[**Android 内存泄露实践分析**](http://blog.csdn.net/ouyang_peng/article/details/52577261)

2016-09-18 19:53 795人阅读 [评论](http://blog.csdn.net/ouyang_peng/article/details/52577261#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/ouyang_peng/article/details/52577261#report)

分类：

Android应用开发*（289）*

目录[(?)[+]](http://blog.csdn.net/ouyang_peng/article/details/52577261)

今天看到一篇关于Android 内存泄露实践分析的文章，感觉不错，讲的还算详细，mark到这里。

**原文发表于:Testerhome；**

**作者：ycwdaaaa ;**

**原文链接：https://testerhome.com/topics/5822**

**定义**

​内存泄漏也称作“存储渗漏”，用[动态存储](http://baike.baidu.com/view/589359.htm)分配函数动态开辟的空间，在使用完毕后未释放，结果导致一直占据该内存单元。直到程序结束。（其实说白了就是该内存空间使用完毕之后未回收）即所谓内存泄漏。

内存泄漏形象的比喻是“[**操作系统**](http://lib.csdn.net/base/operatingsystem)可提供给所有进程的存储空间正在被某个进程榨干”，最终结果是程序运行时间越长，占用存储空间越来越多，最终用尽全部存储空间，整个[系统崩溃](http://baike.baidu.com/view/765121.htm)。所以“内存泄漏”是从操作系统的角度来看的。这里的存储空间并不是指物理内存，而是指虚拟内存大小，这个虚拟内存大小取决于磁盘交换区设定的大小。由程序申请的一块内存，如果没有任何一个[指针](http://baike.baidu.com/view/159417.htm)指向它，那么这块内存就泄漏了。

​ ——来自《百度百科》

**影响**

* 导致OOM
* 糟糕的用户体验
* 鸡肋的App存活率

**成效**

* 内存泄露是一个持续的过程，随着版本的迭代，效果越明显
* 由于某些原因无法改善的泄露（如框架限制），则尽量降低泄露的内存大小
* 内存泄露实施后的版本，一定要验证，不必马上推行到正式版，可作为beta版持续观察是否影响/引发其他功能/问题

内存泄露实施后，项目的收获：

* OOM减少30%以上
* 平均使用内存从80M稳定到40M左右
* 用户体验上升，流畅度提升
* 存活率上升，推送到达率提升

**类型**

* IO
  + FileStream
  + Cursor
* Bitmap
* Context
  + 单例
  + Callback
* Service
  + BraodcastReceiver
  + ContentObserver
* Handler
* Thread

**技巧**

* 慎用Context
  + [Context概念](http://blog.csdn.net/lmj623565791/article/details/40481055)
  + 四大组件Context和Application的context使用参见下表

​

* 善用Reference
  + [Java引用介绍](http://blog.csdn.net/mazhimazh/article/details/19752475)
  + Java四种引用由高到低依次为：强引用  >  软引用  >  弱引用  >  虚引用
  + 表格说明
* 复用ConvertView
  + [复用详解](http://blog.csdn.net/lmj623565791/article/details/24333277)
* 对象释放
  + 遵循谁创建谁释放的原则
  + 示例：显示调用clear列表、对象赋空值

**分析**

​ **原理**

* [Java内存分配机制](http://blog.csdn.net/shimiso/article/details/8595564)
* [Java垃圾回收机制](http://www.cnblogs.com/sunniest/p/4575144.html)

​ **根本原因**

* 关注堆内存

​ **怎么解决**

* 详见方案

​ **实践分析**

* 详见实践

**方案**

* StrictMode
  + 使用方法：AppContext的onCreate()方法加上
* StrictMode.setThreadPolicy(**new** StrictMode.ThreadPolicy
* .Builder()
* .detectAll()
* .penaltyLog()
* .build());
* StrictMode.setVmPolicy(**new** StrictMode.VmPolicy
* .Builder()
* .detectAll()
* .penaltyLog()

.build());

* + 主要检查项：内存泄露、耗时操作等
* Leakcanary
  + [GitHub地址](https://github.com/square/leakcanary)
  + [使用方法](http://www.liaohuqiu.net/cn/posts/leak-canary-read-me/)
* Leakcanary + StrictMode + monkey **（推荐）**
  + 使用阶段：功能测试完成后，稳定性测试开始时
  + 使用方法：安装集成了Leakcanary的包，跑monkey
  + 收获阶段：一段时间后，会发现出现N个泄露
  + 实战分析：逐条分析每个泄露并改善/修复
  + StrictMode：查看日志搜索StrictMode关键字
* Adb命令
  + 手动触发GC
  + 通过adb shell dumpsys meminfo packagename -d查看
  + 查看Activity以及View的数量
  + 越接近0越好
  + 对比进入Activity以及View前的数量和退出Activity以及View后的数量判断
* [**Android**](http://lib.csdn.net/base/android) Monitor
  + [使用介绍](http://wetest.qq.com/lab/view/?id=99)
* MAT
  + [使用介绍](http://blog.csdn.net/xiaanming/article/details/42396507)

**实践（示例）**

**Bitmap泄露**

Bitmap泄露一般会泄露较多内存，视图片大小、位图而定

* 经典场景：App启动图
* 解决内存泄露前后内存相差10M+，可谓惊人
* 解决方案：

App启动图Activity的onDestroy()中及时回收内存

@Override

**protected** **void** onDestroy() {

*// TODO Auto-generated method stub*

**super**.onDestroy();

recycleImageView(imgv\_load\_ad);

}

**public** **static** **void** recycleImageView(View view){

**if**(view==**null**) **return**;

**if**(view **instanceof** ImageView){

Drawable drawable=((ImageView) view).getDrawable();

**if**(drawable **instanceof** BitmapDrawable){

Bitmap bmp = ((BitmapDrawable)drawable).getBitmap();

**if** (bmp != **null** && !bmp.isRecycled()){

((ImageView) view).setImageBitmap(**null**);

bmp.recycle();

bmp=**null**;

}

}

}

}

​

**IO流未关闭**

* 分析：通过日志可知FileOutputStream()未关闭
* 问题代码：

**public** **static** **void** copyFile(File source, File dest) {

FileChannel inChannel = **null**;

FileChannel outChannel = **null**;

Log.i(TAG, "source path: " + source.getAbsolutePath());

Log.i(TAG, "dest path: " + dest.getAbsolutePath());

**try** {

inChannel = **new** FileInputStream(source).getChannel();

outChannel = **new** FileOutputStream(dest).getChannel();

inChannel.transferTo(0, inChannel.size(), outChannel);

} **catch** (IOException e) {

e.printStackTrace();

}

}

* 解决方案：
  + 及时关闭IO流，避免泄露

**public** **static** **void** copyFile(File source, File dest) {

FileChannel inChannel = **null**;

FileChannel outChannel = **null**;

Log.i(TAG, "source path: " + source.getAbsolutePath());

Log.i(TAG, "dest path: " + dest.getAbsolutePath());

**try** {

inChannel = **new** FileInputStream(source).getChannel();

outChannel = **new** FileOutputStream(dest).getChannel();

inChannel.transferTo(0, inChannel.size(), outChannel);

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**if** (inChannel != **null**) {

**try** {

inChannel.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**if** (outChannel != **null**) {

**try** {

outChannel.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

E/StrictMode: A resource was acquired at attached stack trace but never released.

See java.io.Closeable for information on avoiding resource leaks.

java.lang.Throwable: Explicit termination method 'close' not called

at dalvik.system.CloseGuard.open(CloseGuard.java:180)

at java.io.FileOutputStream.<init>(FileOutputStream.java:89)

at java.io.FileOutputStream.<init>(FileOutputStream.java:72)

at com.heyniu.lock.utils.FileUtil.copyFile(FileUtil.java:44)

at com.heyniu.lock.db.BackupData.backupData(BackupData.java:89)

at com.heyniu.lock.ui.HomeActivity$11.onClick(HomeActivity.java:675)

at android.support.v7.app.AlertController$ButtonHandler.handleMessage(AlertController.java:157)

at android.os.Handler.dispatchMessage(Handler.java:102)

at android.os.Looper.loop(Looper.java:148)

at android.app.ActivityThread.main(ActivityThread.java:5417)

at java.lang.reflect.Method.invoke(Native Method)

at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:726)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:616)

**单例模式泄露**

* 分析：通过截图我们发现SplashActivity被ActivityUtil的实例activityStack持有
* 引用代码：

ActivityUtil.getAppManager().add(**this**);

* 持有代码：

**public** **void** add(Activity activity) {

**if** (activityStack == **null**) {

**synchronized** (ActivityUtil.class){

**if** (activityStack == **null**) {

activityStack = **new** Stack<>();

}

}

}

activityStack.add(activity);

}

* 解决方案：
  + 在SplashActivity的onDestroy()生命周期移除引用

@Override

**protected** **void** onDestroy() {

**super**.onDestroy();

ActivityUtil.getAppManager().remove(**this**);

}

**静态变量持有Context实例泄露**

* 分析：长生命周期持有短什么周期引用导致泄露，详见上文**四大组件Context和Application的context使用**
* 示例引用代码：

**private** **static** HttpRequest req;

**public** **static** **void** HttpUtilPost(Context context, **int** TaskId, String url, String requestBody,ArrayList<HttpHeader> Headers, RequestListener listener) {

*// TODO Auto-generated constructor stub*

req = **new** HttpRequest(context, url, TaskId, requestBody, Headers, listener);

req.post();

}

* 解决方案：
  + 改为弱引用
  + **pass：**弱引用随时可能为空，使用前先判空
  + 示例代码：
* **public** **static** **void** cancel(**int** TaskId) {
* **if**(req != **null** && req.get() != **null**){
* req.get().AsyncCancel(TaskId);
* }

}

**private** **static** WeakReference<HttpRequest> req;

**public** **static** **void** HttpUtilPost(Context context, **int** TaskId, String url, String requestBody,ArrayList<HttpHeader> Headers, RequestListener listener) {

*// TODO Auto-generated constructor stub*

req = **new** WeakReference<HttpRequest>(**new** HttpRequest(context, url, TaskId, requestBody, Headers, listener));

req.get().post();

}

* + 改为长生命周期

**private** **static** HttpRequest req;

**public** **static** **void** HttpUtilPost(Context context, **int** TaskId, String url, String requestBody,ArrayList<HttpHeader> Headers, RequestListener listener) {

*// TODO Auto-generated constructor stub*

req = **new** HttpRequest(context.getApplicationContext(), url, TaskId, requestBody, Headers, listener);

req.post();

}

**Context泄露**

**Callback泄露**

**服务未解绑注册泄露**

* 分析：一般发生在注册了某服务，不用时未解绑服务导致泄露
* 引用代码：

**private** **void** initSensor() {

*// 获取传感器管理器*

sm = (SensorManager) container.activity.getSystemService(Context.SENSOR\_SERVICE);

*// 获取距离传感器*

acceleromererSensor = sm.getDefaultSensor(Sensor.TYPE\_PROXIMITY);

*// 设置传感器监听器*

acceleromererListener = **new** SensorEventListener() {

......

};

sm.registerListener(acceleromererListener, acceleromererSensor, SensorManager.SENSOR\_DELAY\_NORMAL);

}

* 解决方案：
  + 在Activity的onDestroy()方法解绑服务

@Override

**protected** **void** onDestroy() {

**super**.onDestroy();

sm.unregisterListener(acceleromererListener,acceleromererSensor);

}

**Handler泄露**

* 分析：由于Activity已经关闭，Handler任务还未执行完成，其引用了Activity的实例导致内存泄露
* 引用代码：

handler.sendEmptyMessage(0);

* 解决方案：
  + 在Activity的onDestroy()方法回收Handler

@Override

**protected** **void** onDestroy() {

**super**.onDestroy();

handler.removeCallbacksAndMessages(**null**);

}

* 图片后续遇到再补上

**异步线程泄露**

* 分析：一般发生在线程执行耗时操作时，如下载，此时Activity关闭后，由于其被异步线程引用，导致无法被正常回收，从而内存泄露
* 引用代码：

**new** Thread() {

**public** **void** run() {

imageArray = loadImageFromUrl(imageUrl);

}.start();

* 解决方案：
  + 把线程作为对象提取出来
  + 在Activity的onDestroy()方法阻塞线程

thread = **new** Thread() {

**public** **void** run() {

imageArray = loadImageFromUrl(imageUrl);

};

thread.start();

@Override

**protected** **void** onDestroy() {

**super**.onDestroy();

**if**(thread != **null**){

thread.interrupt();

thread = **null**;

}

}

**后面**

* 欢迎补充实际中遇到的泄露类型
* 文章如有错误，欢迎指正
* 如有更好的内存泄露分享方法，欢迎一起讨论

**原文发表于:Testerhome；**

**作者：ycwdaaaa ;**

**原文链接：https://testerhome.com/topics/5822**