内存泄露研究（LeakCanary）

日期	作者	内容	版本/备注
2015-11-17	SWD-3 何建华 jianhua.he@tcl.com	create	V1.0

目录

1 什么是内存泄露

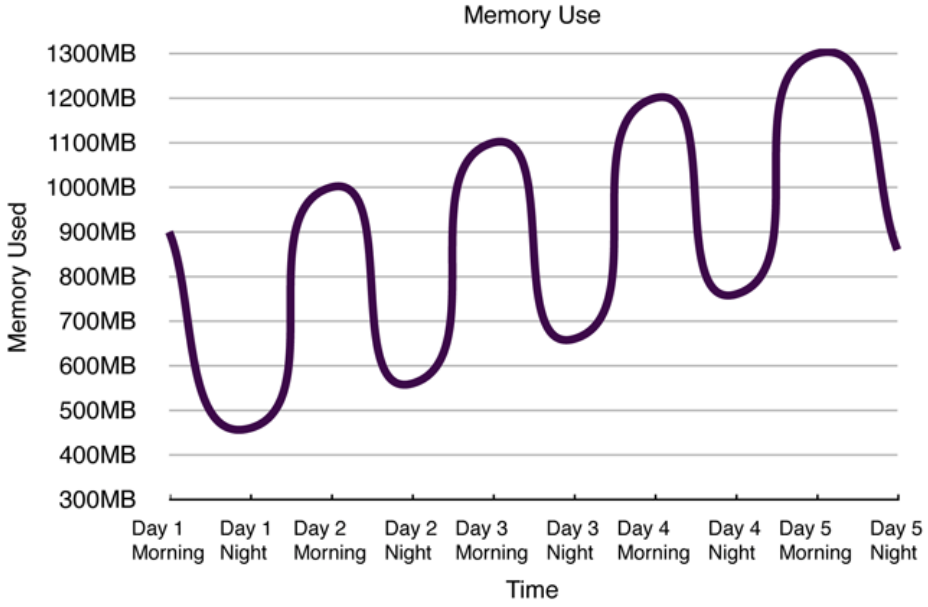
1.1 概念：

内存泄漏也称作“存储渗漏”，用动态存储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。即该内存空间使用完毕之后未回收，就是所谓内存泄漏。

内存泄漏形象的比喻是“操作系统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。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指向它，那么这块内存就泄漏了。

一些对象有着有限的生命周期。当这些对象所要做的事情完成了，我们希望他们会被回收掉。但是如果有一系列对这个对象的引用，那么在我们期待这个对象生命周期结束的时候被收回的时候，它是不会被回收的。它还会占用内存，这就造成了内存泄露。持续累加，内存很快被耗尽。

比如，当 Activity.onDestroy 被调用之后，activity 以及它涉及到的 view 和相关的 bitmap 都应该被回收。但是，如果有一个后台线程持有这个 activity 的引用，那么 activity 对应的内存就不能被回收。这最终将会导致内存耗尽，然后因为 OOM 而 crash。



根据经验，当使用大量的函数对相同的内存块进行处理时，很可能会出现内存泄漏。同时内存泄露是导致内存溢出（OOM）的原因之一，不能简单的去谈他们的不同。

1.2 分类：

以发生的方式来分类，内存泄漏可以分为 4 类：

常发性

发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

偶发性

发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

一次性

发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块且仅一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。

隐式

程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个长期在线的应用，或许需要运行几天，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

1.3 Android 内存泄露例子：

Android 是 java 语言，具有自动垃圾回收机制（GC），不需要开发者过多关注回收对象。所以在很多情况下容易出现内存泄露，造成程序不断耗尽内存（进程最大可以使用的那部分），最终导致 OutOfMemery。

同时对于 OEM 来说，在做定制开发的时候，如果出现内存泄露这样的问题，对于出货手机来说是有很大影响的。

资源对象没关闭造成的内存泄露

资源性对象比如(Cursor，File 文件等)往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 Java 虚拟机内，还存在于 Java 虚拟机外。如果我们仅仅是把它的引用设置为 null,而不关闭它们，往往会造成内存泄露。因为有些资源性对象，比如 SQLiteCursor(在析构函数 finalize(),如果我们没有关闭它，它自己会调 close()关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该立即调用它的 close()函数，将其关闭掉，然后再置为 null.在我们的程序退出时一定要确保我们的资源性对象已经关闭。

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在长时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

未关闭 InputStream/OutputStream

在使用文件或者访问网络资源时，使用了 InputStream/OutputStream 也会导致内存泄露

查询数据库而没有关闭 Cursor

在 Android 中，Cursor 是很常用的一个对象，但在写代码的时候，经常会有人忘记调用 close，或者因为代码逻辑问题状况导致 close 未被调用。

通常，在 Activity 中，我们可以调用 startManagingCursor 或直接使用 managedQuery 让 Activity 自动管理 Cursor 对象。

但需要注意的是，当 Activity 结束后，Cursor 将不再可用！

若操作 Cursor 的代码和 UI 不同步（如后台线程），那没需要先判断 Activity 是否已经结束，或者在调用 OnDestroy 前，先等待后台线程结束。

除此之外，以下也是比较常见的 Cursor 不会被关闭的情况：

```
try {
    Cursor c = queryCursor();
    int a = c.getInt(1);
    ......
    c.close();
    } catch (Exception e) {
    }
```

虽然表面看起来，Cursor.close()已经被调用，但若出现异常，将会跳过 close()，从而导致内存泄露。

所以，我们的代码应该以如下的方式编写：

```
Cursor c = queryCursor();
try {
    int a = c.getInt(1);
    ......
    } catch (Exception e) {
    } finally {
    c.close(); //在 finally 中调用 close(), 保证其一定会被调用
    }
```

注册某个对象后未反注册

注册广播接收器、注册观察者等等，比如：

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。

但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被 GC 回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory,使得 system_process 进程挂掉。

虽然有些系统程序，它本身好像是可以自动取消注册的(当然不及时)，但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

Bitmap 使用

Bitmap 使用后未调用 recycle()

根据 SDK 的描述，调用 recycle 并不是必须的。但在实际使用时，Bitmap 占用的内存是很大的，所以当我们不再使用时，尽量调用 recycle()以释放资源。

Bitmap 使用缩略图，设置一定的采样率。

有时候，我们要显示的区域很小，没有必要将整个图片都加载出来，而只需要记载一个缩小过的图片，这时候可以设置一定的采样率，那么就可以大大减小占用的内存。如下面的代码：

```
private ImageView preview;
BitmapFactory.Options options = newBitmapFactory.Options();
options.inSampleSize = 2;//图片宽高都为原来的二分之一，即图片为原来的四分之一
Bitmap bitmap =BitmapFactory.decodeStream(cr.openInputStream(uri), null, options);
preview.setImageBitmap(bitmap);
```

巧妙的运用软引用（SoftRefrence）

有些时候，我们使用 Bitmap 后没有保留对它的引用，因此就无法调用 Recycle 函数。这时候巧妙的运用软引用，可以使 Bitmap 在内存快不足时得到有效的释放。如下：

```
SoftReference<Bitmap> bitmap_ref = new SoftReference<Bitmap>(BitmapFactory.decodeStream(inputstream));
……
……
if (bitmap_ref .get() != null)
bitmap_ref.get().recycle();
```

构造 Adapter 时，没有使用缓存的 convertView

以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提共了方法：

public View getView(intposition, View convertView, ViewGroup parent)

来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象(初始化时缓存中没有 view 对象则 convertView 是 null)。

由此可以看出，如果我们不去使用 convertView，而是每次都在 getView()中重新实例化一个 View 对象的话，即浪费时间，也造成内存垃圾，给垃圾回收增加压力，如果垃圾回收来不及的话，虚拟机将不得不给该应用进程分配更多的内存，造成不必要的内存开支。

```
public View getView(intposition, View convertView, ViewGroup parent) {
View view = null;
if (convertView != null){
view = convertView;
populate(view, getItem(position));
} else {
```

```
                view = new Xxx(...);
                }
                return view;
        }
```

Context 泄露

```
        private static Drawable sBackground;
        @Override
        protected void onCreate(Bundle state) {
            super.onCreate(state);

            TextView label = new TextView(this);
            label.setText("Leaks are bad");

            if (sBackground == null) {
                sBackground = getDrawable(R.drawable.large_bitmap);
            }
            label.setBackgroundDrawable(sBackground);

            setContentView(label);
}
```

在这段代码中，我们使用了一个 static 的 Drawable 对象。

这通常发生在我们需要经常调用一个 Drawable，而其加载又比较耗时，不希望每次加载 Activity 都去创建这个 Drawable 的情况。

此时，使用 static 无疑是最快的代码编写方式，但是其也非常的糟糕。

当一个 Drawable 被附加到 View 时，这个 View 会被设置为这个 Drawable 的 callback (通过调用 Drawable.setCallback()实现)。

就意味着，这个 Drawable 拥有一个 TextView 的引用，而 TextView 又拥有一个 Activity 的引用。

这就会导致 Activity 在销毁后，内存不会被释放。

使用 handler 时的内存问题

我们知道，Handler 通过发送 Message 与主线程交互，Message 发出之后是存储在 MessageQueue 中的，有些 Message 也不是马上就被处理的。在 Message 中存在一个 target，是 Handler 的一个引用，如果 Message 在 Queue 中存在的时间越长，就会导致 Handler 无法被回收。如果 Handler 是非静态的，则会导致 Activity 或者 Service 不会被回收。 所以正确处理 Handler 等之类的内部类，应该将自己的 Handler 定义为静态内部类。

HandlerThread 的使用也需要注意：

当我们在 activity 里面创建了一个 HandlerThread，代码如下：

```
public classMainActivity extends Activity
{
    @Override
    public void onCreate(BundlesavedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Thread mThread = newHandlerThread("demo",
Process.THREAD_PRIORITY_BACKGROUND);
        mThread.start();
MyHandler mHandler = new MyHandler( mThread.getLooper( ) );
……
……
……
}
    @Override
    public void onDestroy()
    {
    super.onDestroy();
    }
}
```

这个代码存在泄漏问题，因为 HandlerThread 的 run 方法是一个死循环，它不会自己结束，线程的生命周期超过了 activity 生命周期，当横竖屏切换，HandlerThread 线程的数量会随着 activity 重建次数的增加而增加。

应该在 onDestroy 时将线程停止掉：mThread.getLooper().quit();

另外，对于不是 HandlerThread 的线程，也应该确保 activity 消耗后，线程已经终止，可以这样做：在 onDestroy 时调用 mThread.join();

非静态内部类的静态实例容易造成内存泄漏

```
public class MainActivity extends Activity
{
        static Demo sInstance = null;

    @Override
    public void onCreate(BundlesavedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
            if (sInstance == null)
            {
                sInstance= new Demo();
            }
        }
        class Demo
        {
        voiddoSomething()
        {
                System.out.print("dosth.");
        }
        }
}
```

上面的代码中的 sInstance 实例类型为静态实例，在第一个 MainActivity act1 实例创建时，sInstance 会获得并一直持有 act1 的引用。当 MainAcitivity 销毁后重建，因为 sInstance 持有 act1 的引用，所以 act1 是无法被 GC 回收的，进程中会存在 2 个 MainActivity 实例（act1 和重建后的 MainActivity 实例），这个 act1 对象就是一个无用的但一直占用内存的对象，即无法回收的垃圾对象。所以，对于 lauchMode 不是 singleInstance 的 Activity， 应该避免在 activity 里面实例化其非静态内部类的静态实例。

集合中对象没清理造成的内存泄露

我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，如果没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

比如某公司的 ROM 的锁屏曾经就存在内存泄漏问题：

这个泄漏是因为 LockScreen 每次显示时会注册几个 callback，它们保存在 KeyguardUpdateMonitor 的 ArrayList<InfoCallback>、ArrayList<SimStateCallback>等 ArrayList 实例中。但是在 LockScreen 解锁后，这些 callback 没有被 remove 掉，导致 ArrayList 不断增大， callback 对象不断增多。这些 callback 对象的 size 并不大，heap 增长比较缓慢，需要长时间地使用手机才能出现 OOM，由于锁屏是驻留在 system_server 进程里，所以导致结果是手机重启。

经常调用的方法中创建对象

不要在经常调用的方法中创建对象，尤其是忌讳在循环中创建对象。可以适当的使用 hashtable ， vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次 new 之后又丢弃。

2 LeakCanary 检查内存泄露

2.1 简介：

LeakCanary 是一个检测内存泄露的开源类库。你可以在 debug 包种轻松地使用几行代码检测内存泄露。

地址：https://github.com/square/leakcanary

优点：简单，在出现问题的时候自动抓取信息。

2.2 在 Eclipse 上使用方法及其实例：

1， 准备好 Android5.0 SDK(level 22)及其以上的 SDK，因为 LeakCanery 中的代码需要用到 SDK 22 上的东西，否则会编译不过。

2， 开发你的程序。

如这里，专门做了一个内存泄露的例子：

```
public class MainActivity extends Activity {

   private static String TAG = "LeakCanaryTestItem";
   private static LeakDemo mLeakDemo = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        if (mLeakDemo == null) {
         mLeakDemo = new LeakDemo();
         Log.d(TAG, "leakdemo class create");
      }
        mLeakDemo.doSomething();
    }

    class LeakDemo{
      private void doSomething() {
         Log.d(TAG, "leakdemo class doSomething method
```

```
run!");
        }
    }
}
```

3， 下载 leakcanarylib 源码包：

注：本文档为 doc 格式时候，直接点击下面的图标可以下载（下同）。

也可以在这里下载，直接可用：

https://github.com/kevinjh443/LeakCanaryUseDemo

LeakcanarySample-Eclipse-master.zip

4， 设置 leakcanary 为 library

在 lib 工程上右击-》properties-》Android-》 右边勾选 is library-》 apply-》OK 。可以在 project.properties 文件中查看有 `android.library=true` 值。

5， 加入 leakcanary lib 到开发工程中

在开发工程上右击-》properties-》Android-》点击 library 中的 add 按钮-》其中会有目前 eclipse 中存在为 lib 的工程，选择 Leakcanary –》apply –》OK。 可以在 project.properties 文件中存在

“ `android.library.reference.2=../code_temp/LeakcanarySample-Eclipse-master/leakcanarylib` ” 代表已经可以使用了。

6， 创建一个类：

```
public class TestApplication extends Application {

   public static RefWatcher getRefWatcher(Context context)
{
      TestApplication application = (TestApplication)
context
            .getApplicationContext();
      return application.refWatcher;
   }

   private RefWatcher refWatcher;
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        refWatcher = LeakCanary.install(this);
    }
}
```

7， 在 AndroidMenifast.xml 中的<application>的标签中加入如下标签，让 leakcanary 能正常跳出结果页面。

```
<service

android:name="com.squareup.leakcanary.internal.HeapAnalyzerService
            android:enabled="false"
            android:process=":leakcanary" />
        <service

android:name="com.squareup.leakcanary.DisplayLeakService"
            android:enabled="false" />

        <activity

android:name="com.squareup.leakcanary.internal.DisplayLeakActivity
            android:enabled="false"
            android:icon="@drawable/__leak_canary_icon"

android:label="@string/__leak_canary_display_activity_label"
            android:taskAffinity="com.squareup.leakcanary"
            android:theme="@style/__LeakCanary.Base" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```

```
<application

android:name="com.hogee.leakcanarytest.TestApplication"
```

```
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
```

```
<!-- To store the heap dumps and leak analysis results. -->
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"
/>
```

8， 在 values 中加入以下几个文件


__leak_canary_strings.xml


__leak_canary_themes.xml


__leak_canary_int.xml

9， 在 drawable 下加入以下两个图片，这两个图片为出现问题时候显示的图标，即 leakcanary 的图标。

__leak_canary_notification.png __leak_canary_icon.png

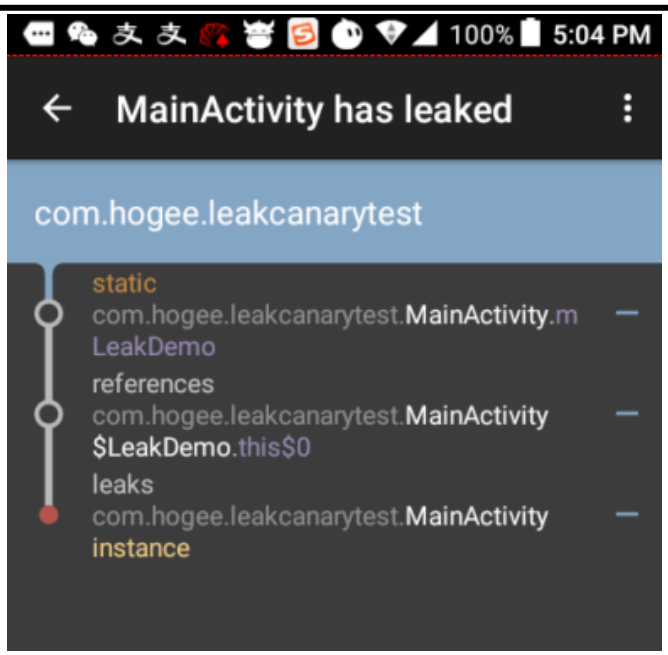
10， 在需要测试的 Activity 中加入如下代码：

```
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        RefWatcher refWatcher =
TestApplication.getRefWatcher(this);
        refWatcher.watch(this);
```

11， 以上过程做完就可以进行测试你的 acticity 以及其他 Context 有没有潜在的内存泄露问题了。

12， 如果有内存泄露会在 luncher 中 和 notification 中显示 leakcanary 的图标，点击查看，即可查看具体在哪个地方有内存泄露了，十分方便。



与此同时，在 adb logcat 中可以查看到如下 log， 即 **leak trace**：

```
D/LeakCanary(31295): In com.hogee.leakcanarytest:1.0:1.
D/LeakCanary(31295): * com.hogee.leakcanarytest.MainActivity has leaked:
D/LeakCanary(31295): * GC ROOT static com.hogee.leakcanarytest.MainActivity.mLeakDemo
D/LeakCanary(31295): * references com.hogee.leakcanarytest.MainActivity$LeakDemo.this$0
D/LeakCanary(31295): * leaks com.hogee.leakcanarytest.MainActivity instance
D/LeakCanary(31295):
D/LeakCanary(31295): * Reference Key: 0678a718-1e6e-4f88-a424-c2b463a2a694
D/LeakCanary(31295): * Device: TCL TCL 5065W 5065W
D/LeakCanary(31295): * Android Version: 5.1.1 API: 22 LeakCanary:
D/LeakCanary(31295): * Durations: watch=5143ms, gc=141ms, heap dump=1790ms, analysis=17925ms
D/kevinjh (31295): notifycation 1 , SDK_INT=22 HONEYCOMB=11
D/kevinjh (31295): notifycation 3
```

13， **分析结论：**根据上图发现，在 MainActivity 的 static 的 mLeakDemo 变量发生了内存泄露了。泄露的细分类型为“**非静态内部类的静态实例容易造成内存泄漏**”。

原因：在第一个 MainActivity act1 实例创建时，mLeakDemo 会获得并一直持有 act1 的引用。当 MainAcitivity 销毁后重建，因为 mLeakDemo 持有 act1 的引用，所以 act1 是无法被 GC 回收的，进程中会存在 2 个 MainActivity 实例（act1 和重建后的 MainActivity 实例），这个 act1 对象就是一个无用的但一直占用内存的对象，即无法回收的垃圾对象。所以，对于 lauchMode 不是 singleInstance 的 Activity， 应该避免在 activity 里面实例化其非静态内部类的静态实例。

注意：

- leakcanary 在检查内存泄露的时候，通常发生内存泄露后约 10 sec 才能在 notification 中跳出来，不能立即显示。
- 可以在进行 monkey test 的时候加入这个检查机制，检查应用的泄露问题，只要发生了泄露，它会一直记录在案，直到人为手动删除这些记录（在 leak 结果显示界面可以删除）。
- 如果正式发布版本再去掉以上添加在项目中的资源和代码即可。也可以设置 `RefWatcher.DISABLED` 来关闭 `LeakCanary` 模块。

2.3 在项目中使用方法（可检查平台代码）：

2.3.1 系统应用使用 LeakCanary

把 LeakCanary 作为 LeakCanaryTest 的静态 java 库，可以参考 android 5.0 上的 keyguard 和 systemui 的关系，keyguard 是 systemui 的库。

LeakCanary 的 Android.mk：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)



LOCAL_MODULE_TAGS := optional



LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_MODULE := LeakCanary

LOCAL_SDK_VERSION := current

#LOCAL_MODULE_PATH := $(TARGET_OUT)/bin
LOCAL_RESOURCE_DIR := \
$(LOCAL_PATH)/res

#LOCAL_SHARED_LIBRARIES := LeakCanary



#include $(BUILD_PACKAGE)
LOCAL_MODULE_CLASS := JAVA_LIBRARIES
```

```
include $(BUILD_STATIC_JAVA_LIBRARY)

# Use the following include to make our test apk.
#include $(call all-makefiles-under,$(LOCAL_PATH))
```

LeakCanaryTest 的 Android.mk：

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_PACKAGE_NAME := LeakCanaryTest

#LOCAL_MODULE := LeakCanaryTest

LOCAL_STATIC_JAVA_LIBRARIES := LeakCanary

LOCAL_SDK_VERSION := current

#LOCAL_MODULE_PATH := $(TARGET_OUT)/bin
LOCAL_RESOURCE_DIR := \
  packages/apps/LeakCanary/res \
  $(LOCAL_PATH)/res

LOCAL_AAPT_FLAGS := --auto-add-overlay --extra-packages com.squareup.leakcanary

#include packages/apps/LeakCanary/Android.mk

include $(BUILD_PACKAGE)
##################################################
#include $(CLEAR_VARS)
```

```
#include $(BUILD_MULTI_PREBUILT)

# Use the following include to make our test apk.
#include $(call all-makefiles-under,$(LOCAL_PATH))
```

编译 LeakCanary 库：

命令： mmm packages/apps/LeakCanary

```
Copying: out/target/common/obj/JAVA_LIBRARIES/LeakCanary_intermediates/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/LeakCanary_intermediates/emma_out/lib/classes-jarjar.jar
Copying: out/target/common/obj/JAVA_LIBRARIES/LeakCanary_intermediates/classes.jar
target Static Jar: LeakCanary (out/target/common/obj/JAVA_LIBRARIES/LeakCanary_intermediates/javalib.jar)
make: Leaving directory `/data/rescode/idol4-fb1116-m8976'

#### make completed successfully (4 seconds) ####
```

编译需要测试的模块。

mmm packages/apps/LeakCanaryTest/ 这里需要注意，在此应用中也对应应该在 AndroidMenifast.xml 中加入 LeakCanary 的 service、权限等信息，和 res 资源文件中的文件。如上一节所述一样。

命令：mmm packages/apps/LeakCanaryTest/

```
Warning: AndroidManifest.xml already defines targetSdkVersion (in http://schemas.android.com/apk/res/android);
Install: out/target/product/idol455/system/app/LeakCanaryTest/LeakCanaryTest.apk
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:2052] out/host/linux-x86/bin/dex2oatd --runtime-arg -Xms64m --runt
ct/idol455/dex_bootjars/system/framework/boot.art --dex-file=out/target/product/idol455/obj/APPS/LeakCanaryTest_
ex-location=/system/app/LeakCanaryTest/LeakCanaryTest.apk --oat-file=out/target/product/idol455/obj/APPS/LeakCan
-android-root=out/target/product/idol455/system --instruction-set=arm64 --instruction-set-variant=generic --inst
nformation --runtime-arg -Xnorelocate --no-generate-debug-info --abort-on-hard-verifier-error --compile-pic
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:692] Dumping image sections
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:696] SectionObjects start=0x70000000 size=9969264 ran
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:696] SectionArtFields start=0x70981e70 size=531744 ra
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:696] SectionArtMethods start=0x70a03b90 size=3076584
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:696] SectionInternedStrings start=0x70cf2d78 size=244
dex2oatd I 18522 18522 art/runtime/gc/space/image_space.cc:696] SectionImageBitmap start=0x70d2f000 size=159744
dex2oatd W 18522 18522 art/dex2oat/dex2oat.cc:1694] Failed to open dex file '.': Failed to find magic in '.'
dex2oatd W 18522 18522 art/dex2oat/dex2oat.cc:1694] Failed to open dex file '/data/rescode/idol4-fb1116-m8976/ve
 Failed to find magic in '/data/rescode/idol4-fb1116-m8976/vendor/tct/buildtools/java-7-openjdk-amd64/lib'
dex2oatd W 18522 18522 art/dex2oat/dex2oat.cc:1694] Failed to open dex file '/data/rescode/idol4-fb1116-m8976/ve
ib': Failed to find magic in '/data/rescode/idol4-fb1116-m8976/vendor/tct/buildtools/java-7-openjdk-amd64/jre/li
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] dex2oat took 202.475ms (threads: 8) arena alloc=4MB java all
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] Code dedupe: 0 collisions, 0 max bucket size, 327923 ns hash
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] Mapping table dedupe: 0 collisions, 0 max bucket size, 6683
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] Vmap table dedupe: 0 collisions, 0 max bucket size, 131456 m
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] GC map dedupe: 0 collisions, 0 max bucket size, 9953 ns hash
dex2oatd I 18522 18522 art/dex2oat/dex2oat.cc:1855] CFI info dedupe: 0 collisions, 0 max bucket size, 0 ns hash
dex2oatd W 18522 18522 art/runtime/runtime.cc:217] Current thread not detached in Runtime shutdown
Install: out/target/product/idol455/system/app/LeakCanaryTest/oat/arm64/LeakCanaryTest.odex
make: Leaving directory `/data/rescode/idol4-fb1116-m8976'

#### make completed successfully (7 seconds) ####
```

编译完成后，然后安装此 LeakCanaryTest .apk 即可进行测试。

2.4 在 Android Studio 上使用：

参考：http://www.liaohuqiu.net/cn/posts/leak-canary-read-me/

2.5 LeakCanary 原理简述

本质：使用 RefWatcher 监控那些本该被回收的对象。

LeakCanary.install() 会返回一个预定义的 RefWatcher，同时也会启用一个 ActivityRefWatcher，用于自动监控调用 Activity.onDestroy() 之后泄露的 activity。

工作机制：

- RefWatcher.watch() 创建一个 KeyedWeakReference 到要被监控的对象。
- 然后在后台线程检查引用是否被清除，如果没有，调用 GC。
- 如果引用还是未被清除，把 heap 内存 dump 到 APP 对应的文件系统中的一个 .hprof 文件中。
- 在另外一个进程中的 HeapAnalyzerService 有一个 HeapAnalyzer 使用 HAHA 解析这个文件。
- 得益于唯一的 reference key, HeapAnalyzer 找到 KeyedWeakReference，定位内存泄露
- HeapAnalyzer 计算 到 GC roots 的最短强引用路径，并确定是否是泄露。如果是的话，建立导致泄露的引用链。
- 引用链传递到 APP 进程中的 DisplayLeakService， 并以通知的形式展示出来。

正式发布：

正式发布的时候使用 RefWatcher.DISABLED 关闭 LeakCanary 功能即可：

```
public class ExampleApplication extends Application {
  public static RefWatcher getRefWatcher(Context context) {
      ExampleApplication application = (ExampleApplication)
context.getApplicationContext();
      return application.refWatcher;
    }

     private RefWatcher refWatcher;

     @Override public void onCreate() {
       super.onCreate();
       refWatcher = installLeakCanary();
   }
```

```
    protected RefWatcher installLeakCanary() {
        return RefWatcher.DISABLED;
    }
}
```