在上一篇[Android内存泄漏的八种可能（上）](http://www.jianshu.com/p/ac00e370f83d)中，我们讨论了八种容易发生内存泄漏的代码。其中，尤其严重的是泄漏Activity对象，因为它占用了大量系统内存。不管内存泄漏的代码表现形式如何，其核心问题在于：

在Activity生命周期之外仍持有其引用。

幸运的是，一旦泄漏发生且被定位到了，修复方法是相当简单的。

**Static Actitivities**

这种泄漏

private static MainActivity activity;

void setStaticActivity() {

activity = this;

}

构造静态变量持有Activity对象很容易造成内存泄漏，因为静态变量是全局存在的，所以当MainActivity生命周期结束时，引用仍被持有。这种写法开发者是有理由来使用的，所以我们需要正确的释放引用让垃圾回收机制在它被销毁的同时将其回收。

Android提供了特殊的Set集合<https://developer.android.com/reference/java/lang/ref/package-summary.html#classes>  
允许开发者控制引用的“强度”。Activity对象泄漏是由于需要被销毁时，仍然被强引用着，只要强引用存在就无法被回收。

可以用弱引用代替强引用。  
<https://developer.android.com/reference/java/lang/ref/WeakReference.html>.

弱引用不会阻止对象的内存释放，所以即使有弱引用的存在，该对象也可以被回收。

private static WeakReference<MainActivity> activityReference;

void setStaticActivity() {

activityReference = new WeakReference<MainActivity>(this);

}

**Static Views**

静态变量持有View

private static View view;

void setStaticView() {

view = findViewById(R.id.sv\_button);

}

由于View持有其宿主Activity的引用，导致的问题与Activity一样严重。弱引用是个有效的解决方法，然而还有另一种方法是在生命周期结束时清除引用，Activity#onDestory()方法就很适合把引用置空。

private static View view;

@Override

public void onDestroy() {

super.onDestroy();

if (view != null) {

unsetStaticView();

}

}

void unsetStaticView() {

view = null;

}

**Inner Class**

这种泄漏

private static Object inner;

void createInnerClass() {

class InnerClass {

}

inner = new InnerClass();

}

与上述两种情况相似，开发者必须注意**少**用非静态内部类，因为[非静态内部类](https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html)持有外部类的隐式引用，容易导致意料之外的泄漏。然而内部类可以访问外部类的私有变量，只要我们注意引用的生命周期，就可以避免意外的发生。

避免静态变量

这样持有内部类的成员变量是可以的。

private Object inner;

void createInnerClass() {

class InnerClass {

}

inner = new InnerClass();

}

**Anonymous Classes**

前面我们看到的都是持有全局生命周期的静态成员变量引起的，直接或间接通过链式引用Activity导致的泄漏。这次我们用AsyncTask

void startAsyncTask() {

new AsyncTask<Void, Void, Void>() {

@Override protected Void doInBackground(Void... params) {

while(true);

}

}.execute();

}

Handler

void createHandler() {

new Handler() {

@Override public void handleMessage(Message message) {

super.handleMessage(message);

}

}.postDelayed(new Runnable() {

@Override public void run() {

while(true);

}

}, Long.MAX\_VALUE >> 1);

}

Thread

void scheduleTimer() {

new Timer().schedule(new TimerTask() {

@Override

public void run() {

while(true);

}

}, Long.MAX\_VALUE >> 1);

}

全部都是因为[匿名类](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)导致的。匿名类是特殊的内部类——写法更为简洁。当需要一次性特殊的子类时，Java提供的语法糖能让表达式最少化。这种很赞很偷懒的写法容易导致泄漏。正如使用内部类一样，只要不跨越生命周期，内部类是完全没问题的。但是，这些类是用于产生后台线程的，这些Java线程是全局的，而且持有创建者的引用（即匿名类的引用），而匿名类又持有外部类的引用。线程是可能长时间运行的，所以一直持有Activity的引用导致当销毁时无法回收。  
这次我们不能通过移除静态成员变量解决，因为线程是于应用生命周期相关的。为了避免泄漏，我们必须舍弃简洁偷懒的写法，把子类声明为静态内部类。

静态内部类不持有外部类的引用，打破了链式引用。

所以对于AsyncTask

private static class NimbleTask extends AsyncTask<Void, Void, Void> {

@Override protected Void doInBackground(Void... params) {

while(true);

}

}

void startAsyncTask() {

new NimbleTask().execute();

}

Handler

private static class NimbleHandler extends Handler {

@Override public void handleMessage(Message message) {

super.handleMessage(message);

}

}

private static class NimbleRunnable implements Runnable {

@Override public void run() {

while(true);

}

}

void createHandler() {

new NimbleHandler().postDelayed(new NimbleRunnable(), Long.MAX\_VALUE >> 1);

}

TimerTask

private static class NimbleTimerTask extends TimerTask {

@Override public void run() {

while(true);

}

}

void scheduleTimer() {

new Timer().schedule(new NimbleTimerTask(), Long.MAX\_VALUE >> 1);

}

但是，如果你坚持使用匿名类，只要在生命周期结束时中断线程就可以。

private Thread thread;

@Override

public void onDestroy() {

super.onDestroy();

if (thread != null) {

thread.interrupt();

}

}

void spawnThread() {

thread = new Thread() {

@Override public void run() {

while (!isInterrupted()) {

}

}

}

thread.start();

}

**Sensor Manager**

这种泄漏

void registerListener() {

SensorManager sensorManager = (SensorManager) getSystemService(SENSOR\_SERVICE);

Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE\_ALL);

sensorManager.registerListener(this, sensor, SensorManager.SENSOR\_DELAY\_FASTEST);

}

使用Android系统服务不当容易导致泄漏，为了Activity与服务交互，我们把Activity作为监听器，引用链在传递事件和回调中形成了。只要Activity维持注册监听状态，引用就会一直持有，内存就不会被释放。

在Activity结束时注销监听器

private SensorManager sensorManager;

private Sensor sensor;

@Override

public void onDestroy() {

super.onDestroy();

if (sensor != null) {

unregisterListener();

}

}

void unregisterListener() {

sensorManager.unregisterListener(this, sensor);

}

**总结**

Activity泄漏的案例我们已经都走过一遍了，其他都大同小异。建议日后遇到类似的情况时，就使用相应的解决方法。内存泄漏只要发生过一次，通过详细的检查，很容易解决并防范于未然。

是时候做最佳实践者了！