Java是垃圾回收语言的一种，其优点是开发者无需特意**管理内存分配**，降低了应用由于**局部故障(segmentation fault)**导致崩溃，同时防止未释放的内存把**堆栈(heap)**挤爆的可能，所以写出来的代码更为安全。

不幸的是，在Java中仍存在很多容易导致内存泄漏的**逻辑可能**(logical leak)。如果不小心，你的Android应用很容易浪费掉未释放的内存，最终导致内存用光的错误抛出(out-of-memory，OOM)。

一般内存泄漏(traditional memory leak)的原因是：由忘记释放分配的内存导致的。（译者注：Cursor忘记关闭等）  
逻辑内存泄漏(logical memory leak)的原因是：当应用不再需要这个对象，当仍未释放该对象的所有引用。

**如果持有对象的强引用，垃圾回收器是无法在内存中回收这个对象。**

在Android开发中，最容易引发的内存泄漏问题的是[Context](http://developer.android.com/reference/android/content/Context.html)。比如[Activity](http://developer.android.com/reference/android/app/Activity.html)的Context，就包含大量的内存引用，例如View Hierarchies和其他资源。一旦泄漏了Context，也意味泄漏它指向的所有对象。Android机器内存有限，太多的内存泄漏容易导致OOM。

检测逻辑内存泄漏需要主观判断，特别是对象的生命周期并不清晰。幸运的是，Activity有着明确的[生命周期](http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle)，很容易发现泄漏的原因。[Activity.onDestroy()](http://developer.android.com/reference/android/app/Activity.html#onDestroy())被视为Activity生命的结束，程序上来看，它应该被销毁了，或者Android系统需要回收这些内存（译者注：当内存不够时，Android会回收看不见的Activity）。  
如果这个方法执行完，在堆栈中仍存在持有该Activity的强引用，垃圾回收器就无法把它标记成已回收的内存，而我们本来目的就是要回收它！  
结果就是Activity存活在它的生命周期之外。

Activity是重量级对象，应该让Android系统来处理它。然而，逻辑内存泄漏总是在不经意间发生。（译者注：曾经试过一个Activity导致20M内存泄漏）。在Android中，导致潜在内存泄漏的陷阱不外乎两种：

* 全局进程(process-global)的static变量。这个无视应用的状态，持有Activity的强引用的怪物。
* 活在Activity生命周期之外的线程。没有清空对Activity的强引用。

检查一下你有没有遇到下列的情况。

**Static Activities**

在类中定义了静态Activity变量，把当前运行的Activity实例赋值于这个静态变量。  
如果这个静态变量在Activity生命周期结束后没有清空，就导致内存泄漏。因为static变量是贯穿这个应用的生命周期的，所以被泄漏的Activity就会一直存在于应用的进程中，不会被垃圾回收器回收。

static Activity activity;

void setStaticActivity() {

activity = this;

}

View saButton = findViewById(R.id.sa\_button);

saButton.setOnClickListener(new View.OnClickListener() {

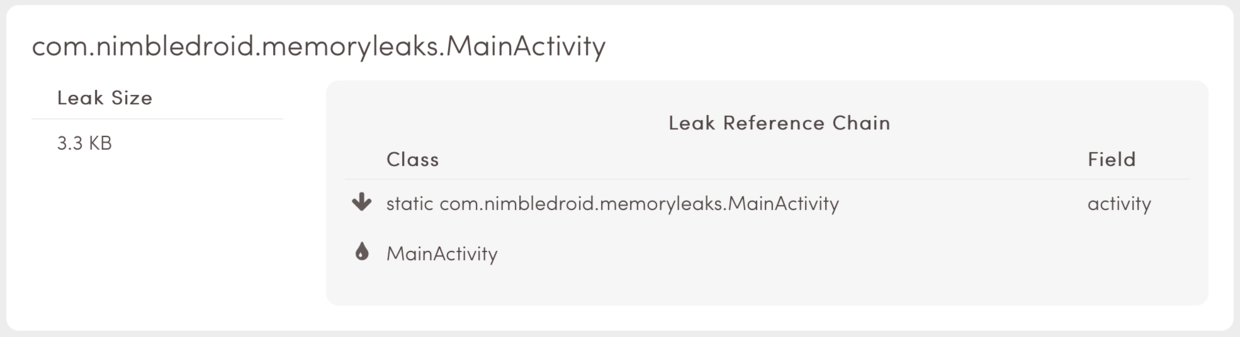
@Override public void onClick(View v) {

setStaticActivity();

nextActivity();

}

});



Memory Leak 1 - Static Activity

**Static Views**

类似的情况会发生在单例模式中，如果Activity经常被用到，那么在内存中保存一个实例是很实用的。正如之前所述，强制延长Activity的生命周期是相当危险而且不必要的，无论如何都不能这样做。

特殊情况：如果一个View初始化耗费大量资源，而且在一个Activity生命周期内保持不变，那可以把它变成static，加载到视图树上(View Hierachy)，[像这样](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L132)，当Activity被销毁时，应当释放资源。（译者注：示例代码中并没有释放内存，把这个static view置null即可，但是还是不建议用这个static view的方法）

static view;

void setStaticView() {

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

}

View svButton = findViewById(R.id.sv\_button);

svButton.setOnClickListener(new View.OnClickListener() {

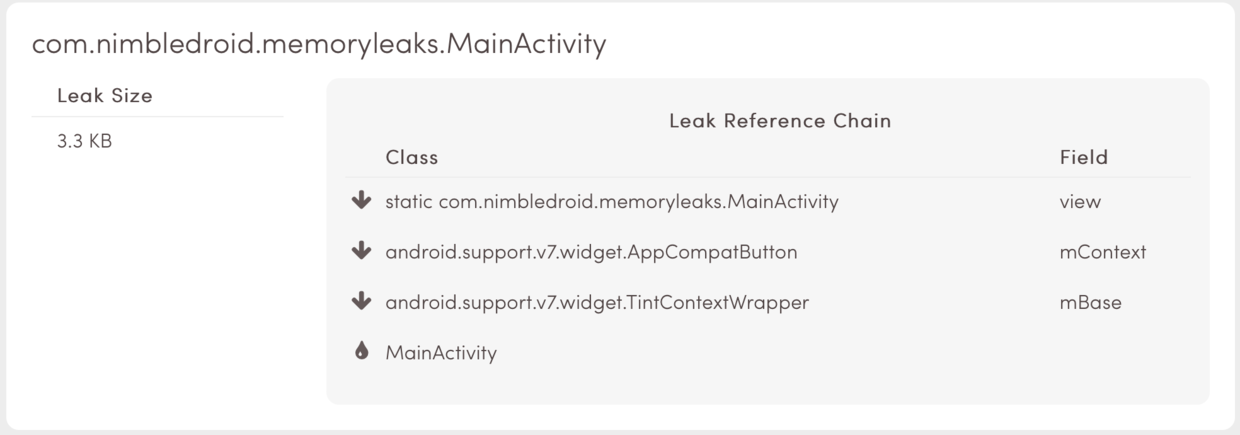
@Override public void onClick(View v) {

setStaticView();

nextActivity();

}

});



Memory Leak 2 - Static View

**Inner Classes**

继续，假设Activity中有个[内部类](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L126)，这样做可以提高可读性和封装性。将如我们创建一个内部类，而且持有一个静态变量的引用，恭喜，内存泄漏就离你不远了（译者注：销毁的时候置空，嗯）。

private static Object inner;

void createInnerClass() {

class InnerClass {

}

inner = new InnerClass();

}

View icButton = findViewById(R.id.ic\_button);

icButton.setOnClickListener(new View.OnClickListener() {

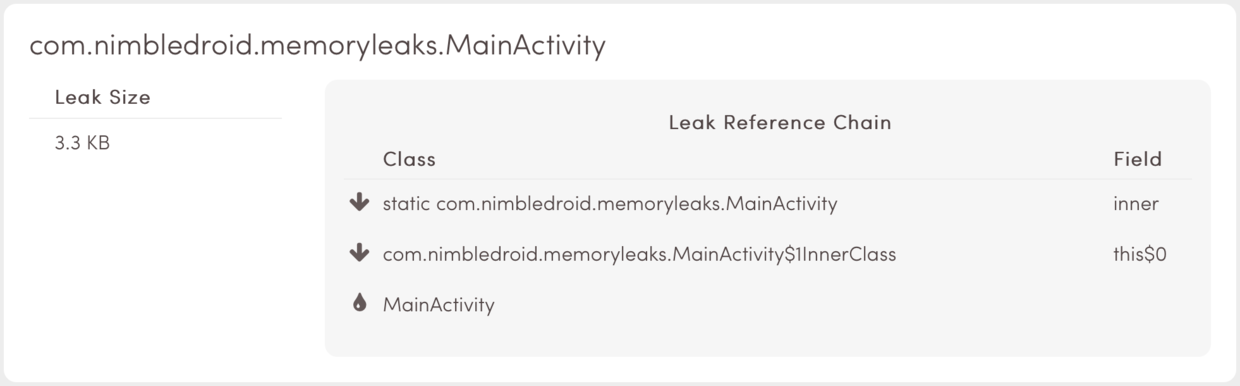
@Override public void onClick(View v) {

createInnerClass();

nextActivity();

}

});



Memory Leak 3 - Inner Class

内部类的优势之一就是可以访问外部类，不幸的是，导致内存泄漏的原因，就是内部类持有外部类实例的强引用。

**Anonymous Classes**

相似地，匿名类也维护了外部类的引用。所以内存泄漏很容易发生，[当你在Activity中定义了匿名的AsyncTsk](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L102)  
。当异步任务在后台执行耗时任务期间，Activity不幸被销毁了（译者注：用户退出，系统回收），这个被AsyncTask持有的Activity实例就不会被垃圾回收器回收，直到异步任务结束。

void startAsyncTask() {

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

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

while(true);

}

}.execute();

}

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

View aicButton = findViewById(R.id.at\_button);

aicButton.setOnClickListener(new View.OnClickListener() {

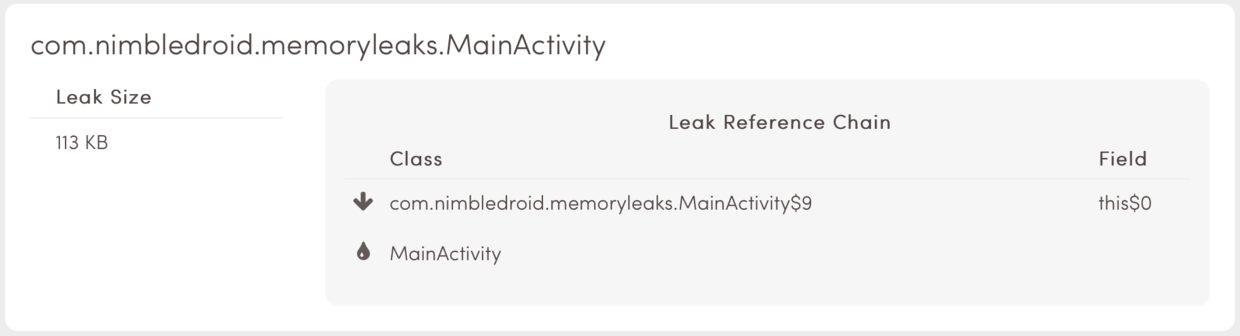
@Override public void onClick(View v) {

startAsyncTask();

nextActivity();

}

});



Memory Leak 4 - AsyncTask

**Handler**

同样道理，[定义匿名的Runnable，用匿名类Handler执行](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L114)。Runnable内部类会持有外部类的隐式引用，被传递到Handler的消息队列MessageQueue中，在Message消息没有被处理之前，Activity实例不会被销毁了，于是导致内存泄漏。

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);

}

View hButton = findViewById(R.id.h\_button);

hButton.setOnClickListener(new View.OnClickListener() {

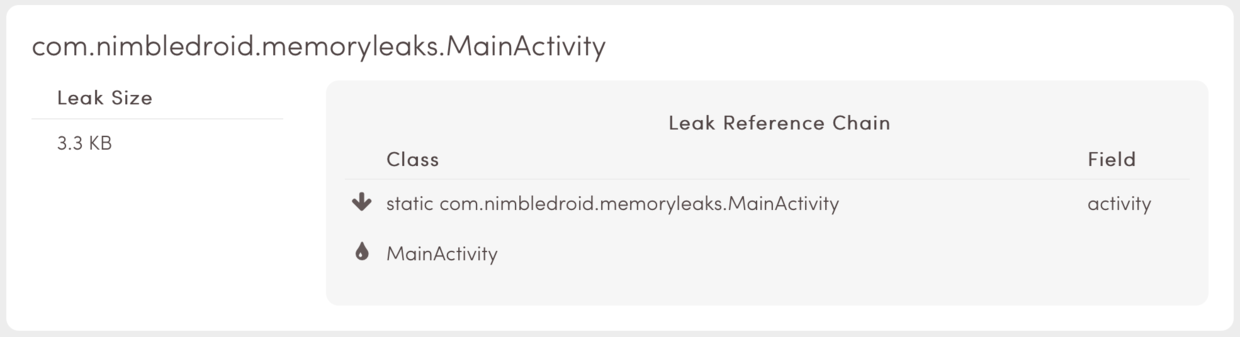
@Override public void onClick(View v) {

createHandler();

nextActivity();

}

});



Memory Leak 5 - Handler

**Threads**

我们再次通过[Thread](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L142)和[TimerTask](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L150)来展现内存泄漏。

void spawnThread() {

new Thread() {

@Override public void run() {

while(true);

}

}.start();

}

View tButton = findViewById(R.id.t\_button);

tButton.setOnClickListener(new View.OnClickListener() {

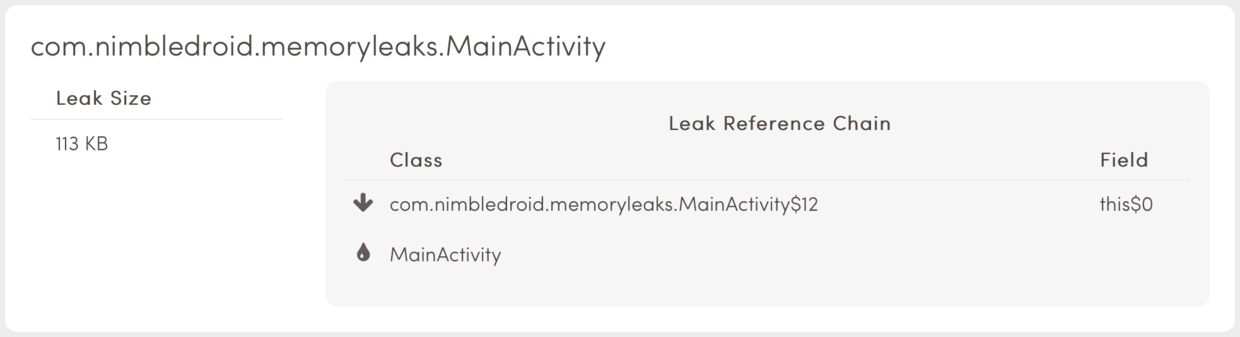
@Override public void onClick(View v) {

spawnThread();

nextActivity();

}

});



Memory Leak 6 - Thread

**TimerTask**

只要是匿名类的实例，不管是不是在工作线程，都会持有Activity的引用，导致内存泄漏。

void scheduleTimer() {

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

@Override

public void run() {

while(true);

}

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

}

View ttButton = findViewById(R.id.tt\_button);

ttButton.setOnClickListener(new View.OnClickListener() {

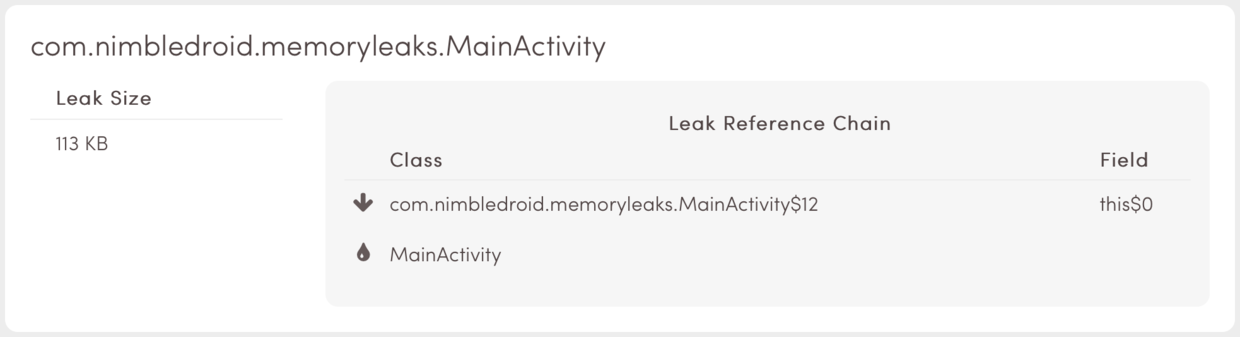
@Override public void onClick(View v) {

scheduleTimer();

nextActivity();

}

});



Memory Leak 7 - TimerTask

**Sensor Manager**

最后，通过[Context.getSystemService(int name)](http://developer.android.com/reference/android/content/Context.html#getSystemService(java.lang.String))可以获取系统服务。这些服务工作在各自的进程中，帮助应用处理后台任务，处理硬件交互。如果需要使用这些服务，可以注册[监听器](https://github.com/NimbleDroid/Memory-Leaks/blob/master/app/src/main/java/com/nimbledroid/memoryleaks/MainActivity.java#L136)，这会导致服务持有了Context的引用，如果在Activity销毁的时候没有注销这些监听器，会导致内存泄漏。

void registerListener() {

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

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

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

}

View smButton = findViewById(R.id.sm\_button);

smButton.setOnClickListener(new View.OnClickListener() {

@Override public void onClick(View v) {

registerListener();

nextActivity();

}

});



Memory Leak 8 - Sensor Manager

**总结**

看过那么多会导致内存泄漏的例子，容易导致吃光手机的内存使垃圾回收处理更为频发，甚至最坏的情况会导致OOM。垃圾回收的操作是很昂贵的开销，会导致肉眼可见的卡顿。所以，实例化的时候注意持有的引用链，并经常进行内存泄漏检查。