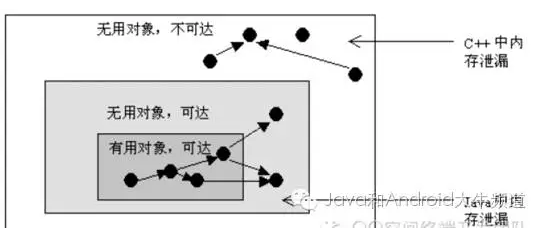
**首先讲讲内存泄漏：**



[**http://mp.weixin.qq.com/s?\_\_biz=MzI0MjE3OTYwMg==&mid=2649548387&idx=1&sn=8b826d2d29e688d813d010568ff99d63&chksm=f1180d1ec66f8408b1c7b5899b83767f0ec1c9641114e060d98524ee2ee5df31a43bfbe924bf&scene=0#wechat\_redirect**](http://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649548387&idx=1&sn=8b826d2d29e688d813d010568ff99d63&chksm=f1180d1ec66f8408b1c7b5899b83767f0ec1c9641114e060d98524ee2ee5df31a43bfbe924bf&scene=0#wechat_redirect)

**首先，要明确，内存泄漏并不是不可达的。JAVA内存泄漏所指范围与C++不一致。GC会自动回收C++中所指的不可达内存。一般JAVA指的内存泄漏只是没有对可达内存进行合理操作，实际上是可达的。**

**前言**

对于C++来说，内存泄漏就是new出来的对象没有delete，俗称野指针；对于Java来说，就是new出来的Object 放在Heap上无法被GC回收；本文通过QQ和Qzone中内存泄漏实例来讲android中内存泄漏分析解法和编写代码应注意的事项。

**Java 中的内存分配**

1. 静态储存区：编译时就分配好，在程序整个运行期间都存在。它主要存放静态数据和常量；
2. 栈区：当方法执行时，会在栈区内存中创建方法体内部的局部变量，方法结束后自动释放内存；
3. 堆区：通常存放 new 出来的对象。由 Java 垃圾回收器回收。

**四种引用类型的介绍**

1. 强引用(StrongReference)：JVM 宁可抛出 OOM ，也不会让 GC 回收具有强引用的对象；
2. 软引用(SoftReference)：只有在内存空间不足时，才会被回的对象；
3. 弱引用(WeakReference)：在 GC 时，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存；
4. 虚引用(PhantomReference)：任何时候都可以被GC回收，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否存在该对象的虚引用，来了解这个对象是否将要被回收。可以用来作为GC回收Object的标志。

注意：在java中会有相应API可以建立除了强引用之外的三种引用。还有ReferenceQueue用于处理三种引用的处理。

<http://www.2cto.com/kf/201403/285767.html>

[**http://www.cnblogs.com/E-star/p/3441846.html**](http://www.cnblogs.com/E-star/p/3441846.html)

我们常说的内存泄漏是指new出来的Object无法被GC回收，即为强引用：  
  
内存泄漏发生时的主要表现为内存抖动，可用内存慢慢变少：  
  
**Andriod中分析内存泄漏的工具MAT**

1. MAT（Memory Analyzer Tools）是一个 Eclipse 插件，它是一个快速、功能丰富的JAVA heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。
2. MAT 插件的下载地址： www.eclipse.org/mat
3. MAT 使用方法介绍：

http://www.cnblogs.com/larack/p/6071209.html

**QQ和Qzone内存泄漏如何监控**

QQ和Qzone 的内存泄漏采用SNGAPM解决方案，SNGAPM是一个性能监控、分析的统一解决方案，它从终端收集性能信息，上报到一个后台，后台将监控类信息聚合展示为图表，将分析类信息进行分析并提单，通知开发者；

1. SNGAPM由App（MagnifierApp）和

web server（MagnifierServer）两部分组成；

1. MagnifierApp在自动内存泄漏检测中是一个衔接检测组件（LeakInspector）和自动化云分析（MagnifierCloud）的中间性平台，它从LeakInspector的内存dump自动化上传MagnifierServer；
2. MagnifierServer后台会定时提交分析任务到MagnifierCloud；
3. MagnifierCloud分析结束之后会更新数据到magnifier web上，同时以bug单形式通知开发者。

**常见的内存泄漏案例**

**case 1. 单例造成的内存泄露**

单例的静态特性导致其生命周期同应用一样长。

*解决方案：*

1. 将该属性的引用方式改为弱引用;
2. 如果传入Context，使用ApplicationContext;

*example：*

泄漏代码片段

private static ScrollHelper mInstance;      
private ScrollHelper() {

}      
public static ScrollHelper getInstance() {          
    if (mInstance == null) {             
       synchronized (ScrollHelper.class) {                  
            if (mInstance == null) {

               mInstance = new ScrollHelper();

           }

       }

   }          
                  
    return mInstance;

}      
/\*\*

\* 被点击的view

\*/  
private View mScrolledView = null;      
public void setScrolledView(View scrolledView) {

   mScrolledView = scrolledView;

}

*Solution：使用WeakReference*

private static ScrollHelper mInstance;      
private ScrollHelper() {

}      
public static ScrollHelper getInstance() {          
    if (mInstance == null) {              
        synchronized (ScrollHelper.class) {                  
            if (mInstance == null) {

               mInstance = new ScrollHelper();

           }

       }

   }          
          
    return mInstance;

}      
/\*\*

\* 被点击的view

\*/  
private WeakReference<View> mScrolledViewWeakRef = null;      
public void setScrolledView(View scrolledView) {

   mScrolledViewWeakRef = new WeakReference<View>(scrolledView);

}

**case 2. InnerClass匿名内部类**

在Java中，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。

*解决方案：*

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;

*example：*

public class LeakAct extends Activity {    
    @Override  
    protected void onCreate(Bundle savedInstanceState) {      
        super.onCreate(savedInstanceState);

       setContentView(R.layout.aty\_leak);

       test();  
    }   
    //这儿发生泄漏      
    public void test() {      
        new Thread(new Runnable() {        
            @Override

           public void run() {          
                while (true) {            
                    try {

                       Thread.sleep(1000);

                   } catch (InterruptedException e) {

                       e.printStackTrace();

                   }

               }

           }

       }).start();

   }  
}

*Solution：*

public class LeakAct 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();

   }

}

**case 3. Activity Context 的不正确使用**

在Android应用程序中通常可以使用两种Context对象：Activity和Application。当类或方法需要Context对象的时候常见的做法是使用第一个作为Context参数。这样就意味着View对象对整个Activity保持引用，因此也就保持对Activty的所有的引用。

假设一个场景，当应用程序有个比较大的Bitmap类型的图片，每次旋转是都重新加载图片所用的时间较多。为了提高屏幕旋转是Activity的创建速度，最简单的方法时将这个Bitmap对象使用Static修饰。 当一个Drawable绑定在View上，实际上这个View对象就会成为这份Drawable的一个Callback成员变量。而静态变量的生命周期要长于Activity。导致了当旋转屏幕时，Activity无法被回收，而造成内存泄露。

*解决方案：*

1. 使用ApplicationContext代替ActivityContext，因为ApplicationContext会随着应用程序的存在而存在，而不依赖于activity的生命周期；
2. 对Context的引用不要超过它本身的生命周期，慎重的对Context使用“static”关键字。Context里如果有线程，一定要在onDestroy()里及时停掉。

*example：*

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);

}

*Solution：*

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 = getApplicationContext().getDrawable(R.drawable.large\_bitmap);

   }

   label.setBackgroundDrawable(sBackground);

   setContentView(label);

}

**case 4. Handler引起的内存泄漏**

当Handler中有延迟的的任务或是等待执行的任务队列过长，由于消息持有对Handler的引用，而Handler又持有对其外部类的潜在引用，这条引用关系会一直保持到消息得到处理，而导致了Activity无法被垃圾回收器回收，而导致了内存泄露。

*解决方案：*

1. 可以把Handler类放在单独的类文件中，或者使用静态内部类便可以避免泄露;
2. 如果想在Handler内部去调用所在的Activity,那么可以在handler内部使用弱引用的方式去指向所在Activity.使用Static + WeakReference的方式来达到断开Handler与Activity之间存在引用关系的目的。

*Solution*

@Override  
protected void doOnDestroy() {          
    super.doOnDestroy();          
    if (mHandler != null) {

       mHandler.removeCallbacksAndMessages(null);

   }

   mHandler = null;

   mRenderCallback = null;

}

**case 5. 注册监听器的泄漏**

系统服务可以通过Context.getSystemService 获取，它们负责执行某些后台任务，或者为硬件访问提供接口。如果Context 对象想要在服务内部的事件发生时被通知，那就需要把自己注册到服务的监听器中。然而，这会让服务持有Activity 的引用，如果在Activity onDestory时没有释放掉引用就会内存泄漏。

*解决方案：*

1. 使用ApplicationContext代替ActivityContext;
2. 在Activity执行onDestory时，调用反注册;

mSensorManager = (SensorManager) this.getSystemService(Context.SENSOR\_SERVICE);

*Solution：*

mSensorManager = (SensorManager) getApplicationContext().getSystemService(Context.SENSOR\_SERVICE);

下面是容易造成内存泄漏的系统服务：

InputMethodManager imm = (InputMethodManager) context.getApplicationContext().getSystemService(Context.INPUT\_METHOD\_SERVICE);

*Solution：*

protected void onDetachedFromWindow() {          
    if (this.mActionShell != null) {  
        this.mActionShell.setOnClickListener((OnAreaClickListener)null);

   }          
    if (this.mButtonShell != null) {   
        this.mButtonShell.setOnClickListener((OnAreaClickListener)null);

   }          
    if (this.mCountShell != this.mCountShell) {  
        this.mCountShell.setOnClickListener((OnAreaClickListener)null);

   }          
    super.onDetachedFromWindow();

}

**case 6. Cursor，Stream没有close，View没有recyle**

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，往往会造成内存泄漏。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，应该调用它的close()函数，将其关闭掉，然后才置为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。

*Solution：*

调用onRecycled()

@Override  
public void onRecycled() {

    reset();

   mSinglePicArea.onRecycled();

}

在View中调用reset()

public void reset() {  
    if (mHasRecyled) {              
        return;

   }

...

   SubAreaShell.recycle(mActionBtnShell);

   mActionBtnShell = null;

...

   mIsDoingAvatartRedPocketAnim = false;          
    if (mAvatarArea != null) {

           mAvatarArea.reset();

   }          
    if (mNickNameArea != null) {

       mNickNameArea.reset();

   }

}

**case 7. 集合中对象没清理造成的内存泄漏**

我们通常把一些对象的引用加入到了集合容器（比如ArrayList）中，当我们不需要该对象时，并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。  
所以要在退出程序之前，将集合里的东西clear，然后置为null，再退出程序。

*解决方案：*

在Activity退出之前，将集合里的东西clear，然后置为null，再退出程序。

*Solution*

private List<EmotionPanelInfo> data;      
public void onDestory() {          
    if (data != null) {

       data.clear();

       data = null;

   }

}

**case 8. WebView造成的泄露**

当我们不要使用WebView对象时，应该调用它的destory()函数来销毁它，并释放其占用的内存，否则其占用的内存长期也不能被回收，从而造成内存泄露。

*解决方案：*

为webView开启另外一个进程，通过AIDL与主线程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，从而达到内存的完整释放。

**case 9. 构造Adapter时，没有使用缓存的ConvertView**

初始时ListView会从Adapter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)