到结束，和static变量的一样；

2）及时置空null；

1. **单例模式模式导致的内存泄漏**

单例都是静态对象，而静态对象的生命周期和app一样。

如果子单例中引用了Activity，那么activity不能正常销毁！

**解决办法：**

1）我们在构造单例模式时，尽量避免使用Activity的上下文，而使用Application的上下文

2）如果要使用Activity的上下文，则应该及时置空释放（如在onDestory时解除注册）

1. **属性动画导致的内存泄漏**

从Android 3.0开始，Google提供了属性动画，属性动画中有一类无线循环的动画，如果在Activity中播放此类动画且没有在onDestroy中停止动画，那么动画会一直播放下去，尽管已经无法在界面上看到动画效果了，并且这个时候Activity的View会被动画持有，而View又持有了Activity，最终Activity无法释放。

**解决办法：**

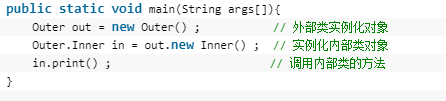
在Activity的onDestroy中调用animator.cancel()来停止动画；

**扩展：属性动画的原理？属性动画与View动画的区别？属性动画在自定义View中的使用？插值器？**

**4、非静态内部类导致内存泄露（常见）**

非静态内部类（包括匿名内部类）默认就会**隐性地持有外部类的引用**，**当非静态内部类对象的生命周期比外部类对象的生命周期长时**，就会导致内存泄露。

从外部调用内部类也可以发现，如下：



常见场景如：handler内部类、Thread内部类等等…

**解决办法：**

1. 使用静态内部类代替非静态内部类（或匿名内部类）
2. 保持内部类对象的生命周期与外部类（Activity）的生命周期一致

**5、Handler导致的内存泄漏**

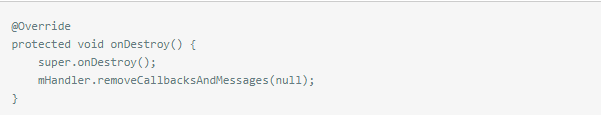
Handler原理：

1. Looper
2. Message
3. MessageQueue

Activity和Looper的生命周期不一样，**Looper伴随整个Application；**当Handler处理延时任务、延时消息时，则会造成内存泄露

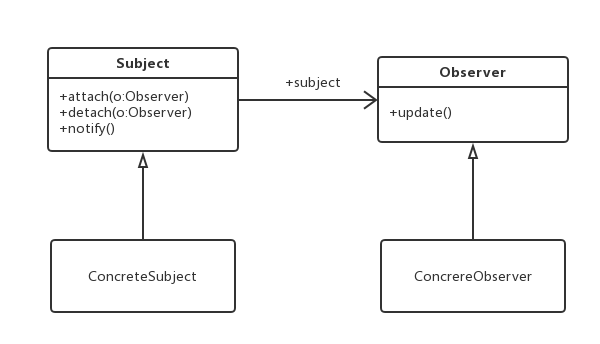
**解决办法：**

1. 使用静态内部类代替非静态内部类
2. 使用WeakReference弱引用代替强引用（注意判空！）
3. **在Activity销毁时就将mHandler的回调和发送的消息给移除掉**



**6、未取消注册或回调导致内存泄露**

观察者模式：



观察者模式UML

如果被观察者为Activity，当该Activity销毁时不及时解除注册的话，则会引起内存泄露！

**解决办法：如果被观察者为Activity，则应该及时的调用观察者的unRegiseter()解除注册**

**7、MVP框架中的内存泄漏**

MVP架构：

1.Model层：获取数据

2.View层：实现类执行回调的逻辑

3.Presenter：层解除M和V的耦合，使M和V通过P层交互。

MVP相对于MVC来说，解除了M和V的耦合，他们俩互不感知，但是P层作为中间交互层不得不持有一个V层的引用和一个M层的实例。而当M层在进行一个耗时的操作时，由于P层是调用M层的逻辑实现一些功能，所以也可以将P层视为是一个耗时的操作。而且前面也说了，**P层会持有一个V层的引用**，如果在这个时候我们想要销毁这个Activity，那么这个Activity因为仍有P在持有Activity的引用从而导致其不会被回收，也就导致了内存泄露！

**解决办法：我们将presenter的生命周期和Activity的生命周期关联起来**

* 在presenter中声明一个onDestroy()方法，在这个方法中将mView（Activity）置为null，然后在presenter中凡是使用到mView的地方，都判断一下是否为空。
* 在activity的onDestroy()方法中调用presenter.onDestroy()，同时也将activity持有的presenter置空

参考文献：<http://blog.csdn.net/yulong0809/article/details/78622428>