内存泄露的例子分析

**内存泄露（Memory Leak）**Java内存泄漏指的是进程中某些对象（垃圾对象）已经没有使用价值了，但是它们却可以直接或间接地引用到gc roots导致无法被GC回收。Dalvik VM具备的GC机制（垃圾回收机制）会在内存占用过多时自动回收，严重时会造成内存溢出OOM。

**内存溢出OOM**当应用程序申请的java heap空间超过Dalvik VM HeapGrowthLimit时，溢出。  
注意：OOM并不代表内存不足，只要申请的heap超过Dalvik VM HeapGrowthLimit时，即使内存充足也会溢出。效果是能让较多进程常驻内存。

# 静态变量引起的内存泄漏

在java中静态变量的生命周期是在类加载时开始，类卸载时结束。换句话说，在android中其生命周期是在进程启动时开始，进程死亡时结束。所以在程序的运行期间，如果进程没有被杀死，静态变量就会一直存在，不会被回收掉。如果静态变量强引用了某个Activity中变量，那么这个Activity就同样也不会被释放,即便是该Activity执行了onDestroy(不要将执行onDestroy和被回收划等号)。这类问题的解决方案为：1.寻找与该静态变量生命周期差不多的替代对象。2.若找不到，将强引用方式改成弱引用。比较典型的例子如下:

## 单例引起的Context内存泄漏

**public class** IMManager {  
 **private** Context **context**;  
 **private static** IMManager *mInstance*;  
  
 **public static** IMManager getInstance(Context context) {  
 **if** (*mInstance* == **null**) {  
 **synchronized** (IMManager.**class**) {  
 **if** (*mInstance* == **null**)  
 *mInstance* = **new** IMManager(context);  
 }  
 }  
 **return** *mInstance*;  
 }  
  
 **private** IMManager(Context context) {  
 **this**.**context** = context;  
 }  
}

当调用getInstance时，如果传入的context是Activity的context。只要这个单例没有被释放，这个Activity也不会被释放。

**解决方案**   
传入Application的context,因为Application的context的生命周期比Activity长，可以理解为Application的context与单例的生命周期一样长，传入它是最合适的。

**public class** IMManager {  
 **private** Context **context**;  
 **private static** IMManager *mInstance*;  
  
 **public static** IMManager getInstance(Context context) {  
 **if** (*mInstance* == **null**) {  
 **synchronized** (IMManager.**class**) {  
 **if** (*mInstance* == **null**)  
 *//将传入的context转换成Application的context   
 mInstance* = **new** IMManager(context.getApplicationContext());  
 }  
 }  
 **return** *mInstance*;  
 }  
  
 **private** IMManager(Context context) {  
 **this**.**context** = context;  
 }  
}

# 非静态内部类引起的内存泄漏

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。

在java里，非静态内部类 和 匿名类 都会潜在的引用它们所属的外部类。但是，静态内部类却不会。

如果这个非静态内部类实例做了一些耗时的操作，就会造成外围对象不会被回收，从而导致内存泄漏。这类问题的解决方案为：1.将内部类变成静态内部类 2.如果有强引用Activity中的属性，则将该属性的引用方式改为弱引用。3.在业务允许的情况下，当Activity执行onDestory时，结束这些耗时任务。

## 内部线程造成的内存泄漏

**public class** LeakAty **extends** Activity {  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.aty\_leak);  
 test();  
 }  
  
 **public void** test() {  
 *//匿名内部类会引用其外围实例LeakAty.this,所以会导致内存泄漏* **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }).start();  
 }  
}

**解决方案**   
将非静态匿名内部类修改为静态匿名内部类

**public class** LeakAty **extends** Activity {  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.aty\_leak);  
 *test*();  
 }  
  
 *//加上static，变成静态匿名内部类* **public static void** test() {  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **while** (**true**) {  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }).start();  
 }  
}

还例如：TimerTask

**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();  
 }  
});

## Handler引起的内存泄漏

**public class** LeakAty **extends** Activity {  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.aty\_leak);  
 fetchData();  
 }  
  
 **private** Handler **mHandler** = **new** Handler() {  
 **public void** handleMessage(android.os.Message msg) {  
 **switch** (msg.**what**) {  
 **case** 0:  
 *// 刷新数据* **break**;  
 **default**:  
 **break**;  
 }  
 };  
 };  
 **private void** fetchData() {  
 *//获取数据*

mHandler.sendEmptyMessage(0);

}  
}

mHandler 为匿名内部类实例，会引用外围对象LeakAty.this,如果该Handler在Activity退出时依然还有消息需要处理，那么这个Activity就不会被回收。

比如尤其是：mHandler.**postDelayed**(sRunnable, 1000 \* 60 \* 10);

**解决方案**

**public class** LeakAty **extends** Activity {  
 **private** TextView **tvResult**;  
 **private** MyHandler **handler**;  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.aty\_leak);  
 **tvResult** = (TextView) findViewById(R.id.tvResult);  
 **handler** = **new** MyHandler(**this**);  
 fetchData();  
 }  
  
 *//第一步，将Handler改成静态内部类。* **private static class** MyHandler **extends** Handler {  
 *//第二步，将需要引用Activity的地方，改成弱引用。* **private** WeakReference<LeakAty> **atyInstance**;  
  
 **public** MyHandler(LeakAty aty) {  
 **this**.**atyInstance** = **new** WeakReference<LeakAty>(aty);  
 }  
  
 @Override  
 **public void** handleMessage(Message msg) {  
 **super**.handleMessage(msg);  
 LeakAty aty = **atyInstance** == **null** ? **null** : **atyInstance**.get();  
 *//如果Activity被释放回收了，则不处理这些消息* **if** (aty == **null** || aty.isFinishing()) {  
 **return**;  
 }  
 aty.**tvResult**.setText(**"fetch data success"**);  
 }  
 }  
  
 **private void** fetchData() {  
 *// 获取数据* **handler**.sendEmptyMessage(0);  
 }  
  
 @Override  
 **protected void** onDestroy() {  
 *//第三步，在Activity退出的时候移除回调* **super**.onDestroy();  
 **handler**.removeCallbacksAndMessages(**null**);  
 }  
}

有人就会有疑问了，这里用弱引用，万一被回收了怎么办？岂不是会空指针。需要说明：你弱引用里面的哥们Activity都被干掉了，你一个Handler还想继续玩什么？

**小结**虽然静态类与非静态类之间的区别并不大，但是对于Android开发者而言却是必须理解的。至少我们要清楚，如果一个内部类实例的生命周期比Activity更长，那么我们千万不要使用非静态的内部类。最好的做法是，使用静态内部类，然后在该类里使用弱引用来指向所在的Activity。

# 资源未关闭引起的内存泄漏

当使用了BraodcastReceiver、Cursor、Bitmap、自定义属性attr等资源时，当不需要使用时，需要及时释放掉，若没有释放，则会引起内存泄漏

# 不要用的监听未移除

1.调用了View.getViewTreeObserver().addOnXXXListener ,而没有调用View.getViewTreeObserver().removeXXXListener

2.传感器

**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();  
 }  
});

1. PhoneStateListener电话状态监听

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无法被垃圾回收。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。

# 无限循环动画

在Activity中播放属性动画中的一类无限循环动画，没有在ondestory中停止动画，Activity会被动画持有而无法释放