目录

[快速查找当前Activity 2](#_Toc57985279)

[快速定位生产闪退问题 2](#_Toc57985280)

[Android库项目中不能使用switch语句进行资源文件的判断 2](#_Toc57985281)

[查看手机对每个应用分配的内存大小 3](#_Toc57985282)

[RxJava中线程切换 3](#_Toc57985283)

[滑动监听 3](#_Toc57985284)

[关于自定义VIEW与性能优化 4](#_Toc57985285)

[关于位运算 4](#_Toc57985286)

[关于组件化 4](#_Toc57985287)

[大家都知道，很多时候我们说"Android16.6ms刷新一次屏幕" 4](#_Toc57985288)

[自定义 View 的流程 7](#_Toc57985289)

[转场动画 11](#_Toc57985290)

[关于在主线程执行耗时操作可能出现的问题 12](#_Toc57985291)

[关于构造器 12](#_Toc57985292)

[关于点击事件的分发 12](#_Toc57985293)

[计算机网络的七层结构 13](#_Toc57985294)

[算法相关 – 指定数的幂 13](#_Toc57985295)

[TextView与EditText同样设置点击事件，EditText不可编辑时，二者存在差异? 14](#_Toc57985296)

[java类库相关 15](#_Toc57985297)

[HTTPS加密概念 17](#_Toc57985298)

[算法题—滑动窗口 19](#_Toc57985299)

[RecyclerView和ListView的异同 19](#_Toc57985300)

[内存泄漏的问题总结 21](#_Toc57985301)

[ViewStub 标签懒加载 24](#_Toc57985302)

[ConstraintLayout下ellipsize无效的问题 25](#_Toc57985303)

[Activity与fragment的生命周期互动 26](#_Toc57985304)

[Android Window Flag介绍 28](#_Toc57985305)

[控件刷新调用树 32](#_Toc57985306)

[Handel源码 33](#_Toc57985307)

[RxJava使用场景介绍 35](#_Toc57985308)

# 快速查找当前Activity

|  |
| --- |
| 命令行：adb shell "dumpsys window w | grep name=" |

# 快速定位生产闪退问题

|  |
| --- |
| 日志输出到电脑指定目录，如d:\locat.text  adb logcat -v time >D:\\logcat.log  捕获闪退日志  adb logcat -v time | find "E/AndroidRuntime" >D:\\logcat.log>D:\\logcat.log  捕获特定的包名下的日志  **adb logcat -v time | find "com.pmg.pmgenius.sichuantianyan" >D:\\logcat.log** |

# Android库项目中不能使用switch语句进行资源文件的判断

|  |
| --- |
| 详情如下：  In a regular Android project, constants in the resource R class are declared like this:  public static final int main=0x7f030004;  However, as of ADT 14, in a library project, they will be declared like this:  public static int main=0x7f030004;  In other words, the constants are not final in a library project.  在常规的Android项目中，资源R类中的常量声明如下：  公共静态最终int main=0x7f030004；  然而，从ADT 14开始，在图书馆项目中，它们的声明如下：  公共静态int main=0x7f030004；  换句话说，这些常量在库项目中不是最终的。 |

# 查看手机对每个应用分配的内存大小

|  |
| --- |
| 命令行：adb shell getprop dalvik.vm.heapsize |

# RxJava中线程切换

|  |
| --- |
| Rxjava通过 observeOn() 的多次调用，程序实现了线程的多次切换。  如：  Observable.just(1, 2, 3, 4) // IO 线程，由 subscribeOn() 指定  .subscribeOn(Schedulers.io())  .observeOn(Schedulers.newThread())  .map(mapOperator) // 新线程，由 observeOn() 指定  .observeOn(Schedulers.io())  .map(mapOperator2) // IO 线程，由 observeOn() 指定  .observeOn(AndroidSchedulers.mainThread)  .subscribe(subscriber); // Android 主线程，由 observeOn()指定  注意：  多次指定上游(Observable)的线程只有第一次指定的有效, 也就是说多次调用subscribeOn() 只有第一次的有效, 其余的会被忽略.  多次指定下游(Observer)的线程是可以的, 也就是说每调用一次observeOn() , 下游的线程就会切换一次  相关文章：https://www.jianshu.com/p/404ca4e3e85d |

# 滑动监听

|  |
| --- |
| binding.recyclerInvitationSearch.addOnScrollListener(new RecyclerView.OnScrollListener() {  @Override  public void onScrolled(@NonNull RecyclerView recyclerView, int dx, int dy) {  super.onScrolled(recyclerView, dx, dy);  if (dy!=0 && !recyclerView.canScrollVertically(1)){  getViewModel().getMoreData();  }  } });  资料参考：   1. 自定义view：[https://www.gcssloop.com/#blog](https://www.gcssloop.com/" \l "blog) |

# 关于自定义VIEW与性能优化

|  |
| --- |
| 尽量避免在频繁调用的方法中调用new创建对象，可以使用对象缓存池进行性能调优。  耗时的算法（递归查找等）可以在子线程进行调用。 |

# 关于位运算

|  |
| --- |
| 清零取反要用与，某位置一可用或  若要取反和交换，轻轻松松用异或 |

# 关于组件化

参考资料：<https://blog.csdn.net/nnmmbb/article/details/82969995>

# 大家都知道，很多时候我们说"Android16.6ms刷新一次屏幕"

那么问题是：这个16.6ms刷新一次屏幕是什么意思呢？是每隔16.6ms都会调用一次View的onDraw么？如果不是？那是什么样的一个刷新机制，这个机制在Android版本迭代中有无变化。

|  |
| --- |
| 每过16.6ms都会回调一次View的onDraw方法吗？ 不是的，这个频率，说的是系统发出屏幕刷新信号频率(但我也不太确定是不是真的16.6ms（在源码里没看到过)），但是在onDraw中收到回调的时机，是不确定的（各种原因，如代码写法、设备性能等）。 那它的刷新机制是怎样的呢？ **屏幕刷新大致流程：**   1. View调用invalidate方法； 2. ViewRootImpl会把doTraversal任务（处理View的测量、布局、绘制），post到Choreographer中（在系统下一次发出同步信号的时候，这个doTraversal任务会被执行）； 3. Choreographer会借助DisplayEventReceiver的scheduleVsync方法，在native层（IDisplayEventConnection.cpp）通过Binder（跨进程）发出一个**REQUEST\_NEXT\_VSYNC**的Tag（发去了哪里不知道，我找不到那个类）； 4. 在DisplayEventDispatcher.cpp中，会看到一个handleEvent方法（可以猜测，系统在下一次发出屏幕刷新信号时，(间接或直接)回调的就是这个方法），它里面会调用android\_view\_DisplayEventReceiver.cpp中的dispatchVsync方法，**这个方法最终去了哪里呢？** 5. 回到java层，在DisplayEventReceiver中会看到一个同名同签名的dispatchVsync方法，并且上面有注释写着：**Called from native code.** (可以知道，这个方法是在native层被调用的)，它里面会调用onVsync方法， 6. 而onVsync方法在Choreographer中内部类的实现，最终是会调用doFrame方法的，这个doFrame方法，里面会把刚刚在ViewRootImpl中post到Choreographer里的doTraversal任务（处理View的测量、布局、绘制）执行！！。   **以上提到的native层代码，分别(按顺序)在以下链接中能看到：**   * <http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/jni/android_view_DisplayEventReceiver.cpp> 关键方法：**nativeScheduleVsync** * <http://androidxref.com/9.0.0_r3/xref/frameworks/base/libs/androidfw/DisplayEventDispatcher.cpp> 关键方法：**scheduleVsync** * <http://androidxref.com/9.0.0_r3/xref/frameworks/base/libs/androidfw/include/androidfw/DisplayEventDispatcher.h> 关键方法：**requestNextVsync** * <http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/gui/DisplayEventReceiver.cpp> 关键方法：**requestNextVsync** * <http://androidxref.com/9.0.0_r3/xref/frameworks/native/libs/gui/IDisplayEventConnection.cpp> 关键方法：**requestNextVsync**  这个机制在版本迭代中有何变化？ 刚刚分别试了下手头上的6.0和9.0系统的测试机， 在一个自定义View中的onDraw方法不断invalidate和打印被回调的时间：  **private long lastTime;**  **@Override**  **public void onDraw(Canvas canvas) {**  **LogUtil.print(SystemClock.uptimeMillis() - lastTime);**  **lastTime = SystemClock.uptimeMillis();**  **invalidate();**  **}**  **发现在6.0的手机是上这样的（随机截取20条log）：**  onDraw: 14  onDraw: 0  onDraw: 13  onDraw: 0  onDraw: 14  onDraw: 1  onDraw: 14  onDraw: 1  onDraw: 13  onDraw: 0  onDraw: 14  onDraw: 1  onDraw: 14  onDraw: 1  onDraw: 13  onDraw: 1  onDraw: 10  onDraw: 1  onDraw: 12  onDraw: 1  **而9.0系统的手机是这样的（随机截取20条log）：**  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 16  onDraw: 15  onDraw: 15  onDraw: 16  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 15  onDraw: 14  onDraw: 15  onDraw: 15  可以看到，在6.0系统上，好像有个缓存机制，不断地invalidate的话，每两次中有一次是**<=**1ms就回调了，可以猜想这一次没有等待系统的屏幕刷新信号就直接回调了onDraw方法。 而9.0系统则比较稳定，onDraw方法每一次被回调都间隔了15ms左右. |

# 自定义 View 的流程

|  |
| --- |
| 1. 要想充分理解自定义View的流程，就必须对View的绘制流程有深刻理解，下面我简单说几点： 2. #### DecorView被加载到Window中  - 从Activity的startActivity开始，最终调用到ActivityThread的handleLaunchActivity方法来创建Activity，首先，会调用performLaunchActivity方法，内部会执行Activity的onCreate方法，从而完成DecorView和Activity的创建。然后，会调用handleResumeActivity，里面首先会调用performResumeActivity去执行Activity的onResume()方法，执行完后会得到一个ActivityClientRecord对象，然后通过r.window.getDecorView()的方式得到DecorView，然后会通过a.getWindowManager()得到WindowManager，最终调用其addView()方法将DecorView加进去。 - WindowManager的实现类是WindowManagerImpl，它内部会将addView的逻辑委托给WindowManagerGlobal，可见这里使用了接口隔离和委托模式将实现和抽象充分解耦。在WindowManagerGlobal的addView()方法中不仅会将DecorView添加到Window中，同时会创建ViewRootImpl对象，并将ViewRootImpl对象和DecorView通过root.setView()把DecorView加载到Window中。这里的ViewRootImpl是ViewRoot的实现类，是连接WindowManager和DecorView的纽带。View的三大流程均是通过ViewRoot来完成的。  #### 了解绘制的整体流程  绘制会从根视图ViewRoot的performTraversals()方法开始，从上到下遍历整个视图树，每个View控件负责绘制自己，而ViewGroup还需要负责通知自己的子View进行绘制操作。  #### 理解MeasureSpec  MeasureSpec表示的是一个32位的整形值，它的高2位表示测量模式SpecMode，低30位表示某种测量模式下的规格大小SpecSize。MeasureSpec是View类的一个静态内部类，用来说明应该如何测量这个View。它由三种测量模式，如下：  - EXACTLY：精确测量模式，视图宽高指定为match\_parent或具体数值时生效，表示父视图已经决定了子视图的精确大小，这种模式下View的测量值就是SpecSize的值。 - AT\_MOST：最大值测量模式，当视图的宽高指定为wrap\_content时生效，此时子视图的尺寸可以是不超过父视图允许的最大尺寸的任何尺寸。 - UNSPECIFIED：不指定测量模式, 父视图没有限制子视图的大小，子视图可以是想要的任何尺寸，通常用于系统内部，应用开发中很少用到。  MeasureSpec通过将SpecMode和SpecSize打包成一个int值来避免过多的对象内存分配，为了方便操作，其提供了打包和解包的方法，打包方法为makeMeasureSpec，解包方法为getMode和getSize。  普通View的MeasureSpec的创建规则如下：  ![image](https://images2015.cnblogs.com/blog/918357/201706/918357-20170618234001337-203688773.png)  对于DecorView而言，它的MeasureSpec由窗口尺寸和其自身的LayoutParams共同决定；对于普通的View，它的MeasureSpec由父视图的MeasureSpec和其自身的LayoutParams共同决定。  #### View绘制流程之Measure  - 首先，在ViewGroup中的measureChildren()方法中会遍历测量ViewGroup中所有的View，当View的可见性处于GONE状态时，不对其进行测量。 - 然后，测量某个指定的View时，根据父容器的MeasureSpec和子View的LayoutParams等信息计算子View的MeasureSpec。 - 最后，将计算出的MeasureSpec传入View的measure方法，这里ViewGroup没有定义测量的具体过程，因为ViewGroup是一个抽象类，其测量过程的onMeasure方法需要各个子类去实现。不同的ViewGroup子类有不同的布局特性，这导致它们的测量细节各不相同，如果需要自定义测量过程，则子类可以重写这个方法。（setMeasureDimension方法用于设置View的测量宽高，如果View没有重写onMeasure方法，则会默认调用getDefaultSize来获得View的宽高）  ##### getSuggestMinimumWidth分析  如果View没有设置背景，那么返回android:minWidth这个属性所指定的值，这个值可以为0；如果View设置了背景，则返回android:minWidth和背景的最小宽度这两者中的最大值。  ##### 自定义View时手动处理wrap\_content时的情形  直接继承View的控件需要重写onMeasure方法并设置wrap\_content时的自身大小，否则在布局中使用wrap\_content就相当于使用match\_parent。此时，可以在wrap\_content的情况下（对应MeasureSpec.AT\_MOST）指定内部宽/高(mWidth和mHeight)。  ##### LinearLayout的onMeasure方法实现解析（这里仅分析measureVertical核心源码）  系统会遍历子元素并对每个子元素执行measureChildBeforeLayout方法，这个方法内部会调用子元素的measure方法，这样各个子元素就开始依次进入measure过程，并且系统会通过mTotalLength这个变量来存储LinearLayout在竖直方向的初步高度。每测量一个子元素，mTotalLength就会增加，增加的部分主要包括了子元素的高度以及子元素在竖直方向上的margin等。  ##### 在Activity中获取某个View的宽高  由于View的measure过程和Activity的生命周期方法不是同步执行的，如果View还没有测量完毕，那么获得的宽/高就是0。所以在onCreate、onStart、onResume中均无法正确得到某个View的宽高信息。解决方式如下：  - Activity/View#onWindowFocusChanged：此时View已经初始化完毕，当Activity的窗口得到焦点和失去焦点时均会被调用一次，如果频繁地进行onResume和onPause，那么onWindowFocusChanged也会被频繁地调用。 - view.post(runnable)： 通过post可以将一个runnable投递到消息队列的尾部，始化好了然后等待Looper调用次runnable的时候，View也已经初始化好了。 - ViewTreeObserver#addOnGlobalLayoutListener：当View树的状态发生改变或者View树内部的View的可见性发生改变时，onGlobalLayout方法将被回调。 - View.measure(int widthMeasureSpec, int heightMeasureSpec)：match\_parent时不知道parentSize的大小，测不出；具体数值时，直接makeMeasureSpec固定值，然后调用view..measure就可以了；wrap\_content时，在最大化模式下，用View理论上能支持的最大值去构造MeasureSpec是合理的。  #### View的绘制流程之Layout  首先，会通过setFrame方法来设定View的四个顶点的位置，即View在父容器中的位置。然后，会执行到onLayout空方法，子类如果是ViewGroup类型，则重写这个方法，实现ViewGroup中所有View控件布局流程。  ##### LinearLayout的onLayout方法实现解析（layoutVertical核心源码）  其中会遍历调用每个子View的setChildFrame方法为子元素确定对应的位置。其中的childTop会逐渐增大，意味着后面的子元素会被放置在靠下的位置。  注意：在View的默认实现中，View的测量宽/高和最终宽/高是相等的，只不过测量宽/高形成于View的measure过程，而最终宽/高形成于View的layout过程，即两者的赋值时机不同，测量宽/高的赋值时机稍微早一些。在一些特殊的情况下则两者不相等：  - 重写View的layout方法,使最终宽度总是比测量宽/高大100px。 - View需要多次measure才能确定自己的测量宽/高，在前几次测量的过程中，其得出的测量宽/高有可能和最终宽/高不一致，但最终来说，测量宽/高还是和最终宽/高相同。  #### View的绘制流程之Draw  ##### Draw的基本流程  绘制基本上可以分为六个步骤：  - 首先绘制View的背景； - 如果需要的话，保持canvas的图层，为fading做准备； - 然后，绘制View的内容； - 接着，绘制View的子View； - 如果需要的话，绘制View的fading边缘并恢复图层； - 最后，绘制View的装饰(例如滚动条等等)。  ##### setWillNotDraw的作用  如果一个View不需要绘制任何内容，那么设置这个标记位为true以后，系统会进行相应的优化。  - 默认情况下，View没有启用这个优化标记位，但是ViewGroup会默认启用这个优化标记位。 - 当我们的自定义控件继承于ViewGroup并且本身不具备绘制功能时，就可以开启这个标记位从而便于系统进行后续的优化。 - 当明确知道一个ViewGroup需要通过onDraw来绘制内容时，我们需要显示地关闭WILL\_NOT\_DRAW这个标记位。   #### Requestlayout，onlayout，onDraw，DrawChild区别与联系？  requestLayout()方法 ：会导致调用 measure()过程 和 layout()过程，将会根据标志位判断是否需要ondraw。  onLayout()方法：如果该View是ViewGroup对象，需要实现该方法，对每个子视图进行布局。  onDraw()方法：绘制视图本身 (每个View都需要重载该方法，ViewGroup不需要实现该方法)。  drawChild()：去重新回调每个子视图的draw()方法。   #### invalidate() 和 postInvalidate()的区别 ？  invalidate()与postInvalidate()都用于刷新View，主要区别是invalidate()在主线程中调用，若在子线程中使用需要配合handler；而postInvalidate()可在子线程中直接调用。 [更详细的内容请查看这里](https://jsonchao.github.io/2018/10/28/Android%20View%E7%9A%84%E7%BB%98%E5%88%B6%E6%B5%81%E7%A8%8B/) |

# 转场动画

|  |
| --- |
| Android stytle.xml  android:activityOpenEnterAnimation 一个activity创建进入的效果。  android:activityOpenExitAnimation 一个activity还没有finish()下退出效果, 比如有俩个activity A与B 首先启动A 然后再启动B 那么A还没有finish() 这时A的退出效果。  android:activityCloseEnterAnimation 表示上一个activity返回进入效果 比如有俩个activity A与B B在最上面，B退出(finish)后 A重新进入的效果。  android:activityCloseExitAnimation 表示的是activity finish()之后的效果 比如有俩个activity A与B B退出后会被finish() 那么B的退出效果在这定义。  相关方法  overridePendingTransition(int enterAnim, int exitAnim)； |

# 关于在主线程执行耗时操作可能出现的问题

|  |
| --- |
| 1. 超时，系统弹窗应用无响应 2. 界面卡顿，动画失效 |

# 关于构造器

|  |
| --- |
| 构造器中尽量避免耗时操作，应该尽量可能地精简。  也不应该处理耗费资源过多的操作，比如开启线程 |

# 关于点击事件的分发

|  |
| --- |
| 事件分发三个方法   1. 分发 dispatchTouchEvent 2. 拦截 onInterceptTouchEvent 3. 处理 onTounchEven   一般情况下，down -> move -> up为一个连续的事件，一般地这个连续事件都由一个view进行处理。  但是有些控件会拦截子view的特定事件，比如recyclerView拦截子view的move事件，导致这个连续的事件不是由一个view进行处理。  RecyclerView的拦截机制  当down事件交给子view处理时，标记此view为后续事件处理view。  但是当检测到move事件的时候，就启动拦截机制，首先第一个move事件会清除子view的标记，第二个move事件开始进行滚动操作。也就是在滑动的时候，recyclerView阻截了事件，传递给子view的是CANCEL事件，同时让自己变成了传递链的末尾，并且消耗了一个motion\_move事件。第二个motion\_move的时候才会真正开始执行该move的滑动。 |

# 计算机网络的七层结构

|  |
| --- |
| **应用层**  与其它计算机进行通讯的一个应用，它是对应应用程序的通信服务的。例如，一个没有通信功能的字处理程序就不能执行通信的代码，从事字处理工作的程序员也不关心OSI的第7层。但是，如果添加了一个传输文件的选项，那么字处理器的程序就需要实现OSI的第7层。示例：TELNET，HTTP，FTP，NFS，SMTP等。  **表示层**  这一层的主要功能是定义数据格式及加密。例如，FTP允许你选择以二进制或ASCII格式传输。如果选择二进制，那么发送方和接收方不改变文件的内容。如果选择ASCII格式，发送方将把文本从发送方的字符集转换成标准的ASCII后发送数据。在接收方将标准的ASCII转换成接收方计算机的字符集。示例：加密，ASCII等。  **会话层**  它定义了如何开始、控制和结束一个会话，包括对多个双向消息的控制和管理，以便在只完成连续消息的一部分时可以通知应用，从而使表示层看到的数据是连续的，在某些情况下，如果表示层收到了所有的数据，则用数据代表表示层。示例：RPC，SQL等。  **传输层**  这层的功能包括是否选择差错恢复协议还是无差错恢复协议，及在同一主机上对不同应用的数据流的输入进行复用，还包括对收到的顺序不对的数据包的重新排序功能。示例：TCP，UDP，SPX。  **网络层**  这层对端到端的包传输进行定义，它定义了能够标识所有结点的逻辑地址，还定义了路由实现的方式和学习的方式。为了适应最大传输单元长度小于包长度的传输介质，网络层还定义了如何将一个包分解成更小的包的分段方法。示例：IP，IPX等。  **数据链路层**  它定义了在单个链路上如何传输数据。这些协议与被讨论的各种介质有关。示例：ATM，FDDI等。  **物理层**  OSI的物理层规范是有关传输介质的特性，这些规范通常也参考了其他组织制定的标准。连接头、帧、帧的使用、电流、编码及光调制等都属于各种物理层规范中的内容。物理层常用多个规范完成对所有细节的定义。示例：Rj45，802.3等。 |

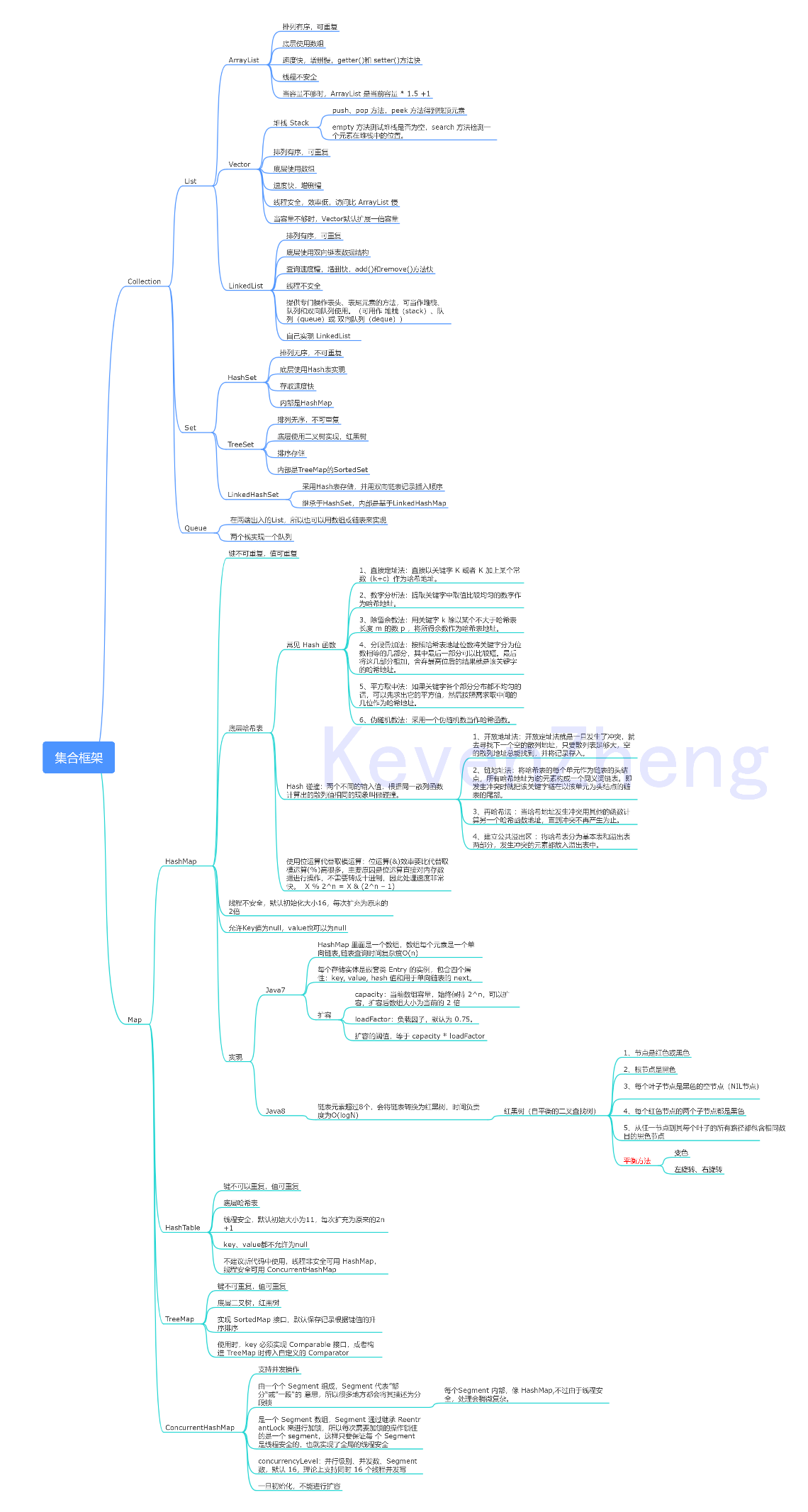
# 算法相关 – 指定数的幂

|  |
| --- |
| https://leetcode-cn.com/problems/power-of-three/solution/3de-mi-by-leetcode/ |

# **TextView与EditText同样设置点击事件，EditText不可编辑时，二者存在差异?**

|  |
| --- |
| TextView点击一次即可响应点击事件  Edittext在初始化后需要第一点击不响应点击事件，第二次点击方响应事件  分析：Edittext需要获取焦点之后才能对事件进行响应 |

# **java类库相关**



# **HTTPS加密概念**

|  |
| --- |
| 名词概念  对称加密算法：对称加密(也叫[私钥](https://baike.baidu.com/item/%E7%A7%81%E9%92%A5" \t "_blank)加密)指加密和解密使用相同[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5" \t "_blank)的加密算法。有时又叫传统[密码算法](https://baike.baidu.com/item/%E5%AF%86%E7%A0%81%E7%AE%97%E6%B3%95)，就是加密密钥能够从解密密钥中推算出来，同时解密密钥也可以从加密密钥中推算出来。而在大多数的[对称算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E7%AE%97%E6%B3%95)中，加密密钥和解密密钥是相同的，所以也称这种加密算法为秘密密钥算法或单密钥算法。它要求发送方和接收方在安全通信之前，商定一个密钥。对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信的安全性至关重要。  非对称加密算法：非对称加密算法需要两个密钥：[公开密钥](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%AF%86%E9%92%A5/7453570)（publickey:简称公钥）和私有密钥（privatekey:简称私钥）。公钥与私钥是一对，如果用公钥对数据进行加密，只有用对应的私钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。 非对称加密算法实现机密信息交换的基本过程是：甲方生成一对[密钥](https://baike.baidu.com/item/%E5%AF%86%E9%92%A5/101144)并将公钥公开，需要向甲方发送信息的其他角色(乙方)使用该密钥(甲方的公钥)对机密信息进行加密后再发送给甲方；甲方再用自己私钥对加密后的信息进行解密。甲方想要回复乙方时正好相反，使用乙方的公钥对数据进行加密，同理，乙方使用自己的私钥来进行解密。  另一方面，甲方可以使用自己的私钥对机密信息进行签名后再发送给乙方；乙方再用甲方的公钥对甲方发送回来的数据进行验签。  甲方只能用其私钥解密由其公钥加密后的任何信息。 非[对称加密算法](https://baike.baidu.com/item/%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95" \t "_blank)的保密性比较好，它消除了最终用户交换密钥的需要。  非对称[密码体制](https://baike.baidu.com/item/%E5%AF%86%E7%A0%81%E4%BD%93%E5%88%B6" \t "_blank)的特点：算法强度复杂、安全性依赖于算法与密钥但是由于其算法复杂，而使得加密解密速度没有对称加密解密的速度快。对称密码体制中只有一种密钥，并且是非公开的，如果要解密就得让对方知道密钥。所以保证其安全性就是保证密钥的安全，而非对称密钥体制有两种密钥，其中一个是公开的，这样就可以不需要像对称密码那样传输对方的密钥了。这样安全性就大了很多。  http原理通俗理解：   1. 从一个聊天软件说起，我们要实现A能发一个hello消息给B：   考虑安全性问题，要实现  A发给B的hello消息包，即使被中间人拦截到了，也无法得知消息的内容  如何做到真正的安全？   1. 问题边界：A与B通信的内容，有且只有A和B有能力看到通信的真正内容   解决方案，很容易就想到了对消息进行加密。  对于A与B这样的简单通信模型，我们很容易做出选择：这就是对称加密算法，密钥S同时扮演加密和解密的角色只要这个密钥S不公开给第三者，同时密钥S足够安全，我们就解决了我们一开始所定问题域了。因为世界上有且只有A与B知道如何加密和解密他们之间的消息。  但是，在WWW环境下，我们的Web服务器的通信模型没有这么简单：  如果服务器端对所有的客户端通信都使用同样的对称加密算法，无异于没有加密。那怎么办呢？即能使用对称加密算法，又不公开密钥？  答案是：Web服务器与每个客户端使用不同的对称加密算法：  另一个问题来了，我们的服务器端怎么告诉客户端该使用哪种对称加密算法？  当然是通过协商。但是协商没有加密，给协商加密，后面会引起套娃的问题。   1. 如何对协商过程进行加密   新问题来了，如何对协商过程进行加密？密码学领域中，有一种称为“非对称加密”的加密算法，特点是私钥加密后的密文，只要是公钥，都可以解密，但是公钥加密后的密文，只有私钥可以解密。私钥只有一个人有，而公钥可以发给所有的人。  虽然服务器端向A、B……的方向还是不安全的，但是至少A、B向服务器端方向是安全的。  好了，如何协商加密算法的问题，我们解决了：使用非对称加密算法进行对称加密算法协商过程。  这下，你明白为什么HTTPS同时需要对称加密算法和非对称加密算法了吧？   1. 协商什么加密算法   要达到Web服务器针对每个客户端使用不同的对称加密算法，同时，我们也不能让第三者知道这个对称加密算法是什么，怎么办？  使用随机数，就是使用随机数来生成对称加密算法。这样就可以做到服务器和客户端每次交互都是新的加密算法、只有在交互的那一该才确定加密算法。  这下，你明白为什么HTTPS协议握手阶段会有这么多的随机数了吧。   1. 如何得到公钥？   细心的人可能已经注意到了如果使用非对称加密算法，我们的客户端A，B需要一开始就持有公钥，要不没法开展加密行为啊。  这下，我们又遇到新问题了，如何让A、B客户端安全地得到公钥？  我能想到的方案只有这些：  方案1. 服务器端将公钥发送给每一个客户端  方案2. 服务器端将公钥放到一个远程服务器，客户端可以请求得到  我们选择方案1，因为方案2又多了一次请求，还要另外处理公钥的放置问题。  公钥被调包了怎么办？又是一个鸡生蛋蛋生鸡问题？  但是方案1有个问题：如果服务器端发送公钥给客户端时，被中间人调包了，怎么办？   1. 使用第三方机构的公钥解决鸡生蛋蛋生鸡问题   显然，让每个客户端的每个浏览器默认保存所有网站的公钥是不现实的。  公钥被调包的问题出现，是因为我们的客户端无法分辨返回公钥的人到底是中间人，还是真的服务器。这其实就是密码学中提的身份验证问题。  我是这样解决的。既然服务器需要将公钥传给客户端，这个过程本身是不安全，那么我们为什么不对这个过程本身再加密一次？可是，你是使用对称加密，还是非对称加密？这下好了，我感觉又进了鸡生蛋蛋生鸡问题了。  问题的难点是如果我们选择直接将公钥传递给客户端的方案，我们始终无法解决公钥传递被中间人调包的问题。  所以，我们不能直接将服务器的公钥传递给客户端，而是第三方机构使用它的私钥对我们的公钥进行加密后，再传给客户端。客户端再使用第三方机构的公钥进行解密。   1. 另一个场景一个场景：第三方机构不可能只给你一家公司制作证书，它也可能会给中间人这样有坏心思的公司发放证书。这样的，中间人就有机会对你的证书进行调包，客户端在这种情况下是无法分辨出是接收的是你的证书，还是中间人的。因为不论中间人，还是你的证书，都能使用第三方机构的公钥进行解密。 2. 数字签名，解决同一机构颁发的不同证书被篡改问题   要解决这个问题，我们首先要想清楚一个问题，辨别同一机构下不同证书的这个职责，我们应该放在哪？  只能放到客户端了。意思是，客户端在拿到证书后，自己就有能力分辨证书是否被篡改了。如何才能有这个能力呢？  我们从现实中找灵感。比如你是HR，你手上拿到候选人的学历证书，证书上写了持证人，颁发机构，颁发时间等等，同时证书上，还写有一个最重要的：证书编号！我们怎么鉴别这张证书是的真伪呢？只要拿着这个证书编号上相关机构去查，如果证书上的持证人与现实的这个候选人一致，同时证书编号也能对应上，那么就说明这个证书是真实的。  我们的客户端能不能采用这个机制呢？  可是，这个“第三方机构”到底是在哪呢？是一个远端服务？不可能吧？如果是个远端服务，整个交互都会慢了。所以，这个第三方机构的验证功能只能放在客户端的本地了。   1. 客户端本地怎么验证证书呢？   答案是证书本身就已经告诉客户端怎么验证证书的真伪。  也就是证书上写着如何根据证书的内容生成证书编号。客户端拿到证书后根据证书上的方法自己生成一个证书编号，如果生成的证书编号与证书上的证书编号相同，那么说明这个证书是真实的。  同时，为避免证书编号本身又被调包，所以使用第三方的私钥进行加密。  但是第三方机构的公钥怎么跑到了客户端的机器中呢？世界上这么多机器。  其实呢，现实中，浏览器和操作系统都会维护一个权威的第三方机构列表（包括它们的公钥）。因为客户端接收到的证书中会写有颁发机构，客户端就根据这个颁发机构的值在本地找相应的公钥。  说到这里，想必大家已经知道上文所说的，证书就是HTTPS中数字证书，证书编号就是数字签名，而第三方机构就是指数字证书签发机构（CA）。  相关链接：<https://www.cnblogs.com/zhenguoli/p/8622933.html> |

# **算法题—滑动窗口**

|  |
| --- |
| 使用变量记录左右下标，对比，寻找最大值 |

# **RecyclerView和ListView的异同**

|  |
| --- |
| ViewHolder是用来保存视图引用的类，无论是ListView亦或是RecyclerView。只不过在ListView中，ViewHolder需要自己来定义，且这只是一种推荐的使用方式，不使用当然也可以，这不是必须的。只不过不使用ViewHolder的话，ListView每次getView的时候都会调用findViewById(int)，这将导致ListView性能展示迟缓。而在RecyclerView中使用RecyclerView.ViewHolder则变成了必须，尽管实现起来稍显复杂，但它却解决了ListView面临的上述不使用自定义ViewHolder时所面临的问题。  我们知道ListView只能在垂直方向上滚动，Android API没有提供ListView在水平方向上面滚动的支持。或许有多种方式实现水平滑动，但是请想念我，ListView并不是设计来做这件事情的。但是RecyclerView相较于ListView，在滚动上面的功能扩展了许多。它可以支持多种类型列表的展示要求，主要如下：  LinearLayoutManager，可以支持水平和竖直方向上滚动的列表。  StaggeredGridLayoutManager，可以支持交叉网格风格的列表，类似于瀑布流或者Pinterest。  GridLayoutManager，支持网格展示，可以水平或者竖直滚动，如展示图片的画廊。  列表动画是一个全新的、拥有无限可能的维度。起初的Android API中，删除或添加item时，item是无法产生动画效果的。后面随着Android的进化，Google的Chat Hasse推荐使用ViewPropertyAnimator属性动画来实现上述需求。 相比较于ListView，RecyclerView.ItemAnimator则被提供用于在RecyclerView添加、删除或移动item时处理动画效果。同时，如果你比较懒，不想自定义ItemAnimator，你还可以使用DefaultItemAnimator。  ListView的Adapter中，getView是最重要的方法，它将视图跟position绑定起来，是所有神奇的事情发生的地方。同时我们也能够通过registerDataObserver在Adapter中注册一个观察者。RecyclerView也有这个特性，RecyclerView.AdapterDataObserver就是这个观察者。ListView有三个Adapter的默认实现，分别是ArrayAdapter、CursorAdapter和SimpleCursorAdapter。然而，RecyclerView的Adapter则拥有除了内置的内DB游标和ArrayList的支持之外的所有功能。RecyclerView.Adapter的实现的，我们必须采取措施将数据提供给Adapter，正如BaseAdapter对ListView所做的那样。  在ListView中如果我们想要在item之间添加间隔符，我们只需要在布局文件中对ListView添加如下属性即可：  android:divider="@android:color/transparent"  android:dividerHeight="5dp"  ListView通过AdapterView.OnItemClickListener接口来探测点击事件。而RecyclerView则通过RecyclerView.OnItemTouchListener接口来探测触摸事件。它虽然增加了实现的难度，但是却给予开发人员拦截触摸事件更多的控制权限。  ListView可以设置选择模式，并添加MultiChoiceModeListener，如下所示：  listView.setChoiceMode(ListView.CHOICE\_MODE\_MULTIPLE\_MODAL);  listView.setMultiChoiceModeListener(new MultiChoiceModeListener() { public boolean onCreateActionMode(ActionMode mode, Menu menu) { ... } public void onItemCheckedStateChanged(ActionMode mode, int position, long id, boolean checked) { ... } public boolean onActionItemClicked(ActionMode mode, MenuItem item) { switch (item.getItemId()) { case R.id.menu\_item\_delete\_crime: CrimeAdapter adapte |

# **内存泄漏的问题总结**

|  |
| --- |
| 1、内类是有危险的编码方式  说道内类就不得不提到 ”this$0“，它是一种奇特的内类成员，每个类实例都具有一个 this$0，当它的内类需要访问它的成员时，内类就会持有外类的 this$0，通过 this$0 就可以访问外部类所有的成员。  解决方案是在 Activity 关闭，即触发 onDestory 时解除内类和外部的引用关系。  2、普通 Hanlder 内部类的问题  这也是一个 this$0 间接引用的问题，对于 Handler 的解决方案一般可以归结为如下三个步骤：  • 1）、把内类声明成 static：用来断绝 this$0 的引用。因为 static 描述的内类从 Java 编译原理的角度看，”内类“与”外类“相互独立，互相都没有访问对方成员变量的能力。  • 2、使用 WeakReference 来引用外部类的实例。  • 3、在外部类（如 Activity）销毁的时候使用 removeCallbackAndMessages 来移除回调和消息。  这里需要在使用过程中注意对 WeakReference 进行判空。  3、登录界面的内存问题  如果在闪屏页跳转到登录界面时没有调用 finish()，则会造成闪屏页的内存泄漏，在碰到这种”过渡界面“的情况时，需要注意不要产生这样的内存 Bug。  4、使用系统服务时产生的内存问题  我们通常都会使用 getSystemService 方法来获取系统服务，但是当在 Activity 中调用时，会默认把 Activity 的 Context 传给系统服务，在某些不确定的情况下，某些系统服务内部会产生异常，从而 hold 住外界传入的 Context。  解决方案是 直接使用 Applicaiton 的 Context 去获取系统服务。  5、把 WebView 类型的泄漏装进垃圾桶进程  我们都知道，对应 WebView 来说，其 网络延时、引擎 Session 管理、Cookies 管理、引擎内核线程、HTML5 调用系统声音、视频播放组件等产生的引用链条无法及时打断，造成的内存问题基本上可以用”无解“来形容。  解决方案是我们可以 把 WebView 装入另一个进程。 具体为在 AndroidManifes 中对当前的 Activity 设置 android:process 属性即可，最后，在 Activity 的 onDestory 中退出进程，这样即可基本上终结 WebView 造成的泄漏。  6、在适当的时候对组件进行注销  我们在平常开发过程中经常需要在Activity创建的时候去注册一些组件，如广播、定时器、事件总线等等。这个时候我们应该在适当的时候对组件进行注销，如 onPause 或 onDestory 方法中。  7、Handler / FrameLayout 的 postDelyed 方法触发的内存问题  不仅在使用 Handler 的 sendMessage 方法时，我们需要在 onDestory 中使用 removeCallbackAndMessage 移除回调和消息，在使用到 Handler / FrameLayout 的 postDelyed 方法时，我们需要调用 removeCallbacks 去移除实现控件内部的延时器对 Runnable 内类的持有。  8、图片放错资源目录也会有内存问题  在做资源适配的时候，因为需要考虑到 APK 的瘦身问题，无法为每张图片在每个 drawable / mipmap 目录下安置一张适配图片的副本。很多同学不知道图片应该放哪个目录，如果放到分辨率低的目录如 hdpi 目录，则可能会造成内存问题，这个时候建议尽量问设计人员要高品质图片然后往高密度目录下方，如 xxhdpi 目录，这样 在低密屏上”放大倍数“是小于1的，在保证画质的前提下，内存也是可控的。也可以使用 Drawable.createFromSream 替换 getResources().getDrawable 来加载，这样便可以绕过 Android 的默认适配规则。  对于已经被用户使用物理“返回键”退回到后台的进程，如果包含了以下 两点，则 不会被轻易杀死。  • 1）、进程包含了服务 startService，而服务本身调用了 startForeground（低版本需通过反射调用）。  • 2）、主 Activity 没有实现 onSaveInstanceState 接口。  但建议 在运行一段时间（如3小时）后主动保存界面进程（位于后台），然后重启它，这样可以有效地降低内存负载。  9、列表 item 被回收时注意释放图片的引用  我们应该在 item 被回收不可见时去释放掉对图片的引用。如果你使用的是 ListView，由于每次 item 被回收后被再次利用都会去重新绑定数据，所以只需在 ImageView 回调其 onDetchFromWindow 方法的时候区释放掉图片的引用即可。如果你使用的是 RecyclerView，因为被回收不可见时第一次选择是放进 mCacheView中，但是这里面的 item 被复用时并不会去执行 bindViewHolder 来重新绑定数据，只有被回收进 mRecyclePool 后拿出来复用才会重新绑定数据。所以此时我们应该在 item 被回收进 RecyclePool 的时候去释放图片的引用，这里我们只要去 重写 Adapter 中的 onViewRecycled 方法 就可以了，代码如下所示：  @Override  public void onViewRecycled(@Nullable VH holder) {  super.onViewRecycled(holder);  if (holder != null) {  //做释放图片引用的操作  }  }  复制代码  10、使用 ViewStub 进行占位  我们应该使用 ViewStub 对那些没有马上用到的资源去做延迟加载，并且还有很多大概率不会出现的 View 更要去做懒加载，这样可以等到要使用时再去为它们分配相应的内存。  11、注意定时清理 App 过时的埋点数据  产品或者运营为了统计数据会在每个版本中不断地增加新的埋点。所以我们需要定期地去清理一些过时的埋点，以此来 适当地优化内存以及CPU的压力。  12、针对匿名内部类 Runnable 造成内存泄漏的处理  我们在做子线程操作的时候，喜欢使用匿名内部类 Runnable 来操作。但是，如果某个 Activity 放在线程池中的任务不能及时执行完毕，在 Activity 销毁时很容易导致内存泄漏。因为这个匿名内部类 Runnable 类持有一个指向 Outer 类的引用，这样一来如果 Activity 里面的 Runnable 不能及时执行，就会使它外围的 Activity 无法释放，产生内存泄漏。从上面的分析可知，只要在 Activity 退出时没有这个引用即可，那我们就通过反射，在 Runnable 进入线程池前先干掉它，代码如下所示：  Field f = job.getClass().getDeclaredField("this$0");  f.setAccessible(true);  f.set(job, null);  复制代码  这个任务就是我们的 Runnable 对象，而 ”this$0“ 就是上面所指的外部类的引用了。这里注意使用 WeakReference 装起来，要执行了先 get 一下，如果是 null 则说明 Activity 已经回收，任务就放弃执行。 |

# ViewStub 标签懒加载

|  |
| --- |
| <ViewStub> 标签实质上是一个宽高都为 0 的不可见 View. 通过延迟加载布局的方式优化布局提升渲染性能.  这里的延迟加载是指初始化时, 程序无需显示该标签所指向的布局文件, 只有在特定的条件下, 所指向的布局文件才需要被渲染, 且此布局文件直接将当前的 <ViewStub> 替换掉. 但这里的替换并不是完全意义上的替换, 布局文件的 layout params 是以 ViewStub 为优先.  当初次渲染布局文件时, ViewStub 控件虽然也占据内存, 但是相比于其他控件, 它所占内存很小. 它主要是作为一个“占位符”, 放置于 View Tree中, 且它本身是不可见的.  <ViewStub  android:id="@+id/view\_stub"  android:layout="@layout/layout\_to\_show"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent" />  要被加载的布局通过 android:layout 属性来设置. 然后在程序中调用 inflate() 方法来加载. 还可以设定 Visibility 为 VISIBLE 或 INVISIBLE, 也会触发 inflate(). 但只有直接使用 inflate() 方法能返回布局文件的根 View. 但是这里只会在首次使用 setVisibility() 会加载要渲染的布局文件. 再次使用只是单纯的设置可见性.  对 inflate() 操作也只能进行一次, 因为 inflate() 的时候是其指向的布局文件替换掉当前 <ViewStub> 标签. 之后, 原来的布局文件中就没有 <ViewStub> 标签了. 因此, 如果多次 inflate() 操作, 会报错: ViewStub must have a non-null ViewGroup viewParent |

# ConstraintLayout下ellipsize无效的问题

|  |
| --- |
| 出现这种问题按照下面两种思路进行排查   1. 布局的约束范围是否有效 2. 使用ellipsize应该结合singleLine使用，而不是使用maxline属性   就目前实验来看，maxline存在兼容问题 |

# Activity与fragment的生命周期互动

|  |
| --- |
|  |

# Android Window Flag介绍

|  |
| --- |
| Window win = getWindow();  win.addFlags(WindowManager.LayoutParams.FLAG\_SHOW\_WHEN\_LOCKED);//覆盖在屏幕锁之上。  PowerManager pm = (PowerManager) getSystemService(Context.POWER\_SERVICE);  if (!pm.isScreenOn()) {//屏幕时候保持高亮  win.addFlags(  WindowManager.LayoutParams.FLAG\_KEEP\_SCREEN\_ON  | WindowManager.LayoutParams.FLAG\_TURN\_SCREEN\_ON  | WindowManager.LayoutParams.FLAG\_ALLOW\_LOCK\_WHILE\_SCREEN\_ON  | WindowManager.LayoutParams.FLAG\_LAYOUT\_INSET\_DECOR  );}  弹出窗口后的内容变暗。  public static final int FLAG\_DIM\_BEHIND       = 0x00000002;  弹出窗口后的内容变模糊。  public static final int FLAG\_BLUR\_BEHIND       = 0x00000004;  弹出窗口不能获得按键输入事件，事件将会往其他窗口传递。  public static final int FLAG\_NOT\_FOCUSABLE     = 0x00000008;  弹出窗口不接受触摸屏事件，事件将会往其他窗口传递。  public static final int FLAG\_NOT\_TOUCHABLE     = 0x00000010;  当设备休眠时，点击触摸屏，弹出窗口将收到这个触摸事件。  public static final int FLAG\_NOT\_TOUCH\_MODAL   = 0x00000020;  触摸事件被系统所消耗，用户不会看到他们点击屏幕有什么反应。  public static final int FLAG\_TOUCHABLE\_WHEN\_WAKING = 0x00000040;  弹出窗口可见时，保持设备常亮不变。  public static final int FLAG\_KEEP\_SCREEN\_ON    = 0x00000080;  弹出窗口占满整个屏幕，忽略周围的装饰边框（例如状态栏）。  public static final int FLAG\_LAYOUT\_IN\_SCREEN   =0x00000100;  允许弹出窗口扩展到屏幕之外。  public static final int FLAG\_LAYOUT\_NO\_LIMITS   =0x00000200;  弹出窗口显示时，隐藏所有的屏幕装饰（例如状态条）。  public static final int FLAG\_FULLSCREEN     = 0x00000400;  此选项将覆盖FLAG\_FULLSCREEN选项，并强制屏幕装饰（如状态条）弹出。  public static final int FLAG\_FORCE\_NOT\_FULLSCREEN   =0x00000800;  弹出窗口半透明的显示处理方法，又称“点透”。通常用于图形处理较差的设备。  public static final int FLAG\_DITHER           = 0x00001000;  弹出窗口不允许屏幕截图。  public static final int FLAG\_SECURE           = 0x00002000;  弹出窗口特殊模式，布局参数用于指示显示比例。  public static final int FLAG\_SCALED           = 0x00004000;  当屏幕有贴着脸时，防止面颊对屏幕造成误操作。  public static final int FLAG\_IGNORE\_CHEEK\_PRESSES   = 0x00008000;  确保窗口内容不会被装饰条（状态栏）盖住。  public static final int FLAG\_LAYOUT\_INSET\_DECOR = 0x00010000;  窗口不能与输入法交互，覆盖输入法窗口。（同时设置FLAG\_NOT\_FOCUSABLE，窗口将能够与输入法交互，输入法窗口覆盖）  public static final int FLAG\_ALT\_FOCUSABLE\_IM = 0x00020000;  当触屏事件发生在弹出窗口之外，可以通过设置此标志接收到一个MotionEvent.ACTION\_OUTSIDE事件。  public static final int FLAG\_WATCH\_OUTSIDE\_TOUCH = 0x00040000;  当屏幕锁定时，弹出窗口可以被看到。这使得应用程序窗口优先于锁屏界面。可配合FLAG\_KEEP\_SCREEN\_ON  public static final int FLAG\_SHOW\_WHEN\_LOCKED = 0x00080000;  系统墙纸显示为弹出窗口背景。窗口必须是半透明的。  public static final int FLAG\_SHOW\_WALLPAPER = 0x00100000;  窗口一旦显示出来，系统将点亮屏幕。  public static final int FLAG\_TURN\_SCREEN\_ON = 0x00200000;  弹出窗口时解除锁屏。锁屏界面非加密才能解锁。如果是加密的，需要设置了FLAG\_SHOW\_WHEN\_LOCKED  public static final int FLAG\_DISMISS\_KEYGUARD = 0x00400000;  锁屏界面淡出时，继续运行它的动画。  public static final int FLAG\_KEEP\_SURFACE\_WHILE\_ANIMATING =0x10000000;  以原始尺寸显示窗口。  public static final int FLAG\_COMPATIBLE\_WINDOW = 0x20000000;  设置此选项的窗口将无条件获得焦点。  public static final int FLAG\_SYSTEM\_ERROR = 0x40000000; |

# 控件刷新调用树

|  |
| --- |
|  |

# Handel源码

|  |
| --- |
| 变量：  interface Callback 用来避免使用子类的方式写一个Handel的回调接口  方法：  handleMessage 核心方法，用来处理一个message  dispatchMessage 分发message，如果message中存在callback则调用handleCallback，handleCallback中会直接调用message中的callback的run方法。如果message不存在callback变量，则会判断handel是否拥有callback变量，存在则调用其中的handleMessage不存在则调用自身的handleMessage方法。  *\* Handle system messages here.  \*/* public void dispatchMessage(@NonNull Message msg) {  if (msg.callback != null) {  *handleCallback*(msg);  } else {  if (mCallback != null) {  if (mCallback.handleMessage(msg)) {  return;  }  }  handleMessage(msg);  } }  构造方法：  public Handler(@Nullable Callback callback, boolean async)  主要逻辑：对mLooper，mQueue，mCallback进行赋值  如果mLooper为null则抛出一个运行时异常  public Handler(@Nullable Callback callback, boolean async) {  if (*FIND\_POTENTIAL\_LEAKS*) {  final Class<? extends Handler> klass = getClass();  if ((klass.isAnonymousClass() || klass.isMemberClass() || klass.isLocalClass()) &&  (klass.getModifiers() & Modifier.*STATIC*) == 0) {  Log.*w*(*TAG*, "The following Handler class should be static or leaks might occur: " +  klass.getCanonicalName());  }  }   mLooper = Looper.*myLooper*();  if (mLooper == null) {  throw new RuntimeException(  "Can't create handler inside thread " + Thread.*currentThread*()  + " that has not called Looper.prepare()");  }  mQueue = mLooper.mQueue;  mCallback = callback;  mAsynchronous = async; }  getMessageName 顾名思义  obtainMessage 从全局消息池中获取一个message，大小为50，源码位置在Message  post 提交一个任务  postAtTime 在设定的时间提交一个任务  postDelayed 延迟一段时间提交一个任务  postAtFrontOfQueue 提交一个任务，Looper下次迭代立刻执行 插队  runWithScissors 同步运行指定任务  简单来说如果是主线程直接调用run方法执行，如果不是则调用相应的handel执行  removeCallbacks 删除一个任务  sendMessage 发送一个消息  sendMessageAtFrontOfQueue 发送一个插队消息  executeOrSendMessage 执行一个消息  如果是在用一个线程直接调用handelMessgae处理消息，否则调用相应的handel进行处理 |

# RxJava使用场景介绍

|  |
| --- |
| 链接：https://www.zhihu.com/question/53151203/answer/276395705  RxJava中包括丰富的操作符，这里结合实际的场景来谈谈Rx中常用的操作符的使用。  我们在项目中使用最多的就是**Retrofit**配合**RxJava**的网络请求处理。但是项目中可能会有一些特殊的需求或设计不合理的接口，需要客户端对接口的调用进行高精度的控制，这时候RxJava的操作符就派上大用场了。  **场景一：使用flatMap注册成功后自动登录**  假设现在有一个这样的需求：服务端有注册和登录两个接口，但是这两个接口的功能是分离的，注册成功并不会自动登录，需要客户端再进行调用登录接口才可以。  现在我们来分析下这个场景：需要在注册成功后调用登录，在没有使用RxJava的时候可能会这样解决：在注册登录的回调里再写一个登录，我来用伪代码来模拟下：  Api.register(username, password, new CallBack() {  public void onSuccess(){  Api.login(username, password, new CallBack() {  public void onSuccess() {  //登陆成功  }  });  }  });  这里没有涉及到异常情况的处理，这样是可以实现这样的功能，但是很容易陷入**缩进地狱**，假设现在有需要在登录前通过网络获取一下ip，那就是又要加一层回调。这样代码不美观且不利于维护，我们来看看用Rx是怎样处理的吧：  Api.register(account, password)  .flatMap(new Function<BizResponse<UserInfo>, ObservableSource<UserLoginResponse>>() {  @Override  public ObservableSource<UserLoginResponse> apply(@NonNull BizResponse<UserInfo> response) throws Exception {  // 注册成功后执行登录方法  return Api.login(account, password);  }  })  .subscribe(new Consumer<UserLoginResponse>() {  @Override  public void accept(@NonNull UserLoginResponse userLoginResponse) throws Exception {  // 成功  }  }, new Consumer<Throwable>() {  @Override  public void accept(@NonNull Throwable throwable) throws Exception {  // 失败  }  });  这里我们用到了一个操作符**flatMap**，它和map很像，不一样的是map从一个对象转化为另一个对象，而flatMap可以从一个对象转化为一个Observable对象，后续继续对它进行操作。  这样咋一看麻烦了很多，其实分析一下其实这段代码更容易维护——**如果需要增加新的操作，比如异步获取ip，就在flatMap之前加入这个操作，flatMap之后的代码不用动；而且有统一的异常入口——里面任何的异常都会在第二个Consumer中捕获。**  **场景二：使用compose简化代码**  在网络请求中一般会加入这两句话  observable.subscribeOn(Schedulers.io())  .observeOn(AndroidSchedulers.mainThread());  在io线程执行网络操作，在主线程执行ui操作。但是每个observable都要写这两句话，能不能简化呢？当然是可以的。这就要用到**compose**操作符。我们可以在RxUtils中定义一个这样的方法：  /\*\*  \* io线程执行，主线程观察  \* .compose(RxUtils.<T>applySchedulers())  \*/  public static <T> ObservableTransformer<T, T> applySchedulers() {  return new ObservableTransformer<T, T>() {  @Override  public ObservableSource<T> apply(@NonNull Observable<T> observable) {  return observable.subscribeOn(Schedulers.io())  .observeOn(AndroidSchedulers.mainThread());  }  };  }  使用的时候在**subscribe**方法之前加入*.compose(RxUtils.<T>applySchedulers())*就可以了。  当然，这个方法可以增加更多的逻辑，比如判断一个接口返回的数据是否异常，一般服务端会返回一个状态来判断：  /\*\*  \* 增加了统一的判断逻辑  \*/  public static <T extends BizResponse> ObservableTransformer<T, T> applyBizSchedulers() {  return new ObservableTransformer<T, T>() {  @Override  public ObservableSource<T> apply(@NonNull Observable<T> observable) {  return observable  .map(new Function<T, T>() {  @Override  public T apply(@NonNull final T t) throws Exception {  if (t.getRcode() != 0) {  throw new RuntimeException(t.getRmsg());  }  return t;  }  })  .subscribeOn(Schedulers.io())  .observeOn(AndroidSchedulers.mainThread());  }  };  }  上面的例子中，如果返回的code不是0，则直接抛出异常，统一在 **Consumer<Throwable>**中处理。  **场景三：使用zip对并发的请求做处理**  现在有一个这样的需求：假设在用户页面，不仅要显示用户的普通信息，还要显示用户的资金信息。这两个信息分别对应的两个接口，同时访问两个接口，**当两个信息都获取成功的时候才算成功，如果有一个没获取成功则算失败，需要重新获取。**  如果没有rx，这个需求实现起来非常复杂，只能按照先后顺序获取，这样就会增加加载的时间。  我们可以通过RxJava中的zip来实现：  Observable.zip(Api.getNewsList(), Api.getBanner(), new BiFunction<List<News>, BizResponse<List<Banner>>, Pair<List<News>, List<Banner>>>() {  @Override  public Pair<List<News>, List<Banner>> apply(List<News> news, List<Banner> banner) throws Exception {  return Pair.create(news, banner);  }  })  .compose(RxUtils.applySchedulers())  .subscribe(...)  **场景四：debounce控制频率**  假设有一个搜索框，在输入后自动搜索输入的内容。咋一看这个需求很简单，只需要监听EditText的输入事件，然后调用搜索接口就可以。但是如果用户输入过快，就会造成接口的频繁调用。有什么好的方式控制接口的调用频率呢？**debounce**就可以解决。  RxTextView.textChanges(search)  .debounce(1, TimeUnit.SECONDS)  通过这段代码，EditText的textChange事件会在输入间隔大于一秒时调用，这样如果用户因正在输入造成的接口调用频繁的问题就解决了。 |

# Android 响应SEND,VIEW

|  |
| --- |
| Send：  <intent-filter android:label="发送" >  <action android:name="android.intent.action.SEND" />  <category android:name="android.intent.category.DEFAULT" />  <data android:mimeType="\*/\*" />  </intent-filter>  View：  <intent-filter android:label="@string/copy\_to\_pmg"  tools:ignore="AppLinkUrlError">  <action android:name="android.intent.action.VIEW" />  <category android:name="android.intent.category.DEFAULT" />  <data android:mimeType="text/plain" />  <data android:mimeType="image/jpeg" />  <data android:mimeType="image/png" />  <data android:mimeType="image/bmp" />  <data android:mimeType="image/tiff" />  <data android:mimeType="image/gif" />  <data android:mimeType="image/jpeg" />  <data android:mimeType="image/x-ms-bmp" />  <data android:mimeType="image/png" />  <data android:mimeType="image/x-pcx" />  <data android:mimeType="image/targa" />  <data android:mimeType="image/x-photoshop" />  <data android:mimeType="video/mpeg" />  <data android:mimeType="audio/mp4"/>  <data android:mimeType="audio/mp3"/>  <data android:mimeType="audio/mp3"/>  <data android:mimeType="video/x-ms-wmv"/>  <data android:mimeType="video/x-ms-wm"/>  <data android:mimeType="video/x-ms-wmx"/>  <data android:mimeType="application/octet-stream" />  <data android:mimeType="application/pdf" />  <data android:mimeType="application/msword" />  <data android:mimeType="application/pdf" />  <data android:mimeType="application/zip" />  <data android:mimeType="application/x-xls" />  <data android:mimeType="application/msword" />  <data android:mimeType="application/x-ppt"/>  <data android:mimeType="application/vnd.ms-powerpoint"/>  </intent-filter> |

# 响应Action\_View并复制

|  |
| --- |
| 1.  if (Intent.*ACTION\_VIEW*.equals(action)){  if (*SCHEME\_FILE*.equals(intent.getScheme()) || *SCHEME\_CONTENT*.equals(intent.getScheme())) {  Uri uri = intent.getData();  copyFileActionViewToQ(uri);  } }  private void copyFileActionViewToQ(Uri path){  Flowable.*just*(path)  .subscribeOn(Schedulers.*io*())  .map((Function<Uri, ResponseModel<String>>) s -> {  try {  File file = *uriToFileCopy*(s,mView.getContext());  if (file == null){  return TaskDetailsBiz.*obtainResponse*(0,mView.getContext().getString(R.string.*share\_file\_fail*,"Q 返回为NULL"),null);  }else{  return TaskDetailsBiz.*obtainResponse*(1,mView.getContext().getString(R.string.*share\_file\_success*),null);  }  }catch (Exception e){  e.printStackTrace();  return TaskDetailsBiz.*obtainResponse*(0,mView.getContext().getString(R.string.*share\_file\_fail*,e.getMessage()),null);  }  })  .observeOn(AndroidSchedulers.*mainThread*())  .subscribe(getRequestSubscriber()); }  private static File uriToFileCopy(Uri uri, Context context) throws IOException{  File file = null;  if(uri == null) return file;  //android10以上转换  if (uri.getScheme().equals(ContentResolver.*SCHEME\_FILE*)) {  file = new File(uri.getPath());  InputStream is = new FileInputStream(file);  File path = FilePathProvider.*getPublicDir*(context, FilePathProvider.*DIR\_ENCLOSURE* +  "/" + DateTimeUtils.*formatDatetime*(new Date(),*FORMAT\_DATE3*));  if (!path.exists()){  path.mkdir();  }  File cache = new File(path.getAbsolutePath(), file.getName());  FileOutputStream fos = new FileOutputStream(cache);  *copyFileStream*(is,fos);  file = cache;  } else if (uri.getScheme().equals(ContentResolver.*SCHEME\_CONTENT*)) {  //把文件复制到沙盒目录  ContentResolver contentResolver = context.getContentResolver();  String displayName = null;  //注释掉的方法可以获取到原文件的文件名，但是比较耗时  Cursor cursor = contentResolver.query(uri, null, null, null, null);  if (cursor.moveToFirst()) {  displayName = cursor.getString(cursor.getColumnIndex(OpenableColumns.*DISPLAY\_NAME*));  }   InputStream is = contentResolver.openInputStream(uri);  File path = FilePathProvider.*getPublicDir*(context, FilePathProvider.*DIR\_ENCLOSURE* +  "/" + DateTimeUtils.*formatDatetime*(new Date(),*FORMAT\_DATE3*));  if (!path.exists()){  path.mkdir();  }  File cache = new File(path.getAbsolutePath(), displayName);  FileOutputStream fos = new FileOutputStream(cache);  *copyFileStream*(is,fos);  file = cache;  }  return file; }  */\*\*  \* 复制文件流  \** ***@param*** *inputStream 输入流  \** ***@param*** *outputStream 输出流  \** ***@throws*** *IOException IO异常  \*/* private static void copyFileStream(InputStream inputStream , OutputStream outputStream) throws IOException{  int byteread = 0;  byte[] buffer = new byte[1024];  while ( (byteread = inputStream.read(buffer)) != -1) {  outputStream.write(buffer, 0, byteread);  }  inputStream.close();  outputStream.close(); } |