**一14一 一14一 一12一 一11一** 一10.5一 一10.5一 '一10一"

**目 录**

[1. Adapter 5](#_Toc32465)

[1.1. ArrayAdapter 5](#_Toc24231)

[1.2. SimpleAdapter 5](#_Toc11508)

[1.3. BaseAdapter 6](#_Toc13836)

[1.4. notifyDataChanged的原理 7](#_Toc18041)

[2. LayoutInflater 10](#_Toc15105)

[2.1. 获得 LayoutInflater 实例的三种方式: 10](#_Toc12587)

[2.2. inflate的三种情况： 11](#_Toc17370)

[3. RadioGroup、RadioButton、ToggleButton、CheckBox 11](#_Toc6905)

[3.1. 基本用法和状态改变监听 11](#_Toc23559)

[3.2. 状态改变监听2 13](#_Toc10700)

[3.3. 自定义图标 13](#_Toc8483)

[3.3.1. 使用button属性自定义CheckBox的图标并通过paddingLeft属性设置图标和文本之间的间隔 13](#_Toc7900)

[3.3.2. 通过drawableLeft属性自定义的图标 14](#_Toc10598)

[4. Switch控件 14](#_Toc12766)

[5. Button调用自己的点击事件 14](#_Toc18079)

[6. Xml里设置点击事件 14](#_Toc19082)

[7. 剪切板 15](#_Toc21040)

[8. TextView&EditText 16](#_Toc13116)

[8.1. 设置进入Activity时不弹出默认不弹出软件键盘 16](#_Toc20885)

[8.2. Edit不可编辑但是可以点击 17](#_Toc30146)

[8.3. Edit可以通过setText编辑但是不可以弹虚拟键盘 17](#_Toc129)

[8.4. Edit移动光标位置 17](#_Toc23456)

[8.5. addTextChangedListener 17](#_Toc29854)

[8.5.1. 函数 17](#_Toc5444)

[8.5.1.1. beforeTextChanged 17](#_Toc28771)

[8.5.1.2. onTextChanged 17](#_Toc26625)

[8.5.1.3. afterTextChanged 18](#_Toc26027)

[8.5.2. 注意事项 18](#_Toc206)

[8.6. InputFilter过滤 18](#_Toc28143)

[8.6.1. 函数 18](#_Toc14879)

[8.6.2. 注意事项 18](#_Toc27977)

[8.7. InputFilter限制指定金额 19](#_Toc21834)

[8.8. 监听文字改变，匹配中文 20](#_Toc32437)

[8.9. 实现显示和隐藏密码 21](#_Toc27740)

[8.10. Edit光标和变化的下划线 21](#_Toc31524)

[8.11. 设置单行 22](#_Toc18246)

[8.12. 禁止回车换行 22](#_Toc14942)

[8.13. 禁止使用emoji（并加上字数限制） 22](#_Toc22477)

[8.14. 限制小数位数 23](#_Toc25546)

[8.15. 添加下划线 24](#_Toc31140)

[8.15.1. 如果是在资源文件里： 24](#_Toc5776)

[8.15.2. 如果是代码里： 24](#_Toc16529)

[8.16. 解决输入框被键盘压住一半的问题 25](#_Toc11186)

[8.16.1. 原理 25](#_Toc11411)

[8.16.2. layer\_chat\_edt.xml 25](#_Toc10770)

[8.16.3. layer\_chat\_btn.xml 26](#_Toc7265)

[8.16.4. activity\_used.xml 26](#_Toc893)

[8.17. 空格、换行 27](#_Toc22718)

[8.17.1. 空格 27](#_Toc11057)

[8.17.2. 换行和回车 27](#_Toc10089)

[8.17.2.1. 概念和由来 27](#_Toc22907)

[8.17.2.2. Android中如何使用 28](#_Toc2755)

[8.18. 其他属性 28](#_Toc11618)

[8.18.1. drawableLeft和drawablePadding 28](#_Toc15190)

[8.18.2. textSize设置文字大小 28](#_Toc9329)

[8.18.3. textCursorDrawable光标颜色 29](#_Toc4838)

[8.18.4. digits实现只能输入特定的字符 29](#_Toc25128)

[8.18.5. inputType设置默认输入中文、英文、数字 29](#_Toc7345)

[8.18.6. ellipsize设置当文字过长时,该控件该如何显示 30](#_Toc19354)

[8.18.7. Ems设置控件一行的容纳字符数 30](#_Toc23815)

[8.18.8. maxLength文本字数控制 30](#_Toc24888)

[8.18.9. textAppearance字体外观 30](#_Toc12907)

[8.18.10. textColorHighlight被选中文字的底色 30](#_Toc20459)

[8.18.11. textColorHint提示信息文字的颜色 30](#_Toc1233)

[8.18.12. textColorLink文字链接的颜色 30](#_Toc15324)

[8.18.13. textScaleX文字之间间隔 30](#_Toc23481)

[8.18.14. lineSpacing行间距 30](#_Toc6811)

[8.18.15. textStyle设置字形 31](#_Toc31285)

[8.18.16. typeface设置字体 31](#_Toc24469)

[8.19. 有时候会诡异换行 31](#_Toc22562)

[9. 虚拟键盘相关 31](#_Toc5347)

[9.1. 打开和关闭输入法 31](#_Toc18966)

[9.1.1. 打开输入法： 31](#_Toc3076)

[9.1.2. 关闭输入法： 32](#_Toc30165)

[9.2. 设置不调用系统键盘 32](#_Toc27907)

[9.3. 监听输入法中的按钮（比如回车） 32](#_Toc24390)

[9.4. 改变回车按钮的显示内容（配合监听回车按钮使用） 33](#_Toc24586)

[9.5. 进入activity时候让Edittext的时候，使其失去焦点 33](#_Toc31297)

[10. SpannableString 34](#_Toc11397)

[10.1. Spannable 34](#_Toc20402)

[10.2. ImageSpan 34](#_Toc16943)

[10.3. ClickableSpan 35](#_Toc30944)

[10.4. 其他超链接效果实现方式 35](#_Toc965)

[10.5. 网址跳转 35](#_Toc2521)

[11. Spinner下拉列表 36](#_Toc10304)

[12. AutoCompleteTextView自动输入 37](#_Toc3155)

[13. ProgressBar 37](#_Toc28337)

[14. ViewPager 37](#_Toc30339)

[14.1. 代码 37](#_Toc572)

[14.2. 使用开源导航条Indicator 38](#_Toc9652)

[14.3. notifyDataSetChanged不管用的解决办法 39](#_Toc5838)

[14.4. ViewPager和Fragment结合使用onResume问题 39](#_Toc25263)

[14.5. 不可滑动ViewPager 40](#_Toc19570)

[15. ListView小技巧 41](#_Toc26152)

[15.1. 隐藏分割线和滚动条 41](#_Toc5756)

[15.2. 回弹 41](#_Toc17869)

[15.3. 添加headerView和footerView 41](#_Toc3299)

[15.4. 自动显示、隐藏布局的ListView 42](#_Toc14003)

[15.5. 改变listView的选中颜色？ 43](#_Toc378)

[15.6. 聊天ListView（多布局） 43](#_Toc14381)

[15.7. ListView异步加载图片乱序问题，原因分析及解决方案 45](#_Toc16365)

[15.8. Holder重用有时会导致问题（需要时刻注意visible） 46](#_Toc4689)

[15.9. 设置listview需要显示在第几项 46](#_Toc22188)

[15.10. 屏蔽ListView的touch事件，让外层容器获取touch事件 47](#_Toc11704)

[15.11. 在tv项目中，禁止listview的item选中 47](#_Toc30763)

[15.12. 让item响应，而不是item中的button响应 47](#_Toc13890)

[15.13. 从底部开始展示 47](#_Toc16156)

[15.13.1. stackFromBottom 47](#_Toc18654)

[15.13.2. transcriptMode 47](#_Toc13719)

[16. ListView源码解析 48](#_Toc11429)

[16.1. Listview继承结构 48](#_Toc15936)

[16.2. Adapter的作用 48](#_Toc22991)

[16.3. RecycleBin机制 49](#_Toc14321)

[16.4. 第一次Layout 51](#_Toc25928)

[16.5. 第二次Layout 54](#_Toc14203)

[16.6. 滑动加载更多数据 55](#_Toc24046)

[17. List、Grid和ExpandableListView 57](#_Toc20581)

[17.1. ListView定位到某一个item上 57](#_Toc20671)

[17.2. GridView属性 57](#_Toc27877)

[17.3. Item里Buttom会抢占焦点，导致item不能点击的问题解决 57](#_Toc4994)

[17.4. 有时候Item里图片等view很多，滚动时会很卡，解决方法： 57](#_Toc30312)

[17.5. ExpandableListView 57](#_Toc12723)

[18. 可拖动排序的GridView 59](#_Toc23375)

[18.1. 引入拖动排序控件 59](#_Toc30127)

[18.2. fragment\_main\_seek\_watch.xml 59](#_Toc9187)

[18.3. item\_main\_seek\_watch.xml 59](#_Toc6269)

[18.4. GridSeekWatchAdapter 60](#_Toc6018)

[18.5. MainSeekWatchFragment 61](#_Toc13478)

[19. RecycleView 65](#_Toc15507)

[19.1. 基本使用 65](#_Toc15603)

[19.1.1. 对比ListView 65](#_Toc4296)

[19.1.2. 最简单的（连分割线都没有） 65](#_Toc32280)

[19.1.3. 设置分割线ItemDecoration 66](#_Toc1199)

[19.1.4. 设置布局管理器LayoutManager 68](#_Toc10882)

[19.1.4.1. GridLayoutManager需要适配自定义DividerGridItemDecoration 68](#_Toc23726)

[19.1.4.2. StaggeredGridLayoutManager（不但有Grid的竖向还有横向） 71](#_Toc10736)

[19.1.5. 设置动画ItemAnimator 71](#_Toc27340)

[19.1.6. 单击和长按事件Click and LongClick 71](#_Toc29310)

[19.2. 监听滚动事件 72](#_Toc28679)

[19.2.1. 监听是否滚动到了最前面 72](#_Toc8001)

[19.2.2. 监听是否滚动到了最后面 73](#_Toc10495)

[19.3. 公用Adapter的封装 73](#_Toc22709)

[19.4. MultiTypeAdapter库 77](#_Toc26932)

[19.4.1. 引用 77](#_Toc1405)

[19.4.2. Java代码 78](#_Toc321)

[19.4.3. Kotlin代码 79](#_Toc11850)

[19.5. RecyclerView常见问题解决方案 80](#_Toc20387)

[19.5.1. 异步加载图片错乱的问题 80](#_Toc5609)

[19.5.2. 无法引用报错 80](#_Toc6423)

[19.5.3. 嵌套自动滚动 80](#_Toc20161)

[19.5.4. 高度设置wrap\_content 无作用的bug，在listview中这么设置会不显示 81](#_Toc27798)

[19.5.5. Item布局宽度设置match\_parent无效果的问题解决 81](#_Toc8525)

[19.6. 整个RecycleView不响应点击事件 81](#_Toc14450)

[19.7. 只点击整个RecyclerView不点击Item 81](#_Toc29526)

[19.8. 九宫格居中平分间距 82](#_Toc8486)

[19.8.1. 分析 82](#_Toc7664)

[19.8.2. item\_dynamic\_img\_nine.xml 83](#_Toc28627)

[19.8.3. DynamicSpaceDecoration.java 83](#_Toc20627)

[19.8.4. 使用 84](#_Toc23943)

[20. RecylcerView 面试十四问 84](#_Toc20931)

[20.1. Adapter相关 84](#_Toc27060)

[20.1.1. Adapter的作用是什么？ 84](#_Toc12833)

[20.1.2. 几个方法是做什么用的？ 84](#_Toc7565)

[20.1.3. 如何理解adapter订阅者模式？ 84](#_Toc27869)

[20.1.4. requestLayout方法 86](#_Toc27428)

[20.1.5. invalidate方法 87](#_Toc2271)

[20.2. ViewHolder相关 87](#_Toc3695)

[20.2.1. ViewHolder作用大概有这些： 87](#_Toc10279)

[20.2.2. 如何理解ViewHolder的复用? 87](#_Toc18353)

[20.2.3. ViewHolder中为何使用SparseArray替代HashMap存储viewId？ 88](#_Toc7985)

[20.2.3.1. HashMap 88](#_Toc16844)

[20.2.3.2. SparseArray 88](#_Toc15449)

[20.3. LayoutManager相关 89](#_Toc19594)

[20.3.1. LayoutManager作用是什么？ 89](#_Toc26523)

[20.3.2. LayoutManager样式有哪些？ 89](#_Toc4587)

[20.3.3. setLayoutManager(LayoutManager layout)源码里做了什么？ 89](#_Toc17701)

[20.4. SpanSizeLookup相关 90](#_Toc26341)

[20.4.1. SpanSizeLookup的作用是干什么的？ 90](#_Toc6542)

[20.4.2. SpanSizeLookup如何使用？ 90](#_Toc6983)

[20.5. ItemDecoration相关 90](#_Toc25777)

[20.5.1. ItemDecoration的用途是什么？ 90](#_Toc2463)

[20.5.2. 自定义ItemDecoration有哪些重写方法？ 90](#_Toc4225)

[20.5.3. 分析一下addItemDecoration()源码？ 91](#_Toc28491)

[20.5.4. 如何实现可以设置自定义分割线 91](#_Toc15179)

[20.6. 上拉加载更多相关 92](#_Toc5666)

[20.6.1. 上拉加载更多的功能是如何做的？ 92](#_Toc8516)

[20.6.2. 网格布局上拉加载如何优化？ 92](#_Toc25690)

[20.7. 绘制原理相关 93](#_Toc1881)

[20.7.1. RecyclerView绘制原理如何理解？ 93](#_Toc19596)

[20.7.2. 性能优化本质是什么？ 93](#_Toc22929)

[20.7.3. RecyclerView绘制原理过程大概是怎样的？ 93](#_Toc10002)

[20.8. Recycler相关 93](#_Toc398)

[20.8.1. Recyler是如何实现ViewHolder的缓存？ 93](#_Toc1806)

[20.8.2. 如何理解recyclerView三级缓存是如何实现的？ 94](#_Toc24037)

[20.8.2.1. 一级缓存：返回布局和内容都都有效的ViewHolder 94](#_Toc28882)

[20.8.2.2. 二级缓存：返回View 94](#_Toc18788)

[20.8.2.3. 三级缓存：返回布局有效，内容无效的ViewHolder 94](#_Toc17553)

[20.8.3. 屏幕滑动状态 95](#_Toc22925)

[20.8.3.1. 屏幕滑动(状态是item状态可见，不可见，即将可见变化)时三级缓存是如何理解的？ 95](#_Toc1913)

[20.8.3.2. 出屏幕时候的情况 96](#_Toc30150)

[20.8.3.3. 进屏幕时候的情况 96](#_Toc18080)

[20.8.3.4. 屏幕往下拉ViewHolder（position=1）进入屏幕的情况 96](#_Toc13572)

[20.9. SnapHelper相关 96](#_Toc27249)

[20.9.1. 主要是做什么用的？ 96](#_Toc7692)

[20.9.2. 什么是Fling操作 96](#_Toc18050)

[20.9.3. 有哪些重要的方法，其作用就是是什么？ 96](#_Toc1660)

[20.9.4. LinearSnapHelper类分析 97](#_Toc20717)

[20.9.4.1. LinearSnapHelper如何使用 97](#_Toc28461)

[20.9.4.2. out[0]和out[1]分别指什么 97](#_Toc1446)

[20.9.4.3. LinearSnapHelper代码中calculateDistanceToFinalSnap作用是什么？ 97](#_Toc28859)

[20.9.4.4. LinearSnapHelper中是如何实现滚动停止的？ 98](#_Toc23918)

[20.9.5. PagerSnapHelper类分析 99](#_Toc8496)

[20.10. 如何实现复杂type首页需求 99](#_Toc15685)

[20.10.1. 如何实现 99](#_Toc25640)

[20.10.2. 如果不封装会出现什么问题和弊端? 100](#_Toc10892)

[20.11. item条目点击事件应在onCreateViewHolder中写 101](#_Toc24804)

[20.12. RecyclerView滑动卡顿 101](#_Toc12610)

[20.12.1. RecyclerView滑动卡顿原因有哪些？ 101](#_Toc30246)

[20.12.2. 如何解决RecyclerView实现画廊卡顿？ 101](#_Toc5363)

[20.12.3. 耗时的操作导致卡顿 102](#_Toc465)

[20.13. RecyclerView常见的优化有哪些 102](#_Toc27221)

[20.13.1. DiffUtil刷新优化 102](#_Toc17056)

[20.13.2. 布局优化 102](#_Toc26284)

[20.13.3. 对itemView中孩子View的点击事件优化 102](#_Toc3182)

[20.13.4. 其他的一些优化点 102](#_Toc23400)

[20.14. 如何处理ViewPager嵌套水平RecyclerView横向滑动到底后不滑动ViewPager？ 103](#_Toc3045)

[21. ScrollView 103](#_Toc18334)

[21.1. scrollTo(),scrollBy(),getScrollX(), getScrollY() 103](#_Toc31893)

[21.2. 判断是否滑到顶部和底部 104](#_Toc30273)

[21.3. 判断滑动位置的地方，可以有两种方式： 105](#_Toc3255)

[21.4. scrollview嵌套viewpager或listview引起滑动冲突的解决方案 106](#_Toc5972)

[21.5. ScrollView自己的方法移动到顶部、底部 106](#_Toc17459)

[21.6. Android版本在23以前，监听ScrollView滚动事件 106](#_Toc29538)

[21.7. 删除边界阴影 107](#_Toc12446)

[21.8. Scrollview、Rv、Lv删除拉到尽头出现的阴影效果 107](#_Toc5120)

[22. ScrollView嵌套ListView 107](#_Toc26180)

[22.1.1. 重写Listview 107](#_Toc20781)

[22.1.2. 固定高度，获得滚动权限 107](#_Toc9859)

[22.1.3. 计算出其高度，然后根据需求设定 108](#_Toc29376)

[23. ScrollView嵌套GridView 109](#_Toc15871)

[24. ScrollView和SmartRefreshLayout嵌套多个RecyclerView 110](#_Toc20044)

[24.1. ScrollView内嵌有多个RecyclerView 111](#_Toc377)

[24.2. SmartRefreshLayout内嵌有多个RecyclerView 112](#_Toc19745)

[25. TableLayout和TableRow的使用 112](#_Toc17997)

[26. CountDownTimer定时器控件 113](#_Toc25424)

[27. Timer类 114](#_Toc23518)

[28. Android闹钟设置的解决方案（Alarm） 114](#_Toc2018)

[28.1. 主要问题 115](#_Toc30080)

[28.2. API19以上AlarmManager机制的修改 115](#_Toc32664)

[28.3. 应用程序被Kill掉时的处理 115](#_Toc12451)

[28.4. 5.0以上的JobScheduler 116](#_Toc5882)

[28.5. 6.0以上Doze模式的处理 117](#_Toc322)

[28.6. 补充 118](#_Toc3445)

[29. 日期、时间、时间戳转换 119](#_Toc24296)

[30. 日期选择器 121](#_Toc17115)

[30.1. 常用方法 124](#_Toc7883)

[31. 时间显示控件TextClock 125](#_Toc395)

[31.1. 注意 125](#_Toc3020)

[31.2. XML中 125](#_Toc21241)

[31.3. 代码中 125](#_Toc4952)

[32. 传感器Sensor 125](#_Toc31099)

[33. Dialog 128](#_Toc22147)

[33.1. ProgressDialog 128](#_Toc16774)

[33.1.1. ProgressDialog和AlertDialog必须先show才能setContentView 129](#_Toc10456)

[33.2. dismiss和cancel的区别 129](#_Toc11711)

[33.3. Dialog遇到横竖屏切换 130](#_Toc4777)

[33.3.1. 基于Dialog实现的对话框 130](#_Toc23680)

[33.3.2. 基于DialogFragment的对话框 131](#_Toc23272)

[33.4. 使dialog居于所有的dialog最上层 132](#_Toc15061)

[33.5. setCancelable和setCanceledOnTouchOutside使用顺序 132](#_Toc31957)

[33.6. 自定义Style实现不同的ui需求 133](#_Toc30702)

[33.7. 封装工具类 134](#_Toc20481)

[34. popupWindow 136](#_Toc1439)

[34.1. 设置基本的view 136](#_Toc7538)

[34.2. 设置动画效果 137](#_Toc5479)

[34.3. 放置位置 138](#_Toc29424)

[34.4. 某个特定位置显隐动画 138](#_Toc16138)

[34.5. 7.0以上适配 139](#_Toc19914)

[34.6. BasePopup库 140](#_Toc9427)

[34.6.1. 使用 140](#_Toc16519)

[34.6.2. 平移动画 141](#_Toc19182)

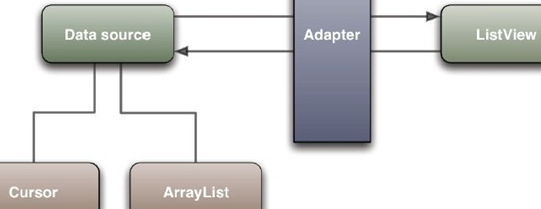
[34.6.2.1. 工具类 141](#_Toc4964)

[34.6.2.2. 自定义PopUpwindow重写show方法 143](#_Toc17014)

[34.6.2.3. 快速集成 144](#_Toc7571)

# Adapter

Adapter是连接后端数据和前端显示的适配器接口，是数据和UI（View）之间一个重要的纽带。在常见的View(ListView,GridView)等地方都需要用到Adapter。如下图直观的表达了Data、Adapter、View三者的关系：



BaseAdapter是一个抽象类，继承它需要实现较多的方法，所以也就具有较高的灵活性；

ArrayAdapter支持泛型操作，最为简单，只能展示一行字。

SimpleAdapter有最好的扩充性，可以自定义出各种效果。

## ArrayAdapter

ArrayAdapter(Context context, int textViewResourceId, List<T> objects)来装配数据；三个参数，依次为this,布局文件，数据源(一个List集合)。

**public** **void** **onCreate**(**Bundle** savedInstanceState){

**super**.onCreate(savedInstanceState);

listView = **new** ListView(**this**);

listView.setAdapter(**new** ArrayAdapter<**String**>(**this**,android.**R**.**layout**.**simple\_expandable\_list\_item\_1**,getData()));

setContentView(listView);

}

**private** List<**String**> **getData**(){

List<**String**> data = **new** ArrayList<**String**>();

data.add("测试数据1");

data.add("测试数据2");

data.add("测试数据3");

data.add("测试数据4");

**return** data;

## SimpleAdapter

SimpleAdapter(context, data, resource, from, to)五个参数，依次为1.this,2.List<Map<String, Object>>，3.布局文件，4.列的名称 5.列的值所要绑定的view



## BaseAdapter

系统要绘制ListView了，他首先获得要绘制的这个列表的长度，然后开始绘制第一行，怎么绘制呢？调用getView()函数。在这个函数里面首先获得一个View（实际上是一个ViewGroup），然后再实例并设置各个组件，显示之。好了，绘制完这一行了。那再绘制下一行，直到绘完为止。

**public** **class** **MyAdapter** **extends** **BaseAdapter**{

**private** List<**Person**> list;

**private** **Context** mContext;

**public** **MyAdapter**(List<**Person**> list, **Context** context) {

**this**.list = list;

mContext = context;

}

@Override

**public** **int** **getCount**() {

**return** list.size();

}

@Override

**public** **Object** **getItem**(**int** position) {

**return** list.get(position);

}

@Override

**public** **long** **getItemId**(**int** position) {

**return** position;

}

@Override

**public** **View** **getView**(**int** position, **View** convertView, **ViewGroup** parent) {

**ViewHolder** holder = **null**;

**if** (convertView == **null**) {

holder = **new** ViewHolder();

convertView = **LayoutInflater**.*from*(mContext).inflate(**R**.**layout**.**item**, **null**);

holder.tv = (**TextView**) convertView.findViewById(**R**.**id**.**tv**);

holder.img = (**ImageView**) convertView.findViewById(**R**.**id**.**img**);

convertView.setTag(holder);

}**else** {

holder = (**ViewHolder**) convertView.getTag();

}

holder.tv.setText(list.get(position).getName());

holder.img.setBackgroundResource(list.get(position).getHeader());

**return** convertView;

}

**class** **ViewHolder**{

**TextView** tv;

**ImageView** img;

}

}

## notifyDataChanged的原理

notifyDataSetChanged是Adater的一个方法，主要用来通知ListView，告诉它Adapter的数据发生了变化，需要更新ListView的显示，所以当Adapter的数据内容改变时会调用notifyDataSetChanged()方法。 直接看看BaseAdapter中notifyDataSetChanged的源码实现，看notifyDataSetChanged是如何工作的

public abstract class BaseAdapter implements ListAdapter, SpinnerAdapter {

private final DataSetObservable mDataSetObservable = new DataSetObservable();

public void notifyDataSetChanged() {

mDataSetObservable.notifyChanged();

}

}

可以看到它调用的是DataSetObservable中的notifyChanged，进入DataSetObservable查看实现

public class DataSetObservable extends Observable<DataSetObserver> {

public void notifyChanged() {

synchronized(mObservers) {

for (int i = mObservers.size() - 1; i >= 0; i--) {

mObservers.get(i).onChanged();

}

}

}

}

其中mObervers是在DataSetObservable的父类Observable中定义：

public abstract class Observable<T> {

protected final ArrayList<T> mObservers = new ArrayList<T>();

}

它是一个DataSetObserver类型的ArrayList,最终执行的是DataSetObserver的onChange函数。再来看看DataSetOberver类的源代码：

public abstract class DataSetObserver {

public void onChanged() {

// Do nothing

}

public void onInvalidated() {

// Do nothing

}

}

从上面的过程可以大致看出，如果需要得到ListView更新的通知，首先实现一个DataSetObserver类，重写其中的onChanged回调方法，然后把这个对象添加（注册）到ArrayList中，这样当我们调用notifyDataSetChanged的时候，它会遍历这个ArrayList取出DataSetObserver对象,回调onChanged方法。也就是说我们最终的刷新ListViewd的工作应该在这个onChanged方法中。

那么疑问就是系统在哪个地方实现了实现一个DataSetObserver类，重写其中的onChanged回调方法，然后把这个对象添加到ArrayList当中的。

其实这个工作在setAdapter中完成的，当为ListView设置一个Adapter的时候，就在这个Adapter中注册了一个回调监听，也就是上面说的实现一个DataSetObserver类，重写其中的onChanged回调方法，然后把这个对象添加到ArrayList当中，当Adapter调用notifyDataSetChanged的时候，就会回调onChanged函数，在onChanged里面进行ListView的更新，这样ListView就进行更新操作。

那么我们来看看ListAdapter中setAdapter的实现：

@Override

public void setAdapter(ListAdapter adapter) {

//这里判断是否已经注册了监听

//如果已经注册，则取消注册

//如果重复调用setAdapter，下面的代码就会执行mDataSetObserver从ArrayList中移除

if (mAdapter != null && mDataSetObserver != null) {

mAdapter.unregisterDataSetObserver(mDataSetObserver);

}

resetList();

mRecycler.clear();

if (mHeaderViewInfos.size() > 0|| mFooterViewInfos.size() > 0) {

mAdapter = new HeaderViewListAdapter(mHeaderViewInfos, mFooterViewInfos, adapter);

} else {

mAdapter = adapter;

}

mOldSelectedPosition = INVALID\_POSITION;

mOldSelectedRowId = INVALID\_ROW\_ID;

// AbsListView#setAdapter will update choice mode states.

super.setAdapter(adapter);

if (mAdapter != null) {

mAreAllItemsSelectable = mAdapter.areAllItemsEnabled();

mOldItemCount = mItemCount;

//得到数据的数量

mItemCount = mAdapter.getCount();

checkFocus();

//看到没有这里定义了一个AdapterDataSetObserver，它就是DataSetObserver的实现类

mDataSetObserver = new AdapterDataSetObserver();

//看到这里应该明白了，这里将这个DataSetObserver实现了类对象添加到ArrayList中，这样就可以回调了

mAdapter.registerDataSetObserver(mDataSetObserver);

mRecycler.setViewTypeCount(mAdapter.getViewTypeCount());

int position;

if (mStackFromBottom) {

position = lookForSelectablePosition(mItemCount - 1, false);

} else {

position = lookForSelectablePosition(0, true);

}

setSelectedPositionInt(position);

setNextSelectedPositionInt(position);

if (mItemCount == 0) {

// Nothing selected

checkSelectionChanged();

}

} else {

mAreAllItemsSelectable = true;

checkFocus();

// Nothing selected

checkSelectionChanged();

}

//会导致调用measure()过程 和 layout()过程

requestLayout();

}

具体的解释直接看代码的注释，这样更方便理解，主要就是mDataSetObserver = new AdapterDataSetObserver()和mAdapter.registerDataSetObserver(mDataSetObserver)这两句，我们上面已经注释了AdapterDataSetObserver就是DataSetObserver的实现类，它重写了onChanged方法。

看AdapterDataSetObserver源代码，在这里就可以看到，notifyDatasetChanged的最终执行的操作是什么了，因为它最终回调了这个onChanged方法。AdapterDataSetObserver是AbsListView的一个内部类

class AdapterDataSetObserver extends AdapterView<ListAdapter>.AdapterDataSetObserver {

@Override

public void onChanged() {

//这里是核心操作

super.onChanged();

if (mFastScroller != null) {

mFastScroller.onSectionsChanged();

}

}

@Override

public void onInvalidated() {

super.onInvalidated();

if (mFastScroller != null) {

mFastScroller.onSectionsChanged();

}

}

}

最终回调的就是这里的onChanged函数，直接看代码，核心操作在super.onChanged()里面。

super.onChanged意思就是执行AdapterView.AdapterDataSetObserver里面的onChanged函数，它是ApdaterView的一个内部类。

class AdapterDataSetObserver extends DataSetObserver {

private Parcelable mInstanceState = null;

@Override

public void onChanged() {

mDataChanged = true;

mOldItemCount = mItemCount;

mItemCount = getAdapter().getCount();

// Detect the case where a cursor that was previously invalidated has

// been repopulated with new data.

if (AdapterView.this.getAdapter().hasStableIds() && mInstanceState != null

&& mOldItemCount == 0 && mItemCount > 0) {

AdapterView.this.onRestoreInstanceState(mInstanceState);

mInstanceState = null;

} else {

rememberSyncState();

}

checkFocus();

//会导致调用measure()过程 和 layout()过程

requestLayout();

}

@Override

public void onInvalidated() {

mDataChanged = true;

if (AdapterView.this.getAdapter().hasStableIds()) {

// Remember the current state for the case where our hosting activity is being

// stopped and later restarted

mInstanceState = AdapterView.this.onSaveInstanceState();

}

// Data is invalid so we should reset our state

mOldItemCount = mItemCount;

mItemCount = 0;

mSelectedPosition = INVALID\_POSITION;

mSelectedRowId = INVALID\_ROW\_ID;

mNextSelectedPosition = INVALID\_POSITION;

mNextSelectedRowId = INVALID\_ROW\_ID;

mNeedSync = false;

checkFocus();

requestLayout();

}

public void clearSavedState() {

mInstanceState = null;

}

}

看到没有，这个类实现了DataSetObserver，正好说明了上面的说法，直接看它的onChanged函数。直接看最后一句requestLayout()，这里就会进行刷新了， 如果细心的话，你应该也会看到在setAdapter中也执行了这个函数，这样就充分说了，执行这个函数，这里是进行了布局和重绘。 这里我们可以知道一点，就是我们上面对ListView的刷新，本质就是调用了requestLayout方法。

# LayoutInflater

在实际开发中LayoutInflater这个类还是非常有用的，它的作用类似于findViewById()。不同点是LayoutInflater是用来找res/layout/下的xml布局文件，并且实例化；而findViewById()是找xml布局文件下的具体widget控件(如Button、TextView等)。 具体作用：

1、对于一个没有被载入或者想要动态载入的界面，都需要使用LayoutInflater.inflate()来载入；

2、对于一个已经载入的界面，就可以使用Activiyt.findViewById()方法来获得其中的界面元素。

## 获得 LayoutInflater 实例的三种方式:

**LayoutInflater** inflater = getLayoutInflater(); **//调用Activity的getLayoutInflater()**

**LayoutInflater** localinflater = (**LayoutInflater**)context.getSystemService(**Context**.**LAYOUT\_INFLATER\_SERVICE**);

**LayoutInflater** inflater2 = **LayoutInflater**.*from*(context);

## inflate的三种情况：

返回item的布局，外边的框架是怎样的完全不管

convertView = mInflater.inflate(**R**.**layout**.**list\_item**, **null**);

返回父布局的LayoutParamrs

convertView = mInflater.inflate(**R**.**layout**.**list\_item**, parent, **false**);

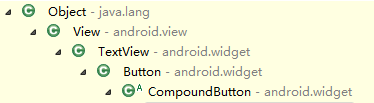
返回整个父布局

convertView = mInflater.inflate(**R**.**layout**.**list\_item**, parent, **true**);

# RadioGroup、RadioButton、ToggleButton、CheckBox

## 基本用法和状态改变监听

这三个控件都继承于CompoundButton，CompoundButton的继承结构如下



<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical" >

<CheckBox

android:id="@+id/cb"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="CB嘎嘎发" />

<RadioGroup

android:id="@+id/rg"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content" >

<RadioButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="男" />

<RadioButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="女" />

</RadioGroup>

<ToggleButton

android:id="@+id/tb"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:textOff="关"

android:textOn="开" />

</LinearLayout>

**public** **class** **MainActivity** **extends** **Activity** **implements** OnCheckedChangeListener{

**private** **CheckBox** cb;

**private** **RadioGroup** rg;

**private** **ToggleButton** tb;

**private** **Context** mContext;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

mContext = **MainActivity**.**this**;

initView();

initEvent();

}

**private** **void** **initView**() {

cb = (**CheckBox**) findViewById(**R**.**id**.**cb**);

rg = (**RadioGroup**) findViewById(**R**.**id**.**rg**);

tb = (**ToggleButton**) findViewById(**R**.**id**.**tb**);

}

**private** **void** **initEvent**() {

//RadioGroup、RadioButton监听

**int** count = rg.getChildCount();

**if** (count > 0) {

**for** (**int** i = 0; i < count; i++) {

**final** **RadioButton** rb = (**RadioButton**) rg.getChildAt(i);

rb.setOnCheckedChangeListener(**new** **CompoundButton**.OnCheckedChangeListener() {

@Override

**public** **void** **onCheckedChanged**(**CompoundButton** buttonView, **boolean** isChecked) {

**if** (isChecked) {

**Toast**.*makeText*(mContext, rb.getText().toString() + "---rb被选中了", **Toast**.**LENGTH\_SHORT**).show();

}

}

});

}

}

}

@Override

**public** **void** **onCheckedChanged**(**CompoundButton** buttonView, **boolean** isChecked) {

**switch** (buttonView.getId()) {

**case** **R**.**id**.**cb**: //CheckBox监听

**if** (isChecked) {

**Toast**.*makeText*(mContext, "cb被选中了", **Toast**.**LENGTH\_SHORT**).show();

}**else** {

**Toast**.*makeText*(mContext, "cb被取消了", **Toast**.**LENGTH\_SHORT**).show();

}

**break**;

**case** **R**.**id**.**tb**: //ToggleButton监听

**if** (isChecked) {

**Toast**.*makeText*(mContext, tb.getTextOn().toString() + "---tb被选中了", **Toast**.**LENGTH\_SHORT**).show();

}**else** {

**Toast**.*makeText*(mContext, tb.getTextOff().toString() + "---tb被取消了", **Toast**.**LENGTH\_SHORT**).show();

}

**break**;

**default**:

**break**;

}

}

}

## 状态改变监听2

public void setOnCheckedChangeListener(RadioGroup.OnCheckedChangeListener listener) {  
 mOnCheckedChangeListener = listener;  
}

public interface OnCheckedChangeListener {  
public void onCheckedChanged(RadioGroup group, @IdRes int checkedId);  
}

rg\_period\_container.setOnCheckedChangeListener **{** \_, checkedId **->** when (checkedId) {  
 R.id.*rb\_0* -> request.feeCycleType = 0  
 R.id.*rb\_1* -> request.feeCycleType = 1  
 R.id.*rb\_2* -> request.feeCycleType = 2  
 }  
**}**

## 自定义图标

### 使用button属性自定义CheckBox的图标并通过paddingLeft属性设置图标和文本之间的间隔

<?xml version="1.0" encoding="utf-8"?>  
<selector xmlns:android="http://schemas.android.com/apk/res/android">  
 <item android:drawable="@drawable/icon\_eye\_open" android:state\_checked="true" />  
 <item android:drawable="@drawable/icon\_eye\_close" android:state\_checked="false" />  
</selector>

<CheckBox

android:id="@+id/cb"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:button="@drawable/cb\_sel"

android:text="CB嘎嘎发" />

这种方法看上去还不错，万万没想到碰到一个诡异的bug,问题是这样的，图片遮住了文字刚开始以为是屏幕适配的问题，也按着这个思路去解决了。但是后来发现在4.4及以上的系统上没有问题（没那么多系统的手机，4.4及以上没发现问题,唯一有的低版本4.0和4.2都有这个bug）。要是通过paddingLeft属性处理的话在高版本的系统上又会显得间距较大。

后来又惊奇的发现如果继承的是Activity的话在低版本的手机上会出现同样的问题，如果继承的是v7包里的AppCompatActivity则问题没有再出现。是代码的bug么，问题应该是paddingLeft计算的起始位置不同，跪求解答啊。无奈由于一些原因我们代码只能继承Activity，so使用另一种方法吧。

### 通过drawableLeft属性自定义的图标

可以使用TextView的drawableLeft属性实现效果，同时可以使用drawablePadding设置图标和文本之间的距离。使用drawable属性需要将button和background的值设置为null

<ToggleButton

android:id="@+id/tb"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:background="@null"

android:button="@null"

android:drawablePadding="7dp"

android:drawableLeft="@drawable/cb\_sel"

android:textOff="关"

android:textOn="开" />

# Switch控件

设置监听

switch\_open.setOnCheckedChangeListener **{** \_, isChecked **->** if (isChecked) {  
 ToastUtils.showShort("勾选")  
 } else {  
 ToastUtils.showShort("未勾选")  
 }  
**}**

自定义样式

<!--使用方法： 在布局文件增加android:theme="@style/MySwitch",  
如果是SwitchCompat 控件，增加app:theme="@style/MySwitch"-->  
<style name="MySwitch" parent="Theme.AppCompat.Light">  
 <item name="colorControlActivated">@color/colorPrimary</item><!-- 打开时的拇指按钮的颜色 -->  
 <item name="android:colorForeground">@color/colorAccent</item><!-- 关闭时的轨迹的颜色-->  
</style>

# Button调用自己的点击事件

button.performClick();

performClick() 是使用代码主动去调用控件的点击事件（模拟人手去触摸控件）

# Xml里设置点击事件

我们知道在XML中可以添加一个onClick属性，值是一个字符串，然后在对应的activity中定义出这个方法就可以直接调用。比如：

<Button

android:id="@+id/button1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:onClick="onButtonClick"

android:text="Button" />

在相应的Activity中实现onButtonClick方法，就可以实现Button绑定点击监听事件，非常简单方便。

public void onButtonClick(View view){

Toast.makeText(MainActivity.this, view.getId()+"", Toast.LENGTH\_LONG).show();

}

我们现在来分析一下它的原理，自定义控件，道XML中的属性会解析成attributeSet来给view使用。

case R.styleable.View\_onClick:  
 //解析onClick的属性  
 if (context.isRestricted()) {  
 throw new IllegalStateException("The android:onClick attribute cannot " + "be used within a restricted context");  
 }  
 //得到设定的值，也就是方法名  
 final String handlerName = a.getString(attr);  
 if (handlerName != null) {  
 //如果不为空，给view设一个onClickListener  
 setOnClickListener(new OnClickListener() {  
 private Method mMethod;  
 public void onClick(View v) {  
 if (mMethod == null) {  
 try {

//这里是通过反射来取对应的方法，context其实就是activity, 方法名是handlerName,方法的参数是view  
 mMethod = getContext().getClass().getMethod(handlerName, View.class);  
 } catch (NoSuchMethodException e) {  
 int id = getId();  
 String idText = id == NO\_ID ? "" : " with id '" +

getContext().getResources().getResourceEntryName(id) + "'";  
 throw new IllegalStateException(

"Could not find a method " + handlerName + "(View) in the activity " + getContext().getClass() + " for onClick handler" + " on view " + View.this.getClass() + idText, e);  
 }  
 }  
 try {  
 //找到方法后就直接反射调用  
 mMethod.invoke(getContext(), View.this);  
 } catch (IllegalAccessException e) {  
 throw new IllegalStateException("Could not execute non " + "public method of the activity", e);  
 } catch (InvocationTargetException e) {  
 throw new IllegalStateException("Could not execute " + "method of the activity", e);  
 }  
 }  
 });  
 }  
break;

# 剪切板

* 复制文字到剪切板

import android.content.ClipData;  
import android.content.ClipboardManager;  
import android.content.Context;  
public class ClipboardUtil {  
 public static void putStringInClipboard(Context context, String text) {  
 //获取剪贴板管理器：  
 ClipboardManager cm = (ClipboardManager) context.getSystemService(Context.*CLIPBOARD\_SERVICE*);  
 // 创建普通字符型ClipData  
 ClipData mClipData = ClipData.*newPlainText*("Label", text);  
 // 将ClipData内容放到系统剪贴板里。  
 cm.setPrimaryClip(mClipData);  
 }  
}

* 如需自由复制TextView等控件的文字，只需要在该控件上加上这句就好了android:textIsSelectable="true"，或者java代码加 setTextIsSelectable(true)。长按控件自动弹出弹窗，效果如图：



* ClipData.newPlainText("Label", "Content")方法只在ClipData对象中创建了一个ClipData.Item对象，如果想向ClipData对象中添加多个Item应该通过ClipData对象的addItem()方法添加。
* 从剪贴板中获取ClipData数据：ClipboardManager.getPrimaryClip();

# TextView&EditText

## 设置进入Activity时不弹出默认不弹出软件键盘

在AndroidMainfest.xml中设置该activity的windowSoftInputMode属性为adjustUnspecified|stateHidden

<activity android:name=".Main"

android:windowSoftInputMode="adjustUnspecified|stateHidden" />

属性值的介绍：

"stateUnspecified"

这个是软件盘行为默认的设置。软键盘的状态(是否它是隐藏或可见)没有被指定。系统将选择一个合适的状态或依赖于主题的设置。

"stateUnchanged"

当主窗口出现在前面时，软键盘被保持它上次是什么状态，无论上次是可见或隐藏。

"stateHidden"

当用户选择该Activity时，软键盘被隐藏——也就是说，当用户确定导航到该Activity时，不管他离开的Activity的软键盘是可见还是隐藏都会被隐藏，不过当用户离开一个Activity而导致另一个被覆盖的Activity显示出来时，软键盘会使用默认的设置。

"stateAlwaysHidden"

当该Activity主窗口获取焦点时，软键盘总是被隐藏的，不管是确认导航到该Activity还是Activity被覆盖后显示出来。

"stateVisible"

当用户导航到Activity主窗口时，软键盘是可见的。不过当用户离开一个Activity而导致另一个被覆盖的Activity显示出来时，软键盘会使用默认的设置。

"stateAlwaysVisible"

当该Activity主窗口获取焦点时，软键盘总是显示的，不管是确认导航到该Activity还是Activity被覆盖后显示出来。

"adjustUnspecified"

这个是软件盘行为默认的设置。它不被指定是否该Activity主窗口调整大小以便留出软键盘的空间，或是否窗口上的内容得到屏幕上当前的焦点是可见的。系统将自动选择这些模式中一种主要依赖于是否窗口的内容有任何布局视图能够滚动他们的内容。如果有这样的一个视图，这个窗口将调整大小，这样的假设可以使滚动窗口的内容在一个较小的区域中可见的。

"adjustResize"

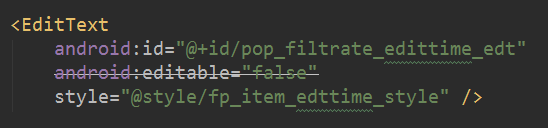
该Activity主窗口总是被调整屏幕的大小以便留出软键盘的空间

"adjustPan"

该Activity主窗口并不调整屏幕的大小以便留出软键盘的空间。相反，当前窗口的内容将自动移动以便当前焦点从不被键盘覆盖和用户能总是看到输入内容的部分。这个通常是不期望比调整大小，因为用户可能关闭软键盘以便获得与被覆盖内容的交互操作。

## Edit不可编辑但是可以点击

android:editable="false"



## Edit可以通过setText编辑但是不可以弹虚拟键盘

edt.setKeyListener(null);

注意, 这里不是setOnKeyListener, 而是setKeyListener. 此方法是TextView的成员, 调用后的效果完全符合预期, 并且获得焦点后不会弹出输入法.

## Edit移动光标位置

mEditText\_ps.setText(newStr);

mEditText\_ps.setSelection(newStr.length());

ditable editable = mEditText\_ps.getEditableText();

**Selection**.setSelection(editable, editable.length());

## addTextChangedListener

### 函数

TextWatcher是Android TextView文字改变监听接口，内部有以下几个函数：

* public void beforeTextChanged(CharSequence s, int start, int count, int after);
* public void onTextChanged(CharSequence s, int start, int before, int count);
* public void afterTextChanged(Editable s);

所有继承自TextView的类

textView.addTextChangedListener(new TextWatcher() {

@Override

public void onTextChanged(CharSequence s, int start, int before, int count) {}

@Override

public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

@Override

public void afterTextChanged(Editable s) {}

});

调用顺序：beforeTextChanged -> onTextChanged -> afterTextChanged

#### beforeTextChanged

* s 旧文本内容，即文本改变之前的内容
* start 将要修改文本的起始偏移量
* count 将要替换老文本的长度
* after 将要替换新文本的长度

#### onTextChanged

* s 新文本内容，即文本改变之后的内容
* start 被修改文本的起始偏移量
* before 被替换旧文本长度
* count 替换的新文本长度

#### afterTextChanged

* s 修改后的内容

### 注意事项

* 在回调方法中获取的新字符串虽然已经设置到TextView中,但是还未在UI界面中更新.
* 有2种情况会触发TextWatcher的3个回调方法,setText()和EditText键盘输入
* 如果在回调方法中调用setText(),会进入无限循环,需要增加判断条件
* afterTextChanged中去改变Editable s的值会触发TextWatcher的3个回调方法,会进入无限循环,需要增加判断条件
* Editable s和CharSequence s获取的数据为去除空格和回车之后字符串.

## InputFilter过滤

### 函数

InputFilter是系统提供的一个接口,是一个过滤器,修改传入的字符串,返回值类型为CharSequence.

public class mInputFilter implements InputFilter {

@Override

public CharSequence filter(CharSequence source, int start, int end,

Spanned dest, int dstart, int dend) {

return null;

}

}

* CharSequence source 用来替换的新的字符串,包括键盘输入/setText()/粘贴的内容
* int start source的起始位置
* int end source的结束位置
* Spanned dest 修改前的字符串,setText()时dest为EMPTY\_SPANNED,添加或修改时为输入框内的文本内容
* int dstart dest选中范围的起始位置
* int dend dest选中范围的结束位置

### 注意事项

* setFilters方法为替换过程,如果直接添加我们自定义的过滤器会导致原本textview自带的过滤方法失效,比如设置了maxLength属性会无效.具体方法如下

//获取到原来的InputFilter数组

textView.getFilters();

... //将获取到的InputFilter数组元素加入新建的数组中

//设置新的

textView.setFilters(inputFilters);

* 过滤操作在beforeTextChanged回调方法之前进行
* 过滤操作是对InputFilter数组的元素进行遍历,需要考虑元素顺序对过滤的影响

int n = mFilters.length;

for (int i = 0; i < n; i++) {

CharSequence out = mFilters[i].filter(text, 0, text.length(),

EMPTY\_SPANNED, 0, 0);

if (out != null) {

text = out;

}

}

* 根据源码得知,当返回值为null时,不对输入数据进行任何过滤操作,数据不变

## InputFilter限制指定金额

/\* 重写输入框类型 \*/

**private** InputFilter mInputFilter = **new** InputFilter() {

**public** **static** **final** **int** **MAX\_VALUE** = 10000; //最大金额

**public** **static** **final** **int** **PONTINT\_LENGTH** = 2; //小数位数

**Pattern** p = **Pattern**.*compile*("[0-9]\*"); //除数字外的其他的字符串

/\*\*

\* arg0 新输入的字符串

\* arg1 新输入的字符串起始下标，一般为0

\* arg2 新输入的字符串终点下标，一般为arg0长度-1

\* arg3 输入之前文本框内容

\* arg4 原内容起始坐标，一般为0

\* arg5 原内容终点坐标，一般为arg3长度-1

\*/

@Override

**public** CharSequence **filter**(CharSequence arg0, **int** arg1, **int** arg2,

Spanned arg3, **int** arg4, **int** arg5) {

//新字符串

**String** inputtext = arg0.toString().trim();

//之前的字符串

**String** oldtext = arg3.toString().trim();

//验证删除等按键

**if**("".equals(inputtext)){

**return** **null**;

}

//判断第一次输入0第二次又输入0-9

**Matcher** in = p.matcher(inputtext);

**if**(oldtext.equals("0") && in.matches() && !inputtext.equals(".")){

**return** "";

}

//判断是否第一次直接输入小数点

**if**(inputtext.equals(".") && oldtext.equals("")){

**return** "0" + inputtext;

}

//验证非数字或者小数点的情况

**Matcher** m = p.matcher(arg0);

**if**(oldtext.contains(".")){

//已经存在小数点的情况下，只能输入数字

**if**(!m.matches()){

**return** **null**;

}

}**else**{

//未输入小数点的情况下，可以输入小数点和数字

**if**(!m.matches() && !arg0.equals(".") ){

**return** **null**;

}

}

//验证输入金额的大小

**if**(!arg0.toString().equals("")){

**double** dold = **Double**.*parseDouble*(oldtext+inputtext);

**if**(dold > **MAX\_VALUE**){

**Toast**.*makeText*(mContext, "输入的消费金额不能大于"+**MAX\_VALUE**, **Toast**.**LENGTH\_SHORT**).show();

**return** arg3.subSequence(arg4, arg5);

}**else** **if**(dold == **MAX\_VALUE**){

**if**(inputtext.equals(".")){

**Toast**.*makeText*(mContext, "输入的消费金额不能大于"+**MAX\_VALUE**, **Toast**.**LENGTH\_SHORT**).show();

**return** arg3.subSequence(arg4, arg5);

}

}

}

//验证小数位精度是否正确

**if**(oldtext.contains(".")){

**int** index = oldtext.indexOf(".");

**int** len = arg5 - index;

//小数位只能2位

**if**(len > **PONTINT\_LENGTH**){

CharSequence newText = arg3.subSequence(arg4, arg5);

**return** newText;

}

}

**return** arg3.subSequence(arg4, arg5) +inputtext;

}

};

//使用---TextView.setFilters(InputFilter[] filters)

edt\_shopkeeper\_phoneno.setFilters(**new** InputFilter[]{mInputFilter});

## 监听文字改变，匹配中文

mEditText\_ps.addTextChangedListener(**new** TextWatcher() {

@Override

**public** **void** **onTextChanged**(CharSequence s, **int** start, **int** before, **int** count) {

**Log**.e("on", s.toString());

**String** regexStr = "[\u4E00-\u9FA5]";//中文的编码

**String** newStr = StringFilter(s.toString(), regexStr);//去掉中文

**String** text = mEditText\_ps.getText().toString().trim();//如果是中文，text的值会包含中文

**if** (!newStr.equals(text)) {

mEditText\_ps.setText(newStr);

mEditText\_ps.setSelection(newStr.length());

}

}

@Override

**public** **void** **beforeTextChanged**(CharSequence s, **int** start, **int** count, **int** after) {

**Log**.e("before", s.toString());

}

@Override

**public** **void** **afterTextChanged**(Editable s) {

**Log**.e("after", s.toString());

}

});

**public** **static** **String** **StringFilter**(**String** str, **String** regEx)**throws** PatternSyntaxException{

//String regEx = "[/\\:\*?<>|\"\n\t]"; //要过滤掉的字符

**Pattern** p = **Pattern**.compile(regEx);

**Matcher** m = p.matcher(str);

**return** m.replaceAll("").trim();

}

## 实现显示和隐藏密码

**if** (mEditText\_ps.getTransformationMethod() **instanceof** **PasswordTransformationMethod**) {//判断是否是密码状态

mEditText\_ps.setTransformationMethod(**HideReturnsTransformationMethod**.getInstance());//设置为普通状态

}**else** {

mEditText\_ps.setTransformationMethod(**PasswordTransformationMethod**.getInstance());

}

Editable editable = mEditText\_ps.getEditableText();//光标移动到edittext末尾

**Selection**.setSelection(editable, editable.length);

## Edit光标和变化的下划线

et\_underline\_unselected.xml

<?xml version="1.0" encoding="utf-8"?>  
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
 <item  
 android:bottom="0dp"  
 android:left="-2dp"  
 android:right="-2dp"  
 android:top="-2dp">  
 <shape>  
 <solid android:color="@android:color/transparent" />  
 <stroke  
 android:width="1dp"  
 android:color="@color/gray" />  
 <padding android:bottom="4dp" />  
 </shape>  
 </item>  
</layer-list>

et\_underline\_selected.xml

<?xml version="1.0" encoding="utf-8"?>  
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
 <item  
 android:bottom="0dp"  
 android:left="-2dp"  
 android:right="-2dp"  
 android:top="-2dp">  
 <shape>  
 <solid android:color="@android:color/transparent" />  
 <stroke  
 android:width="1dp"  
 android:color="@color/red\_ff" />  
 <padding android:bottom="4dp" />  
 </shape>  
 </item>  
</layer-list>

et\_underline\_selector.xml

<?xml version="1.0" encoding="utf-8"?>  
<selector xmlns:android="http://schemas.android.com/apk/res/android">  
 <item android:state\_focused="false" android:drawable="@drawable/et\_underline\_unselected" />  
 <item android:state\_focused="true" android:drawable="@drawable/et\_underline\_selected" />  
</selector>

ed\_cursor.xml

<?xml version="1.0" encoding="utf-8"?>  
<shape xmlns:android="http://schemas.android.com/apk/res/android"  
 android:shape="rectangle">  
 <size android:width="2dp" />  
 <solid android:color="@color/green\_08" />  
</shape>

使用

<EditText  
 android:layout\_width="match\_parent"  
 android:layout\_height="40dp"  
 android:background="@drawable/et\_underline\_selector"  
 android:textCursorDrawable="@drawable/ed\_cursor" />

## 设置单行

singleLine

和layout\_width一起使用，当文本不能全部显示时，TextView中后面用…来表示，EditText中光标可拖动

maxLines = “1”

还需要和inputType同时使用，TextView不会显示...,EditText中光标可拖动

## 禁止回车换行

监听回车键即可

edt.setOnEditorActionListener(new TextView.OnEditorActionListener() {

@Override

public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {

return (event.getKeyCode() == KeyEvent.KEYCODE\_ENTER);

}

});

## 禁止使用emoji（并加上字数限制）

工具类

public class EmojiFilterUtil {  
 /\*\*  
 \* 限制emoji  
 \*/  
 public static InputFilter[] getFilter() {  
 InputFilter emojiFilter = new InputFilter() {  
 @Override  
 public CharSequence filter(CharSequence source, int start, int end, Spanned dest, int dstart, int dend) {  
 Pattern emoji = Pattern.compile("[\ud83c\udc00-\ud83c\udfff]|[\ud83d\udc00-\ud83d\udfff]|[\u2600-\u27ff]",  
 Pattern.UNICODE\_CASE | Pattern.CASE\_INSENSITIVE);  
 Matcher emojiMatcher = emoji.matcher(source);  
 if (emojiMatcher.find()) {  
 ToastUtil.shortShow("不支持输入表情");  
 return "";  
 }  
 return null;  
 }  
 };  
 return new InputFilter[]{emojiFilter};  
 }  
 /\*\*  
 \* 限制emoji和字数  
 \*/  
 public static InputFilter[] getFilter(int lengh) {  
 InputFilter emojiFilter = new InputFilter() {  
 @Override  
 public CharSequence filter(CharSequence source, int start, int end, Spanned dest, int dstart, int dend) {  
 Pattern emoji = Pattern.compile("[\ud83c\udc00-\ud83c\udfff]|[\ud83d\udc00-\ud83d\udfff]|[\u2600-\u27ff]",  
 Pattern.UNICODE\_CASE | Pattern.CASE\_INSENSITIVE);  
 Matcher emojiMatcher = emoji.matcher(source);  
 if (emojiMatcher.find()) {  
 ToastUtil.shortShow("不支持输入表情");  
 return "";  
 }  
 return null;  
 }  
 };  
 return new InputFilter[]{new InputFilter.LengthFilter(lengh), emojiFilter};  
 }  
}

使用

edt.setFilters(EmojiFilterUtil.getFilter(50));

edt.setFilters(EmojiFilterUtil.getFilter());

## 限制小数位数

*/\*\*  
 \* 限制小数位数  
 \*/*private static void limitXDecimal(EditText et, int place) {  
 et.addTextChangedListener(  
 new TextWatcher() {  
 @Override  
 public void beforeTextChanged(CharSequence s, int start, int count, int after) {  
 }  
 @Override  
 public void onTextChanged(CharSequence s, int start, int before, int count) {  
 }  
 @Override  
 public void afterTextChanged(Editable s) {  
 if (s == null) return;  
 String str = s.toString();  
 if (TextUtils.*isEmpty*(str)) return;  
 String strBehind = StrNumUtil.*splitBehind*(str, "\\.");  
 if (strBehind.length() <= place) return;  
 ToastUtil.*shortShow*("最多输入" + place + "位小数！");  
 et.setText(str.substring(0, str.length() - (strBehind.length() - place)));  
 }  
 }  
 );  
}

## 添加下划线

### 如果是在资源文件里：

1、字符串资源中设置下划线属性(用<u>标签,最简单的方法)

<resources>

<string name="hello"><u>phone:0123456</u></string>

</resources>

2、TextView设置autoLink属性(会主动调用系统浏览器,跳转链接)

<TextView xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/text1"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:autoLink="all"

android:text="@string/link\_text\_auto" />

### 如果是代码里：

1、使用TextView的Paint的属性

tvTest.getPaint().setFlags(Paint.UNDERLINE\_TEXT\_FLAG); //下划线

tvTest.getPaint().setAntiAlias(true);//抗锯齿

2、使用SpannableString类( 富文本)

SpannableString content = new SpannableString(str);

content.setSpan(new UnderLineSpan, 0, str.length(), 0);

3、使用Html.fromHtml()

TextView textView = (TextView)findViewById(R.id.tv\_test);

textView.setText(Html.fromHtml("<u>"+"0123456"+"</u>"));

4、代码里面自定义超链接样式：

TextView tv=new TextView(this);

tv.setText(Html.fromHtml("<a href=\"http://blog.csdn.net/CAIYUNFREEDOM\">自定义的超链接样式</a>"));

// 在单击链接时凡是有要执行的动作，都必须设置MovementMethod对象

tv.setMovementMethod(LinkMovementMethod.getInstance());

CharSequence text = tv.getText();

if (text instanceof Spannable){

int end = text.length();

Spannable sp = (Spannable)tv.getText();

URLSpan[] urls = sp.getSpans( 0 , end, URLSpan.class );

SpannableStringBuilder style = new SpannableStringBuilder(text);

style.clearSpans(); // should clear old spans

for (URLSpan url : urls){

URLSpan myURLSpan = new URLSpan(url.getURL());

style.setSpan(myURLSpan,sp.getSpanStart(url),sp.getSpanEnd(url),Spannable.SPAN\_EXCLUSIVE\_EXCLUSIVE);

}

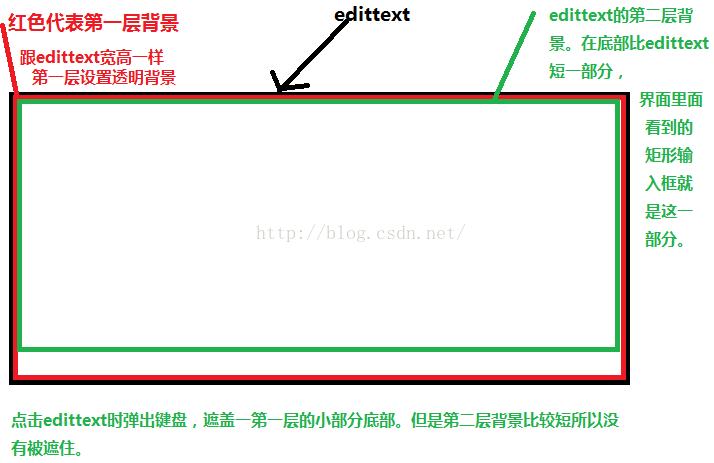
tv.setText(style);

}

## 解决输入框被键盘压住一半的问题

### 原理

EditText设置2层背景，底层背景正常位置，第二层有框的背景设置 bottom = 10dp使第二层背景距离底部10dp,从而第二层背景不在被遮盖。弹出键盘时，遮盖第一层的小部分底部，但是第二层比较短，所以没有遮住，满足需求



改动前后的对比图如下：

### layer\_chat\_edt.xml

<?xml version="1.0" encoding="UTF-8"?>

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

<item>

<shape>

<solid android:color="#00000000" />

</shape>

</item>

<item android:bottom="10dp">

<shape android:shape="rectangle">

<solid android:color="#00000000" />

<stroke

android:width="1dp"

android:color="#dfdfdf" />

<corners android:radius="5dp" />

</shape>

</item>

</layer-list>

### layer\_chat\_btn.xml

<?xml version="1.0" encoding="UTF-8"?>

<layer-list xmlns:android="http://schemas.android.com/apk/res/android">

<item>

<shape>

<solid android:color="#00000000" />

</shape>

</item>

<item android:bottom="10dp">

<shape android:shape="rectangle">

<solid android:color="#54afef" />

<corners android:radius="5dp" />

</shape>

</item>

</layer-list>

### activity\_used.xml

<!-- 输入评论控件-->

<LinearLayout

android:id="@+id/ll\_chat\_container"

android:layout\_width="match\_parent"

android:layout\_height="44dp"

android:background="#ffffff"

android:gravity="center\_vertical"

android:orientation="horizontal"

android:paddingLeft="10dp"

android:paddingRight="10dp"

android:paddingTop="4dp">

<!--评论输入框-->

<EditText

android:id="@+id/edt\_chat\_content"

android:layout\_width="0dp"

android:layout\_height="match\_parent"

android:layout\_weight="1"

android:background="@drawable/layer\_chat\_edt"

android:maxLength="50"

android:paddingBottom="10dp"

android:paddingLeft="10dp"

android:paddingRight="10dp"

android:singleLine="true"

android:textColor="#333333"

android:textSize="12sp" />

<!--评论发送按钮-->

<TextView

android:id="@+id/tv\_chat\_send"

android:layout\_width="50dp"

android:layout\_height="match\_parent"

android:layout\_alignParentBottom="true"

android:layout\_marginLeft="10dp"

android:gravity="center"

android:background="@drawable/layer\_chat\_btn"

android:paddingBottom="10dp"

android:text="发送"

android:textColor="#ffffff" />

</LinearLayout>

## 空格、换行

### 空格

在Android布局中进行使用到空格，以便实现文字的对齐。那么在Android中如何表示一个空格呢？

Xml中

空格：&#160;

窄空格：&#8201;

半个汉字宽度的空格：&#8194;

一个中文字但用起来会比中文字宽一点点：&#8195;

一个中文大小的空格：\u3000

因为&#160; &#8201;在不同机型有不同表现，所以中文对齐还是建议使用&#8194;或者\u3000

strings中

<string name="str1">姓&#8194;&#8194;名：</string>

android:text="@string/str1"

代码中

使用setText("\u3000"+xx);或setText("xx\u3000xx");

例子

android:text="真实姓名:"

android:text="身&#8194;份&#8194;证:"

 IMG_256

android:text="姓&#8194;&#8194;名:

android:text="姓\u3000名:"

android:text="身份证:"

### 换行和回车

#### 概念和由来

'\r'是回车（carriage return），使光标到行首

'\n'是换行（line feed），使光标下移一格

在计算机还没有出现之前，有一种叫做电传打字机（Teletype Model 33）的玩意，每秒钟可以打10个字符。但是它有一个问题，就是打完一行换行的时候，要用去0.2秒，正好可以打两个字符。要是在这0.2秒里面，又有新的字符传过来，那么这个字符将丢失。于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做“回车”，告诉打字机把打印头定位在左边界；另一个叫做“换行”，告诉打字机把纸向下移一行。这就是“换行”和“回车”的来历，从它们的英语名字上也可以看出一二。

后来，计算机发明了，这两个概念也就被般到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。

分歧

Unix系统里，每行结尾只有“<换行>”，即'\n'

Windows系统里面，每行结尾是“<换行><回车>”，即'\r\n'

Mac系统里，每行结尾是“<回车>”即'\r

直接后果

Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；

Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号。

#### Android中如何使用

Android是基于Linux内核的，在处理换行符时使用的是'\n'。

* Xml中，在文字的前面加入"\n"，例如： android:text="姓发斯蒂芬第三方\n的就开了房间电力科技"
* 代码中，使用setText("姓发\n科技");或setText("姓发" + "\n" + "科技");
* strings中，<string name="str1">姓发\n科技</string>

## 其他属性

### drawableLeft和drawablePadding

Xml中

android:background="@drawable/head\_sexage\_shape"

android:drawableLeft="@mipmap/iconfont\_nan"

android:drawablePadding="5dp"

* 可以设置背景样式
* drawablePadding是指文字和图片的距离

代码中

Drawable drawableNv= mContext.getResources().getDrawable(R.mipmap.icon\_nv);

drawableNv.setBounds(0, 0, drawableNv.getMinimumWidth(), drawableNv.getMinimumHeight());

tv.setCompoundDrawables(drawableNv, null, null, null);

或者

tv.setCompoundDrawablesWithIntrinsicBounds(R.mipmap.icon\_nv, 0, 0, 0);

### textSize设置文字大小

setTextSize(int size)默认的单位是sp，可以用另一个设置字体大小的方法：setTextSize(int unit, int size)

第一个参数可设置如下静态变量：

TypedValue.COMPLEX\_UNIT\_PX : Pixels

TypedValue.COMPLEX\_UNIT\_SP : Scaled Pixels

TypedValue.COMPLEX\_UNIT\_DIP : Device Independent Pixels

### textCursorDrawable光标颜色



### digits实现只能输入特定的字符

android:digits="01234aBc"

控制输入所有数字、英文、英文符号

<string name="password\_digits">0123456789abcdefghigklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ`~!@#$%^\*()-\_+=[]{}|:;,.?&lt;&gt;&amp;\'\"\/\\</string>

android:digits="@string/password\_digits"

* 单引号、双引号、左斜杆、有斜杠需要转义\'\"\/\\
* &<>在xml中需要用替代语句&lt;&gt;&amp;

### inputType设置默认输入中文、英文、数字

代码中

默认中文：

mEditText.setInputType(EditorInfo.TYPE\_CLASS\_TEXT);

默认英文：

mEditText.setInputType(EditorInfo.TYPE\_TEXT\_VARIATION\_URI);

默认数字

mEditText.setInputType(EditorInfo.TYPE\_CLASS\_NUMBER);

Xml中

number 用来输入数字

numberDecimal 可以带小数点的浮点格式

textCapCharacters 输入的全部字母大写

textCapWords 输入的每个单词首字母大写

textCapSentences 输入的每个句子首字母大写

textPassword 用来输入密码，所有输入的字符都会变成星号\*

textVisiblePassword 用来输入密码，输入的字符原样显示

textEmailAddress 用来输入Email地址

### ellipsize设置当文字过长时,该控件该如何显示

android:ellipsize设置当文字过长时,该控件该如何显示。有如下值设置：”start”—-省略号显示在开头;”end” ——省略号显示在结尾;”middle”—-省略号显示在中间;”marquee” ——以跑马灯的方式显示(动画横向移动)

### Ems设置控件一行的容纳字符数

em是一个印刷排版的单位，表示字宽的单位。 em字面意思为：equal M（和M字符一致的宽度为一个单位）简称em。ems是em的复数表达。设置maxEms是指控件宽度最大为X个字符的宽度，一行中最大只能显示设置的宽度，超过就换行

规律

* 只有在android:layout\_width=“wrap\_content”时，才会生效；“match\_parent”时，则不会生效
* 印刷版的M（不是代码中的M）占一个ems；一个汉字大概占一个ems；一个数字、字母大概占半个ems
* 以汉字为例：设你输入的参数为n， 屏幕显示的数量为m。值为1-5时，m = n；6-11时，m = n+1；12-18时，m = n+2。

属性

* android:ems设置TextView的宽度为N个字符的宽度
* android:maxEms设置TextView的宽度为最长为N个字符的宽度。与ems同时使用时覆盖ems选项。
* android:minEms设置TextView的宽度为最短为N个字符的宽度。与ems同时使用时覆盖ems选项。

### maxLength文本字数控制

android:maxLength限制显示的文本长度，超出部分不显示。

### textAppearance字体外观

android:textAppearance设置文字外观。如 “?android:attr/textAppearanceLargeInverse”这里引用的是系统自带的一个外观，?表示系统是否有这种外观，否则使用默认的外观。可设置的值如下：textAppearanceButton/textAppearanceInverse/textAppearanceLarge/textAppearanceLargeInverse/textAppearanceMedium/textAppearanceMediumInverse/textAppearanceSmall/textAppearanceSmallInverse

### textColorHighlight被选中文字的底色

android:textColorHighlight被选中文字的底色，默认为蓝色

### textColorHint提示信息文字的颜色

android:textColorHint设置提示信息文字的颜色，默认为灰色。与hint一起使用。

### textColorLink文字链接的颜色

android:textColorLink文字链接的颜色.

### textScaleX文字之间间隔

android:textScaleX设置文字之间间隔，默认为1.0f。

### lineSpacing行间距

android:lineSpacingExtra

android:lineSpacingExtra表示额外的行间距数值，单位通常为dp。如android:lineSpacingExtra=”1dp”。

android:lineSpacingExtra的值可以为负数，小数和0。如果值为正数表示增加行间距，如果值为负数表示减少行间距，如果值为0，则没有变化。

android:lineSpacingMultiplier

android:lineSpacingMultiplier表示行间距的倍数，没有单位。如android:lineSpacingMultiplier=”1.2”。

android:lineSpacingMultiplier的值可以为任意浮点数。如果值大于1.0表示增加行间距，如果值小于1.0表示减少行间距。

一起使用时

先增加android:lineSpacingMultiplier设置的倍数，再加上android:lineSpacingExtra设置的额外的间距。

代码中

public void setLineSpacing(float add, float mult);

参数add表示要增加的间距数值，对应android:lineSpacingExtra参数。

参数mult表示要增加的间距倍数，对应android:lineSpacingMultiplier参数。

### textStyle设置字形

android:textStyle设置字形[bold(粗体) 0, italic(斜体) 1, bolditalic(又粗又斜) 2] 可以设置一个或多个，用“|”隔开

### typeface设置字体

android:typeface设置文本字体，必须是以下常量值之一：normal 0, sans 1, serif 2, monospace(等宽字体) 3]

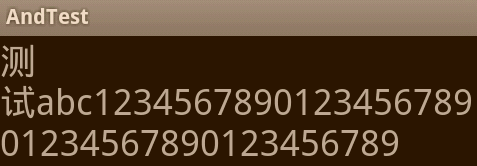
### 加粗

textView .setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));

textView.setTypeface(Typeface.defaultFromStyle(Typeface.NORMAL));

## 有时候会诡异换行

TextView在遇到非英文的时候，可能出现很诡异的换行，例如，如果TextView的text是"测试abc123456789012345678901234567890123456789"，那么会发生下图的状况。



花了一下午研究Android的源代码，终于发现了原因所在。这个逻辑是在android.text.StaticLayout的方法generate中实现的（代码比较复杂，看得挺吃力的。。。）。  
 Android中判断两个字符之间是否可以换行（称为换行点，breakpoint），依据的方法是Unicode Line Breaking Algorithm。  
 根据Unicode Line Breaking Algorithm，"测"和"试“之间是可以换行的（所有的中文字符都一样，包括全角的标点符号等），而"试"和"a"之间是不可以换行的，后面的所有英文字符和数字之间也是不可以换行的。  
 generate中，当发现在TextView的宽度中塞不下这一行中更多的字符时，如果之前找到了可以换行的点，那么就在最后一个换行点（在这个例子中是"测"和“试”之间）进行换行；如果没有换行点，那么就直接在当前字符位置换行（也就是说，虽然“试abc123456789012345678901234567890123456789”这个长字符串找不到换行点，但是也只能强制在能显示出来的最后一个字换行了）。  
 解决办法：将TextView换为EditText即可

<EditText  
 android:id="@+id/item\_chat\_card\_email"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:background="@null"  
 android:enabled="false" />

# 虚拟键盘相关

## 打开和关闭输入法

手动控制输入法的开关状态也能提升用户体验，比如：

* 有的搜索框会有一个清除按钮，点击清除按钮时就应该弹出输入法，因为用户清除搜索内容的目的多数时候是需要输入新的内容；
* 执行搜索时应该隐藏输入法，因为显示输入法时会遮挡搜索结果，用户体验不太好；
* 闹钟来时或者有其它window弹出时应该隐藏输入法，因为输入法也是window，如果不隐藏可能导致输入法遮挡住了其它window等用户体验不太友好的问题。

### 打开输入法：

**public** **static** **void** **showSoftInput**(**Context** context, **EditText** edit) {

edit.setFocusable(**true**);

edit.setFocusableInTouchMode(**true**);

edit.requestFocus();

**InputMethodManager** inputManager = (**InputMethodManager**) context.getSystemService(**Context**.**INPUT\_METHOD\_SERVICE**);

inputManager.showSoftInput(edit, 0);

}

### 关闭输入法：

**public** **static** **void** **hideSoftInput**(**Activity** activity) {

**View** view = activity.getWindow().peekDecorView();

**if** (view != **null**) {

**InputMethodManager** inputmanger = (**InputMethodManager**) activity.getSystemService(**Context**.**INPUT\_METHOD\_SERVICE**);

inputmanger.hideSoftInputFromWindow(view.getWindowToken(), 0);

}

}

**public** **static** **void** **hideSoftInput**(**Context** context, **EditText** edit) {

edit.clearFocus();

**InputMethodManager** inputmanger = (**InputMethodManager**) context.getSystemService(**Context**.**INPUT\_METHOD\_SERVICE**);

inputmanger.hideSoftInputFromWindow(edit.getWindowToken(), 0);

}

关于收起虚拟键盘，网上能找到的大多是这个：

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

imm.hideSoftInputFromWindow(editText.getWindowToken(), 0) ;

不过这有个问题，比如我们在一个页面中有很多editText，到最后提交的时候我们并不知道实际光标停留在哪个editText。除非一个一个遍历判断。那么该怎么办呢？我后来找到这样的方法：

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

if(imm != null) {

      imm.hideSoftInputFromWindow(getWindow().getDecorView().getWindowToken(), 0);

}

这个便不用提供绑定的editText。完美解决

## 设置不调用系统键盘

**if** (android.os.**Build**.**VERSION**.**SDK\_INT** <= 10) {

textAmount.setInputType(InputType.**TYPE\_NULL**);

} **else** {

**this**.getWindow().setSoftInputMode(WindowManager.**LayoutParams**.**SOFT\_INPUT\_STATE\_ALWAYS\_HIDDEN**);

**try** {

**Class**<**EditText**> cls = **EditText**.**class**;

**Method** setShowSoftInputOnFocus;

setShowSoftInputOnFocus = cls.getMethod("setShowSoftInputOnFocus", **boolean**.**class**);

setShowSoftInputOnFocus.setAccessible(**true**);

setShowSoftInputOnFocus.invoke(textAmount, **false**);

} **catch** (**Exception** e) {

e.printStackTrace();

}

}

## 监听输入法中的按钮（比如回车）

比如搜狗输入法的右下角有一个回车按钮，我们希望用户点击它时也执行确认功能，可以通过监听EditText的按键点击事件来实现：

mInputEditTxt.setOnKeyListener(new OnKeyListener() {

@Override

public boolean onKey(View v, int keyCode, KeyEvent event) {

if (keyCode == KeyEvent.KEYCODE\_ENTER && event.getAction() == KeyEvent.ACTION\_UP) {

System.out.println("手指弹起时执行确认功能");

return true;

}

return false;

}

});

## 改变回车按钮的显示内容（配合监听回车按钮使用）

必须同时使用inputType

android:imeOptions附加功能，设置右下角IME动作与编辑框相关的动作，如actionDone右下角将显示一个“完成”，而不设置默认是一个回车符号。这个在EditView中再详细说明，此处无用。

android:imeActionId设置IME动作ID。

android:imeActionLabel设置IME动作标签。

**代码中**

/\*\*

\* IME\_ACTION\_SEARCH 搜索

\* IME\_ACTION\_SEND 发送

\* IME\_ACTION\_NEXT 下一步

\* IME\_ACTION\_DONE 完成

\* IME\_ACTION\_GO 前进

\* ME\_ACTION\_NONE 无动作

\* IME\_ACTION\_PREVIOUS上一项

\* IME\_ACTION\_UNSPECIFIED 未指定

\*/

mInputEditTxt.setImeOptions(EditorInfo.IME\_ACTION\_SEARCH);

**Layout中**

actionDone 完成

actionGo 前进

actionNext 下一项

actionNone 无动作

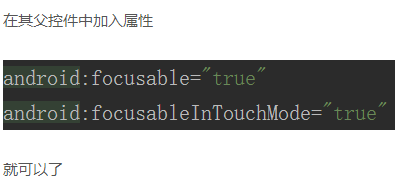
actionPrevious 上一项

actionSearch 搜索

actionUnspecified 未指定

actionSend 发送

## 进入activity时候让Edittext的时候，使其失去焦点



# SpannableString

## Spannable

/两次加大字体，设置字体为红色（big会加大字号，font可以定义颜色）

textView1.setText(Html.fromHtml("北京市发布霾黄色预警，<font color='#ff0000'><big><big>外出携带好</big></big></font>口罩"));

//设置字体大小为3级标题，设置字体为红色

textView2.setText(Html.fromHtml("北京市发布霾黄色预警，<h3><font color='#ff0000'>外出携带好</font></h3>口罩"));

//设置字体大小为58（单位为物理像素），设置字体为红色，字体背景为黄色

textView3.setText("北京市发布霾黄色预警，外出携带好口罩");

Spannable span = new SpannableString(textView3.getText());

span.setSpan(new AbsoluteSizeSpan(58), 11, 16, Spannable.SPAN\_EXCLUSIVE\_EXCLUSIVE);

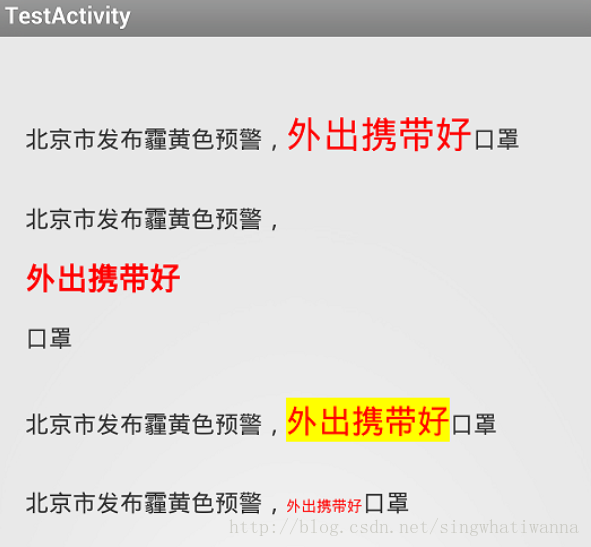
span.setSpan(new ForegroundColorSpan(Color.RED), 11, 16, Spannable.SPAN\_EXCLUSIVE\_EXCLUSIVE);

span.setSpan(new BackgroundColorSpan(Color.YELLOW), 11, 16, Spannable.SPAN\_EXCLUSIVE\_EXCLUSIVE);

textView3.setText(span);

//两次缩小字体，设置字体为红色（small可以减小字号）

textView4.setText(Html.fromHtml("北京市发布霾黄色预警，<font color='#ff0000'><small><small>外出携带好</small></small></font>口罩"));



说明：上述代码分别用不同方式实现了类似的效果，通过对比，我们应该可以理解各种方法的不同，想写出更多效果Spannable是必须的，因为font支持的属性很有限，比如说font不支持size属性，所以请多多了解Spannable。也许有时候我们想把文本加到xml里面，这也是可以的，只不过有一点我们需要注意，那就是要对<、>、'、"这四个字符进行转换，因为它们四个是HTML的标签。具体来说，左括号<要用&lt;代替，右括号>要用&gt;代替，单引号和双引号要用\进行转义。这里给出一个示例，其效果和上面代码中的效果是一致的：<string name="text">北京市发布霾黄色预警，&lt;font color=\'#ff0000\'&gt;&lt;big&gt;&lt;big&gt;外出携带好&lt;/big&gt;&lt;/big&gt;&lt;/font&gt;口罩</string>，另外，你必须显式地调用textView.setText(Html.fromHtml(getString(R.string.text)))来使其生效。

## ImageSpan

if (!str.*isNullOrBlank*()) {  
 val placeholder = "a"//需要注意图片的占位符  
 val kong = " "  
 val strRes = "$str$kong$placeholder$kong$placeholder"  
 val spanStr = SpannableString(strRes)  
 val imageSpan1 = ImageSpan(tv.*context*, R.drawable.*icon\_card\_qiye*)  
 val imageSpan2 = ImageSpan(tv.*context*, R.drawable.*icon\_card\_sihiming*)  
 val len = strRes.length  
 spanStr.setSpan(imageSpan1, len - 3, len - 2, Spannable.*SPAN\_EXCLUSIVE\_EXCLUSIVE*)  
 spanStr.setSpan(imageSpan2, len - 1, len, Spannable.*SPAN\_EXCLUSIVE\_EXCLUSIVE*)  
 tv.*text* = spanStr  
}



## ClickableSpan

SpannableString spannableString = new SpannableString("点我打开链接");

ClickableSpan clickableSpan = new ClickableSpan() {

@Override

public void onClick(View widget) {

Intent intent = new Intent(demo3.this,MainActivity.class);

startActivity(intent);

}

};

spannableString.setSpan(clickableSpan, 0, 6, Spanned.SPAN\_EXCLUSIVE\_EXCLUSIVE);

tv\_openActivity.setText(spannableString);

tv\_openActivity.setMovementMethod(LinkMovementMethod.getInstance());

## 其他超链接效果实现方式

String text1="<font color='red'><i>你好啊，陌生人</i></font><br/>";

text1+="<a href='http://www.baidu.com'>百度</a><br/>";

tv\_one.setText(Html.fromHtml(text1));//没有自动识别超链接，所以要配置

//产生链接效果

tv\_one.setMovementMethod(LinkMovementMethod.getInstance());

String text2="我的网站：http：//www.baidu.com ";

text2+="我的电话15102365487";

//因为在layout里设置了text2的autoLink（自动识别超链接，所以也会有超链接）

tv\_two.setText(text2);

tv\_two.setMovementMethod(LinkMovementMethod.getInstance());

## 网址跳转

public class MyHttpClickableSpan extends ClickableSpan {  
 //有无协议名http、https、ftp的网址都能匹配  
 public static Pattern *HTTP\_PATTERN* = Pattern.*compile*("((http[s]?|ftp)://[a-zA-Z0-9.\\-]+\\.([a-zA-Z]{2,4})(:\\d+)?(/[a-zA-Z0-9.\\-~!@#$%^&\*+?:\_/=<>]\*)?)|(www.[a-zA-Z0-9.\\-]+\\.([a-zA-Z]{2,4})(:\\d+)?(/[a-zA-Z0-9.\\-~!@#$%^&\*+?:\_/=<>]\*)?)");

//匹配http、https的网址  
 public static Pattern *HTTP\_PATTERN2* = Pattern.*compile*("([hH][tT]{2}[pP]://|[hH][tT]{2}[pP][sS]://)[-A-Za-z0-9+&@#/%?=~\_|!:,.;]+[-A-Za-z0-9+&@#/%=~\_|]");

private String url;  
 private Context mContext;  
 private long lastClickTime = 0;  
 public MyHttpClickableSpan(Context context, String url) {  
 this.url = url;  
 mContext = context;  
 }  
 @Override  
 public void updateDrawState(TextPaint ds) {  
 super.updateDrawState(ds);  
 //设置文本的颜色  
 ds.setColor(ContextCompat.*getColor*(mContext, R.color.*text\_blue*));  
 //超链接形式的下划线，false 表示不显示下划线，true表示显示下划线,其实默认也是true，如果要下划线的话可以不设置  
 ds.setUnderlineText(false);  
 }  
 //点击事件，自由操作  
 @Override  
 public void onClick(View widget) {  
 long now = System.*currentTimeMillis*();  
 if (now - lastClickTime > 2000) {  
 lastClickTime = now;  
 Intent intent = new Intent();  
 intent.setAction("android.intent.action.VIEW");  
 String temp = url;  
 if (temp.startsWith("http")) temp = "http://" + temp;  
 Uri content\_url = Uri.*parse*(temp);  
 intent.setData(content\_url);  
 mContext.startActivity(intent);  
 }  
 }  
}

*/\*\*  
 \* 网址变色  
 \*/*private void dealSpContentByHttp(String content, SpannableString spContent) {  
 //判断是否是网址  
 Matcher matcher = MyHttpClickableSpan.*HTTP\_PATTERN*.matcher(content);  
 while (matcher.find()) {  
 spContent.setSpan(new MyHttpClickableSpan(mContext, matcher.group()), matcher.start(), matcher.end(), Spanned.*SPAN\_EXCLUSIVE\_EXCLUSIVE*);  
 }  
}

contentTv.setMovementMethod(LinkMovementMethod.*getInstance*());  
contentTv.setText(spContent);

# Spinner下拉列表

sp\_one = (Spinner) findViewById(R.id.sp\_one);

String[] strArr = new String[]{"北京","香港","厦门","广州"};

ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, R.layout.simple\_spinner, strArr);

sp\_one.setAdapter(adapter);

SimpleAdapter adapter = new SimpleAdapter(this, list, R.layout.mysp, new String[]{"icon","text"}, new int[]{R.id.img\_one,R.id.tv\_one});

sp\_two.setAdapter(adapter);

sp\_two.setOnItemSelectedListener(new OnItemSelectedListener() {

# AutoCompleteTextView自动输入

**AutoCompleteTextView** auto\_textview = (**AutoCompleteTextView**) findViewById(**R**.**id**.**auto\_textview**);

**String**[] str = {"哈德良","哈德良长城","凯撒大帝","凯撒广场"};

**ArrayAdapter**<**String**> adapter = **new** ArrayAdapter<**String**>(**MainActivity**.**this**, **R**.**layout**.**auto\_textview**, str);

auto\_textview.setAdapter(adapter);

1.class Xc extends Thread {

public void run() {

for (int i = 0; i <= 60; i++) {

djsview = ("重新发送：(" + (60 - i) + ")");

Message msg = new Message();

msg.what = 1;

msg.obj = djsview;

if (i == 60) {

msg.what = 2;

}

handler.sendMessage(msg);

handler.sendEmptyMessage(3);//发空的msg过去，不接收值

handler.sendEmptyMessageDelayed(4, 1000);//延迟1秒钟

handler.sendMessageDelayed(msg, 2000);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

2.Handler handler = new Handler() {

public void handleMessage(android.os.Message msg) {

if (msg.what == 1) {

btn.setText(msg.obj + "");

} else if (msg.what == 2) {

3.Xc djs = new Xc();

djs.start();

# ProgressBar

<ProgressBar

style="?android:attr/progressBarStyleHorizontal"

android:max="100"

android:progress="75"

android:progressDrawable="@drawable/myprogress\_style" />

# ViewPager

## 代码

**public** **class** **MainActivity** **extends** **Activity** **implements** OnPageChangeListener{

...

viewpager = (**ViewPager**) findViewById(**R**.**id**.**viewpager**);

adapter = **new** ViewPagerAdapter(views);

viewpager.setAdapter(adapter);

viewpager.setOnPageChangeListener(**this**);

}

@Override

**public** **void** **onPageScrollStateChanged**(**int** arg0) {}

@Override

**public** **void** **onPageScrolled**(**int** arg0, **float** arg1, **int** arg2) {}

@Override

**public** **void** **onPageSelected**(**int** arg0) {

**if** (arg0 == views.size() - 1) {

btn\_welcome.setVisibility(**View**.**VISIBLE**);

}**else** {

btn\_welcome.setVisibility(**View**.**INVISIBLE**);

}

}

}

**public** **class** **ViewPaperAdapter** **extends** **PagerAdapter** {

**private** List<**View**> viewsList;

**public** **ViewPaperAdapter**(List<**View**> viewsList){

**this**.viewsList = viewsList;

}

@Override

**public** **int** **getCount**() {

**return** **this**.viewsList.size();

}

@Override

**public** **boolean** **isViewFromObject**(**View** arg0, **Object** arg1) {

**return** arg0==arg1;

}

@Override

**public** **Object** **instantiateItem**(**ViewGroup** container, **int** position) {

container.addView(viewsList.get(position));

**return** viewsList.get(position);

}**//初始化position位置的界面**

@Override

**public** **void** **destroyItem**(**ViewGroup** container, **int** position, **Object** object) {

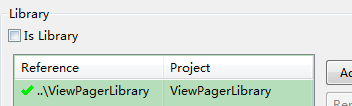
container.removeView(viewsList.get(position));

}

}

## 使用开源导航条Indicator

**使用com.viewpagerindicator引用包**



**Xml里**

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical" >

<com.viewpagerindicator.TabPageIndicator

android:id="@+id/indicator"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content" />

<android.support.v4.view.ViewPager

android:id="@+id/viewpager"

android:layout\_width="match\_parent"

android:layout\_height="0dp"

android:layout\_weight="1" />

</LinearLayout>

**实例化和关联viewpager**

viewPager = (**ViewPager**) findViewById(**R**.**id**.**viewpager**);

indicator = (**TabPageIndicator**) findViewById(**R**.**id**.**indicator**);

viewPager.setAdapter(**new** MyAdapter(getSupportFragmentManager()));

indicator.setViewPager(viewPager);

**Adapter里要复写title方法**

@Override

**public** CharSequence **getPageTitle**(**int** position) {

**return** strings[position % strings.length].toUpperCase();

}

**Manifest里边activity的theme属性就是indicator的样式，可点进去修改**

<activity

android:name=".MainActivity"

android:label="@string/app\_name"

android:theme="@style/Theme.PageIndicatorDefaults" >

## notifyDataSetChanged不管用的解决办法

在Adapter中重写getItemPosition方法，当数据改变时想要动态的更新ViewPager的显示时，在调用notifyDataSetChanged却发现ViewPager并没有进行相应的数据更新；重写getItemPosition让它返回POSITOIN\_ NONE;这样做的目的是notifyDataSetChanged时返回空，这样就会从数据重新填充，再外部更新数据然后调用notifyDataSetChanged更新就达到了目的了。

## ViewPager和Fragment结合使用onResume问题

ViewPager中的Fragment会提前加载左右两个，提前触发onResume方法，当遇到滑动到某个Fragment才刷新该Fragment的需求时，解决办法：复写Fragment的setUserVisibleHint方法

setUserVisibleHint优先于onViewCreated，并且顾名思义，对用户可见性发生变化时被调用，所以：

//在不切换fragment情况下，onResume情况不会触发setUserVisibleHint，所以需要在onResume中容错  
override fun setUserVisibleHint(isVisibleToUser: Boolean) {  
 super.setUserVisibleHint(isVisibleToUser)  
 if (!index.*isNullOrBlank*()) {  
 if (isVisibleToUser) {  
 refreshList()  
 //如果setUserVisibleHint中刷了，onResume中不要再刷  
 isUserVisibleHintFresh = true  
 /\*setUserVisibleHint永远会优先于onResume回调  
 找不到上级分发方法了，只能500毫秒后自动重置isUserVisibleHintFresh\*/  
 Handler().postDelayed(**{** isUserVisibleHintFresh = false **}**, 500)  
 }  
 }  
}  
override fun onResume() {  
 super.onResume()  
 //如果setUserVisibleHint中没有刷，onResume中刷  
 if (!isUserVisibleHintFresh) {  
 refreshList()  
 }  
}

private var isUserVisibleHintFresh = false

## 不可滑动ViewPager

public class NoScrollViewPager extends ViewPager {  
 private boolean isScroll = true;  
 public NoScrollViewPager(Context context, AttributeSet attrs) {  
 super(context, attrs);  
 }  
 public NoScrollViewPager(Context context) {  
 super(context);  
 }  
 */\*\*  
 \* 1.dispatchTouchEvent一般情况不做处理  
 \* ,如果修改了默认的返回值,子孩子都无法收到事件  
 \*/* @Override  
 public boolean dispatchTouchEvent(MotionEvent ev) {  
 return super.dispatchTouchEvent(ev); // return true;不行  
 }  
 */\*\*  
 \* 是否拦截  
 \* 拦截:会走到自己的onTouchEvent方法里面来  
 \* 不拦截:事件传递给子孩子  
 \*/* @Override  
 public boolean onInterceptTouchEvent(MotionEvent ev) {  
 // return false;//可行,不拦截事件,  
 // return true;//不行,孩子无法处理事件  
 //return super.onInterceptTouchEvent(ev);//不行,会有细微移动  
 if (isScroll) {  
 return super.onInterceptTouchEvent(ev);  
 } else {  
 return false;  
 }  
 }  
 */\*\*  
 \* 是否消费事件  
 \* 消费:事件就结束  
 \* 不消费:往父控件传  
 \*/* @Override  
 public boolean onTouchEvent(MotionEvent ev) {  
 //return false;// 可行,不消费,传给父控件  
 //return true;// 可行,消费,拦截事件  
 //super.onTouchEvent(ev); //不行,  
 //虽然onInterceptTouchEvent中拦截了,  
 //但是如果viewpage里面子控件不是viewgroup,还是会调用这个方法.  
 if (isScroll) {  
 return super.onTouchEvent(ev);  
 } else {  
 return true;// 可行,消费,拦截事件  
 }  
 }  
 public void setScroll(boolean scroll) {  
 isScroll = scroll;  
 }  
}

## 多次setAdapter，需清除缓存

//在setAdapter之前要清除FragmentManager缓存的fragment  
if (mViewPager.getAdapter() != null) {  
 FragmentManager fm = fragment.getChildFragmentManager();  
 FragmentTransaction ft = fm.beginTransaction();  
 Bundle bundle = new Bundle();  
 int index = mViewPager.getAdapter().getCount();  
 String key = "index";  
 while (index >= 0) {  
 bundle.putInt(key, index);  
 ft.remove(fm.getFragment(bundle, key));  
 index--;  
 }  
 ft.commit();  
}

# ListView小技巧

## 隐藏分割线和滚动条

android:scrollbars="none"

android:divider="@null";

listView.setDivider(null);

## 回弹

网上有很多通过重写ListView来实现弹性效果的方法，比如增加HeaderView或者使用ScrollView进行嵌套，方法有很多，不过这里可以使用一种非常简单的方法来实现这个效果。虽然不如那些方法可定制化高、效果丰富，但主要目的是学会如何从源代码中找到问题的解决办法。我们在查看ListView源代码的时候可以发现，ListView中有一个控制滑动到边缘的处理方法，如下所示。

protected boolean overScrollBy(int deltaX, int deltaY, int scrollX, int scrollY, int scrollRangeX, int scrollRangeY, int maxOverScrollX, int maxOverScrollY, boolean isTouchEvent)

可以看见这样一个参数：maxOverScrollY，注释中这样写道——Number of pixels to overscroll by in either direction along the Y axis。由此可以发现，虽然它的默认值是0，但其实只要修改这个参数的值，就可以让ListView具有弹性了！所以，既然我们不知道为什么Google不采用这样的修改，那我们就自己来修改一下吧。重写这个方法，并将maxOverScrollY改为设置的值——mMaxOverDistance，代码如下所示。

@Override

protected boolean overScrollBy(int deltaX, int deltaY, int scrollX, int scrollY, int scrollRangeX, int scrollRangeY, int maxOverScrollX, int maxOverScrollY, boolean isTouchEvent) {

return super.overScrollBy(deltaX, deltaY, scrollX, scrollY, scrollRangeX, scrollRangeY, maxOverScrollX, mMaxOverDistance, isTouchEvent);

}

这样，通过对这个值得修改，就实现了一个具有弹性的ListView了

## 添加headerView和footerView

addHeaderView()方法：主要是向listView的头部添加布局

addFooterView()方法：主要是向listView的底部添加布局

* 需要注意的是添加布局的时候应该添加从父容器开始添加，而不能直接添加父容器中的子控件。例如：从一个xml布局文件中添加一个button控件，只能将整个布局xml文件添加进去。而不能单单只添加button控件。
* 当添加头部和底部布局还有另外一个重载方法就是addHeaderView(headView, null, false) 和addFooterView(footerView, null, false)方法，这个方法与上面的方法区别在于：当给listView设置点击事件的时候，可以控制添加的布局是否可触发点击事件。
* addFooterView()方法必须放在listview.setadapter前面，给listview添加头部必须在绑定adapter前添加，否则会报错。原因是当我们在调用setAdapter方法时android会判断当前listview是否已经添加header，如果已经添加则会生成一个新的tempadapter，这个新的tempadapter包含我们设置的adapter所有内容以及listview的header和footer。所以当我们在给listview添加了header后在程序中调用listview.getadapter时返回的是tempadapter而不是我们通过setadapter传进去的adapter。如果没有设置adapter则tempadapter与我们自己的adapter是一样的。
* listview.getadapter().getcount()方法返回值会比我们预期的要大，原因是添加了header。
* 我们自定义adapter里面的getitem方法里面返回的position是不包括header的，是我们自定义adapter中数据position编号从0开始，也就是说与我们传进去的list的位置是一样的。
* 而Activity中listview的onitemclick方法：public void onItemClick(AdapterView<?> arg0, View arg1, int arg2,long arg3)，arg2是当前click的位置，这个位置是指在tempadapter中的位置，从0开始如果listview中添加了header则0代表header。也就是说当我们在使用点击事件时，listView列中的位置为arg2-1一般在开发中，为了达到我们想要的动态添加的效果主要的做法是：在listview.setadapter之前添加所需要的控件，然后使用removeFooterView()方法移除控件。在这里需要注意的是，每对listView的动态操作都要进行一次removeFooterView()方法移除控件。否则listView会自动添加空白行，从而影响显示的效果。
* item 内如果有button等控件时，在监听listview的onitemclick事件时，焦点会被item内的button、imagebutton等控 件抢走，从而导致在listview设置了onitemclick事件后不会被触发。解决方法是在初始化item的时候屏蔽掉其内部button等控件的 焦点获取，具体方法可以在自定义item的根控件中调用：setDescendantFocusability(ViewGroup.FOCUS\_BLOCK\_DESCENDANTS); 这样就能阻塞字控件抢夺焦点，listview的onitemclick就能被正确触发，同时对item内部的button等控件也没有影响，他们在被点击时照样可以触发自身的点击事件。

**关于FooterView的添加和删除**

* 每次总是先remove掉FooterView
* 若有需求再add上FooterView

我们通常在加载数据时，为了省流量不会一次性把数据全部下完，一般是分段下载。分段下载一般会在listview最后面放一个进度条表示正在加载数据，当数据加载完时，我们又要清除它。这时候就要注意了。

mLoadingLayout = (FrameLayout) View.inflate(this, R.layout.load, null);

listView.addFooterView(mLoadingLayout);

listView.requestFocus();

这是listview尾部添加一个进度条。

listView.removeFooterView(mLoadingLayout);

这是移除尾部的进度条。

有时候在移除时回报空指针，但listview不为null ，mLoadingLayout也不为null，但还是报空指针，原因是因为listview要分为三部分。一是头部，二是中间部，三是尾部。在设置了头部或尾部时，必须要有中间部才能真正意义上的生效。没生效就去移除就会报空指针错误。所以在 listView.removeFooterView(mLoadingLayout);时必须先调用 listView.setAdapter(adapter)（设置中间部），adapter可以数据可以为0但不可为null

## 自动显示、隐藏布局的ListView

当我们在ListView上滑动的时候，顶部的ActionBar或者Toolbar就会相应的隐藏或者显示

我们知道，让一个布局显示或者隐藏并带有动画效果，可以通过属性动画来很方便地实现，所以这个效果的关键就在于如何获得ListView的各种滑动事件。所以借助View的OnTouchListener接口来监听ListView的滑动，通过比较与上次坐标的大小，来判断滑动的方向，并通过滑动的方向来判断是否需要显示或隐藏对应的布局。在开始判断滑动事件之前，我们还要做一些准备工作，首先需要给ListView增加一个HeaderView，避免第一个Item被Toolbar遮挡，代码如下所示

**View** header = **new** View(**this**);

header.setLayoutParams(**new** **AbsListView**.LayoutParams(**ViewGroup**.**LayoutParams**.**MATCH\_PARENT**,*getTitleBarHeight*(**this**)));

lv.addHeaderView(header);

在代码中，通过静态方法**getTitleBarHeight(Activity activity)**获取Title栏的高度，并设置给HeaderView。另外，定义一个mTouchSlop变量来获取系统认为的最低滑动距离，即超过这个距离的移动，系统就将其定义为滑动状态了，对这个值得获取非常简单，代码如下所示。

mTouchSlop = ViewConfiguration.get(this).getScaledTouchSlop();

最后就是移动判断和动画了

lv.setOnTouchListener(**new** OnTouchListener() {

@Override

**public** **boolean** **onTouch**(**View** v, **MotionEvent** event) {

**switch** (event.getAction()) {

**case** **MotionEvent**.**ACTION\_DOWN**:

mFirstY = event.getY();

**break**;

**case** **MotionEvent**.**ACTION\_MOVE**:

mCurrentY = event.getY();

**if** (mCurrentY - mFirstY > ***mTouchSlop***) {// 手指下滑

toolbarAnim(0);

} **else** **if** (mFirstY - mCurrentY >***mTouchSlop***) {// 手指上划

toolbarAnim(1);

}

**break**;

}

**return** **false**;

}

});

**private** **void** **toolbarAnim**(**int** flag) {

**ObjectAnimator** mAnimator = **null**;

**if** (mAnimator != **null** && mAnimator.isRunning()) {

mAnimator.cancel();

}

**if** (flag == 0) {//从当前位置移动到0，也就是显示

mAnimator = **ObjectAnimator**.*ofFloat*(fl, "translationY", fl.getTranslationY(), 0);

} **else** {//从当前位置移动到看不见，也就是隐藏

mAnimator = **ObjectAnimator**.*ofFloat*(fl, "translationY", fl.getTranslationY(), -fl.getHeight());

}

mAnimator.start();

}

## 改变listView的选中颜色？

mListView.setSelector(null);//会报空指针

试试下面这个：

mListView.setSelector(new ColorDrawable(Color.TRANSPARENT));

## 聊天ListView（多布局）

public class ChatListViewAdapter extends BaseAdapter {

private List<ChatListViewBean> mData;

private LayoutInflater mInflater;

public ChatListViewAdapter(Context context, List<ChatListViewBean> data) {

this.mData = data;

mInflater = LayoutInflater.from(context);

}

@Override

public int getCount() {

return mData.size();

}

@Override

public Object getItem(int position) {

return mData.get(position);

}

@Override

public long getItemId(int position) {

return position;

}

@Override

public int getItemViewType(int position) {

return mData.get(position).getType();

}

@Override

public int getViewTypeCount() {

return 2;

}

@Override

public View getView(int position, View convertView, ViewGroup parent) {

ViewHolder viewHolder = null;

if (convertView == null) {

viewHolder = new ViewHolder();

if (getItemViewType(position) == 0) {

convertView = mInflater.inflate(R.layout.chat\_item\_itemin, null);

viewHolder.icon = (ImageView) convertView.findViewById(R.id.icon\_in);

viewHolder.text = (TextView) convertView.findViewById(R.id.text\_in);

} else {

convertView = mInflater.inflate(R.layout.chat\_item\_itemout, null);

viewHolder.icon = (ImageView) convertView.findViewById(R.id.icon\_out);

viewHolder.text = (TextView) convertView.findViewById(R.id.text\_out);

}

convertView.setTag(viewHolder);

} else {

viewHolder = (ViewHolder) convertView.getTag();

}

viewHolder.icon.setImageBitmap(mData.get(position).getIcon());

viewHolder.text.setText(mData.get(position).getText());

return convertView;

}

public final class ViewHolder {

public ImageView icon;

public TextView text;

}

}

## ListView异步加载图片乱序问题，原因分析及解决方案

恩？怎么会这个样子，当滑动ListView的时候，图片竟然会自动变来变去，而且图片显示的位置也不正确，简直快乱成一锅粥了！可是我们所有的逻辑都很简单呀，怎么会导致出现这种图片自动变来变去的情况？很遗憾，这是由于Listview内部的工作机制所导致的，如果你对Listview的工作机制不了解，那么就会很难理解这种现象，不过好在上篇文章中我已经讲解过ListView的工作原理了，因此下面就让我们一起分析一下这个问题出现的原因。

**原因分析**

上篇文章中已经提到了ListView之所以能够实现加载成百上千条数据都不会OOM，最主要在于它内部优秀的实现机制。虽然作为普通的使用者，我们大可不必关心ListView内部到底是怎么实现的，但是当你了解了它的内部原理之后，很多之前难以解释的问题都变得有理有据了。

ListView在借助RecycleBin机制的帮助下，实现了一个生产者和消费者的模式，不管有任意多条数据需要显示，ListView中的子View其实来来回回就那么几个，移出屏幕的子View会很快被移入屏幕的数据重新利用起来

那么这里我们就可以思考一下了，目前数据源当中大概有60个图片的URL地址，而根据ListView的工作原理，显然不可能为每张图片都单独分配一个ImageView控件，ImageView控件的个数其实就比一屏能显示的图片数量稍微多一点而已，移出屏幕的ImageView控件会进入到RecycleBin当中，而新进入屏幕的元素则会从RecycleBin中获取ImageView控件。

那么，每当有新的元素进入界面时就会回调getView()方法，而在getView()方法中会开启异步请求从网络上获取图片，注意网络操作都是比较耗时的，也就是说当我们快速滑动ListView的时候就很有可能出现这样一种情况，某一个位置上的元素进入屏幕后开始从网络上请求图片，但是还没等图片下载完成，它就又被移出了屏幕。这种情况下会产生什么样的现象呢？根据ListView的工作原理，被移出屏幕的控件将会很快被新进入屏幕的元素重新利用起来，而如果在这个时候刚好前面发起的图片请求有了响应，就会将刚才位置上的图片显示到当前位置上，因为虽然它们位置不同，但都是共用的同一个ImageView实例，这样就出现了图片乱序的情况。

但是还没完，新进入屏幕的元素它也会发起一条网络请求来获取当前位置的图片，等到图片下载完的时候会设置到同样的ImageView上面，因此就会出现先显示一张图片，然后又变成了另外一张图片的情况，那么刚才我们看到的图片会自动变来变去的情况也就得到了解释。

问题原因已经分析出来了，但是这个问题该怎么解决呢？说实话，ListView异步加载图片的问题并没有什么标准的解决方案，很多人都有自己的一套解决思路，这里我准备给大家讲解三种比较经典的解决办法，大家通过任何一种都可以解决这个问题，但是我们每多学习一种思路，水平就能够更进一步的提高。

**解决方案：使用findViewWithTag**

findViewWithTag算是一种比较简单易懂的解决方案，使用findViewWithTag并不需要修改太多的代码，只需要改动ImageAdapter这一个类就可以了，如下所示：

public class ImageAdapter extends ArrayAdapter<String> {

......

@Override

public View getView(int position, View convertView, ViewGroup parent) {

.....

ImageView image = (ImageView) view.findViewById(R.id.image);

image.setImageResource(R.drawable.empty\_photo);

image.setTag(url);

BitmapDrawable drawable = getBitmapFromMemoryCache(url);

if (drawable != null) {

image.setImageDrawable(drawable);

} else {

BitmapWorkerTask task = new BitmapWorkerTask();

task.execute(url);

}

return view;

}

......

/\*\*

\* 异步下载图片的任务。

\*/

class BitmapWorkerTask extends AsyncTask<String, Void, BitmapDrawable> {

...

@Override

protected void onPostExecute(BitmapDrawable drawable) {

ImageView imageView = (ImageView) mListView.findViewWithTag(imageUrl);

if (imageView != null && drawable != null) {

imageView.setImageDrawable(drawable);

}

}

......

}

}

## Holder重用有时会导致问题（需要时刻注意visible）

@Override

public void convert(CommonViewHolder holder, ContractDetailsBean contractDetailsBean) {

int flag = contractDetailsBean.getFlag();

switch (flag) {

case 0://默认的左右文字

holder.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container, View.VISIBLE)

.setText(R.id.fragment\_pt\_contract\_detiails\_item\_tv\_left, contractDetailsBean.getLeftStr())

.setText(R.id.fragment\_pt\_contract\_detiails\_item\_tv\_right, contractDetailsBean.getRightStr())

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container1, View.GONE)

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container2, View.GONE);

break;

case 1://分隔块

holder.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container1, View.VISIBLE)

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container, View.GONE)

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container2, View.GONE);

break;

case 2://edittext

holder.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container2, View.VISIBLE)

.setText(R.id.fragment\_pt\_contract\_detiails\_item\_tv\_left2, contractDetailsBean.getLeftStr())

.setText(R.id.fragment\_pt\_contract\_detiails\_item\_tv\_right2, contractDetailsBean.getRightStr())

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container1, View.GONE)

.setVisiable(R.id.fragment\_pt\_contract\_detiails\_item\_container, View.GONE);

break;

}

}

## 设置listview需要显示在第几项

listview是以item为一个单元来显示的，默认显示第一个item（索引为0），当需要指定具体显示的item时可以通过以下代码来实现：

mlistView.setSelection(10);

但是这个方法是瞬间完成的移动，除此之外还可以使用以下代码来实现平滑移动

mlistView.smoothScrollBy(distance, duration);

mlistView.smoothScrollByOffset(offset);

mlistView.smoothScrollToPosition(position);

## 屏蔽ListView的touch事件，让外层容器获取touch事件

在代码中设置

lv.setFocusable(false);  
lv.setClickable(false);  
lv.setPressed(false);  
lv.setEnabled(false);

## 在tv项目中，禁止listview的item选中

在xml中，设置android:focusable="false"无用，所以需要在代码中设置

listView.setFocusable(false);  
listView.setFocusableInTouchMode(false);  
listView.setClickable(false);

## 让item响应，而不是item中的button响应

可以使用descendantFocusability来解决，该属性是当一个为view获取焦点时，定义viewGroup和其子控件两者之间的关系。属性的值有三种：

* beforeDescendants：viewgroup会优先其子类控件而获取到焦点
* afterDescendants：viewgroup只有当其子类控件不需要获取焦点时才获取焦点
* blocksDescendants：viewgroup会覆盖子类控件而直接获得焦点

通常我们用到的是第三种，即在Item布局的根布局加上android:descendantFocusability="blocksDescendants"

## 从底部开始展示

### stackFromBottom

ListView有一个属性叫做stackFromBottom，默认该属性是false。该属性的含义是列表是否从ListView的最底部开始展示，若为true，则从最底部展示，否则从最顶部开始展示。

xml中

<ListView

android:id="@+id/listview"

android:stackFromBottom="true"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

</ListView>

代码中

lv.setStackFromBottom(false);

注意

如果所有子View的高度加起来也没有ListView控件高的话，会在ListView上部留空白

### transcriptMode

normal // 如果当前的最后一个 Item 在 ListView 显示范围内，内容变化时就从滚动底部；否则不滚动到底部

disabled // 禁用

alwaysScroll // 总是滚动到最新一条

使用注意

datas从0加入数据后，不起作用

连续添加两次数据，不起作用

使用方法：先加一条数据，延时100毫秒再加其余数据

@Override

public void onChatList(FeedChatResponse response) {

datas.clear();

if (response != null) {

List<FeedChatResponse.Rows> rows = response.getRows();

if (StrNumUtil.notEmptyList(rows)) {

if (rows.size() > 1) {

datas.add(rows.get(0));

adapter.notifyDataSetChanged();

new Handler().postDelayed(new Runnable() {

@Override

public void run() {

for (int i = 1; i < rows.size(); i++) {

datas.add(rows.get(i));

}

adapter.notifyDataSetChanged();

}

},100);

}

}

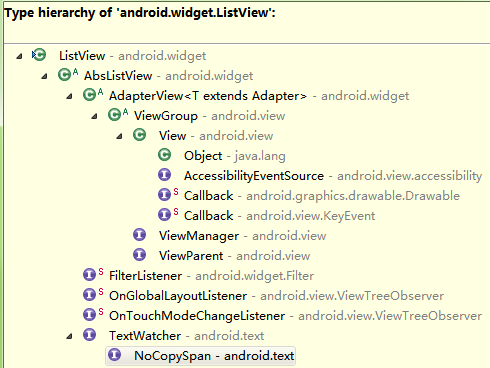
}

adapter.notifyDataSetChanged();

}

# ListView源码解析

## Listview继承结构



ListView的继承结构还是相当复杂的，它是直接继承自的AbsListView，而AbsListView有两个子实现类，一个是ListView，另一个就是GridView，因此我们从这一点就可以猜出来，ListView和GridView在工作原理和实现上都是有很多共同点的。

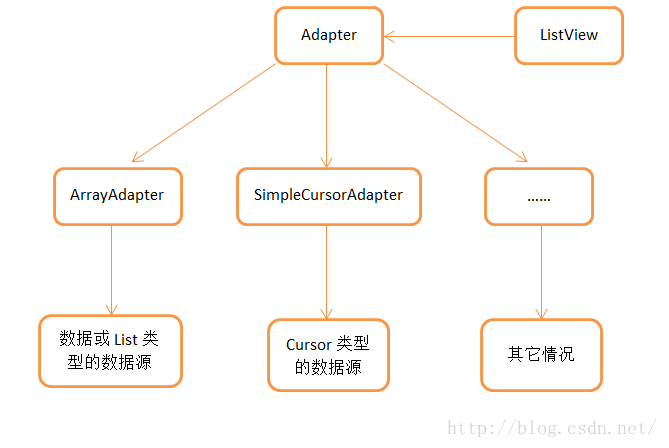
## Adapter的作用

Adapter相信大家都不会陌生，我们平时使用ListView的时候一定都会用到它。那么话说回来大家有没有仔细想过，为什么需要Adapter这个东西呢？总感觉正因为有了Adapter，ListView的使用变得要比其它控件复杂得多。那么这里我们就先来学习一下Adapter到底起到了什么样的一个作用。

其实说到底，控件就是为了交互和展示数据用的，只不过ListView更加特殊，它是为了展示很多很多数据用的，但是ListView只承担交互和展示工作而已，至于这些数据来自哪里，ListView是不关心的。因此，我们能设想到的最基本的ListView工作模式就是要有一个ListView控件和一个数据源。

不过如果真的让ListView和数据源直接打交道的话，那ListView所要做的适配工作就非常繁杂了。因为数据源这个概念太模糊了，我们只知道它包含了很多数据而已，至于这个数据源到底是什么样类型，并没有严格的定义，有可能是数组，也有可能是集合，甚至有可能是数据库表中查询出来的游标。所以说如果ListView真的去为每一种数据源都进行适配操作的话，一是扩展性会比较差，内置了几种适配就只有几种适配，不能动态进行添加。二是超出了它本身应该负责的工作范围，不再是仅仅承担交互和展示工作就可以了，这样ListView就会变得比较臃肿。

那么显然Android开发团队是不会允许这种事情发生的，于是就有了Adapter这样一个机制的出现。顾名思义，Adapter是适配器的意思，它在ListView和数据源之间起到了一个桥梁的作用，ListView并不会直接和数据源打交道，而是会借助Adapter这个桥梁来去访问真正的数据源，与之前不同的是，Adapter的接口都是统一的，因此ListView不用再去担心任何适配方面的问题。而Adapter又是一个接口(interface)，它可以去实现各种各样的子类，每个子类都能通过自己的逻辑来去完成特定的功能，以及与特定数据源的适配操作，比如说ArrayAdapter可以用于数组和List类型的数据源适配，SimpleCursorAdapter可以用于游标类型的数据源适配，这样就非常巧妙地把数据源适配困难的问题解决掉了，并且还拥有相当不错的扩展性。简单的原理示意图如下所示：



当然Adapter的作用不仅仅只有数据源适配这一点，还有一个非常非常重要的方法也需要我们在Adapter当中去重写，就是getView()方法

## RecycleBin机制

那么在开始分析ListView的源码之前，还有一个东西是我们提前需要了解的，就是RecycleBin机制，这个机制也是ListView能够实现成百上千条数据都不会OOM最重要的一个原因。其实RecycleBin的代码并不多，只有300行左右，它是写在AbsListView中的一个内部类，所以所有继承自AbsListView的子类，也就是ListView和GridView，都可以使用这个机制。那我们来看一下RecycleBin中的主要代码，这里的RecycleBin代码并不全，我只是把最主要的几个方法提了出来，如下所示：

**class** **RecycleBin** {

**private** RecyclerListener mRecyclerListener;

**private** **int** mFirstActivePosition;

**private** **View**[] mActiveViews = **new** **View**[0]; //存储View

**private** **ArrayList**<**View**>[] mScrapViews; //存储废弃View

**private** **int** mViewTypeCount;

**private** **ArrayList**<**View**> mCurrentScrap; //存储废弃View

/\*\*

\* fillActiveViews() 这个方法接收两个参数，第一个参数表示要存储的view的数量，第二个参数表示ListView中第一个可见元素的position值。

\* RecycleBin当中使用mActiveViews这个数组来存储View，调用这个方法后就会根据传入的参数来将ListView中的指定元素存储到mActiveViews数组当中。

\*/

**void** **fillActiveViews**(**int** childCount, **int** firstActivePosition) {

**if** (mActiveViews.length < childCount) {

mActiveViews = **new** **View**[childCount];

}

mFirstActivePosition = firstActivePosition;

**final** **View**[] activeViews = mActiveViews;

**for** (**int** i = 0; i < childCount; i++) {

**View** child = getChildAt(i);

**AbsListView**.**LayoutParams** lp = (**AbsListView**.**LayoutParams**) child.getLayoutParams();

**if** (lp != **null** && lp.viewType != ITEM\_VIEW\_TYPE\_HEADER\_OR\_FOOTER) {

activeViews[i] = child;

}

}

}

/\*\*

\* getActiveView() 这个方法和fillActiveViews()是对应的，用于从mActiveViews数组当中获取数据。该方法接收一个position参数，

\* 表示元素在ListView当中的位置，方法内部会自动将position值转换成mActiveViews数组对应的下标值。需要注意的是，mActiveViews当中所存储的View，

\* 一旦被获取了之后就会从mActiveViews当中移除，下次获取同样位置的View将会返回null，也就是说mActiveViews不能被重复利用

\*/

**View** **getActiveView**(**int** position) {

**int** index = position - mFirstActivePosition;

**final** **View**[] activeViews = mActiveViews;

**if** (index >= 0 && index < activeViews.length) {

**final** **View** match = activeViews[index];

activeViews[index] = **null**;

**return** match;

}

**return** **null**;

}

/\*\*

\* addScrapView() 用于将一个废弃的View进行缓存，该方法接收一个View参数，当有某个View确定要废弃掉的时候(比如滚动出了屏幕)，

\* 就应该调用这个方法来对View进行缓存，RecycleBin当中使用mScrapViews和mCurrentScrap这两个List来存储废弃View。

\*/

**void** **addScrapView**(**View** scrap) {

**AbsListView**.**LayoutParams** lp = (**AbsListView**.**LayoutParams**) scrap.getLayoutParams();

**if** (lp == **null**) {

**return**;

}

**int** viewType = lp.viewType;

**if** (!shouldRecycleViewType(viewType)) {

**if** (viewType != ITEM\_VIEW\_TYPE\_HEADER\_OR\_FOOTER) {

removeDetachedView(scrap, **false**);

}

**return**;

}

**if** (mViewTypeCount == 1) {

dispatchFinishTemporaryDetach(scrap);

mCurrentScrap.add(scrap);

} **else** {

dispatchFinishTemporaryDetach(scrap);

mScrapViews[viewType].add(scrap);

}

**if** (mRecyclerListener != **null**) {

mRecyclerListener.onMovedToScrapHeap(scrap);

}

}

/\*\*

\* getScrapView 用于从废弃缓存中取出一个View，这些废弃缓存中的View是没有顺序可言的，因此getScrapView()方法中的算法也非常简单，

\* 就是直接从mCurrentScrap当中获取尾部的一个scrap view进行返回。

\*/

**View** **getScrapView**(**int** position) {

**ArrayList**<**View**> scrapViews;

**if** (mViewTypeCount == 1) {

scrapViews = mCurrentScrap;

**int** size = scrapViews.size();

**if** (size > 0) {

**return** scrapViews.remove(size - 1);

} **else** {

**return** **null**;

}

} **else** {

**int** whichScrap = mAdapter.getItemViewType(position);

**if** (whichScrap >= 0 && whichScrap < mScrapViews.length) {

scrapViews = mScrapViews[whichScrap];

**int** size = scrapViews.size();

**if** (size > 0) {

**return** scrapViews.remove(size - 1);

}

}

}

**return** **null**;

}

/\*\*

\* setViewTypeCount() 我们都知道Adapter当中可以重写一个getViewTypeCount()来表示ListView中有几种类型的数据项，

\* 而setViewTypeCount()方法的作用就是为每种类型的数据项都单独启用一个RecycleBin缓存机制。实际上，

\* getViewTypeCount()方法通常情况下使用的并不是很多，所以我们只要知道RecycleBin当中有这样一个功能就行了。

\*/

**public** **void** **setViewTypeCount**(**int** viewTypeCount) {

**if** (viewTypeCount < 1) {

**throw** **new** IllegalArgumentException("Can't have a viewTypeCount < 1");

}

**ArrayList**<**View**>[] scrapViews = **new** **ArrayList**[viewTypeCount];

**for** (**int** i = 0; i < viewTypeCount; i++) {

scrapViews[i] = **new** ArrayList<**View**>();

}

mViewTypeCount = viewTypeCount;

mCurrentScrap = scrapViews[0];

mScrapViews = scrapViews;

}

}

## 第一次Layout

不管怎么说，ListView即使再特殊最终还是继承自View的，因此它的执行流程还将会按照View的规则来执行，View的执行流程无非就分为三步，onMeasure()用于测量View的大小，onLayout()用于确定View的布局，onDraw()用于将View绘制到界面上。而在ListView当中，onMeasure()并没有什么特殊的地方，因为它终归是一个View，占用的空间最多并且通常也就是整个屏幕。onDraw()在ListView当中也没有什么意义，因为ListView本身并不负责绘制，而是由ListView当中的子元素来进行绘制的。那么ListView大部分的神奇功能其实都是在onLayout()方法中进行的了，因此我们本篇文章也是主要分析的这个方法里的内容。如果你到ListView源码中去找一找，你会发现ListView中是没有onLayout()这个方法的，这是因为这个方法是在ListView的父类AbsListView中实现的，代码如下所示：

@Override

protected void onLayout(boolean changed, int l, int t, int r, int b) {

super.onLayout(changed, l, t, r, b);

mInLayout = true;

if (changed) {

int childCount = getChildCount();

for (int i = 0; i < childCount; i++) {

getChildAt(i).forceLayout();

}

mRecycler.markChildrenDirty();

}

layoutChildren();

mInLayout = false;

}

如果ListView的大小或者位置发生了变化，那么changed变量就会变成true，此时会要求所有的子布局都强制进行重绘。除此之外倒没有什么难理解的地方了，不过我们注意到，在第16行调用了layoutChildren()这个方法，从方法名上我们就可以猜出这个方法是用来进行子元素布局的，不过进入到这个方法当中你会发现这是个空方法，没有一行代码。这当然是可以理解的了，因为子元素的布局应该是由具体的实现类来负责完成的，而不是由父类完成。那么进入ListView的layoutChildren()方法，代码如下所示：

。。。。。。。

这段代码比较长，我们挑重点的看。首先可以确定的是，ListView当中目前还没有任何子View，数据都还是由Adapter管理的，并没有展示到界面上，因此第19行getChildCount()方法得到的值肯定是0。接着在第81行会根据dataChanged这个布尔型的值来判断执行逻辑，dataChanged只有在数据源发生改变的情况下才会变成true，其它情况都是false，因此这里会进入到第90行的执行逻辑，调用RecycleBin的fillActiveViews()方法。按理来说，调用fillActiveViews()方法是为了将ListView的子View进行缓存的，可是目前ListView中还没有任何的子View，因此这一行暂时还起不了任何作用。

接下来在第114行会根据mLayoutMode的值来决定布局模式，默认情况下都是普通模式LAYOUT\_NORMAL，因此会进入到第140行的default语句当中。而下面又会紧接着进行两次if判断，childCount目前是等于0的，并且默认的布局顺序是从上往下，因此会进入到第145行的fillFromTop()方法，我们跟进去瞧一瞧：

private View fillFromTop(int nextTop) {

mFirstPosition = Math.min(mFirstPosition, mSelectedPosition);

mFirstPosition = Math.min(mFirstPosition, mItemCount - 1);

if (mFirstPosition < 0) {

mFirstPosition = 0;

}

return fillDown(mFirstPosition, nextTop);

}

从这个方法的注释中可以看出，它所负责的主要任务就是从mFirstPosition开始，自顶至底去填充ListView。而这个方法本身并没有什么逻辑，就是判断了一下mFirstPosition值的合法性，然后调用fillDown()方法，那么我们就有理由可以猜测，填充ListView的操作是在fillDown()方法中完成的。进入fillDown()方法，代码如下所示：

private View fillDown(int pos, int nextTop) {

View selectedView = null;

int end = (getBottom() - getTop()) - mListPadding.bottom;

while (nextTop < end && pos < mItemCount) {

// is this the selected item?

boolean selected = pos == mSelectedPosition;

View child = makeAndAddView(pos, nextTop, true, mListPadding.left, selected);

nextTop = child.getBottom() + mDividerHeight;

if (selected) {

selectedView = child;

}

pos++;

}

return selectedView;

}

可以看到，这里使用了一个while循环来执行重复逻辑，一开始nextTop的值是第一个子元素顶部距离整个ListView顶部的像素值，pos则是刚刚传入的mFirstPosition的值，而end是ListView底部减去顶部所得的像素值，mItemCount则是Adapter中的元素数量。因此一开始的情况下nextTop必定是小于end值的，并且pos也是小于mItemCount值的。那么每执行一次while循环，pos的值都会加1，并且nextTop也会增加，当nextTop大于等于end时，也就是子元素已经超出当前屏幕了，或者pos大于等于mItemCount时，也就是所有Adapter中的元素都被遍历结束了，就会跳出while循环。那么while循环当中又做了什么事情呢？值得让人留意的就是第18行调用的makeAndAddView()方法，进入到这个方法当中，代码如下所示：

private View makeAndAddView(int position, int y, boolean flow, int childrenLeft,

        boolean selected) {

    View child;

    if (!mDataChanged) {

        // Try to use an exsiting view for this position

        child = mRecycler.getActiveView(position);

        if (child != null) {

            // Found it -- we're using an existing child

            // This just needs to be positioned

            setupChild(child, position, y, flow, childrenLeft, selected, true);

            return child;

        }

    }

    // Make a new view for this position, or convert an unused view if possible

    child = obtainView(position, mIsScrap);

    // This needs to be positioned and measured

    setupChild(child, position, y, flow, childrenLeft, selected, mIsScrap[0]);

    return child;

}

这里在第19行尝试从RecycleBin当中快速获取一个active view，不过很遗憾的是目前RecycleBin当中还没有缓存任何的View，所以这里得到的值肯定是null。那么取得了null之后就会继续向下运行，到第28行会调用obtainView()方法来再次尝试获取一个View，这次的obtainView()方法是可以保证一定返回一个View的，于是下面立刻将获取到的View传入到了setupChild()方法当中。那么obtainView()内部到底是怎么工作的呢？我们先进入到这个方法里面看一下：

View obtainView(int position, boolean[] isScrap) {

    isScrap[0] = false;

    View scrapView;

    scrapView = mRecycler.getScrapView(position);

    View child;

    if (scrapView != null) {

        child = mAdapter.getView(position, scrapView, this);

        if (child != scrapView) {

            mRecycler.addScrapView(scrapView);

            if (mCacheColorHint != 0) {

                child.setDrawingCacheBackgroundColor(mCacheColorHint);

            }

        } else {

            isScrap[0] = true;

            dispatchFinishTemporaryDetach(child);

        }

    } else {

        child = mAdapter.getView(position, null, this);

        if (mCacheColorHint != 0) {

            child.setDrawingCacheBackgroundColor(mCacheColorHint);

        }

    }

    return child;

}

obtainView()方法中的代码并不多，但却包含了非常非常重要的逻辑，不夸张的说，整个ListView中最重要的内容可能就在这个方法里了。那么我们还是按照执行流程来看，在第19行代码中调用了RecycleBin的getScrapView()方法来尝试获取一个废弃缓存中的View，同样的道理，这里肯定是获取不到的，getScrapView()方法会返回一个null。这时该怎么办呢？没有关系，代码会执行到第33行，调用mAdapter的getView()方法来去获取一个View。那么mAdapter是什么呢？当然就是当前ListView关联的适配器了。而getView()方法又是什么呢？还用说吗，这个就是我们平时使用ListView时最最经常重写的一个方法了,View作为obtainView()的结果进行返回，并最终传入到setupChild()方法当中,那么我们继续往下看：

。。。。。。。。。。。

setupChild()方法当中的代码虽然比较多，但是我们只看核心代码的话就非常简单了，刚才调用obtainView()方法获取到的子元素View，这里在第40行调用了addViewInLayout()方法将它添加到了ListView当中。那么根据fillDown()方法中的while循环，会让子元素View将整个ListView控件填满然后就跳出，也就是说即使我们的Adapter中有一千条数据，ListView也只会加载第一屏的数据，剩下的数据反正目前在屏幕上也看不到，所以不会去做多余的加载工作，这样就可以保证ListView中的内容能够迅速展示到屏幕上。

那么到此为止，第一次Layout过程结束。

## 第二次Layout

虽然我在源码中并没有找出具体的原因，但如果你自己做一下实验的话就会发现，即使是一个再简单的View，在展示到界面上之前都会经历至少两次onMeasure()和两次onLayout()的过程。其实这只是一个很小的细节，平时对我们影响并不大，因为不管是onMeasure()或者onLayout()几次，反正都是执行的相同的逻辑，我们并不需要进行过多关心。但是在ListView中情况就不一样了，因为这就意味着layoutChildren()过程会执行两次，而这个过程当中涉及到向ListView中添加子元素，如果相同的逻辑执行两遍的话，那么ListView中就会存在一份重复的数据了。因此ListView在layoutChildren()过程当中做了第二次Layout的逻辑处理，非常巧妙地解决了这个问题，下面我们就来分析一下第二次Layout的过程。其实第二次Layout和第一次Layout的基本流程是差不多的，那么我们还是从layoutChildren()方法开始看起：

。。。。。。

同样还是在第19行，调用getChildCount()方法来获取子View的数量，只不过现在得到的值不会再是0了，而是ListView中一屏可以显示的子View数量，因为我们刚刚在第一次Layout过程当中向ListView添加了这么多的子View。下面在第90行调用了RecycleBin的fillActiveViews()方法，这次效果可就不一样了，因为目前ListView中已经有子View了，这样所有的子View都会被缓存到RecycleBin的mActiveViews数组当中，后面将会用到它们。

接下来将会是非常非常重要的一个操作，在第113行调用了detachAllViewsFromParent()方法。这个方法会将所有ListView当中的子View全部清除掉，从而保证第二次Layout过程不会产生一份重复的数据。那有的朋友可能会问了，这样把已经加载好的View又清除掉，待会还要再重新加载一遍，这不是严重影响效率吗？不用担心，还记得我们刚刚调用了RecycleBin的fillActiveViews()方法来缓存子View吗，待会儿将会直接使用这些缓存好的View来进行加载，而并不会重新执行一遍inflate过程，因此效率方面并不会有什么明显的影响。

那么我们接着看，在第141行的判断逻辑当中，由于不再等于0了，因此会进入到else语句当中。而else语句中又有三个逻辑判断，第一个逻辑判断不成立，因为默认情况下我们没有选中任何子元素，mSelectedPosition应该等于-1。第二个逻辑判断通常是成立的，因为mFirstPosition的值一开始是等于0的，只要adapter中的数据大于0条件就成立。那么进入到fillSpecific()方法当中，代码如下所示：

。。。。。。

fillSpecific()这算是一个新方法了，不过其实它和fillUp()、fillDown()方法功能也是差不多的，主要的区别在于，fillSpecific()方法会优先将指定位置的子View先加载到屏幕上，然后再加载该子View往上以及往下的其它子View。那么由于这里我们传入的position就是第一个子View的位置，于是fillSpecific()方法的作用就基本上和fillDown()方法是差不多的了，这里我们就不去关注太多它的细节，而是将精力放在makeAndAddView()方法上面。再次回到makeAndAddView()方法，代码如下所示：

。。。。。。

仍然还是在第19行尝试从RecycleBin当中获取Active View，然而这次就一定可以获取到了，因为前面我们调用了RecycleBin的fillActiveViews()方法来缓存子View。那么既然如此，就不会再进入到第28行的obtainView()方法，而是会直接进入setupChild()方法当中，这样也省去了很多时间，因为如果在obtainView()方法中又要去infalte布局的话，那么ListView的初始加载效率就大大降低了。注意在第23行，setupChild()方法的最后一个参数传入的是true，这个参数表明当前的View是之前被回收过的，那么我们再次回到setupChild()

。。。。。。

可以看到，setupChild()方法的最后一个参数是recycled，然后在第32行会对这个变量进行判断，由于recycled现在是true，所以会执行attachViewToParent()方法，而第一次Layout过程则是执行的else语句中的addViewInLayout()方法。这两个方法最大的区别在于，如果我们需要向ViewGroup中添加一个新的子View，应该调用addViewInLayout()方法，而如果是想要将一个之前detach的View重新attach到ViewGroup上，就应该调用attachViewToParent()方法。那么由于前面在layoutChildren()方法当中调用了detachAllViewsFromParent()方法，这样ListView中所有的子View都是处于detach状态的，所以这里attachViewToParent()方法是正确的选择。经历了这样一个detach又attach的过程，ListView中所有的子View又都可以正常显示出来了，那么第二次Layout过程结束。

## 滑动加载更多数据

经历了两次Layout过程，虽说我们已经可以在ListView中看到内容了，然而关于ListView最神奇的部分我们却还没有接触到，因为目前ListView中只是加载并显示了第一屏的数据而已。比如说我们的Adapter当中有1000条数据，但是第一屏只显示了10条，ListView中也只有10个子View而已，那么剩下的990是怎样工作并显示到界面上的呢？这就要看一下ListView滑动部分的源码了，因为我们是通过手指滑动来显示更多数据的。

由于滑动部分的机制是属于通用型的，即ListView和GridView都会使用同样的机制，因此这部分代码就肯定是写在AbsListView当中的了。那么监听触控事件是在onTouchEvent()方法当中进行的，我们就来看一下AbsListView中的这个方法：

。。。。。。。。。。

这个方法中的代码就非常多了，因为它所处理的逻辑也非常多，要监听各种各样的触屏事件。但是我们目前所关心的就只有手指在屏幕上滑动这一个事件而已，对应的是ACTION\_MOVE这个动作，那么我们就只看这部分代码就可以了。

可以看到，ACTION\_MOVE这个case里面又嵌套了一个switch语句，是根据当前的TouchMode来选择的。那这里我可以直接告诉大家，当手指在屏幕上滑动时，TouchMode是等于TOUCH\_MODE\_SCROLL这个值的，至于为什么那又要牵扯到另外的好几个方法，这里限于篇幅原因就不再展开讲解了，喜欢寻根究底的朋友们可以自己去源码里找一找原因。

这样的话，代码就应该会走到第78行的这个case里面去了，在这个case当中并没有什么太多需要注意的东西，唯一一点非常重要的就是第92行调用的trackMotionScroll()方法，相当于我们手指只要在屏幕上稍微有一点点移动，这个方法就会被调用，而如果是正常在屏幕上滑动的话，那么这个方法就会被调用很多次。那么我们进入到这个方法中瞧一瞧，代码如下所示：

。。。。。。。。。。

这个方法接收两个参数，deltaY表示从手指按下时的位置到当前手指位置的距离，incrementalDeltaY则表示据上次触发event事件手指在Y方向上位置的改变量，那么其实我们就可以通过incrementalDeltaY的正负值情况来判断用户是向上还是向下滑动的了。如第34行代码所示，如果incrementalDeltaY小于0，说明是向下滑动，否则就是向上滑动。

下面将会进行一个边界值检测的过程，可以看到，从第43行开始，当ListView向下滑动的时候，就会进入一个for循环当中，从上往下依次获取子View，第47行当中，如果该子View的bottom值已经小于top值了，就说明这个子View已经移出屏幕了，所以会调用RecycleBin的addScrapView()方法将这个View加入到废弃缓存当中，并将count计数器加1，计数器用于记录有多少个子View被移出了屏幕。那么如果是ListView向上滑动的话，其实过程是基本相同的，只不过变成了从下往上依次获取子View，然后判断该子View的top值是不是大于bottom值了，如果大于的话说明子View已经移出了屏幕，同样把它加入到废弃缓存中，并将计数器加1。

接下来在第76行，会根据当前计数器的值来进行一个detach操作，它的作用就是把所有移出屏幕的子View全部detach掉，在ListView的概念当中，所有看不到的View就没有必要为它进行保存，因为屏幕外还有成百上千条数据等着显示呢，一个好的回收策略才能保证ListView的高性能和高效率。紧接着在第78行调用了offsetChildrenTopAndBottom()方法，并将incrementalDeltaY作为参数传入，这个方法的作用是让ListView中所有的子View都按照传入的参数值进行相应的偏移，这样就实现了随着手指的拖动，ListView的内容也会随着滚动的效果。

然后在第84行会进行判断，如果ListView中最后一个View的底部已经移入了屏幕，或者ListView中第一个View的顶部移入了屏幕，就会调用fillGap()方法，那么因此我们就可以猜出fillGap()方法是用来加载屏幕外数据的，进入到这个方法中瞧一瞧，如下所示：

abstract void fillGap(boolean down);

OK，AbsListView中的fillGap()是一个抽象方法，那么我们立刻就能够想到，它的具体实现肯定是在ListView中完成的了。回到ListView当中，fillGap()方法的代码如下所示：

。。。。。。。。。。

down参数用于表示ListView是向下滑动还是向上滑动的，可以看到，如果是向下滑动的话就会调用fillDown()方法，而如果是向上滑动的话就会调用fillUp()方法。那么这两个方法我们都已经非常熟悉了，内部都是通过一个循环来去对ListView进行填充，所以这两个方法我们就不看了，但是填充ListView会通过调用makeAndAddView()方法来完成，又是makeAndAddView()方法，但这次的逻辑再次不同了，所以我们还是回到这个方法瞧一瞧：

。。。。。。。。。。

不管怎么说，这里首先仍然是会尝试调用RecycleBin的getActiveView()方法来获取子布局，只不过肯定是获取不到的了，因为在第二次Layout过程中我们已经从mActiveViews中获取过了数据，而根据RecycleBin的机制，mActiveViews是不能够重复利用的，因此这里返回的值肯定是null。

既然getActiveView()方法返回的值是null，那么就还是会走到第28行的obtainView()方法当中，代码如下所示：

。。。。。。。。。。

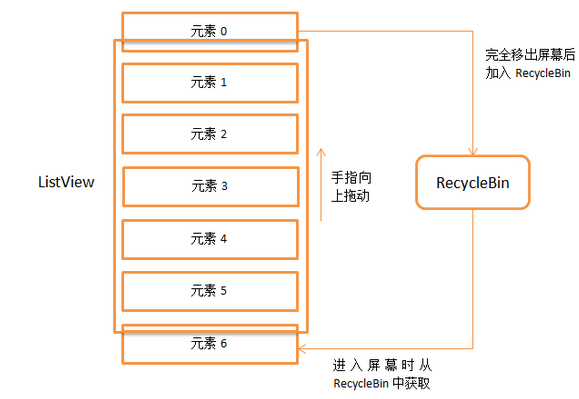
这里在第19行会调用RecyleBin的getScrapView()方法来尝试从废弃缓存中获取一个View，那么废弃缓存有没有View呢？当然有，因为刚才在trackMotionScroll()方法中我们就已经看到了，一旦有任何子View被移出了屏幕，就会将它加入到废弃缓存中，而从obtainView()方法中的逻辑来看，一旦有新的数据需要显示到屏幕上，就会尝试从废弃缓存中获取View。所以它们之间就形成了一个生产者和消费者的模式，那么ListView神奇的地方也就在这里体现出来了，不管你有任意多条数据需要显示，ListView中的子View其实来来回回就那么几个，移出屏幕的子View会很快被移入屏幕的数据重新利用起来，因而不管我们加载多少数据都不会出现OOM的情况，甚至内存都不会有所增加。

那么另外还有一点是需要大家留意的，这里获取到了一个scrapView，然后我们在第22行将它作为第二个参数传入到了Adapter的getView()方法当中。那么第二个参数是什么意思呢？我们再次看一下一个简单的getView()方法示例：

。。。。。。。。。。

第二个参数就是我们最熟悉的convertView呀，难怪平时我们在写getView()方法是要判断一下convertView是不是等于null，如果等于null才调用inflate()方法来加载布局，不等于null就可以直接利用convertView，因为convertView就是我们之间利用过的View，只不过被移出屏幕后进入到了废弃缓存中，现在又重新拿出来使用而已。然后我们只需要把convertView中的数据更新成当前位置上应该显示的数据，那么看起来就好像是全新加载出来的一个布局一样之后的代码又都是我们熟悉的流程了，从缓存中拿到子View之后再调用setupChild()方法将它重新attach到ListView当中，因为缓存中的View也是之前从ListView中detach掉的，这部分代码就不再重复进行分析了。

为了方便大家理解，这里我再附上一张图解说明：



# List、Grid和ExpandableListView

## ListView定位到某一个item上

mListView.setSelection(mData.size())//定位到最后一行

## GridView属性

android:horizontalSpacing="5dp" 水平间距

android:numColumns="2" 一行两项

android:gravity 设置此组件中的内容在组件中的位置

android:stretchMode="columnWidth" 缩放模式

android:verticalSpacing="5dp" 竖直间距

**Gridview在高版本的sdk中有gravity属性，设置android:gravity="center\_horizontal"才能居中**

## Item里Buttom会抢占焦点，导致item不能点击的问题解决

原因：listview里有判断，孩子不为空，并且有焦点时，不能触发点击事件

解决方案一：

Buttom设置 android:focusable="false"

解决方案二：

Buttom的容器里设置android:descendantFocusability="blocksDescendants"

beforeDescendants：viewgroup会优先其子类控件而获取到焦点

afterDescendants：viewgroup只有当其子类控件不需要获取焦点时才获取焦点

blocksDescendants：viewgroup会覆盖子类控件而直接获得焦点

## 有时候Item里图片等view很多，滚动时会很卡，解决方法：

思路：ListView滑动时，取消所有加载项；ListView滑动停止后，加载可见项

## ExpandableListView

**public** **class** **ExpandableAdapter** **extends** **BaseExpandableListAdapter**{

**Activity** activity;

**public** **ExpandableAdapter**(**Activity** a){

activity = a;

}

@Override

**public** **int** **getGroupCount**() {

**return** groupArray.size();

}

@Override

**public** **int** **getChildrenCount**(**int** groupPosition) {

**return** childArray.get(groupPosition).size();

}

@Override

**public** **Object** **getGroup**(**int** groupPosition) {

**return** groupArray.get(groupPosition);

}

@Override

**public** **Object** **getChild**(**int** groupPosition, **int** childPosition) {

**return** childArray.get(groupPosition).get(childPosition);

}

@Override

**public** **long** **getGroupId**(**int** groupPosition) {

**return** groupPosition;

}

@Override

**public** **long** **getChildId**(**int** groupPosition, **int** childPosition) {

**return** childPosition;

}

@Override

**public** **View** **getGroupView**(**int** groupPosition, **boolean** isExpanded,

**View** convertView, **ViewGroup** parent) {

**String** str = groupArray.get(groupPosition);

**return** getGenericView(str);

}

@Override

**public** **View** **getChildView**(**int** groupPosition, **int** childPosition,

**boolean** isLastChild, **View** convertView, **ViewGroup** parent) {

**String** str = childArray.get(groupPosition).get(childPosition);

**return** getGenericView(str);

}

@Override

**public** **boolean** **isChildSelectable**(**int** groupPosition, **int** childPosition) {

**return** **true**;

}

@Override

**public** **boolean** **hasStableIds**() {

**return** **false**;

}

**public** **TextView** **getGenericView**(**String** string){

**AbsListView**.**LayoutParams** layoutParams = **new** **AbsListView**.LayoutParams(

**ViewGroup**.**LayoutParams**.**FILL\_PARENT**, 64 );

**TextView** text = **new** TextView(activity);

text.setLayoutParams(layoutParams);

text.setGravity(**Gravity**.**CENTER\_VERTICAL** | **Gravity**.**LEFT**);

text.setPadding(36 , 0 , 0 , 0 );

text.setText(string);

**return** text;

}

}

}

# 可拖动排序的GridView

## 引入拖动排序控件

//拖动排序gridView  
implementation 'com.huxq17.handygridview:handygridview:1.2.0'

## fragment\_main\_seek\_watch.xml

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:background="@color/theme\_gray\_color">  
 <com.huxq17.handygridview.HandyGridView  
 android:id="@+id/gv"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:numColumns="3"  
 android:scrollbars="vertical"  
 tools:listitem="@layout/item\_main\_seek\_watch" />  
 <TextView  
 android:id="@+id/tv\_bottom\_ok"  
 style="@style/btn\_314\_44\_back\_blue\_textff16"  
 android:layout\_alignParentBottom="true"  
 android:layout\_marginBottom="20dp"  
 android:text="编辑完成"  
 android:visibility="gone" />  
</RelativeLayout>

## item\_main\_seek\_watch.xml

<?xml version="1.0" encoding="utf-8"?>  
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:paddingTop="5dp"  
 android:paddingBottom="5dp">  
 <LinearLayout  
 android:id="@+id/ll\_container"  
 android:layout\_width="105dp"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="bottom|center\_horizontal"  
 android:layout\_marginTop="10dp"  
 android:background="@drawable/selector\_press\_white\_gray\_r5"  
 android:orientation="vertical">  
 <ImageView  
 android:id="@+id/img\_logo"  
 android:layout\_width="105dp"  
 android:layout\_height="105dp"  
 tools:src="@drawable/url\_default\_icon" />  
 </LinearLayout>  
 <LinearLayout  
 android:layout\_width="120dp"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="center\_horizontal"  
 android:orientation="vertical">  
 <ImageView  
 android:id="@+id/img\_delete"  
 android:layout\_width="30dp"  
 android:layout\_height="30dp"  
 android:layout\_alignParentRight="true"  
 android:layout\_gravity="right"  
 android:background="?selectableItemBackgroundBorderless"  
 android:padding="3dp"  
 android:src="@drawable/discover\_unselect" />  
 </LinearLayout>  
</FrameLayout>

## GridSeekWatchAdapter

public class GridSeekWatchAdapter extends BaseAdapter implements OnItemMovedListener {  
 private boolean inEditMode = false;  
 private Context context;  
 private List<MainSeekWatchListItemBean> mListDatas;  
 private List<String> mDeleteDatas;  
 public GridSeekWatchAdapter(Context context, List<MainSeekWatchListItemBean> list, List<String> mDeleteDatas) {  
 this.context = context;  
 this.mListDatas = list;  
 this.mDeleteDatas = mDeleteDatas;  
 }  
 public void setInEditMode(boolean inEditMode) {  
 this.inEditMode = inEditMode;  
 notifyDataSetChanged();  
 }  
 @Override  
 public int getCount() {  
 return mListDatas.size();  
 }  
 @Override  
 public MainSeekWatchListItemBean getItem(int position) {  
 return mListDatas.get(position);  
 }  
 @Override  
 public long getItemId(int position) {  
 return position;  
 }  
 @Override  
 public View getView(final int position, View convertView, ViewGroup parent) {  
 ViewHolder holder;  
 if (convertView == null) {  
 holder = new ViewHolder();  
 convertView = LayoutInflater.*from*(context).inflate(R.layout.*item\_main\_seek\_watch*, null);  
 holder.img\_delete = convertView.findViewById(R.id.*img\_delete*);  
 convertView.setTag(holder);  
 } else {  
 holder = (ViewHolder) convertView.getTag();  
 }  
 MainSeekWatchListItemBean item = getItem(position);  
 if (item != null) {  
 GlideLoadUtils.*getInstance*().glideLoadTopRound(context, item.getLogo(), holder.img\_logo, R.drawable.*url\_default\_icon*);  
 holder.img\_delete.setVisibility(inEditMode ? View.*VISIBLE* : View.*INVISIBLE*);  
 holder.img\_delete.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 mDeleteDatas.add(mListDatas.get(position).getFocuId());//保存删除的id  
 mListDatas.remove(position);  
 notifyDataSetChanged();  
 }  
 });  
 }  
 return convertView;  
 }  
 @Override  
 public void onItemMoved(int from, int to) {  
 MainSeekWatchListItemBean bean = mListDatas.remove(from);  
 mListDatas.add(to, bean);  
 }  
 @Override  
 public boolean isFixed(int position) {  
 //return position == 0; //When postion==0,the item can not be dragged.  
 return false;  
 }  
 class ViewHolder {  
 ImageView img\_logo;  
 View img\_delete;  
 }  
}

## MainSeekWatchFragment

class MainSeekWatchFragment : BaseFragment() {  
 override fun initView() = R.layout.*fragment\_main\_seek\_watch* override fun initData() {  
 //initData会调用多次的处理办法  
 if (mListDatas != null) return  
 index = *arguments*?.getString(KEY\_STR1) ?: ""  
 slideTabTitles = *arguments*?.getParcelableArrayList(KEY\_STR2)  
 mListDatas = ArrayList()  
 initGridView()  
 //退出编辑,调用接口，然后刷新数据  
 tv\_bottom\_ok.setOnClickListener **{** completeEdit() **}** }  
 //在不切换fragment情况下，onResume情况不会触发setUserVisibleHint，所以需要在onResume中容错  
 override fun setUserVisibleHint(isVisibleToUser: Boolean) {  
 super.setUserVisibleHint(isVisibleToUser)  
 if (!index.*isNullOrBlank*()) {  
 if (gv.*mode* == HandyGridView.MODE.TOUCH) return//编辑模式不刷新  
 if (isVisibleToUser) {  
 getListDatas()  
 //如果setUserVisibleHint中刷了，onResume中不要再刷  
 isUserVisibleHintFresh = true  
 //setUserVisibleHint永远会优先于onResume回调找不到上级分发方法了，只能300毫秒后自动重置isUserVisibleHintFresh  
 Handler().postDelayed(**{** isUserVisibleHintFresh = false **}**, 300)  
 }  
 }  
 }  
 override fun onResume() {  
 super.onResume()  
 if (gv.*mode* == HandyGridView.MODE.TOUCH) return//编辑模式不刷新  
 //如果setUserVisibleHint中没有刷，onResume中刷  
 if (!isUserVisibleHintFresh) getListDatas()  
 }  
  
 */\*\*  
 \* 初始化列表  
 \*/* private fun initGridView() {  
 mAdapter = GridSeekWatchAdapter(*context*, mListDatas, mDeleteDatas)  
 gv.*apply* **{** *adapter* = mAdapter  
 setAutoOptimize(true)//自动优化，在屏幕可以滚动的时候，固定为长按模式  
 *mode* = HandyGridView.MODE.LONG\_PRESS  
 setScrollSpeed(1200)// 每秒移动的像素点个数，默认750，可不设置  
 //点击事件  
 setOnItemClickListener **{** parent, view, position, id **->** Toast.makeText(*context*, mListDatas!![position].type, Toast.*LENGTH\_SHORT*).show()  
 **}** //长按事件  
 setOnItemLongClickListener **{** parent, view, position, id **->** if (gv.*isLongPressMode*) {//long press enter edit mode.  
 setGridMode(HandyGridView.MODE.TOUCH)//长按后，进入编辑状态，变为触摸即可拖动模式  
 true  
 }  
 false  
 **}** //拖动时item变大变变小  
 setOnItemCapturedListener(object : OnItemCapturedListener {  
 override fun onItemCaptured(v: View, position: Int) {  
 v.*scaleX* = 1.2f  
 v.*scaleY* = 1.2f  
 }  
 override fun onItemReleased(v: View, position: Int) {  
 v.*scaleX* = 1f  
 v.*scaleY* = 1f  
 }  
 })  
 **}** }  
 */\*\*  
 \* 设置列表的编辑模式，控制编辑时不可切换  
 \*/* private fun setGridMode(mode: HandyGridView.MODE) {  
 gv.*mode* = mode  
 mAdapter.setInEditMode(mode == HandyGridView.MODE.TOUCH)  
 //编辑模式下，显示编辑完成按钮，禁止其他tab点击，禁止viewpageer滑动,禁止item点击事件  
 val pf = *parentFragment* if (mode == HandyGridView.MODE.TOUCH) {  
 tv\_bottom\_ok.*visibility* = View.*VISIBLE* gv.*onItemClickListener* = null  
 pf?.*let* **{** if (pf is MainSeekFragment) pf.dealMainSeekWatchFragmentEdit(true) **}** } else {  
 tv\_bottom\_ok.*visibility* = View.*GONE* gv.setOnItemClickListener **{** parent, view, position, id **->** Toast.makeText(*context*, mListDatas!![position].type, Toast.*LENGTH\_SHORT*).show()  
 **}** pf?.*let* **{** if (pf is MainSeekFragment) pf.dealMainSeekWatchFragmentEdit(false) **}** }  
 }  
 */\*\*  
 \* 编辑完成  
 \*/* fun completeEdit() {  
 setGridMode(HandyGridView.MODE.LONG\_PRESS)  
 val deleteArray = mDeleteDatas.*toTypedArray*()  
 val updateArray = mListDatas!!.*map* **{ it**.focuId **}**.*toTypedArray*()  
 ApiRequest.discoveryFocusUpdate(*context*, object : OnNetWorkBackListener<JustCodeAndMsgInfo> {  
 override fun Success(result: JustCodeAndMsgInfo?) {  
 if (result != null) {  
 mHandler.post **{** dismissLoadingDialog()  
 getListDatas()//成功也获取数据  
 **}** }  
 }  
 override fun Failure(obj: Any?) {  
 dismissLoadingDialog()  
 ToastUtils.showShort(obj?.toString() ?: "未知错误")  
 getListDatas()//错误时，重新获取数据  
 }  
 override fun otherData(result: String?) {  
 dismissLoadingDialog()  
 ToastUtils.showShort(result ?: "未知错误")  
 getListDatas()//错误时，重新获取数据  
 }  
 }, deleteArray, updateArray)  
 }  
 */\*\*  
 \* 获取数据  
 \*/* private fun getListDatas() {  
 if (false) {  
 mListDatas!!.clear()  
 mDeleteDatas.clear()  
 mListDatas!!.addAll(DemoUtil.getMainSeekWatchList())  
 mAdapter.notifyDataSetChanged()  
 return  
 }  
 showLoadingDialog()  
 ApiRequest.discoveryFocus(*context*, object : OnNetWorkBackListener<DiscoveryFocusInfo> {  
 override fun Success(result: DiscoveryFocusInfo?) {  
 if (result != null) {  
 mHandler.post **{** dismissLoadingDialog()  
 mDeleteDatas.clear()  
 mListDatas!!.clear()  
 mListDatas!!.addAll(result.*data*.*records*.*map* **{** val typeName = slideTabTitles?.*find* **{** it2 **-> it**.*appType* == it2.*type* **}**?.*name* ?: ""  
MainSeekWatchListItemBean(typeName, **it**.*appImg*, **it**.*appName*, **it**.*appUrl*, **it**.*focusId*)  
 **}**)  
 mAdapter.notifyDataSetChanged()  
 **}** }  
 }  
 override fun Failure(obj: Any?) {  
 dismissLoadingDialog()  
 ToastUtils.showShort(obj?.toString() ?: "未知错误")  
 }  
 override fun otherData(result: String?) {  
 dismissLoadingDialog()  
 ToastUtils.showShort(result ?: "未知错误")  
 }  
 })  
 }  
 //region declare variable  
 private val mHandler = Handler()  
 private var mListDatas: ArrayList<MainSeekWatchListItemBean>? = null  
 private var mDeleteDatas: ArrayList<String> = ArrayList()  
 private lateinit var mAdapter: GridSeekWatchAdapter  
 private var isUserVisibleHintFresh = false  
 private var index: String? = null  
 private var slideTabTitles: List<DiscoveryTitlesInfo.DataBean>? = null  
 companion object {  
 private const val KEY\_STR1 = "str1Key"  
 private const val KEY\_STR2 = "str2Key"  
 fun newInstance(index: String = "", slideTabTitles: ArrayList<DiscoveryTitlesInfo.DataBean>): MainSeekWatchFragment {  
 val fragment = MainSeekWatchFragment()  
 val bundle = Bundle()  
 bundle.*apply* **{** putString(KEY\_STR1, index)  
 putParcelableArrayList(KEY\_STR2, slideTabTitles)  
 **}** fragment.*arguments* = bundle  
 return fragment  
 }  
 }  
 //endregion  
}

# RecycleView

## 基本使用

### 对比ListView

有了ListView、GridView为什么还要RecyclerView这样的控件呢？整体上看RecyclerView架构，提供了一种插拔式的体验，高度的解耦，异常的灵活，通过设置它提供的不同LayoutManager，ItemDecoration , ItemAnimator实现令人瞠目的效果。

* 你想要控制其显示的方式，请通过布局管理器LayoutManager
* 你想要控制Item间的间隔（可绘制），请通过ItemDecoration
* 你想要控制Item增删的动画，请通过ItemAnimator
* 你想要控制点击、长按事件，请自己写（擦，这点尼玛。）

鉴于我们对于ListView的使用特别的熟悉，对比下RecyclerView的使用代码：

mRecyclerView = findView(R.id.id\_recyclerview);//初始化

mRecyclerView.setLayoutManager([LayoutManager](eclipse-javadoc:%E2%98%82=RecycleViewDemo/D:\\/workspace3\\/RecycleViewDemo\\/libs\\/android-support-v7-recyclerview.jar<android.support.v7.widget(RecyclerView.class%E2%98%83RecyclerView~setLayoutManager~Landroid.support.v7.widget.RecyclerView$LayoutManager;%E2%98%82LayoutManager) layout);//设置布局管理器

mRecyclerView.setAdapter(adapter)//设置adapter

mRecyclerView.setItemAnimator([ItemAnimator](eclipse-javadoc:%E2%98%82=RecycleViewDemo/D:\\/workspace3\\/RecycleViewDemo\\/libs\\/android-support-v7-recyclerview.jar<android.support.v7.widget(RecyclerView.class%E2%98%83RecyclerView~setItemAnimator~Landroid.support.v7.widget.RecyclerView$ItemAnimator;%E2%98%82ItemAnimator) animator);//设置Item增加、移除动画

mRecyclerView.addItemDecoration([ItemDecoration](eclipse-javadoc:%E2%98%82=RecycleViewDemo/D:\\/workspace3\\/RecycleViewDemo\\/libs\\/android-support-v7-recyclerview.jar<android.support.v7.widget(RecyclerView.class%E2%98%83RecyclerView~addItemDecoration~Landroid.support.v7.widget.RecyclerView$ItemDecoration;%E2%98%82ItemDecoration) decor);//添加分割线

ok，相比较于ListView的代码，ListView可能只需要去设置一个adapter就能正常使用了。而RecyclerView基本需要上面一系列的步骤，那么为什么会添加这么多的步骤呢？那么就必须解释下RecyclerView的这个名字了，从它类名上看，RecyclerView代表的意义是，我只管Recycler View，也就是说RecyclerView只管回收与复用View，其他的你可以自己去设置。可以看出其高度的解耦，给予你充分的定制自由（所以你才可以轻松的通过这个控件实现ListView,GirdView，瀑布流等效果）。

### 最简单的（连分割线都没有）

**class** **HomeAdapter** **extends** **Adapter**<**HomeAdapter**.**MyViewHolder**> {

**private** List<**String**> mDatas;

**private** **LayoutInflater** mInflater;

**public** **HomeAdapter**(**Context** context, List<**String**> datas) {

mInflater = **LayoutInflater**.*from*(context);

mDatas = datas;

}

@Override

**public** **MyViewHolder** **onCreateViewHolder**(**ViewGroup** parent, **int** viewType) {

**View** view = mInflater.inflate(**R**.**layout**.**item\_home**, parent, **false**);

**MyViewHolder** holder = **new** MyViewHolder(view);

**return** holder;

}

@Override

**public** **void** **onBindViewHolder**(**MyViewHolder** holder, **int** position) {

holder.tv.setText(mDatas.get(position));

}

@Override

**public** **int** **getItemCount**() {

**return** mDatas.size();

}

@Override

**public** **int** **getItemViewType**(**int** position) {

**return** **super**.**getItemViewType**(position);

}

**class** **MyViewHolder** **extends** **ViewHolder** {

**TextView** tv;

**public** **MyViewHolder**(**View** view) {

**super**(view);

tv = (**TextView**) view.findViewById(**R**.**id**.**id\_num**);

}

}

}

mRecyclerView = (**RecyclerView**) findViewById(**R**.**id**.**id\_recyclerview**);

mAdapter = **new** HomeAdapter(**this**, mDatas);

mRecyclerView.setLayoutManager(**new** LinearLayoutManager(**this**));//设置布局管理器

mRecyclerView.setAdapter(mAdapter);

看起来好丑，Item间应该有个分割线，当你去找时，你会发现RecyclerView并没有支持divider这样的属性。那么怎么办，你可以给Item的布局去设置margin，当然了这种方式不够优雅，我们文章开始说了，我们可以自由的去定制它，当然我们的分割线也是可以定制的。

### 设置分割线ItemDecoration

我们可以通过该方法添加分割线： mRecyclerView.addItemDecoration() ，该方法的参数为RecyclerView.ItemDecoration，该类为抽象类，官方目前并没有提供默认的实现类（我觉得最好能提供几个）。当我们调用mRecyclerView.addItemDecoration()方法添加decoration的时候，RecyclerView在绘制的时候，去会绘制decorator，即调用该类的onDraw和onDrawOver方法，

* onDraw方法先于drawChildren
* onDrawOver在drawChildren之后，一般我们选择复写其中一个即可。
* getItemOffsets 可以通过outRect.set()为每个Item设置一定的偏移量，主要用于绘制Decorator。

接下来我们看一个系统提供的RecyclerView.ItemDecoration的实现类，该类很好的实现了RecyclerView添加分割线（当使用LayoutManager为LinearLayoutManager时）。

**public** **class** **DividerItemDecoration** **extends** **ItemDecoration**{

**//获取到的Drawable，在getItemOffsets中，outRect去设置了绘制的范围。onDraw中实现了真正的绘制。**

**private** **static** **final** **int**[] **ATTRS** = **new** **int**[]{

android.**R**.**attr**.**listDivider**

};

**public** **static** **final** **int** **HORIZONTAL\_LIST** = **LinearLayoutManager**.**HORIZONTAL**;

**public** **static** **final** **int** **VERTICAL\_LIST** = **LinearLayoutManager**.**VERTICAL**;

**private** **Drawable** mDivider;//分割线图片

**private** **int** mOrientation;//方向

**public** **DividerItemDecoration**(**Context** context, **int** orientation) {

**final** **TypedArray** a = context.obtainStyledAttributes(**ATTRS**);

mDivider = a.getDrawable(0);

a.recycle();

setOrientation(orientation);**//设置方向**

}

**//设置方向**

**public** **void** **setOrientation**(**int** orientation) {

**if** (orientation != **HORIZONTAL\_LIST** && orientation != **VERTICAL\_LIST**) {

**throw** **new** IllegalArgumentException("invalid orientation");

}

mOrientation = orientation;

}

@Override

**public** **void** **onDraw**(**Canvas** c, **RecyclerView** parent) {

**Log**.*v*("recyclerview - itemdecoration", "onDraw()");

**if** (mOrientation == **VERTICAL\_LIST**) {

drawVertical(c, parent);//竖着画

} **else** {

drawHorizontal(c, parent);

}

}

**public** **void** **drawVertical**(**Canvas** c, **RecyclerView** parent) {

**final** **int** left = parent.getPaddingLeft();

**final** **int** right = parent.getWidth() - parent.getPaddingRight();

**final** **int** childCount = parent.getChildCount();

**for** (**int** i = 0; i < childCount; i++) {

**final** **View** child = parent.getChildAt(i);

**final** **RecyclerView**.**LayoutParams** params = (**RecyclerView**.**LayoutParams**) child.getLayoutParams();

**final** **int** top = child.getBottom() + params.bottomMargin;

**final** **int** bottom = top + mDivider.getIntrinsicHeight();

mDivider.setBounds(left, top, right, bottom);

mDivider.draw(c);

}

}

**public** **void** **drawHorizontal**(**Canvas** c, **RecyclerView** parent) {

**final** **int** top = parent.getPaddingTop();

**final** **int** bottom = parent.getHeight() - parent.getPaddingBottom();

**final** **int** childCount = parent.getChildCount();

**for** (**int** i = 0; i < childCount; i++) {

**final** **View** child = parent.getChildAt(i);

**final** **RecyclerView**.**LayoutParams** params = (**RecyclerView**.**LayoutParams**) child.getLayoutParams();

**final** **int** left = child.getRight() + params.rightMargin;

**final** **int** right = left + mDivider.getIntrinsicHeight();

mDivider.setBounds(left, top, right, bottom);

mDivider.draw(c);

}

}

@Override

**public** **void** **getItemOffsets**(**Rect** outRect, **int** itemPosition, **RecyclerView** parent) {

**if** (mOrientation == **VERTICAL\_LIST**) {

outRect.set(0, 0, 0, mDivider.getIntrinsicHeight());

} **else** {

outRect.set(0, 0, mDivider.getIntrinsicWidth(), 0);

}

}

}

我们在原来的代码中添加一句：

mRecyclerView.addItemDecoration(new DividerItemDecoration(this,DividerItemDecoration.VERTICAL\_LIST));

该分割线是系统默认的，你可以在theme.xml中找到该属性的使用情况。那么，使用系统的listDivider有什么好处呢？就是方便我们去随意的改变，该属性我们可以直接声明在：

<!-- Application theme. -->

<style name="AppTheme" parent="AppBaseTheme">

<item name="android:listDivider">@drawable/divider\_bg</item>

</style>

然后自己写个drawable即可，下面我们换一种分隔符：

<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"

android:shape="rectangle" >

<gradient

android:centerColor="#ff00ff00"

android:endColor="#ff0000ff"

android:startColor="#ffff0000"

android:type="linear" />

<size android:height="4dp"/>

</shape>

### 设置布局管理器LayoutManager

上面实现了类似ListView样子的Demo，通过使用其默认的LinearLayoutManager。

RecyclerView.LayoutManager是一个抽象类，好在系统提供了3个实现类：

* LinearLayoutManager 现行管理器，支持横向、纵向。
* GridLayoutManager 网格布局管理器
* StaggeredGridLayoutManager 瀑布就式布局管理器

上面我们已经初步体验了下LinearLayoutManager，接下来看GridLayoutManager。

#### GridLayoutManager需要适配自定义DividerGridItemDecoration

我们尝试去实现类似GridView，秒秒钟的事情：

mRecyclerView.setLayoutManager(new GridLayoutManager(this,4));

只需要修改LayoutManager即可，还是很nice的。当然了，改为GridLayoutManager以后，对于分割线，前面的DividerItemDecoration就不适用了，主要是因为它在绘制的时候，比如水平线，针对每个child的取值为：

final int left = parent.getPaddingLeft();

final int right = parent.getWidth() - parent.getPaddingRight();

因为每个Item一行，这样是没问题的。而GridLayoutManager时，一行有多个childItem，这样就多次绘制了，并且GridLayoutManager时，Item如果为最后一列（则右边无间隔线）或者为最后一行（底部无分割线）。

针对上述，我们编写了DividerGridItemDecoration。

**public** **class** **DividerGridItemDecoration** **extends** **RecyclerView**.**ItemDecoration** {

**private** **static** **final** **int**[] **ATTRS** = **new** **int**[] { android.**R**.**attr**.**listDivider** };

**private** **Drawable** mDivider;

**public** **DividerGridItemDecoration**(**Context** context) {

**final** **TypedArray** a = context.obtainStyledAttributes(**ATTRS**);

mDivider = a.getDrawable(0);

a.recycle();

}

@Override

**public** **void** **onDraw**(**Canvas** c, **RecyclerView** parent, **State** state) {

drawHorizontal(c, parent);

drawVertical(c, parent);

}

**// 计算出列数**

**private** **int** **getSpanCount**(**RecyclerView** parent) {

**int** spanCount = -1;

**LayoutManager** layoutManager = parent.getLayoutManager();

**if** (layoutManager **instanceof** **GridLayoutManager**) {

spanCount = ((**GridLayoutManager**) layoutManager).getSpanCount();

} **else** **if** (layoutManager **instanceof** **StaggeredGridLayoutManager**) {

spanCount = ((**StaggeredGridLayoutManager**) layoutManager).getSpanCount();

}

**return** spanCount;

}

**public** **void** **drawHorizontal**(**Canvas** c, **RecyclerView** parent) {

**int** childCount = parent.getChildCount();

**for** (**int** i = 0; i < childCount; i++) {

**final** **View** child = parent.getChildAt(i);

**final** **RecyclerView**.**LayoutParams** params = (**RecyclerView**.**LayoutParams**) child.getLayoutParams();

**final** **int** left = child.getLeft() - params.leftMargin;

**final** **int** right = child.getRight() + params.rightMargin + mDivider.getIntrinsicWidth();

**final** **int** top = child.getBottom() + params.bottomMargin;

**final** **int** bottom = top + mDivider.getIntrinsicHeight();

mDivider.setBounds(left, top, right, bottom);

mDivider.draw(c);

}

}

**public** **void** **drawVertical**(**Canvas** c, **RecyclerView** parent) {

**final** **int** childCount = parent.getChildCount();

**for** (**int** i = 0; i < childCount; i++) {

**final** **View** child = parent.getChildAt(i);

**final** **RecyclerView**.**LayoutParams** params = (**RecyclerView**.**LayoutParams**) child.getLayoutParams();

**final** **int** top = child.getTop() - params.topMargin;

**final** **int** bottom = child.getBottom() + params.bottomMargin;

**final** **int** left = child.getRight() + params.rightMargin;

**final** **int** right = left + mDivider.getIntrinsicWidth();

mDivider.setBounds(left, top, right, bottom);

mDivider.draw(c);

}

}

**private** **boolean** **isLastColum**(**RecyclerView** parent, **int** pos, **int** spanCount, **int** childCount) {

**LayoutManager** layoutManager = parent.getLayoutManager();

**if** (layoutManager **instanceof** **GridLayoutManager**) {

**if** ((pos + 1) % spanCount == 0){**// 如果是最后一列，则不需要绘制右边**

**return** **true**;

}

} **else** **if** (layoutManager **instanceof** **StaggeredGridLayoutManager**) {

**int** orientation = ((**StaggeredGridLayoutManager**) layoutManager).getOrientation();

**if** (orientation == **StaggeredGridLayoutManager**.**VERTICAL**) {

**if** ((pos + 1) % spanCount == 0){**// 如果是最后一列，则不需要绘制右边**

**return** **true**;

}

} **else** {

childCount = childCount - childCount % spanCount;

**if** (pos >= childCount){**// 如果是最后一列，则不需要绘制右边**

**return** **true**;

}

}

}

**return** **false**;

}

**private** **boolean** **isLastRaw**(**RecyclerView** parent, **int** pos, **int** spanCount, **int** childCount) {

**LayoutManager** layoutManager = parent.getLayoutManager();

**if** (layoutManager **instanceof** **GridLayoutManager**) {

childCount = childCount - childCount % spanCount;

**if** (pos >= childCount)**// 如果是最后一行，则不需要绘制底部**

**return** **true**;

} **else** **if** (layoutManager **instanceof** **StaggeredGridLayoutManager**) {

**int** orientation = ((**StaggeredGridLayoutManager**) layoutManager).getOrientation();

**// StaggeredGridLayoutManager 且纵向滚动**

**if** (orientation == **StaggeredGridLayoutManager**.**VERTICAL**) {

childCount = childCount - childCount % spanCount;

**// 如果是最后一行，则不需要绘制底部**

**if** (pos >= childCount)

**return** **true**;

} **else**{

**// StaggeredGridLayoutManager 且横向滚动**

**// 如果是最后一行，则不需要绘制底部**

**if** ((pos + 1) % spanCount == 0) {

**return** **true**;

}

}

}

**return** **false**;

}

@Override

**public** **void** **getItemOffsets**(**Rect** outRect, **int** itemPosition, **RecyclerView** parent) {

**int** spanCount = getSpanCount(parent);

**int** childCount = parent.getAdapter().getItemCount();

**if** (isLastRaw(parent, itemPosition, spanCount, childCount)){**// 如果是最后一行，则不需要绘制底部**

outRect.set(0, 0, mDivider.getIntrinsicWidth(), 0);

} **else** **if** (isLastColum(parent, itemPosition, spanCount, childCount)){**// 如果是最后一列，则不需要绘制右边**

outRect.set(0, 0, 0, mDivider.getIntrinsicHeight());

} **else** {

outRect.set(0, 0, mDivider.getIntrinsicWidth(), mDivider.getIntrinsicHeight());

}

}

}

主要在getItemOffsets方法中，去判断如果是最后一行，则不需要绘制底部；如果是最后一列，则不需要绘制右边，整个判断也考虑到了StaggeredGridLayoutManager的横向和纵向，所以稍稍有些复杂。最重要还是去理解，如何绘制什么的不重要。一般如果仅仅是希望有空隙，还是去设置item的margin方便。

#### StaggeredGridLayoutManager（不但有Grid的竖向还有横向）

使用GridLayoutManager只有默认的竖向，但StaggeredGridLayoutManager有横向，即横向的GridLayoutManager

mRecyclerView.setLayoutManager(**new** StaggeredGridLayoutManager(4, **StaggeredGridLayoutManager**.**VERTICAL**));

mRecyclerView.setLayoutManager(**new** StaggeredGridLayoutManager(4, **StaggeredGridLayoutManager**.**HORIZONTAL**));

### 设置动画ItemAnimator

mRecyclerView.setItemAnimator(new DefaultItemAnimator());//默认动画

更新数据集不是用adapter.notifyDataSetChanged()而是 notifyItemInserted(position)与notifyItemRemoved(position) 否则没有动画效果。上述为adapter中添加了两个方法：

public void addData(int position) {

mDatas.add(position, "Insert One");

notifyItemInserted(position);

}

public void removeData(int position) {

mDatas.remove(position);

notifyItemRemoved(position);

}

### 单击和长按事件Click and LongClick

RecycleView没有提供ClickListener和LongClickListener。 不过我们也可以自己去添加，只是会多了些代码而已。 实现的方式比较多，你可以通过mRecyclerView.addOnItemTouchListener去监听然后去判断手势， 当然你也可以通过adapter中自己去提供回调，这里我们选择后者，前者的方式，大家有兴趣自己去实现。那么代码也比较简单：

**Adapter中**

**public** **interface** OnItemClickLitener {

**void** **onItemClick**(**View** view, **int** position);

**void** **onItemLongClick**(**View** view, **int** position);

}

**private** OnItemClickLitener mOnItemClickLitener;

**public** **void** **setOnItemClickLitener**(OnItemClickLitener mOnItemClickLitener) {

**this**.mOnItemClickLitener = mOnItemClickLitener;

}

**。。。。。。。。。。**

@Override

**public** **void** **onBindViewHolder**(**final** **MyViewHolder** holder, **final** **int** position) {

holder.tv.setText(mDatas.get(position));

**if** (mOnItemClickLitener != **null**) {**// 如果设置了回调，则设置点击事件**

holder.itemView.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**int** pos = holder.getPosition();

mOnItemClickLitener.onItemClick(holder.itemView, pos);

}

});

holder.itemView.setOnLongClickListener(**new** OnLongClickListener() {

@Override

**public** **boolean** **onLongClick**(**View** v) {

**int** pos = holder.getPosition();

mOnItemClickLitener.onItemLongClick(holder.itemView, pos);

**//如果为false按钮点击时间中有一个临界值，当我单下按钮的时间等于这个临界值的时候，点击事件和长点击事件会同时触发**

**return** **true**;

}

});

}

}

**Activity中**

mAdapter.setOnItemClickLitener(**new** OnItemClickLitener() {

@Override

**public** **void** **onItemClick**(**View** view, **int** position) {

**Toast**.*makeText*(**MainActivity**.**this**, position + " click", **Toast**.**LENGTH\_SHORT**).show();

mAdapter.removeData(position);

}

@Override

**public** **void** **onItemLongClick**(**View** view, **int** position) {

**Toast**.*makeText*(**MainActivity**.**this**, position + " long click", **Toast**.**LENGTH\_SHORT**).show();

mAdapter.removeData(position);

}

});

## 监听滚动事件

RecyclerView是viewgroup，不是AbsListView，所以不能实现AbsListView.OnScrollListener

mRecyclerView.addOnScrollListener(**new** **RecyclerView**.OnScrollListener() {

@Override

**public** **void** **onScrolled**(**RecyclerView** recyclerView, **int** dx, **int** dy) {

**int** resid = mDatas.get(mRecyclerView.getChildPosition(mRecyclerView.getChildAt(0)));

**//mImg.setImageBitmap(decodeSampledBitmapFromResource(getResources(), resid, 20, 20));**

mImg.setImageResource(resid);

}

@Override

**public** **void** **onScrollStateChanged**(**RecyclerView** recyclerView, **int** newState) {

**super**.onScrollStateChanged(recyclerView, newState);

}

});

### 监听是否滚动到了最前面

mRecyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {  
 override fun onScrollStateChanged(recyclerView: RecyclerView?, newState: Int) {  
 super.onScrollStateChanged(recyclerView, newState)  
 // 当停止滑动时  
 if (newState == RecyclerView.*SCROLL\_STATE\_IDLE*) {  
 //设置什么布局管理器,就获取什么的布局管理器  
 val manager = mRecyclerView.*layoutManager* as LinearLayoutManager?  
 //获取最前一个完全显示的ItemPosition ,角标值  
 val firstVisibleItem = manager?.findFirstVisibleItemPosition() ?: 0  
 // 判断是否滚动到顶部  
 if (firstVisibleItem <= 1) {  
 //隐藏xx条未读新消息View  
 tv\_noread\_new\_msg.*visibility* = View.*GONE* }  
 }  
 }  
})

### 监听是否滚动到了最后面

mRecyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {  
 override fun onScrollStateChanged(recyclerView: RecyclerView?, newState: Int) {  
 super.onScrollStateChanged(recyclerView, newState)  
 // 当停止滑动时  
 if (newState == RecyclerView.*SCROLL\_STATE\_IDLE*) {  
 if (recycleViewIsInBottom()) {  
 }  
 }  
 }  
})

//判断RecyclerView是否滚到了底部  
private fun recycleViewIsInBottom(): Boolean {  
 if (false) {

//屏幕中最后一个可见子项的position  
 val lastVisibleItemPosition = linearLayoutManager!!.findLastCompletelyVisibleItemPosition()   
 val visibleItemCount = linearLayoutManager!!.*childCount*//当前屏幕所看到的子项个数  
 val totalItemCount = linearLayoutManager!!.*itemCount*//当前RecyclerView的所有子项个数  
 return (visibleItemCount > 0 && lastVisibleItemPosition >= totalItemCount - 1)  
 } else {  
 //得到当前显示的最后一个item的view  
 val lastChildView = linearLayoutManager!!.getChildAt(linearLayoutManager!!.*childCount* - 1)  
 //得到lastChildView的bottom坐标值  
 val lastChildBottom = lastChildView.*bottom* //得到RecyclerView的底部坐标减去底部padding值，也就是显示内容最底部的坐标  
 val recyclerBottom = mRecyclerView.*bottom* - mRecyclerView.*paddingBottom* //通过这个lastChildView得到这个view当前的position值  
 val lastPosition = linearLayoutManager!!.getPosition(lastChildView)  
 //判断lastChildView的bottom值跟recyclerBottom 且判断lastPosition是不是最后一个position

if (lastChildBottom <= recyclerBottom && lastPosition >= linearLayoutManager!!.*itemCount* - 1) {  
 return true  
 }  
 return false  
 }  
}

## 获取滚动距离

### ComputeVerGridLM

public class ComputeVerGridLM extends GridLayoutManager {  
 private HashMap<Integer, Integer> heightMap = new HashMap<>();  
  
 public ComputeVerGridLM(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {  
 super(context, attrs, defStyleAttr, defStyleRes);  
 }  
  
 public ComputeVerGridLM(Context context, int spanCount) {  
 super(context, spanCount);  
 }  
  
 public ComputeVerGridLM(Context context, int spanCount, int orientation, boolean reverseLayout) {  
 super(context, spanCount, orientation, reverseLayout);  
 }  
  
 @Override  
 public void onLayoutCompleted(RecyclerView.State state) {  
 super.onLayoutCompleted(state);  
 int firstVisible = findFirstVisibleItemPosition();  
 int lastVisible = findLastVisibleItemPosition();  
 if (firstVisible >= 0 && lastVisible >= 0 && lastVisible >= firstVisible) {  
 for (int i = firstVisible; i < lastVisible; i++) {  
 View childAt = getChildAt(i);  
 if (childAt != null) {  
 LayoutParams layoutParams = (LayoutParams) childAt.getLayoutParams();  
 if (layoutParams.getSpanIndex() == 0) { //每行的最左边一个，一行只累加一个高度  
 heightMap.put(i, childAt.getHeight());  
 } else {  
 heightMap.put(i, 0);  
 }  
 }  
 }  
 }  
 }  
  
 @Override  
 public int computeVerticalScrollOffset(RecyclerView.State state) {  
 if (getChildCount() == 0) return 0;  
 try {  
 int firstVisiblePosition = findFirstVisibleItemPosition();  
 View viewByPosition = findViewByPosition(firstVisiblePosition);  
 int offset = 0;  
 for (int i = 0; i < firstVisiblePosition; i++) {  
 offset += heightMap.get(i) != null ? heightMap.get(i) : 0;  
 }  
 offset -= viewByPosition != null ? viewByPosition.getTop() : 0;  
 return offset;  
 } catch (Exception e) {  
 return 0;  
 }  
 }  
}

### ComputeVerLineLM

public class ComputeVerLineLM extends LinearLayoutManager {  
 private HashMap<Integer, Integer> heightMap = new HashMap<>();  
  
 public ComputeVerLineLM(Context context) {  
 super(context);  
 }  
  
 public ComputeVerLineLM(Context context, int orientation, boolean reverseLayout) {  
 super(context, orientation, reverseLayout);  
 }  
  
 public ComputeVerLineLM(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {  
 super(context, attrs, defStyleAttr, defStyleRes);  
 }  
  
  
 @Override  
 public void onLayoutCompleted(RecyclerView.State state) {  
 super.onLayoutCompleted(state);  
 int firstVisible = findFirstVisibleItemPosition();  
 int lastVisible = findLastVisibleItemPosition();  
 if (firstVisible >= 0 && lastVisible >= 0 && lastVisible >= firstVisible) {  
 for (int i = firstVisible; i < lastVisible; i++) {  
 View childAt = getChildAt(i);  
 if (childAt != null) {  
 heightMap.put(i, childAt.getHeight());  
 }  
 }  
 }  
 }  
  
 @Override  
 public int computeVerticalScrollOffset(RecyclerView.State state) {  
 if (getChildCount() == 0) return 0;  
 try {  
 int firstVisiblePosition = findFirstVisibleItemPosition();  
 View viewByPosition = findViewByPosition(firstVisiblePosition);  
 int offset = 0;  
 for (int i = 0; i < firstVisiblePosition; i++) {  
 offset += heightMap.get(i) != null ? heightMap.get(i) : 0;  
 }  
 offset -= viewByPosition != null ? viewByPosition.getTop() : 0;  
 return offset;  
 } catch (Exception e) {  
 return 0;  
 }  
 }  
}

### 使用

rv.addOnScrollListener(new RecyclerView.OnScrollListener() {  
 @Override  
 public void onScrollStateChanged(@NonNull RecyclerView recyclerView, int newState) {  
 super.onScrollStateChanged(recyclerView, newState);  
 int offset = recyclerView.computeVerticalScrollOffset();  
 upIv.setVisibility(offset > 500 ? View.*VISIBLE* : View.*GONE*);  
 }  
});

## 公用Adapter的封装

构造一个比较完整的Adapter至少需要完成以下三件事情

* 新建ViewHolder类来存储Item视图及其子视图
* onCreateViewHolder通过视图Id加载不同Item视图并生成ViewHolder用来保存每个列表Item视图
* onBindViewHolder更新列表Item视图（填充model数据）

**LGViewHolder.java**

**public** **class** **LGViewHolder** **extends** **RecyclerView**.**ViewHolder** {

**private** **SparseArray**<**View**> mViews;

**private** **View** mConvertView;**//缓存itemView内部的子View**

**public** **LGViewHolder**(**View** itemView) {

**super**(itemView);

mConvertView = itemView;

mViews = **new** SparseArray<>();

}

/\*\*

\* 加载layoutId视图并用LGViewHolder保存

\* **@param** parent

\* **@param** layoutId

\* **@return**

\*/

**protected** **static** **LGViewHolder** **getViewHolder**(**ViewGroup** parent, **int** layoutId) {

**View** itemView = **LayoutInflater**.*from*(parent.getContext()).inflate(layoutId, parent, **false**);

**return** **new** LGViewHolder(itemView);

}

/\*\*

\* 根据ItemView的id获取子视图View

\* **@param** viewId

\* **@return**

\*/

**public** **View getView**(**int** viewId) {

**View** view = mViews.get(viewId);

**if** (view == **null**) {

view = mConvertView.findViewById(viewId);

mViews.put(viewId, view);

}

**return** view;

}

}

**LGRecycleViewAdapter.java**

**public** **abstract** **class** **LGRecycleViewAdapter**<T> **extends** **RecyclerView**.**Adapter**<**LGViewHolder**> {

**private** **final** **String** TAG = "LGRecycleViewAdapter";

**//存储监听回调**

**private** **SparseArray**<ItemClickListener> onClickListeners;

**private** List<T> dataList;

**public** **interface** ItemClickListener {

**void** **onItemClicked**(**View** view,**int** position);

}

**public** **LGRecycleViewAdapter**(List<T> dataList) {

**this**.dataList = dataList;

onClickListeners = **new** SparseArray<>();

}

/\*\*

\* 存储viewId对应的回调监听实例listener

\*/

**public** **void** **setOnItemClickListener**(**int** viewId,ItemClickListener listener) {

ItemClickListener mlistener = onClickListeners.get(viewId);

**if**(mlistener == **null**){

onClickListeners.put(viewId, listener);

}

}

/\*\*

\* 获取列表控件的视图id(由子类负责完成)

\*/

**public** **abstract** **int** **getLayoutId**(**int** viewType);

/\*\*

\* 更新itemView视图(由子类负责完成)

\*/

**public** **abstract** **void** **convert**(**LGViewHolder** holder, T t, **int** position);

**public** T **getItem**(**final** **int** position){

**if**(dataList == **null**) **return** **null**;

**return** dataList.get(position);

}

@Override

**public** **LGViewHolder** **onCreateViewHolder**(**ViewGroup** parent, **int** viewType) {

**int** layoutId = getLayoutId(viewType);

**LGViewHolder** viewHolder = **LGViewHolder**.*getViewHolder*(parent, layoutId);

**return** viewHolder;

}

@Override

**public** **void** **onBindViewHolder**(**LGViewHolder** holder, **final** **int** position) {

T itemModel = dataList.get(position);

convert(holder, itemModel, position);**//更新itemView视图**

**//设置点击监听**

**for** (**int** i = 0; i < onClickListeners.size(); ++i){

**int** id = onClickListeners.keyAt(i);

**View** view = holder.getView(id);

**if**(view == **null**) **continue**;

**final** ItemClickListener listener = onClickListeners.get(id);

view.setOnClickListener(**new** **View**.OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**if**(listener != **null**){

listener.onItemClicked(v,position);

}

}

});

}

}

@Override

**public** **int** **getItemCount**() {

**if** (dataList == **null**)

**return** 0;

**return** dataList.size();

}

**public** **void** **destroyAdapter**(){

**if**(onClickListeners != **null**) onClickListeners.clear();

onClickListeners = **null**;

**if**(dataList != **null**) dataList.clear();

dataList = **null**;

}

}

**使用方法如下：单视图方式：定义adapter**

**public** **class** **MainAdapter** **extends** **LGRecycleViewAdapter**<**String**> {

**public** **MainAdapter**(List<**String**> list) {

**super**(list);

}

@Override

**public** **int** **getLayoutId**(**int** viewType) {

**return** **R**.**layout**.**item\_view\_main1**;

}

@Override

**public** **void** **convert**(**LGViewHolder** holder, **String** s, **final** **int** position) {

**TextView** textView = (**TextView**) holder.getView(**R**.**id**.**id\_text**);

textView.setText(s);

}

}

**在需要时使用：**

mAdapter = **new** MainAdapter(mDatas);

initEvent();

mRecyclerView.setAdapter(mAdapter);

**private** **void** **initEvent**() {

mAdapter.setOnItemClickListener(**R**.**id**.**id\_text0**, **new** ItemClickListener() {

@Override

**public** **void** **onItemClicked**(**View** view, **int** position) {

**Toast**.*makeText*(**MainActivity0**.**this**, position + "文本框", **Toast**.**LENGTH\_SHORT**).show();

}

});

mAdapter.setOnItemClickListener(**R**.**id**.**id\_img0**, **new** ItemClickListener() {

@Override

**public** **void** **onItemClicked**(**View** view, **int** position) {

**Toast**.*makeText*(**MainActivity0**.**this**, position + "图片框", **Toast**.**LENGTH\_SHORT**).show();

}

});

mAdapter.setOnItemClickListener(**R**.**id**.**id\_btn0**, **new** ItemClickListener() {

@Override

**public** **void** **onItemClicked**(**View** view, **int** position) {

**Toast**.*makeText*(**MainActivity0**.**this**, position + "按钮", **Toast**.**LENGTH\_SHORT**).show();

}

});

}

可以看出我们轻松的实现了同一Item视图不同子视图的点击监听，并且在MainAdapter子类中只需通过getLayoutId告诉父类Item视图对应的视图id，并在convert方法中只更新视图即可.如果需要支持多视图模式则只需在子类中重现getViewType即可，代码如下：

**支持多视图模式的MainAdapter.java**

和ListView实现聊天的原理一样，

**public** **class** **MainAdapter** **extends** **LGRecycleViewAdapter**<**String**> {

List<**String**> list;

**public** **MainAdapter**(List<**String**> list) {

**super**(list);

**this**.list = list;

}

@Override

**public** **int** **getLayoutId**(**int** viewType) {

**if** (viewType == 1) {

**return** **R**.**layout**.**item\_view\_main1**;

}**else** {

**return** **R**.**layout**.**item\_view\_main12**;

}

}

@Override

**public** **void** **convert**(**LGViewHolder** holder, **String** s, **final** **int** position) {

**if** (getItemViewType(position) == 1) {

**TextView** textView = (**TextView**) holder.getView(**R**.**id**.**id\_text0**);

textView.setText(s);

}**else** {

**TextView** textView = (**TextView**) holder.getView(**R**.**id**.**id\_text02**);

textView.setText(s);

}

}

@Override

**public** **int** **getItemViewType**(**int** position) {

**String** typeFlag = list.get(position);

**if**(typeFlag.equals("a")||typeFlag.equals("c"))

**return** 1;

**return** 2;

}

}

## MultiTypeAdapter库

### 引用

dependencies {

implementation 'me.drakeet.multitype:multitype:4.0.0-alpha3'

}

### Java代码

private void initRecyclerView() {  
 adapter = new MultiTypeAdapter(items);  
 adapter.register(Category.class, new CategoryViewBinder());  
 adapter.register(ChargeItem.class, new ChargeItemViewBinder());  
 GridLayoutManager layoutManager = new GridLayoutManager(this, 2);  
 SpanSizeLookup spanSizeLookup = new SpanSizeLookup() {  
 @Override  
 public int getSpanSize(int position) {  
 Object item = items.get(position);  
 return (item instanceof Category) ? 2 : 1;  
 }  
 };  
 layoutManager.setSpanSizeLookup(spanSizeLookup);  
 rvCharge.setLayoutManager(layoutManager);  
 rvCharge.setAdapter(adapter);  
 rvCharge.setNestedScrollingEnabled(false);  
}  
public void onDataResult(ChargeTypeResponse chargeTypeResponse) {  
 items.clear();  
 List<ChargeTypeResponse.Row> rows = chargeTypeResponse.getRow();  
 for (ChargeTypeResponse.Row row : rows) {  
 items.add(new Category(row.getConfigTypeName()));  
 items.addAll(row.getConfigList());  
 }  
 adapter.notifyDataSetChanged();  
}  
private class CategoryViewHolder extends RecyclerView.ViewHolder {  
 @BindView(R.id.*tv\_charge\_title*)  
 TextView tvChargeTitle;  
 CategoryViewHolder(View itemView) {  
 super(itemView);  
 ButterKnife.*bind*(this, itemView);  
 }  
}  
private class ChargeItemViewHolder extends RecyclerView.ViewHolder {  
 @BindView(R.id.*tv\_description*)  
 TextView tvDescription;  
 ChargeItemViewHolder(View itemView) {  
 super(itemView);  
 ButterKnife.*bind*(this, itemView);  
 }  
}  
private class CategoryViewBinder extends ItemViewBinder<Category, CategoryViewHolder> {  
 @NonNull  
 @Override  
 protected CategoryViewHolder onCreateViewHolder(@NonNull LayoutInflater inflater, @NonNull ViewGroup parent) {  
 View root = inflater.inflate(R.layout.*item\_charge\_title*, parent, false);  
 return new CategoryViewHolder(root);  
 }  
 @Override  
 protected void onBindViewHolder(@NonNull CategoryViewHolder holder, @NonNull Category category) {  
 holder.tvChargeTitle.setText(category.title);  
 }  
}  
private class ChargeItemViewBinder extends ItemViewBinder<ChargeItem, ChargeItemViewHolder> {  
 @NonNull  
 @Override  
 protected ChargeItemViewHolder onCreateViewHolder(@NonNull LayoutInflater inflater, @NonNull ViewGroup parent) {  
 View root = inflater.inflate(R.layout.*item\_charge*, parent, false);  
 return new ChargeItemViewHolder(root);  
 }  
 @Override  
 protected void onBindViewHolder(@NonNull final ChargeItemViewHolder holder, @NonNull final ChargeItem chargeItem) {  
 holder.tvDescription.setText(chargeItem.getConfigDesc());  
 }  
}

### Kotlin代码

ruleListSource.clear()  
ruleListSource.add(jichaBean)  
ruleListSource.add(fenxiaoBean)  
mRuleListAdapter.notifyDataSetChanged()

private val ruleListSource = ArrayList<Any>()  
private val mRuleListAdapter by *lazy* **{** MultiTypeAdapter(ruleListSource).*apply* **{** register(GroupRuleJiChaBean::class.*java*, JiChaViewBinder())  
 register(GroupRuleFenXiaoBean::class.*java*, FenXiaoViewBinder())  
 **}  
}**  
inner class JiChaViewBinder : ItemViewBinder<GroupRuleJiChaBean, JiChaViewBinder.JiChaViewHolder>() {  
 override fun onCreateViewHolder(inflater: LayoutInflater, parent: ViewGroup): JiChaViewHolder {  
 return JiChaViewHolder(inflater.inflate(R.layout.*in\_group\_rule\_jicha*, parent, false))  
 }  
 override fun onBindViewHolder(holder: JiChaViewHolder, bean: GroupRuleJiChaBean) {  
 holder.tv\_rule.setOnClickListener **{}** }  
 inner class JiChaViewHolder(view: View) : RecyclerView.ViewHolder(view) {  
 val tv\_rule: TextView = view.findViewById(R.id.*tv\_rule*)  
 }  
}  
inner class FenXiaoViewBinder : ItemViewBinder<GroupRuleFenXiaoBean, FenXiaoViewBinder.FenXiaoViewHolder>() {  
 override fun onCreateViewHolder(inflater: LayoutInflater, parent: ViewGroup): FenXiaoViewHolder {  
 return FenXiaoViewHolder(inflater.inflate(R.layout.*in\_group\_rule\_fenxiao*, parent, false))  
 }  
 override fun onBindViewHolder(holder: FenXiaoViewHolder, bean: GroupRuleFenXiaoBean) {  
 holder.tv\_rule.setOnClickListener **{ }** }  
 inner class FenXiaoViewHolder(view: View) : RecyclerView.ViewHolder(view) {  
 val tv\_rule: TextView = view.findViewById(R.id.*tv\_rule*)  
 }  
}

## RecyclerView常见问题解决方案

### 异步加载图片错乱的问题

* RecyclerView用的是我们自定义的内部类ViewHolder来复用的，也就是复用的是ViewHoler
* 当屏幕下滑，item1滑出可视区域，将item1的ViewHolder对象给item8复用，那么此时item1中ViewHolder对象中持有的变量都是item1的。
* item1中的ViewHolder对象，在onBindViewHolder(MyViewHolder holder, int position)方法中对holder进行更新，但是如果在这里调用glide去从url加载图片到holder中的imageView对象的话，就有可能因为网络延迟，导致图片加载不出来，那么item8就会先显示item1的图片，过一会延迟之后，显示正确的item8该显示的图片

String imageUrl = dataSource.get(position);

if (imageUrl == null) {

//当ViewHolder复用的时候，如果当前返回的图片url为null，为了防止上一个复用的viewHolder图片遗留，要clear

Glide.with(fragment).clear(holder.ivImage);

holder.ivImage.setImageDrawable(null);

holder.ivImage.setTag(R.id.image\_tag, position);

return;

}

Object tag = holder.ivImage.getTag(R.id.image\_tag);

//如果tag不是Null,并且同时tag不等于当前的position。说明当前的viewHolder是复用来的

if (tag != null && (int)tag != position) {

Glide.with(fragment).clear(holder.ivImage);

}

Glide.with(fragment)

.load(imageUrl)

.apply(options)

.into(holder.ivImage);

//给ImageView设置唯一标记

holder.ivImage.setTag(R.id.image\_tag, position);

### 无法引用报错

在eclipse中使用RecyclerView只能导入API21.0.0中的包才可以使用。解决办法：进入sdk\extras\android\m2repository\com\android\support\recyclerview-v7\21.0.0目录，找到ecyclerview-v7-21.0.0.aar这个文件。用解压软件解压此文件里面有classes.jar，把这个jar包放到项目中的libs文件下即可。这个jar包就是RecyclerView要使用的，可以给它改个名字例如android-support-v7-recyclerview.jar , 便于查看。

### 嵌套自动滚动

ScrollView或者RecyclerView1 嵌套RecyclerView2进入页面自动跳转到recyclerView2上面页面会自动滚动

貌似是RecyclerView 自动获得了焦点，两种解决办法

一，recyclerview2去除焦点

recyclerview2.setFocusableInTouchMode(false);

recyclerview2.requestFocus();

二，让处于ScrollView或者RecyclerView1 顶端的某个控件获得焦点即可，比如顶部的一个textview

tv\_goodsName.setFocusableInTouchMode(true);

tv\_goodsName.requestFocus();

三，内部的Recyclerview加

android:overScrollMode="never"

四，父布局覆盖子VIew获取焦点：

android:descendantFocusability="blocksDescendants"

**可能会出现的问题**：有的手机上出现了RecyclerView会出现显示不全的情况。针对这种情形，使用网上的方法一种是使用 RelativeLayout 包裹 RecyclerView 并设置属性：android:descendantFocusability="blocksDescendants"

### 高度设置wrap\_content 无作用的bug，在listview中这么设置会不显示

这是RecyclerView兼容包的bug，23.2.0后官方已经修复了。所以直接在gradle里设置用23.2.0及以上的RecyclerView。PS：如果修改后构建时报了其他奇怪的错，可以试试把兼容包也升级一下：

compile 'com.android.support:cardview-v7:23.2.0'

compile 'com.android.support:recyclerview-v7:23.2.0'

compile 'com.android.support:appcompat-v7:23.2.0'

compile 'com.android.support:design:23.2.0'

### Item布局宽度设置match\_parent无效果的问题解决

//在layoutManager代码里重新设置MATCH\_PARENT  
LinearLayoutManager layoutManager = new LinearLayoutManager(getActivity()) {  
 @Override  
 public RecyclerView.LayoutParams generateDefaultLayoutParams() {  
 return new RecyclerView.LayoutParams(  
 ViewGroup.LayoutParams.*MATCH\_PARENT*,  
 ViewGroup.LayoutParams.*WRAP\_CONTENT*);  
 }  
};  
layoutManager.setOrientation(LinearLayoutManager.*VERTICAL*);  
recyclerview.setLayoutManager(layoutManager);

### 同界面或者跨Fragment焦点失去后，回到Rv会自动滚动的问题

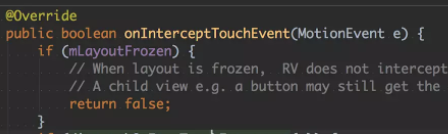
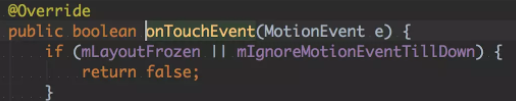
解决了方法：在RecyclerView的父布局上增加“

android:focusable="true"

android:focusableInTouchMode="true"

## 整个RecycleView不响应点击事件

通过RecycleView的onInterceptTouchEvent（）和onTouchEvent（）这2个方法，发现了一个参数mLayoutFrozen，如果他为true的时候，recycleview这2个方法都会返回false，也就不会拦截这个事件了



并且setAdapter(Adapter)会把这个参数设为false



于我在setAdapter之后再setLayoutFrozen(true),这样就达到需求了

## 只点击整个RecyclerView不点击Item

public class RvTotalClickListener implements RecyclerView.OnItemTouchListener {  
 private View.OnClickListener mListener;  
 private GestureDetector mGestureDetector;  
 public RvTotalClickListener(Context context, View.OnClickListener listener) {  
 mListener = listener;  
 mGestureDetector = new GestureDetector(context, new GestureDetector.SimpleOnGestureListener() {  
 @Override  
 public boolean onSingleTapUp(MotionEvent e) {  
 return true;  
 }  
 @Override  
 public void onLongPress(MotionEvent e) {}  
 });  
 }  
 @Override  
 public boolean onInterceptTouchEvent(RecyclerView view, MotionEvent e) {  
 if (view != null && mListener != null && mGestureDetector.onTouchEvent(e)) {  
 mListener.onClick(view);  
 return true;  
 }  
 return false;  
 }  
 @Override  
 public void onTouchEvent(@NonNull RecyclerView view, @NonNull MotionEvent motionEvent) {}  
 @Override  
 public void onRequestDisallowInterceptTouchEvent(boolean disallowIntercept) {}  
}

//使用

mRv.addOnItemTouchListener(new RvTotalClickListener(context, view -> toast("")));

## 九宫格居中平分间距

### 分析

item的宽度先设置为match\_parent，然后通过getItemOffsets设置ItemView的内嵌偏移长度



想实现左右靠边，中间平分的效果，

switch (position % 3) {  
 case 0:  
 outRect.right = 2 \* space;  
 break;  
 case 1:  
 outRect.left = space;  
 outRect.right = space;  
 break;  
 case 2:  
 outRect.left = 2 \* space;  
 break;  
 }

想实现完全平分的效果，

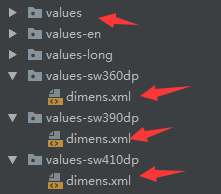
outRect.left = space;  
outRect.right = space;

### item\_dynamic\_img\_nine.xml

<?xml version="1.0" encoding="utf-8"?>  
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
  
 <com.imlianka.lkapp.util.RoundImageView  
 android:id="@+id/item\_nine\_grid\_img"  
 android:layout\_width="match\_parent"  
 android:scaleType="centerCrop"  
 android:layout\_height="@dimen/dynamic\_nine\_grid\_img\_wh"  
 android:src="@drawable/loading\_defaultpic"  
 app:corner="6dp" />  
</FrameLayout>

### DynamicSpaceDecoration.java

图片的高度和宽度一样，因此宽度要根据屏幕大小多设置几个资源文件夹

public class DynamicSpaceDecoration extends RecyclerView.ItemDecoration {  
 private int space;  
 public DynamicSpaceDecoration(Context context) {  
 int screenWidth = ScreenWidthHeightUtil.*getWidthPixels*();  
 int dynamic\_nine\_grid\_img\_wh = GetValueUtil.*getDimensInt*(context, R.dimen.*dynamic\_nine\_grid\_img\_wh*);  
 int dynamic\_margin\_lr = GetValueUtil.*getDimensInt*(context, R.dimen.*dynamic\_margin\_lr*);  
 space = (screenWidth - 2 \* dynamic\_margin\_lr - 3 \* dynamic\_nine\_grid\_img\_wh) / 6;  
 }  
 //设置ItemView的内嵌偏移长度  
 @Override  
 public void getItemOffsets(Rect outRect, View view, RecyclerView parent, RecyclerView.State state) {  
 super.getItemOffsets(outRect, view, parent, state);  
 int position = parent.getChildAdapterPosition(view);  
 switch (position % 3) {  
 case 0:  
 outRect.right = 2 \* space;  
 break;  
 case 1:  
 outRect.left = space;  
 outRect.right = space;  
 break;  
 case 2:  
 outRect.left = 2 \* space;  
 break;  
 }  
 if (position > 2) outRect.top = 3 \* space;  
 }  
}

### 使用

if (gridView.getItemDecorationCount() == 0) {

gridView.addItemDecoration(new DynamicSpaceDecoration(mContext));

}

## 从底部开始展示

linearLayoutManager.setStackFromEnd(true);

# RecylcerView 面试十四问

## Adapter相关

RecyclerView大概结构如下所示

* RecyclerView.Adapter - 处理数据集合并负责绑定视图
* ViewHolder - 持有所有的用于绑定数据或者需要操作的View
* LayoutManager - 负责摆放视图等相关操作
* ItemDecoration - 负责绘制Item附近的分割线
* ItemAnimator - 为Item的一般操作添加动画效果，如，增删条目等

### Adapter的作用是什么？

* 根据不同ViewType创建与之相应的的Item-Layout
* 访问数据集合并将数据绑定到正确的View上

### 几个方法是做什么用的？

* public VH onCreateViewHolder(ViewGroup parent, int viewType)

创建Item视图，并返回相应的ViewHolder

* public void onBindViewHolder(VH holder, int position)

绑定数据到正确的Item视图上。

* public int getItemCount()

返回该Adapter所持有的Itme数量

* public int getItemViewType(int position)

用来获取当前项Item(position参数)是哪种类型的布局

### 如何理解adapter订阅者模式？

当时据集合发生改变时，我们通过调用.notifyDataSetChanged()，来刷新列表，因为这样做会触发列表的重绘。注意这里需要理解什么是订阅者模式,notifyDataSetChanged()源码:

public final void notifyDataSetChanged() {

mObservable.notifyChanged();

}

接着查看notifyChanged()源码,被观察者AdapterDataObservable，内部持有观察者AdapterDataObserver集合

static class AdapterDataObservable extends Observable<AdapterDataObserver> {

public boolean hasObservers() {

return !mObservers.isEmpty();

}

public void notifyChanged() {

for (int i = mObservers.size() - 1; i >= 0; i--) {

mObservers.get(i).onChanged();

}

}

public void notifyItemRangeChanged(int positionStart, int itemCount) {

for (int i = mObservers.size() - 1; i >= 0; i--) {

mObservers.get(i).onItemRangeChanged(positionStart, itemCount);

}

}

public void notifyItemRangeInserted(int positionStart, int itemCount) {

for (int i = mObservers.size() - 1; i >= 0; i--) {

mObservers.get(i).onItemRangeInserted(positionStart, itemCount);

}

}

}

观察者AdapterDataObserver，具体实现为RecyclerViewDataObserver，当数据源发生变更时，最终会调用到requestLayout()，及时响应界面变化

public static abstract class AdapterDataObserver {

public void onChanged() {

// Do nothing

}

public void onItemRangeChanged(int positionStart, int itemCount) {

// Do nothing

}

public void onItemRangeInserted(int positionStart, int itemCount) {

// Do nothing

}

...

}

private class RecyclerViewDataObserver extends AdapterDataObserver {  
 RecyclerViewDataObserver() {  
 }  
 @Override  
 public void onChanged() {  
 assertNotInLayoutOrScroll(null);  
 mState.mStructureChanged = true;  
 processDataSetCompletelyChanged(true);  
 if (!mAdapterHelper.hasPendingUpdates()) {  
 requestLayout();  
 }  
 }  
 @Override  
 public void onItemRangeChanged(int positionStart, int itemCount) {  
 assertNotInLayoutOrScroll(null);  
 if (mAdapterHelper.onItemRangeChanged(positionStart, itemCount)) {  
 triggerUpdateProcessor();  
 }  
 }  
 @Override  
 public void onItemRangeInserted(int positionStart, int itemCount) {  
 assertNotInLayoutOrScroll(null);  
 if (mAdapterHelper.onItemRangeInserted(positionStart, itemCount)) {  
 triggerUpdateProcessor();  
 }  
 }

void triggerUpdateProcessor() {  
 if (*POST\_UPDATES\_ON\_ANIMATION* && mHasFixedSize && mIsAttached) {  
 ViewCompat.*postOnAnimation*(RecyclerView.this, mUpdateChildViewsRunnable);  
 } else {  
 mAdapterUpdateDuringMeasure = true;  
 requestLayout();  
 }  
 }

...  
}

接着查看setAdapter()源码中的setAdapterInternal(adapter, false, true)方法

public void setAdapter(Adapter adapter) {

// bail out if layout is frozen

setLayoutFrozen(false);

setAdapterInternal(adapter, false, true);

requestLayout();

}

private void setAdapterInternal(Adapter adapter, boolean compatibleWithPrevious, boolean removeAndRecycleViews) {

if (adapter != null) {

//注册一个观察者RecyclerViewDataObserver

adapter.registerAdapterDataObserver(mObserver);

adapter.onAttachedToRecyclerView(this);

}

...

}

在setAdapter时将观察者注入，当notify方法被调用，Adapter内部的被观察者会遍历通知已经注册的观察者的对应方法，这时界面就会响应变更。

### requestLayout方法

会调用View的requestLayout方法

public void requestLayout() {  
 ...  
 mPrivateFlags |= *PFLAG\_FORCE\_LAYOUT*;  
 mPrivateFlags |= *PFLAG\_INVALIDATED*;  
 if (mParent != null && !mParent.isLayoutRequested()) {  
 mParent.requestLayout();  
 }  
 ...  
}

requestLayout方法会设置View的标记位：PFLAG\_FORCE\_LAYOUT表示当前View要进行重新布局，PFLAG\_INVALIDATED表示要进行重新绘制，并一层层往上调用父布局的requestLayout方法，最后调用ViewRootImpl中的requestLayout -> scheduleTraversals -> doTraversa -> TraversalRunnable -> performTraversals,开始View的三大流程，然后被标记的View就会进行测量、布局和绘制流程，调用的方法为onMeasure、onLayout和onDraw。

### invalidate方法

它的过程和requestLayout方法方法很像，但是invalidate方法没有标记PFLAG\_FORCE\_LAYOUT，所以不会执行测量和布局流程，而只是对需要重绘的View进行重绘，也就是只会调用onDraw方法，不会调用onMeasure和onLayout方法。

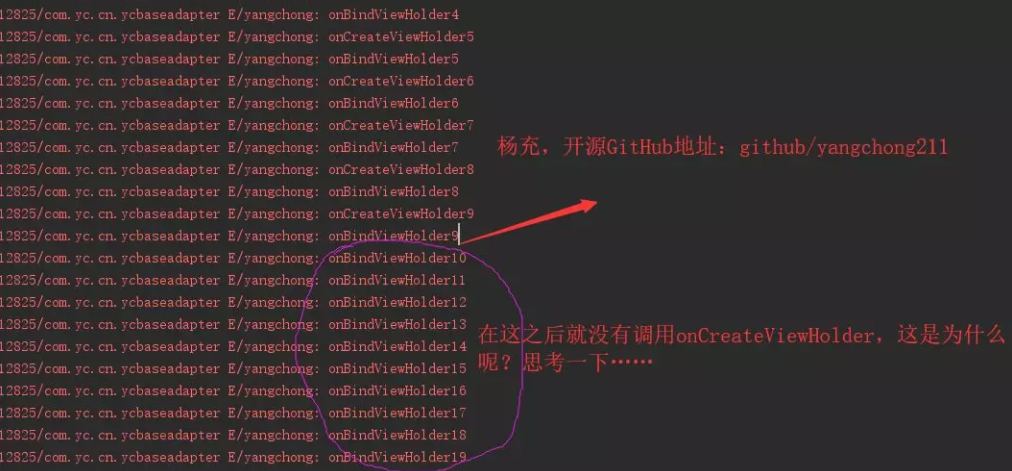
## ViewHolder相关

### ViewHolder作用大概有这些：

* adapter应当拥有ViewHolder的子类，并且ViewHolder内部应当存储一些子view，避免时间代价很大的findViewById操作
* 其RecyclerView内部定义的ViewHolder类包含很多复杂的属性，内部使用场景也有很多，而我们经常使用的也就是onCreateViewHolder()方法和onBindViewHolder()方法。onCreateViewHolder()方法在RecyclerView需要一个新类型时创建一个ViewHolder；onBindViewHolder()方法则当RecyclerView需要在特定位置的item展示数据时调用。

### 如何理解ViewHolder的复用?

在复写RecyclerView.Adapter的时候，需要我们复写两个方法：onCreateViewHolder，onBindViewHolder。这两个方法从字面上看就是创建ViewHolder和绑定ViewHolder的意思。模拟场景：只有一种ViewType，上下滑动的时候需要的ViewHolder种类是只有一种，但是需要的ViewHolder对象数量并不止一个。所以在后面创建了9个ViewHolder之后，需要的数量够了，无论怎么滑动，都只需要复用以前创建的对象就行了。为什么会出现这种情况呢?看到了下面log之后，第一反应是在这个ViewHolder对象的数量“够用”之后就停止调用onCreateViewHolder方法，但是onBindViewHolder方法每次都会调用的.



查看一下createViewHolder源代码，发现这里并没有限制

public final VH createViewHolder(ViewGroup parent, int viewType) {

TraceCompat.beginSection(TRACE\_CREATE\_VIEW\_TAG);

final VH holder = onCreateViewHolder(parent, viewType);

holder.mItemViewType = viewType;

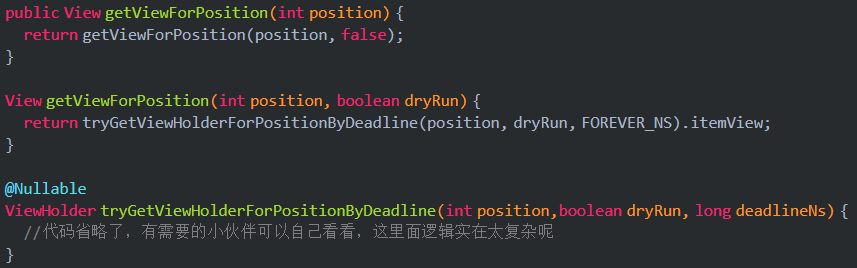
TraceCompat.endSection();

return holder;

}

对于ViewHolder对象的数量“够用”之后就停止调用onCreateViewHolder方法，可以查看

* 获取为给定位置初始化的视图。
* 此方法应由{@link LayoutManager}实现使用，以获取视图来表示来自{@LinkAdapter}的数据。
* 如果共享池可用于正确的视图类型，则回收程序可以重用共享池中的废视图或分离视图。如果适配器没有指示给定位置上的数据已更改，则回收程序将尝试发回一个以前为该数据初始化的报废视图，而不进行重新绑定。



### ViewHolder中为何使用SparseArray替代HashMap存储viewId？

#### HashMap

基本上就是一个 HashMap.Entry 的数组（Entry 是 HashMap 的一个内部类）

Entry 类中包含以下字段：

1. 一个非基本数据类型的 key
2. 一个非基本数据类型的 value
3. 保存对象哈希值的hashCode
4. 指向下一个 Entry 的指针

当有键值对插入时，HashMap 会发生什么 ?

1. 首先，键的哈希值被计算出来，然后这个值会赋给 Entry 类中对应的 hashCode 变量。
2. 然后，使用这个哈希值找到它将要被存入的数组中“桶”的索引。
3. 如果该位置的“桶”中已经有一个元素，那么新的元素会被插入到“桶”的头部，next 指向上一个元素——本质上使“桶”形成链表。
4. 现在，当你用 key 去查询值时，时间复杂度是 O(1)。虽然时间上 HashMap 更快，但同时它也花费了更多的内存空间。

缺点:

* 自动装箱的存在意味着每一次插入都会有额外的对象创建。这跟垃圾回收机制一样也会影响到内存的利用。
* HashMap.Entry 对象本身是一层额外需要被创建以及被垃圾回收的对象。
* “桶” 在 HashMap 每次被压缩或扩容的时候都会被重新安排。这个操作会随着对象数量的增长而变得开销极大
* 在Android中，当涉及到快速响应的应用时（比如RecyclerView），内存至关重要，因为持续地分发和释放内存会出发垃圾回收机制，这会拖慢应用运行。垃圾回收机制会影响应用性能表现，垃圾回收时间段内，应用程序是不会运行的，最终应用使用上就显得卡顿。

#### SparseArray

它里面也用了两个数组。一个int[] mKeys和Object[] mValues。从名字都可以看得出来一个用来存储key一个用来保存value的。

当保存一对键值对的时候：

* key（不是它的hashcode）保存在mKeys[]的下一个可用的位置上。所以不会再对key自动装箱了。
* value保存在mValues[]的下一个位置上，value还是要自动装箱的，如果它是基本类型。

查找的时候：

* 查找key还是用的二分法查找。也就是说它的时间复杂度还是O(logN)
* 知道了key的index，也就可以用key的index来从mValues中检索出value。

相较于HashMap,我们舍弃了Entry和Object类型的key,放弃了HashCode并依赖于二分法查找。虽然查找更慢，但在添加和删除操作的时候有更好的性能开销，更适合于RecyclerView使用

## LayoutManager相关

### LayoutManager作用是什么？

LayoutManager的职责是摆放Item的位置，并且负责决定何时回收和重用Item。RecyclerView 允许自定义规则去放置子 view，这个规则的控制者就是 LayoutManager。一个 RecyclerView 如果想展示内容，就必须设置一个 LayoutManager

### LayoutManager样式有哪些？

* LinearLayoutManager 水平或者垂直的Item视图。
* GridLayoutManager 网格Item视图。
* StaggeredGridLayoutManager 交错的网格Item视图。

### setLayoutManager(LayoutManager layout)源码里做了什么？

分析：当之前设置过 LayoutManager 时，移除之前的视图，并缓存视图在 Recycler 中，将新的 mLayout 对象与 RecyclerView 绑定，更新缓存 View 的数量。最后去调用 requestLayout ，重新请求 measure、layout、draw。

public void setLayoutManager(LayoutManager layout) {  
 if (layout == mLayout) {  
 return;  
 }  
 stopScroll();  
 if (mLayout != null) {  
 if (mItemAnimator != null) {  
 mItemAnimator.endAnimations();  
 }  
 mLayout.removeAndRecycleAllViews(mRecycler);  
 mLayout.removeAndRecycleScrapInt(mRecycler);  
 mRecycler.clear();  
 if (mIsAttached) {  
 mLayout.dispatchDetachedFromWindow(this, mRecycler);  
 }  
 mLayout.setRecyclerView(null);  
 mLayout = null;  
 } else {  
 mRecycler.clear();  
 }  
 mChildHelper.removeAllViewsUnfiltered();  
 mLayout = layout;  
 if (layout != null) {  
 if (layout.mRecyclerView != null) {  
 throw new IllegalArgumentException("LayoutManager " + layout  
 + " is already attached to a RecyclerView:"  
 + layout.mRecyclerView.exceptionLabel());  
 }  
 mLayout.setRecyclerView(this);  
 if (mIsAttached) {  
 mLayout.dispatchAttachedToWindow(this);  
 }  
 }  
 mRecycler.updateViewCacheSize();  
 requestLayout();  
}

## SpanSizeLookup相关

### SpanSizeLookup的作用是干什么的？

GridLayoutManager 可以实现网格布局，还可以用来设置网格中指定Item所占的列数，类似于合并单元格的功能，我们需要用到GridLayoutManager.SpanSizeLookup这个类，该类是一个抽象类，里面包含了一个getSpanSize(int position)的抽象方法，该方法的返回值就是指定position所占的列数

### SpanSizeLookup如何使用？

先是定义了一个6列的网格布局，然后通过GridLayoutManager.SpanSizeLookup这个类来动态的指定某个item应该占多少列。比如getSpanSize返回6，就表示当前position索引处的item占用6列，那么显示就只会展示一个ItemView【占用6列】。比如getSpanSize返回3，就表示当前position索引处的item占用3列

GridLayoutManager manager = new GridLayoutManager(this, 6);

manager.setSpanSizeLookup(new GridLayoutManager.SpanSizeLookup() {

@Override

public int getSpanSize(int position) {

SpanModel model = mDataList.get(position);

if (model.getType() == 1) {

return 6;

} else if(model.getType() == 2){

return 3;

} else {

return 1;

}

}

});

## ItemDecoration相关

### ItemDecoration的用途是什么？

recyclerView.addItemDecoration(new DividerItemDecoration(this,DividerItemDecoration.VERTICAL\_LIST));可以改变Item之间的偏移量或者对Item进行装饰（分隔线）。当然，你也可以对RecyclerView设置多个ItemDecoration，列表展示的时候会遍历所有的ItemDecoration并调用里面的绘制方法，对Item进行装饰。

### 自定义ItemDecoration有哪些重写方法？

该抽象类常见的方法如下所示：

* public void onDraw(Canvas c, RecyclerView parent)

装饰的绘制在Item条目绘制之前调用，所以这有可能被Item的内容所遮挡

* public void onDrawOver(Canvas c, RecyclerView parent)

装饰的绘制在Item条目绘制之后调用，因此装饰将浮于Item之上

* public void getItemOffsets(Rect outRect, int itemPosition, RecyclerView parent)

与padding或margin类似，LayoutManager在测量阶段会调用该方法，计算出每一个Item的正确尺寸并设置偏移量。

### 分析一下addItemDecoration()源码？

通过下面代码可知，mItemDecorations是一个ArrayList，我们将ItemDecoration也就是分割线对象，添加到其中。

可以看到，当通过这个方法添加分割线后，会指定添加分割线在集合中的索引，然后再重新请求 View 的测量、布局、绘制。注

public void addItemDecoration(ItemDecoration decor) {

addItemDecoration(decor, -1);

}

//主要看这个方法，我的GitHub：https://github.com/yangchong211/YCBlogs

public void addItemDecoration(ItemDecoration decor, int index) {

if (mLayout != null) {

mLayout.assertNotInLayoutOrScroll("Cannot add item decoration during a scroll or layout");

}

if (mItemDecorations.isEmpty()) {

setWillNotDraw(false);

}

if (index < 0) {

mItemDecorations.add(decor);

} else {

// 指定添加分割线在集合中的索引

mItemDecorations.add(index, decor);

}

markItemDecorInsetsDirty();

// 重新请求 View 的测量、布局、绘制

requestLayout();

}

在RecyclerView绘制时，会分别调用 ItemDecoration 对象的 onDraw和onDrawOver 方法。这两个抽象方法，由我们继承 ItemDecoration 来自己实现，他们区别就是 onDraw 在 item view 绘制之前调用，onDrawOver 在 item view 绘制之后调用。所以绘制顺序就是 ItemDecoration 的 onDraw ->ItemView的 onDraw -> ItemDecoration 的 onDrawOver。

### 如何实现可以设置自定义分割线

需要实现的分割线功能

可以设置分割线的颜色，宽度，以及到左右两边的宽度间距。item默认分割线的颜色不可改变，那么只有重写onDraw方法，通过设置画笔point颜色来绘制分割线颜色。而设置分割线左右的间隔是通过getItemOffsets方法实现的。

几个重要的方法说明

需要自定义类实现RecyclerView.ItemDecoration类，并选择重写合适方法。注意下面这三个方法有着强烈的因果关系！

//获取当前view的位置信息，该方法主要是设置条目周边的偏移量

public void getItemOffsets(Rect outRect, View view, RecyclerView parent, State state)

//在item背后draw

public void onDraw(Canvas c, RecyclerView parent, State state)

//在item上边draw

public void onDrawOver(Canvas c, RecyclerView parent, State state)

注意的是三个方法的调用顺序

* 首先调用的是getItemOffsets会被多次调用，在layoutManager每次测量可摆放的view的时候回调用一次，在当前状态下需要摆放多少个view这个方法就会回调多少次。
* 其次会调用onDraw方法，ItemDecoration的onDraw方法是在RecyclerView的onDraw方法中调用的，注意这时候传入的canvas是RecyclerView的canvas，要时刻注意这点，它是和RecyclerView的边界是一致的。这个时候绘制的内容相当于背景，会被item覆盖。
* 最后调用的是onDrawOver方法，ItemDecoration的onDrawOver方法是在RecyclerView的draw方法中调用的，同样传入的是RecyclerView的canvas，这时候onlayout已经调用，所以此时绘制的内容会覆盖item。

为每个item实现索引的思路

* 要实现上面的可以设置分割线颜色和宽度，肯定是要绘制的，也就是需要使用到onDraw方法。那么在getItemOffsets方法中需要让view摆放位置距离bottom的距离是分割线的宽度。
* 然后通过parent.getChildCount()方法拿到当前显示的view的数量[注意，该方法并不会获取不显示的view的数量]，循环遍历后，直接用paint画笔进行绘制[注意至于分割线的颜色就是需要设置画笔的颜色]。

## 上拉加载更多相关

### 上拉加载更多的功能是如何做的？

1. 添加recyclerView的滑动事件

首先给recyclerView添加滑动监听事件。那么我们知道，上拉加载时，需要具备两个条件。第一个是监听滑动到最后一个item，第二个是滑动到最后一个并且是向上滑动。

设置滑动监听器，RecyclerView自带的ScrollListener，获取最后一个完全显示的itemPosition，然后判断是否滑动到了最后一个item，

1. 上拉加载分页数据

然后开始调用更新上拉加载更多数据的方法。注意这里的刷新数据，可以直接用notifyItemRangeInserted方法，不要用notifyDataSetChanged方法。

1. 设置上拉加载的底部footer布局

* 在adapter中，可以上拉加载时处理footerView的逻辑
* 在getItemViewType方法中设置最后一个Item为FooterView
* 在onCreateViewHolder方法中根据viewType来加载不同的布局
* 最后在onBindViewHolder方法中设置一下加载的状态显示就可以
* 由于多了一个FooterView，所以要记得在getItemCount方法的返回值中加上1。

1. 显示和隐藏footer布局

一般情况下，滑动底部最后一个item，然后显示footer上拉加载布局，然后让其加载500毫秒，最后加载出下一页数据后再隐藏起来。

那么如何实现自动进行上拉刷新？

设置滑动监听，判断是否滑动到底部，也就是最后一条数据，当滑动到最后时就开始加载下一页数据，并且显示加载下一页loading。当加载数据成功后，则直接隐藏该布局。

那么如何实现手动上拉刷新呢？

在上面步骤的基础上进行修改，当滑动到最后一个数据时，展示上拉加载更多布局。然后设置它的点击事件，点击之后开始加载下一页数据，当加载完成后，则直接隐藏该布局。

### 网格布局上拉加载如何优化？

如果是网格布局，那么上拉刷新的view则不是居中显示，到加载更多的进度条显示在了一个Item上，如果想要正常显示的话，进度条需要横跨两个Item，这该怎么办呢？

在adapter中的onAttachedToRecyclerView方法中处理网格布局情况，代码如下所示，主要逻辑是如果当前是footer的位置，那么该item占据2个单元格，正常情况下占据1个单元格。

@Override

public void onAttachedToRecyclerView(@NonNull RecyclerView recyclerView) {

super.onAttachedToRecyclerView(recyclerView);

RecyclerView.LayoutManager manager = recyclerView.getLayoutManager();

if (manager instanceof GridLayoutManager) {

final GridLayoutManager gridManager = ((GridLayoutManager) manager);

gridManager.setSpanSizeLookup(new GridLayoutManager.SpanSizeLookup() {

@Override

public int getSpanSize(int position) {

// 如果当前是footer的位置，那么该item占据2个单元格，正常情况下占据1个单元格

return getItemViewType(position) == footType ? gridManager.getSpanCount() : 1;

}

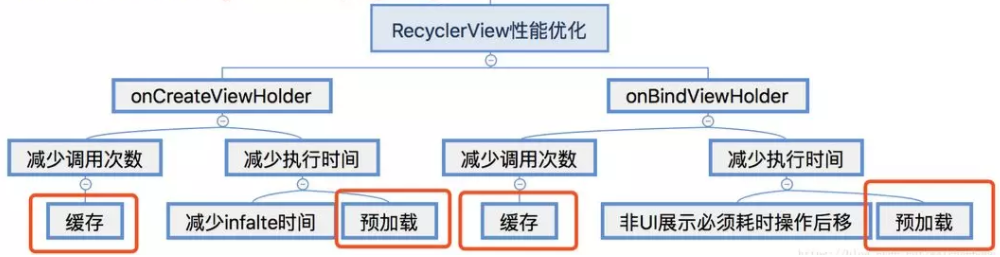
});

}

}

## 绘制原理相关

### RecyclerView绘制原理如何理解？



### 性能优化本质是什么？

RecyclerView做性能优化要说复杂也复杂，比如说布局优化，缓存，预加载，复用池，刷新数据等等。其优化的点很多，在这些看似独立的点之间，其实存在一个枢纽：Adapter。因为所有的ViewHolder的创建和内容的绑定都需要经过Adapter的两个函数onCreateViewHolder和onBindViewHolder。因此性能优化的本质就是要减少这两个函数的调用时间和调用的次数。

如果我们想对RecyclerView做性能优化，必须清楚的了解到我们的每一步操作背后，onCreateViewHolder和onBindViewHolder调用了多少次。

### RecyclerView绘制原理过程大概是怎样的？

简化问题RecyclerView以LinearLayoutManager为例，忽略ItemDecoration、ItemAnimator、Measure过程，假设RecyclerView的width和height是确定的，Recycler忽略mViewCacheExtension

* LayoutManager：接管RecyclerView的Measure，Layout，Draw的过程
* Recycler：缓存池
* Adapter：ViewHolder的生成器和内容绑定器。

绘制过程简介

1. RecyclerView.requestLayout开始发生绘制，忽略Measure的过程
2. 在Layout的过程会通过LayoutManager.fill去将RecyclerView填满
3. LayoutManager.fill会调用LayoutManager.layoutChunk去生成一个具体的ViewHolder
4. 然后LayoutManager就会调用Recycler.getViewForPosition向Recycler去要ViewHolder
5. Recycler首先去一级缓存（Cache）里面查找是否命中，如果命中直接返回。如果一级缓存没有找到，则去三级缓存查找，如果三级缓存找到了则调用Adapter.bindViewHolder来绑定内容，然后返回。如果三级缓存没有找到，那么就通过Adapter.createViewHolder创建一个ViewHolder，然后调用Adapter.bindViewHolder绑定其内容，然后返回为Recycler。
6. 一直重复步骤3-5，知道创建的ViewHolder填满了整个RecyclerView为止

## Recycler相关

### Recyler是如何实现ViewHolder的缓存？

public final class Recycler {  
 final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<>();  
 ArrayList<ViewHolder> mChangedScrap = null;  
 final ArrayList<ViewHolder> mCachedViews = new ArrayList<ViewHolder>();  
 private final List<ViewHolder> mUnmodifiableAttachedScrap = Collections.*unmodifiableList*(mAttachedScrap);  
 private int mRequestedCacheMax = *DEFAULT\_CACHE\_SIZE*;  
 int mViewCacheMax = *DEFAULT\_CACHE\_SIZE*;  
 RecycledViewPool mRecyclerPool;  
 private ViewCacheExtension mViewCacheExtension;  
 static final int *DEFAULT\_CACHE\_SIZE* = 2;

...  
}

RecyclerView在Recyler里面实现ViewHolder的缓存，Recycler里面的实现缓存的主要包含以下5个对象：

1. ArrayList mAttachedScrap：未与RecyclerView分离的ViewHolder列表,如果仍依赖于 RecyclerView （比如已经滑动出可视范围，但还没有被移除掉），但已经被标记移除的 ItemView 集合会被添加到 mAttachedScrap 中

* 按照id和position来查找ViewHolder

1. ArrayList mChangedScrap：表示数据已经改变的viewHolder列表,存储 notifXXX 方法时需要改变的 ViewHolder,

* 按照position和id进行匹配

1. ArrayList mCachedViews：缓存ViewHolder，主要用于解决RecyclerView滑动抖动时的情况，还有用于保存Prefetch的ViewHoder

* 最大的数量为：mViewCacheMax = mRequestedCacheMax + extraCache（extraCache是由prefetch的时候计算出来的）

1. ViewCacheExtension mViewCacheExtension：开发者可自定义的一层缓存，是虚拟类ViewCacheExtension的一个实例，开发者可实现方法getViewForPositionAndType(Recycler recycler, int position, int type)来实现自己的缓存。

* 位置固定
* 内容不变
* 数量有限

1. ViewCacheExtension mRecyclerPool： ViewHolder缓存池，在有限的mCachedViews中如果存不下ViewHolder时，就会把ViewHolder存入RecyclerViewPool中。

* 按照Type来查找ViewHolder
* 每个Type默认最多缓存5个

### 如何理解recyclerView三级缓存是如何实现的？

RecyclerView在设计的时候讲上述5个缓存对象分为了3级。每次创建ViewHolder的时候，会按照优先级依次查询缓存创建ViewHolder。每次将ViewHolder缓存到Recycler缓存的时候，也会按照优先级依次缓存进去。三级缓存分别是：

#### 一级缓存：返回布局和内容都都有效的ViewHolder

* 按照position或者id进行匹配；
* 命中一级缓存无需onCreateViewHolder和onBindViewHolder；
* mAttachScrap在adapter.notifyXxx的时候用到；
* mChanedScarp在每次View绘制的时候用到，因为getViewHolderForPosition非调用多次，后面讲；
* mCachedView：用来解决滑动抖动的情况，默认值为2

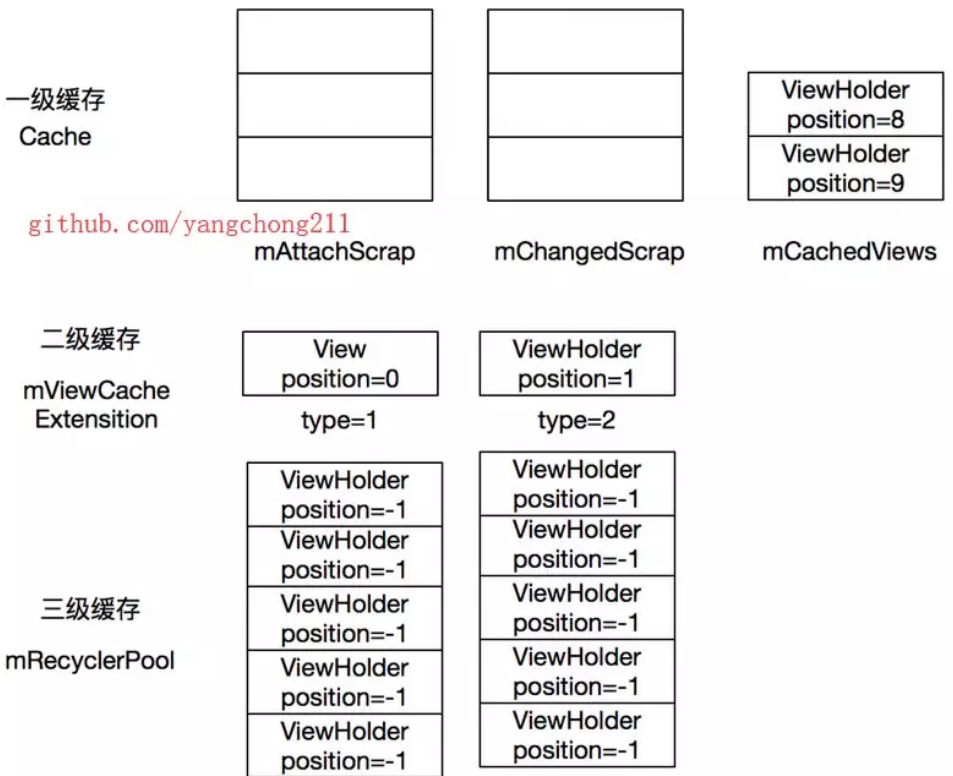
#### 二级缓存：返回View

* 按照position和type进行匹配
* 直接返回View
* 需要自己继承ViewCacheExtension实现
* 位置固定，内容不发生改变的情况，比如说Header如果内容固定，就可以使用

#### 三级缓存：返回布局有效，内容无效的ViewHolder

* 按照type进行匹配，每个type缓存值默认=5
* layout是有效的，但是内容是无效的
* 多个RecycleView可共享,可用于多个RecyclerView的优化

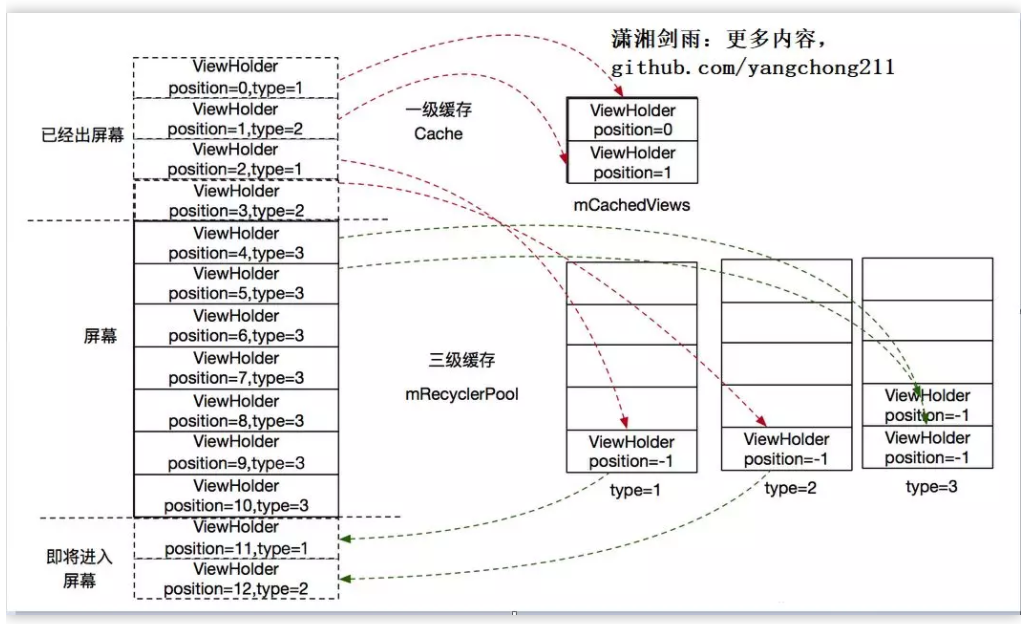
图解：



### 屏幕滑动状态

#### 屏幕滑动(状态是item状态可见，不可见，即将可见变化)时三级缓存是如何理解的？

如图所示



由于ViewCacheExtension在实际使用的时候较少用到，因此本例中忽略二级缓存。mChangedScrap和mAttchScrap是RecyclerView内部控制的缓存，本例暂时忽略。

* RecyclerView包含三部分：已经出屏幕，在屏幕里面，即将进入屏幕，我们滑动的方向是向上
* RecyclerView包含三种Type：1，2，3。屏幕里面的都是Type=3
* 红色的线代表已经出屏幕的ViewHolder与Recycler的交互情况
* 绿色的线代表，即将进入屏幕的ViewHolder进入屏幕时候，ViewHolder与Recycler的交互情况

#### 出屏幕时候的情况

* 当ViewHolder（position=0，type=1）出屏幕的时候，由于mCacheViews是空的，那么就直接放在mCacheViews里面，ViewHolder在mCacheViews里面布局和内容都是有效的，因此可以直接复用。 ViewHolder（position=1，type=2）同步骤1
* 当ViewHolder（position=2，type=1）出屏幕的时候由于一级缓存mCacheViews已经满了，因此将其放入RecyclerPool（type=1）的缓存池里面。此时ViewHolder的内容会被标记为无效，当其复用的时候需要再次通过Adapter.bindViewHolder来绑定内容。 ViewHolder（position=3，type=2）同步骤3

#### 进屏幕时候的情况

* 当ViewHolder（position=3-10，type=3）进入屏幕绘制的时候，由于Recycler的mCacheViews里面找不到position匹配的View，同时RecyclerPool里面找不到type匹配的View，因此，其只能通过adapter.createViewHolder来创建ViewHolder，然后通过adapter.bindViewHolder来绑定内容。
* 当ViewHolder（position=11，type=1）进入屏幕的时候，发现ReccylerPool里面能找到type=1的缓存，因此直接从ReccylerPool里面取来使用。由于内容是无效的，因此还需要调用bindViewHolder来绑定布局。同时ViewHolder（position=4，type=3）需要出屏幕，其直接进入RecyclerPool（type=3）的缓存池中
* ViewHolder（position=12，type=2）同步骤6

#### 屏幕往下拉ViewHolder（position=1）进入屏幕的情况

* 由于mCacheView里面的有position=1的ViewHolder与之匹配，直接返回。由于内容是有效的，因此无需再次绑定内容
* ViewHolder（position=0）同步骤8

## SnapHelper相关

### 主要是做什么用的？

在某些场景下，卡片列表滑动浏览[有的叫轮播图]，希望当滑动停止时可以将当前卡片停留在屏幕某个位置，比如停在左边，以吸引用户的焦点。那么可以使用RecyclerView + Snaphelper来实现

SnapHelper旨在支持RecyclerView的对齐方式，也就是通过计算对齐RecyclerView中TargetView 的指定点或者容器中的任何像素点。SnapHelper类重要的方法：

### 什么是Fling操作

手指在屏幕上滑动 RecyclerView然后松手，RecyclerView中的内容会顺着惯性继续往手指滑动的方向继续滚动直到停止，这个过程叫做 Fling 。 Fling 操作从手指离开屏幕瞬间被触发，在滚动停止时结束。

### 有哪些重要的方法，其作用就是是什么？

calculateDistanceToFinalSnap抽象方法

计算最终对齐要移动的距离，计算二个参数对应的 ItemView 当前的坐标与需要对齐的坐标之间的距离。该方法返回一个大小为 2 的 int 数组，分别对应out[0] 为 x 方向移动的距离，out[1] 为 y 方向移动的距离。

@SuppressWarnings("WeakerAccess")

@Nullable

public abstract int[] calculateDistanceToFinalSnap(@NonNull LayoutManager layoutManager, @NonNull View targetView);

findSnapView抽象方法

找到要对齐的View，该方法会找到当前 layoutManager 上最接近对齐位置的那个 view ，该 view 称为 SanpView ，对应的 position 称为 SnapPosition 。如果返回 null ，就表示没有需要对齐的 View ，也就不会做滚动对齐调整。

@SuppressWarnings("WeakerAccess")

@Nullable

public abstract View findSnapView(LayoutManager layoutManager);

findTargetSnapPosition抽象方法

找到需要对齐的目标View的的Position，更加详细一点说就是该方法会根据触发 Fling 操作的速率（参数 velocityX 和参数 velocityY ）来找到 RecyclerView 需要滚动到哪个位置，该位置对应的 ItemView 就是那个需要进行对齐的列表项。我们把这个位置称为 targetSnapPosition ，对应的 View 称为 targetSnapView 。如果找不到 targetSnapPosition ，就返回RecyclerView.NO\_POSITION 。

public abstract int findTargetSnapPosition(LayoutManager layoutManager, int velocityX, int velocityY);

其他

* attachToRecyclerView: 将SnapHelper attach 到指定的RecyclerView 上。
* calculateScrollDistance: 根据每个方向给定的速度估算滑动的距离，用于Fling 操作。
* onFling:根据给定的x和 y 轴上的速度处理Fling。

### LinearSnapHelper类分析

#### LinearSnapHelper如何使用

LinearSnapHelper 使当前Item居中显示，常用场景是横向的RecyclerView，类似ViewPager效果，但是又可以快速滑动（滑动多页）。

几行代码就可以用RecyclerView实现一个类似ViewPager的效果，并且效果还不错。可以快速滑动多页，当前页居中显示，并且显示前一页和后一页的部分。

LinearSnapHelper snapHelper = new LinearSnapHelper();

snapHelper.attachToRecyclerView(mRecyclerView);

#### out[0]和out[1]分别指什么

* out[0]是x方向对齐要移动的距离
* out[1]是y方向对齐要移动的距离。

#### LinearSnapHelper代码中calculateDistanceToFinalSnap作用是什么？

* 如果是水平方向滚动的，则计算水平方向需要移动的距离，否则水平方向的移动距离为0
* 如果是竖直方向滚动的，则计算竖直方向需要移动的距离，否则竖直方向的移动距离为0
* distanceToCenter方法主要作用是：计算水平或者竖直方向需要移动的距离

@Override

public int[] calculateDistanceToFinalSnap(RecyclerView.LayoutManager layoutManager, @NonNull View targetView) {

int[] out = new int[2];

if (layoutManager.canScrollHorizontally()) {

out[0] = distanceToCenter(layoutManager, targetView, getHorizontalHelper(layoutManager));

} else {

out[0] = 0;

}

if (layoutManager.canScrollVertically()) {

out[1] = distanceToCenter(layoutManager, targetView,getVerticalHelper(layoutManager));

} else {

out[1] = 0;

}

return out;

}

* 接着看看distanceToCenter方法
  + 计算对应的view的中心坐标到RecyclerView中心坐标之间的距离
  + 首先是找到targetView的中心坐标
  + 接着也就是找到容器【RecyclerView】的中心坐标
  + 两个中心坐标的差值就是targetView需要滚动的距离

private int distanceToCenter( RecyclerView.LayoutManager layoutManager, View targetView, OrientationHelper helper) {

final int childCenter = helper.getDecoratedStart(targetView) + (helper.getDecoratedMeasurement(targetView) / 2);

final int containerCenter;

if (layoutManager.getClipToPadding()) {

containerCenter = helper.getStartAfterPadding() + helper.getTotalSpace() / 2;

} else {

containerCenter = helper.getEnd() / 2;

}

return childCenter - containerCenter;

}

#### LinearSnapHelper中是如何实现滚动停止的？

SnapHelper继承了 RecyclerView.OnFlingListener，实现了onFling方法。

获取RecyclerView要进行fling操作需要的最小速率，为啥呢？因为只有超过该速率，ItemView才会有足够的动力在手指离开屏幕时继续滚动下去。该方法返回的是一个布尔值！

@Override

public boolean onFling(int velocityX, int velocityY) {

LayoutManager layoutManager = mRecyclerView.getLayoutManager();

if (layoutManager == null) {

return false;

}

RecyclerView.Adapter adapter = mRecyclerView.getAdapter();

if (adapter == null) {

return false;

}

int minFlingVelocity = mRecyclerView.getMinFlingVelocity();

return (Math.abs(velocityY) > minFlingVelocity || Math.abs(velocityX) > minFlingVelocity)

&& snapFromFling(layoutManager, velocityX, velocityY);

}

接着看看snapFromFling方法源代码，就是通过该方法实现平滑滚动并使得在滚动停止时itemView对齐到目的坐标位置

1. 首先layoutManager必须实现ScrollVectorProvider接口才能继续往下操作
2. 然后通过createSnapScroller方法创建一个SmoothScroller，这个东西是一个平滑滚动器，用于对ItemView进行平滑滚动操作
3. 根据x和y方向的速度来获取需要对齐的View的位置，需要子类实现
4. 最终通过 SmoothScroller 来滑动到指定位置
5. 总结一下可知：snapFromFling()方法会先判断layoutManager是否实现了ScrollVectorProvider接口，如果没有实现该接口就不允许通过该方法做滚动操作。接下来就去创建平滑滚动器SmoothScroller的一个实例，layoutManager可以通过该平滑滚动器来进行滚动操作。SmoothScroller需要设置一个滚动的目标位置，将通过findTargetSnapPosition()方法来计算得到的targetSnapPosition给它，告诉滚动器要滚到这个位置，然后就启动SmoothScroller进行滚动操作。

private boolean snapFromFling(@NonNull LayoutManager layoutManager, int velocityX, int velocityY) {

if (!(layoutManager instanceof ScrollVectorProvider)) {

return false;

}

RecyclerView.SmoothScroller smoothScroller = createSnapScroller(layoutManager);

if (smoothScroller == null) {

return false;

}

int targetPosition = findTargetSnapPosition(layoutManager, velocityX, velocityY);

if (targetPosition == RecyclerView.NO\_POSITION) {

return false;

}

smoothScroller.setTargetPosition(targetPosition);

layoutManager.startSmoothScroll(smoothScroller);

return true;

}

接着看下createSnapScroller这个方法源码

1. 先判断layoutManager是否实现了ScrollVectorProvider这个接口，没有实现该接口就不创建SmoothScroller
2. 这里创建一个LinearSmoothScroller对象，然后返回给调用函数，也就是说，最终创建出来的平滑滚动器就是这个LinearSmoothScroller
3. 在创建该LinearSmoothScroller的时候主要考虑两个方面：

* 第一个是滚动速率，由calculateSpeedPerPixel()方法决定；
* 第二个是在滚动过程中，targetView即将要进入到视野时，将匀速滚动变换为减速滚动，然后一直滚动目的坐标位置，使滚动效果更真实，这是由onTargetFound()方法决定。

@Nullable

protected LinearSmoothScroller createSnapScroller(LayoutManager layoutManager) {

if (!(layoutManager instanceof ScrollVectorProvider)) {

return null;

}

return new LinearSmoothScroller(mRecyclerView.getContext()) {

@Override

protected void onTargetFound(View targetView, RecyclerView.State state, Action action) {

int[] snapDistances = calculateDistanceToFinalSnap(mRecyclerView.getLayoutManager(),

targetView);

final int dx = snapDistances[0];

final int dy = snapDistances[1];

final int time = calculateTimeForDeceleration(Math.max(Math.abs(dx), Math.abs(dy)));

if (time > 0) {

action.update(dx, dy, time, mDecelerateInterpolator);

}

}

@Override

protected float calculateSpeedPerPixel(DisplayMetrics displayMetrics) {

return MILLISECONDS\_PER\_INCH / displayMetrics.densityDpi;

}

};

}

### PagerSnapHelper类分析

PagerSnapHelper看名字可能就能猜到，使RecyclerView像ViewPager一样的效果，每次只能滑动一页（LinearSnapHelper支持快速滑动）, PagerSnapHelper也是Item居中对齐。最简单的使用就是，如下代码：

PagerSnapHelper snapHelper = new PagerSnapHelper();

snapHelper.attachToRecyclerView(mRecyclerView);

## 如何实现复杂type首页需求

### 如何实现

通常写一个多Item列表的方法，根据不同的ViewType 处理不同的item，如果逻辑复杂，这个类的代码量是很庞大的。如果版本迭代添加新的需求，修改代码很麻烦，后期维护困难。

主要操作步骤在onCreateViewHolder中根据viewType参数，也就是getItemViewType的返回值来判断需要创建的ViewHolder类型。在onBindViewHolder方法中对ViewHolder的具体类型进行判断，分别为不同类型的ViewHolder进行绑定数据与逻辑处理，代码如下所示：

public class HomeAdapter extends RecyclerView.Adapter {

public static final int TYPE\_BANNER = 0;

public static final int TYPE\_AD = 1;

@Override

public int getItemViewType(int position) {

if(position == 0){

return TYPE\_BANNER;//banner在开头

}else {

return mData.get(position).type;//type 的值为TYPE\_AD，TYPE\_IMAGE，TYPE\_AD，等其中一个

}

}

@Override

public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {

LayoutInflater inflater = LayoutInflater.from(parent.getContext());

switch (viewType){

case TYPE\_BANNER:

return new BannerViewHolder(inflater.inflate(R.layout.home\_banner\_layout,null));

case TYPE\_AD:

return new BannerViewHolder(inflater.inflate(R.layout.home\_ad\_item\_layout,null));

}

return null;

}

@Override

public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {

int type = getItemViewType(position);

switch (type){

case TYPE\_BANNER:

// banner 逻辑处理

break;

case TYPE\_AD:

// 广告逻辑处理

break;

// ... 此处省去N行代码

}

}

public static class BannerViewHolder extends RecyclerView.ViewHolder{

public BannerViewHolder(View itemView) {

super(itemView);

}

}

public static class NewViewHolder extends RecyclerView.ViewHolder{

public VideoViewHolder(View itemView) {

super(itemView);

}

}

}

### 如果不封装会出现什么问题和弊端?

RecyclerView 可以用ViewType来区分不同的item,也可以满足需求，但还是存在一些问题，比如：

1，在item过多逻辑复杂列表界面，Adapter里面的代码量庞大，逻辑复杂，后期难以维护。

2，每次增加一个列表都需要增加一个Adapter，重复搬砖，效率低下。

3，无法复用adapter，假如有多个页面有多个type，那么就要写多个adapter。

4，要是有局部刷新，那么就比较麻烦了，比如广告区也是一个九宫格的RecyclerView，点击局部刷新当前数据，比较麻烦。

因此最好使用封装好的库，比如MultiTypeAdapter库

## item条目点击事件应在onCreateViewHolder中写

rv设置item条目点击事件有两种方式：1.在onCreateViewHolder中写；2.在onBindViewHolder中写。由于在onBindViewHolder() 中会频繁创建新的 onClickListener 实例，没有必要，因此，建议实际开发中应该在 onCreateViewHolder() 中每次为新建的 View 设置一次就行。

1.在onCreateViewHolder中写@NonNull

@Override

public MyViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {

final View view = LayoutInflater.from(mContext).inflate(R.layout.item\_me\_gv\_grid, parent, false);

final MyViewHolder holder = new MyViewHolder(view);

view.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

if (listener != null) {

listener.onItemClick(view, holder.getLayoutPosition());

}

}

});

return holder;

}

2.在onBindViewHolder中写@Override

public void onBindViewHolder(@NonNull final MyViewHolder holder, int position) {

holder.itemView.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

if (listener != null) {

listener.onItemClick(holder.itemView, holder.getAdapterPosition());

}

}

});

}

## RecyclerView滑动卡顿

### RecyclerView滑动卡顿原因有哪些？

* 第一种：嵌套布局滑动冲突。导致嵌套滑动难处理的关键原因在于当子控件消费了事件, 那么父控件就不会再有机会处理这个事件了, 所以一旦内部的滑动控件消费了滑动操作, 外部的滑动控件就再也没机会响应这个滑动操作了
* 第二种：嵌套布局层次太深，比如六七层等。测量，绘制布局可能会导致滑动卡顿
* 第三种：比如用RecyclerView实现画廊，加载比较大的图片，如果快速滑动，则可能会出现卡顿，主要是加载图片需要时间
* 第四种：在onCreateViewHolder或者在onBindViewHolder中做了耗时的操作导致卡顿。

### 如何解决RecyclerView实现画廊卡顿？

RecyclerView 滑动时不让 Glide 加载图片。滚动停止后才开始恢复加载图片。

//RecyclerView.SCROLL\_STATE\_IDLE //空闲状态

//RecyclerView.SCROLL\_STATE\_FLING //滚动状态

//RecyclerView.SCROLL\_STATE\_TOUCH\_SCROLL //触摸后状态

recyclerView.addOnScrollListener(new RecyclerView.OnScrollListener() {

@Override

public void onScrollStateChanged(@NonNull RecyclerView recyclerView, int newState) {

super.onScrollStateChanged(recyclerView, newState);

if (newState == RecyclerView.SCROLL\_STATE\_IDLE) {

LoggerUtils.e("initRecyclerView"+ "恢复Glide加载图片");

Glide.with(ImageBrowseActivity.this).resumeRequests();

}else {

LoggerUtils.e("initRecyclerView"+"禁止Glide加载图片");

Glide.with(ImageBrowseActivity.this).pauseRequests();

}

}

});

### 耗时的操作导致卡顿

按stackoverflow上面比较通俗的解释：RecyclerView.Adapter里面的onCreateViewHolder()方法和onBindViewHolder()方法对时间都非常敏感。类似I/O读写，Bitmap解码一类的耗时操作，最好不要在它们里面进行。

RecyclerView常见的优化有哪些？实际开发中都是怎么做的，优化前后对比性能上有何提升？

## RecyclerView常见的优化有哪些

### DiffUtil刷新优化

分页拉取远端数据，对拉取下来的远端数据进行缓存，提升二次加载速度；对于新增或者删除数据通过 DiffUtil 来进行局部刷新数据，而不是一味地全局刷新数据。

### 布局优化

减少 xml 文件 inflate 时间

这里的 xml 文件不仅包括 layout 的 xml，还包括 drawable 的 xml，xml 文件 inflate 出 ItemView 是通过耗时的 IO 操作，尤其当 Item 的复用几率很低的情况下，随着 Type 的增多，这种 inflate 带来的损耗是相当大的，此时我们可以用代码去生成布局，即 new View() 的方式，只要搞清楚 xml 中每个节点的属性对应的 API 即可。

减少 View 对象的创建

一个稍微复杂的 Item 会包含大量的 View，而大量的 View 的创建也会消耗大量时间，所以要尽可能简化 ItemView；设计 ItemType 时，对多 ViewType 能够共用的部分尽量设计成自定义 View，减少 View 的构造和嵌套。

### 对itemView中孩子View的点击事件优化

onBindViewHolder() 中频繁创建新的 onClickListener 实例没有必要，建议实际开发中应该在 onCreateViewHolder() 中每次为新建的 View 设置一次就行。

### 其他的一些优化点

* 如果 Item 高度是固定的话，可以使用 RecyclerView.setHasFixedSize(true); 来避免 requestLayout 浪费资源；
  + 注释说当知道Adapter内Item的改变不会影响RecyclerView宽高的时候，可以设置为true让RecyclerView避免重新计算大小。设置为true,再调用notifyDataSetChanged()，发现大小重新计算了，看来理解出现错误了。
  + 当我们确定Item的改变不会影响RecyclerView的宽高的时候可以设置setHasFixedSize(true)，并通过Adapter的增删改插方法去刷新RecyclerView，而不是通过notifyDataSetChanged()。
  + 其实可以直接设置为true，当需要改变宽高的时候就用notifyDataSetChanged()去整体刷新一下，因为调用notifyDataSetChanged()，大小一定会重新计算。
* 设置 RecyclerView.addOnScrollListener(listener); 来对滑动过程中停止加载的操作。
* ((SimpleItemAnimator) rv.getItemAnimator()).setSupportsChangeAnimations(false)，如果不要求动画，可以通过把默认动画关闭来提升效率。
* 通过重写 RecyclerView.onViewRecycled(holder) 来回收资源。
* 通过 RecycleView.setItemViewCacheSize(size); 来加大 RecyclerView 的缓存，用空间换时间来提高滚动的流畅性。
* 如果多个 RecycledView 的 Adapter 是一样的，比如嵌套的 RecyclerView 中存在一样的 Adapter，可以通过设置 RecyclerView.setRecycledViewPool(pool); 来共用一个 RecycledViewPool。

## 如何处理ViewPager嵌套水平RecyclerView横向滑动到底后不滑动ViewPager？

继承RecyclerView，重写dispatchTouchEvent，根据ACTION\_MOVE的方向判断是否调用父布局的parentView.requestDisallowInterceptTouchEvent(boolean disallowIntercept)去阻止父view拦截点击事件

/\*\*

\* 设置父布局是否获取点击事件，flag为true时表示父布局获取

\* 即requestDisallowInterceptTouchEvent(flase)；表示拦截传向子布局

\*/

**private** **void** **setParentScrollAble**(**boolean** flag) {

**if** (parentView != **null**) {

parentView.requestDisallowInterceptTouchEvent(!flag);

}

}

# RecycleView、Sv、Lv去掉拉到尽头的阴影效果

适用于2.3及以上的 否则不用设置  
android:overScrollMode="never"

# ScrollView

## [scrollTo(),scrollBy(),getScrollX(), getScrollY()](http://blog.csdn.net/bigconvience/article/details/26697645)

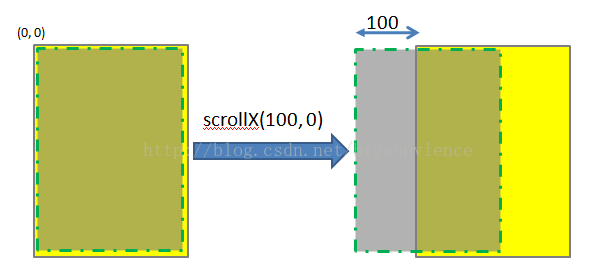
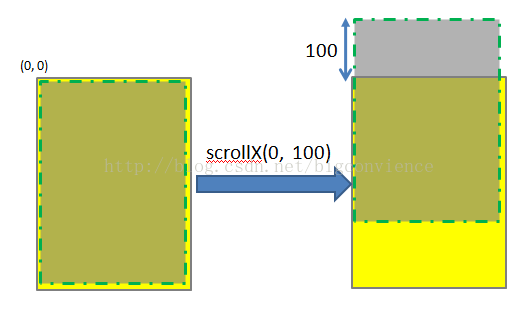
当我们编写一些自定义的滑动控件时，会用到一些api如scrollTo(),scrollBy(),getScrollX(), getScrollY()。由于常常会对函数getScrollX(), getScrollY()返回的值的含义产生混淆，尤其是正负关系，因此本文将使用几幅图来对这些函数进行讲解以方便大家记忆。

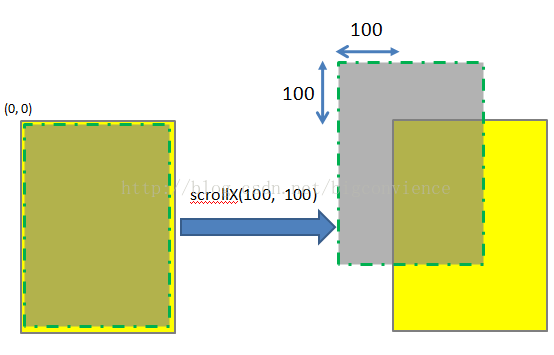
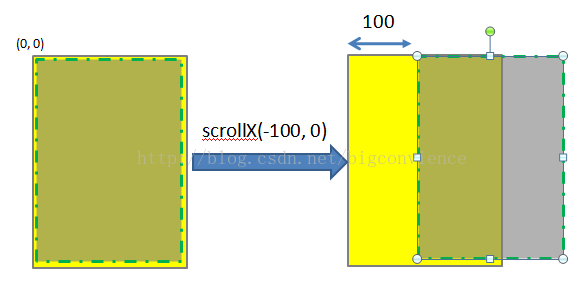
**注意：**

调用View的scrollTo()和scrollBy()是用于滑动View中的内容，而不是把某个View的位置进行改变。如果想改变莫个View在屏幕中的位置，可以public void offsetLeftAndRight(int offset)用于左右移动方法或public void offsetTopAndBottom(int offset)用于上下移动。如：

button.offsetLeftAndRignt(300)//将button控件向左移动300个像素。

scrollTo(int x, int y) 是将View中内容滑动到相应的位置，参考的坐标系原点为parent View的左上角。调用scrollTo(100, 0)表示将View中的内容移动到x = 100， y = 0的位置，如下图所示。注意，图中黄色矩形区域表示的是一个ScrollView(parent View)，绿色虚线矩形为ScrollView(parent view)中的内容。一般情况下两者的大小一致，本文为了显示方便，将虚线框画小了一点。图中的黄色区域的位置始终不变，发生位置变化的是显示的内容。

scrollBy(int x, int y)其实是对scrollTo的包装，移动的是相当位置。 scrollTo(int x, int y)的源码和scrollBy(int x, int y)源码如下所示.

public void scrollTo(int x, int y) {

if (mScrollX != x || mScrollY != y) {

int oldX = mScrollX;

int oldY = mScrollY;

mScrollX = x;

mScrollY = y;

invalidateParentCaches();

onScrollChanged(mScrollX, mScrollY, oldX, oldY);

if (!awakenScrollBars()) {

postInvalidateOnAnimation();

}

}

}

public void scrollBy(int x, int y) {

scrollTo(mScrollX + x, mScrollY + y);

}

可见，mScrollX和mScrollY是View类中专门用于记录滑动位置的变量。这两个函数最终调用onScrollChanged()函数，感兴趣者可以参考他们的源代码。

理解了scrollTo(int x, int y)和scrollBy(int x, int y)的用法，就不难理解getScrollX() 和getScrollY()。这两个函数的源码如下所示：

public final int getScrollX() {

return mScrollX;

}

public final int getScrollY() {

return mScrollY;

}

smoothScrollBy() 、smoothscrollTo()和他们一样的用法参数，只是多了平滑滑动的动画

## 判断是否滑到顶部和底部

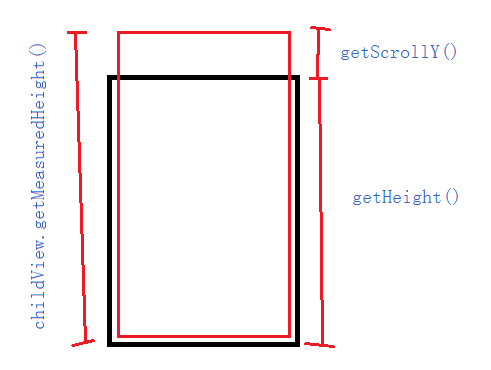
**滚动到顶部判断:**

getScrollY() == 0

**滚动到底部判断:**

View childView = getChildAt(0);

childView.getMeasuredHeight() == getScrollY() + getHeight();



**其中**

* getChildAt表示得到ScrollView的child View，
* childView.getMeasuredHeight()表示得到子View的高度,
* getScrollY()表示得到y轴的滚动距离，
* getHeight()为scrollView的高度
* getScrollY()达到最大时加上scrollView的高度就的就等于它内容的高度了.

## 判断滑动位置的地方，可以有两种方式：

**实现OnTouchListener来监听是否滑动到最顶部、底部**

**private** OnTouchListener oTouchListener = **new** OnTouchListener() {

@Override

**public** **boolean** **onTouch**(**View** v, **MotionEvent** event) {

**switch** (event.getAction()) {

**case** **MotionEvent**.**ACTION\_UP**:

**if** (childView != **null** && childView .getMeasuredHeight() == sc.getScrollY() + sc.getHeight()) {

**Toast**.*makeText*(**MainActivity**.**this**, "到底了", **Toast**.**LENGTH\_SHORT**).show();

} **else** **if** (sc.getScrollY() == 0) {

**Toast**.*makeText*(**MainActivity**.**this**, "到顶了", **Toast**.**LENGTH\_SHORT**).show();

}

**break**;

}

**return** **false**;

}

};

**重写ScrollView的onScrollChanged的方法，在onScrollChanged函数中判断**

**public** **class** **MyScrollView** **extends** **ScrollView** {

**public** **MyScrollView**(**Context** context) {

**super**(context);

}

**public** **MyScrollView**(**Context** context, AttributeSet attributeSet) {

**super**(context,attributeSet);

}

@Override

**protected** **void** **onScrollChanged**(**int** l, **int** t, **int** oldl, **int** oldt) {

**View** view = (**View**)getChildAt(0);

**if**(view.getHeight() == getHeight()+getScrollY()) {

**Log**.*e*("tag", "到底了");

} **else** **if** (getScrollY() == 0) {

**Log**.*e*("tag", "到顶了");

} **else** {

**super**.onScrollChanged(l,t,oldl,oldt);

}

}

}

## scrollview嵌套viewpager或listview引起滑动冲突的解决方案

两种解决方法：

1. 重写scrollview类的onInterceptTouchEvent方法，如果发现是水平滑动的话，那就在该方法内返回false，这样就不拦截viewpager了，会执行viewpager的onTouchEvent方法；
2. 重写viewpager中的onTouchEvent方法，如果MotionEvent.ACTION\_DOWN或者MotionEvent.ACTION\_MOVE的水平滑动就设置getParent().requestDisallowInterceptTouchEvent(true)，如果是竖直滑动就设置为false.

## ScrollView自己的方法移动到顶部、底部

scrollView.fullScroll(ScrollView.FOCUS\_DOWN);//滚动到底部

scrollView.fullScroll(ScrollView.FOCUS\_UP);//滚动到顶部

需要注意的是，该方法不能直接被调用，因为Android很多函数都是基于消息队列来同步，所以需要一部操作，addView完之后，不等于马上就会显示，而是在队列中等待处理，虽然很快，但是如果立即调用fullScroll， view可能还没有显示出来，所以会失败。应该通过handler或view本身在新线程中更新

handler.post(new Runnable() {

@Override

public void run() {

scrollView.fullScroll(ScrollView.FOCUS\_DOWN);

}

});

scrollView.post(new Runnable() {

@Override

public void run() {

scrollView.fullScroll(ScrollView.FOCUS\_DOWN);

}

});

## Android版本在23以前，监听ScrollView滚动事件

在android手机版本23以后，可以设置sc.setOnScrollChangeListener()即可，而在23以前可以通过继承ScrollView的子类中复写onScrollChanged()方法实现，以下以监听HorizontalScrollView横向滚动举例：

public class ObservableScrollView extends ScrollView {

//滚动监听回调

private OnScrollChangedListener onScrollChangedListener;

public ObservableScrollView(Context context, AttributeSet attrs, int defStyle) {

super(context, attrs, defStyle);

}

public ObservableScrollView(Context context, AttributeSet attrs) {

super(context, attrs);

}

public ObservableScrollView(Context context) {

super(context);

}

/\*\*

\* 复写protected的系统滚动监听

\*/

@Override

protected void onScrollChanged(int l, int t, int oldl, int oldt) {

super.onScrollChanged(l, t, oldl, oldt);

if(this.onScrollChangedListener != null) {

//将滚动事件通过回调传给监听者

onScrollChangedListener.onScrollChanged(t, oldt);

}

}

/\*\*

\* 提供给监听者的设置回调

\*/

public void setOnScrollChangedListener(OnScrollChangedListener onScrollChangedListener) {

this.onScrollChangedListener = onScrollChangedListener;

}

/\*\*

\* 滚动监听回调接口

\*/

public interface OnMyScrollChangedListener {

void onScrollChanged(int l, int oldl);

}

}

## 删除边界阴影

* 在xml中添加：android:fadingEdge="none"
* 代码中添加：setHorizontalFadingEdgeEnabled(false);

# ScrollView嵌套ListView

现象：listview里数据源里好多数据，但ListView只是显示1行，listview不能滚动，或者固定高度，不能滚动

原因：scrollview无法正确测量listview的高度

**解决办法两种：**

### 重写Listview

onMeasure里设置超级大的高度Size：Integer.MAX\_VALUE >> 2，且模式为：MeasureSpec.AT\_MOST。这样listview 能显示完，不能滚动，子项很多时会很长，适用于子项较少的情况。有时会出问题，就是默认显示的首项是ListView，需要手动把ScrollView滚动至最顶端。sc.smoothScrollTo(0, 0);

**public** **void** **onMeasure**(**int** widthMeasureSpec, **int** heightMeasureSpec) {

**int** expandSpec = **MeasureSpec**.makeMeasureSpec(**Integer**.**MAX\_VALUE** >> 2, **MeasureSpec**.**AT\_MOST**);

**super**.onMeasure(widthMeasureSpec, expandSpec);

}

### 固定高度，获得滚动权限

Listview高度是固定的，一条子项或几十条子项高度都是300dp，但是子项很多时，listview能滚动，适用于子项较多的情况

**public** **class** **ListViewForScro** **extends** **ListView**{

**private** **ScrollView** parentScrollView;

**public** **void** **setParent**(**ScrollView** scrollView){

parentScrollView = scrollView;

}

**public** **ListViewForScro**(**Context** context) {

**this**(context, **null**);

}

**public** **ListViewForScro**(**Context** context, AttributeSet attrs) {

**this**(context, attrs, 0);

}

**public** **ListViewForScro**(**Context** context, AttributeSet attrs, **int** defStyleAttr) {

**super**(context, attrs, defStyleAttr);

}

@Override

**public** **boolean** **onInterceptTouchEvent**(**MotionEvent** ev) {

**switch** (ev.getAction()) {

**case** **MotionEvent**.**ACTION\_DOWN**:

setParentScrollAble(**false**);//当手指触到listview的时候，让父ScrollView交出ontouch权限

**break**;

**case** **MotionEvent**.**ACTION\_CANCEL**:

setParentScrollAble(**true**);//当手指松开时，让父ScrollView重新拿到onTouch权限

**break**;

**default**:

**break**;

}

**return** **super**.onInterceptTouchEvent(ev);

}

/\*\*

\* 设置父布局是否获取点击事件，flag为true时表示父布局获取

\* 即requestDisallowInterceptTouchEvent(flase)；表示拦截传向子布局

\*/

**private** **void** **setParentScrollAble**(**boolean** flag) {

**if** (parentScrollView != **null**) {

parentScrollView.requestDisallowInterceptTouchEvent(!flag);//这里的parentScrollView就是外面的那个scrollview

}

}

}

<com.cgy.scolllist.ListViewForScro

android:id="@+id/lv"

android:layout\_width="match\_parent"

android:layout\_height="300dp" />

scrollview = (**ScrollView**) findViewById(**R**.**id**.**sc**);

listview = (**ListViewForScro**) findViewById(**R**.**id**.**lv**);

listview.setParent(scrollview);

listview.setAdapter(**new** ArrayAdapter<>(**MainActivity**.**this**, **R**.**layout**.**item**, list));

### 计算出其高度，然后根据需求设定

scrollview = (**ScrollView**) findViewById(**R**.**id**.**sc**);

listview = (**ListViewForScro**) findViewById(**R**.**id**.**lv**);

listview.setParent(scrollview);

listview.setAdapter(**new** ArrayAdapter<>(**MainActivity**.**this**, **R**.**layout**.**item**, list));

fixListViewHeight(lv);

/\*\*

\* 计算出并修正listview的高度，并使其小于等于某值

\*/

**private** **void** **fixListViewHeight**(**ListView** listView) {

ListAdapter listAdapter = listView.getAdapter();

**int** totalHeight = 0;

**if** (listAdapter == **null**) {

**return**;

}

**for** (**int** i = 0, len = listAdapter.getCount(); i < len; i++) {

**View** listViewItem = listAdapter.getView(i , **null**, listView);

// 计算子项View 的宽高

listViewItem.measure(0, 0);

// 计算所有子项的高度和

totalHeight += listViewItem.getMeasuredHeight();

}

**ViewGroup**.**LayoutParams** params = listView.getLayoutParams();

// listView.getDividerHeight()获取子项间分隔符的高度

// params.height设置ListView完全显示需要的高度

totalHeight += (listView.getDividerHeight() \* (listAdapter.getCount() - 1));

**if** (totalHeight > 500) {

totalHeight = 500;

}

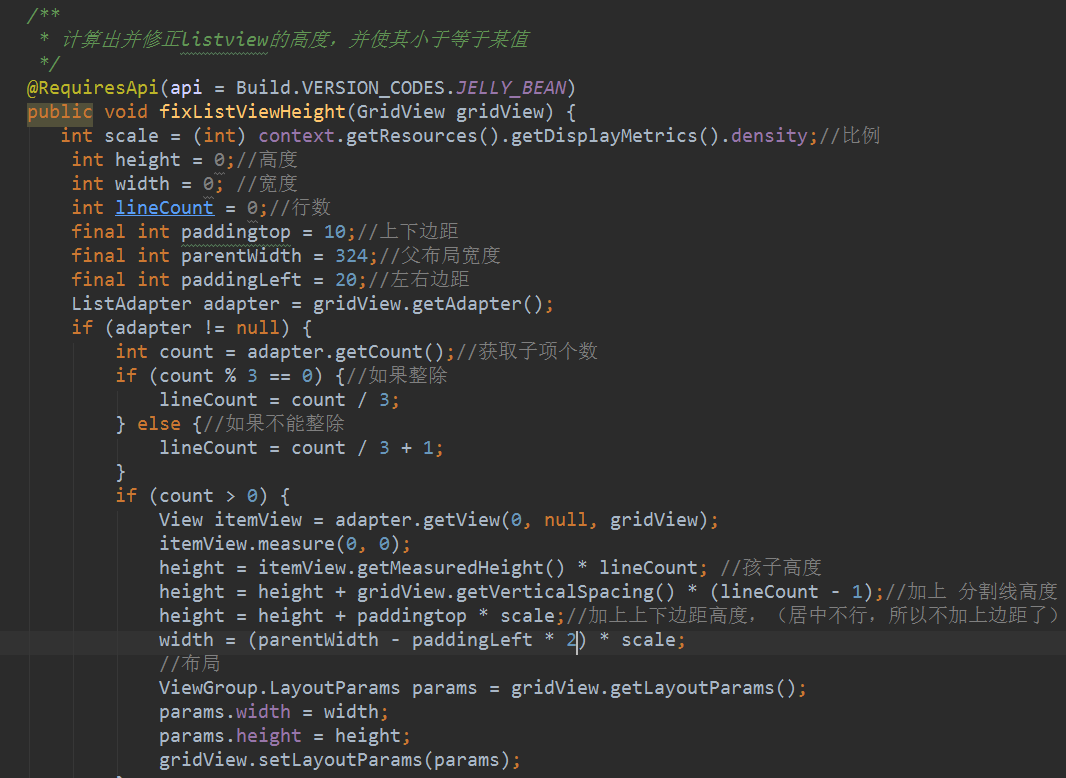
params.height = totalHeight;

listView.setLayoutParams(params);

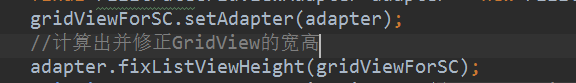
}

# ScrollView嵌套GridView

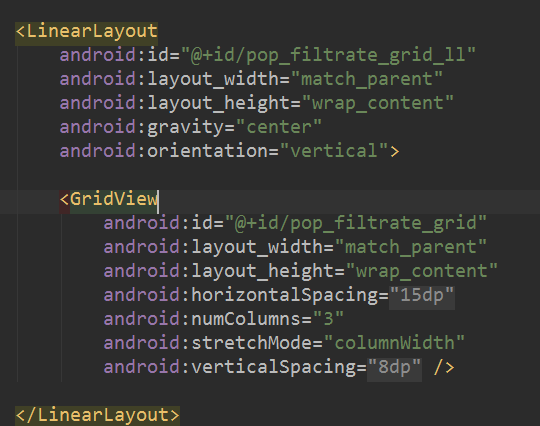
GridView在ScrollView中先要设置左右上下边距的解决方案:由于在Scrollview中，会导致就算Scollview的包裹层的layoutparams也无效，只能自己计算，在adapter中添加方法



**代码中**



**Xml中**



# ScrollView和SmartRefreshLayout嵌套多个RecyclerView

## ScrollView内嵌有多个RecyclerView

<?xml version="1.0" encoding="utf-8"?>  
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/scroll\_view"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent">  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical">  
  
 <!--其它的View-->  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content">  
  
 <TextView  
 android:id="@+id/header\_view"  
 android:layout\_width="match\_parent"  
 android:layout\_height="80dp"  
 android:gravity="center"  
 android:text="这个是RecyclerView" />  
 </LinearLayout>  
  
 <!--RecyclerView1-->  
 <RelativeLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:descendantFocusability="blocksDescendants"  
 android:focusable="true"  
 android:focusableInTouchMode="true">  
  
 <android.support.v7.widget.RecyclerView  
 android:id="@+id/recycler\_view\_one"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:nestedScrollingEnabled="false" />  
 </RelativeLayout>  
  
 <!--RecyclerView2-->  
 <RelativeLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:descendantFocusability="blocksDescendants"  
 android:focusable="true"  
 android:focusableInTouchMode="true">  
  
 <android.support.v7.widget.RecyclerView  
 android:id="@+id/recycler\_view\_two"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:layout\_marginTop="10dp"  
 android:nestedScrollingEnabled="false" />  
 </RelativeLayout>  
 </LinearLayout>  
</ScrollView>

## SmartRefreshLayout内嵌有多个RecyclerView

<?xml version="1.0" encoding="utf-8"?>  
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical">  
  
 <com.scwang.smartrefresh.layout.SmartRefreshLayout  
 android:id="@+id/smart\_refresh\_layout"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 app:srlEnableHeaderTranslationContent="false"  
 app:srlEnableLoadmore="true">  
  
 <com.scwang.smartrefresh.header.MaterialHeader  
 android:id="@+id/header"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content" />  
  
 <!--sc 内嵌多个rv-->  
 <include layout="@layout/demo\_sc\_nested\_rvs"/>  
  
 <com.scwang.smartrefresh.layout.footer.ClassicsFooter  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content" />  
 </com.scwang.smartrefresh.layout.SmartRefreshLayout>  
</FrameLayout>

# [TableLayout和TableRow的使用](http://blog.csdn.net/kavensu/article/details/7933085)

**private** **void** **init**() {

context = **TableActivityDemo**.**this**;

**TableLayout** table = (**TableLayout**) findViewById(**R**.**id**.**table**);

table.setStretchAllColumns(**true**);**//可拉伸**

table.setShrinkAllColumns(**true**);**//可收缩**

**int** a = 126976;

**for** (**int** i = a; i < a + 600000; i+=100000) {

**TableRow** row = **new** TableRow(context);

row.addView(setview(i));

row.addView(setview2(i));

row.addView(setview2(i));

table.addView(row, **new** **TableLayout**.LayoutParams(**LayoutParams**.**MATCH\_PARENT**, 30));

}

}

**private** **View** **setview**(**int** a) {

**TextView** tView = **new** TextView(context);

tView.setText("嘎嘎发 " + a);

tView.setTextColor(**Color**.parseColor("#"+ a));

**return** tView;

}

**private** **View** **setview2**(**int** a) {

**Button** btn = **new** Button(context);

btn.setText("服服服 " + a);

btn.setTextColor(**Color**.parseColor("#"+ (a-6543)));

**return** btn;

}

1）android:shrinkColumns="可收缩的column"，

2）android:stretchColumns="可伸展的column"。

android:shrinkColumns和android:stretchColumns的值都是以0开始的index，但必须是string值，即用"1,2,5"来表示。可以用"\*"来表示all columns。而且同一column可以同时设置为shrinkable和stretchable。

android:stretchColumns属性的作用是使选择的列尽可能填满剩余的空间，注意的是列号是从0开始算的，这里如果改为"0,1"则TableRow中的 TextView和EditText将各占一半横向空间

如果使用TableLayout类的setColumnShrinkable/setColumnStretchable (int columnIndex, boolean isShrinkable)就麻烦些了，需要一个一个column来设置。也可以使用TableLayout的setShrinkAllColumns/setStretchAllColumns来设置all columns。

判断这些column是否shrinkable或stretchable，可以调用isColumnShrinkable/isColumnStretchable(int columnIndex)，isShrinkAllColumns()/isStretchAllColumns()。

# CountDownTimer定时器控件

**class** **MyTime** **extends** **CountDownTimer**{

**public** **MyTime**(**long** millisInFuture, **long** countDownInterval) {

**super**(millisInFuture, countDownInterval);

}

@Override

**public** **void** **onTick**(**long** millisUntilFinished) {

btn.setText("重新发送:("+millisUntilFinished/1000+")");

}

@Override

**public** **void** **onFinish**() {

btn.setEnabled(**true**);

btn.setText("发送验证码");

}

}

**private** **MyTime** myTime;

myTime = **new** MyTime(10\*1000, 1000);

myTime.start();

//myTime.cancel();

# Timer类

**Timer** timer = **new** Timer();

timer.schedule(**new** MyTask(), 1000, 2000);

**class** **MyTask extends** **TimerTask** {

@Override

**public** **void** **run**() {

**System**.**out**.println("dddd");

}

}

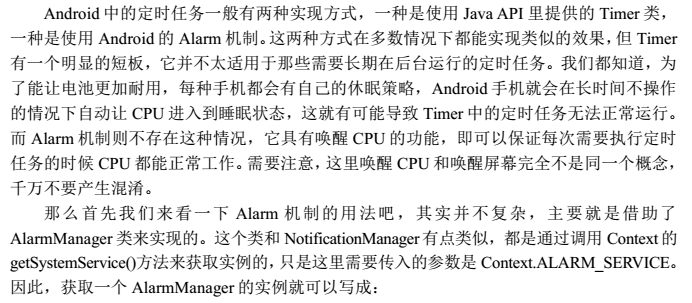
timer.cancel();

这样，就可以在1秒钟之后开始执行mytask，每两秒钟执行一次。

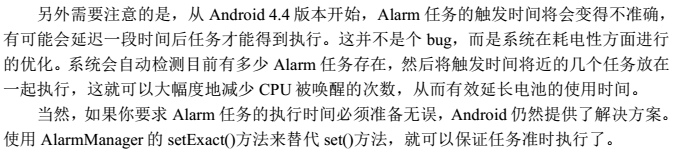
当然，timer的功能也可以通过自己构造线程，然后在线程中用sleep来模拟停止一段时间，然后再执行某个动作。

其实，看一下timertask的源码就立即可以知道，timertask就是实现了runnable接口的。也就是说，通过timer来间隔一段时间执行一个操作，也是通过一个线程来做到的

# Android闹钟设置的解决方案（Alarm）







## 主要问题

* API19开始AlarmManager的机制修改。
* 应用程序被Kill掉后,设置的闹钟不响。
* 6.0以上进入Doze模式会使JobScheduler停止工作。
* 手机设置重启后，闹钟失效问题。

## API19以上AlarmManager机制的修改

API19之前AlarmManager提供了三个设置闹钟的方法，由于业务需求闹钟只需要一次性，所以采用set(int type，long startTime，PendingIntent pi)；这个方法。从API 19开始，AlarmManager的机制都是非准确传递，操作系统将会转换闹钟，来最小化唤醒和电池使用。由于之前的程序，没有对API19以上的闹钟设置做处理，导致在4.4以上的手机设置闹钟无响应（应用程序没有被杀死的情况也没有闹钟）。因些，设置闹钟需要根据API的版本进行分别处理设置。代码如下：

AlarmManager am = (AlarmManager) getActivity().getSystemService(Context.ALARM\_SERVICE);

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.KITKAT) {

am.setExact(AlarmManager.RTC\_WAKEUP, TimeUtils

.stringToLong(recordTime, TimeUtils.NO\_SECOND\_FORMAT), sender);

}else {

am.set(AlarmManager.RTC\_WAKEUP, TimeUtils

.stringToLong(recordTime, TimeUtils.NO\_SECOND\_FORMAT), sender);

}

这样，保证闹钟在应用程序没有被Kill掉的情况闹钟。

## 应用程序被Kill掉时的处理

应用程序被Kill掉后，设置的闹钟失效，这里利用守护进程以及灰色保活来保证后台闹钟服务不被Kill掉。当应用程序以及闹钟服务被Kill掉，守护进程以及灰色保活来重新启动闹钟服务，并且重新设置闹钟。关于守护进程的处理，这里采用开源的守护进程库。Android-AppDaemon,在闹钟服务的onCreat加入Android-AppDaemon这个开源的守护进程。代码如下：

@Override

public void onCreate() {

super.onCreate();

Daemon.run(DaemonService.this, DaemonService.class, Daemon.INTERVAL\_ONE\_MINUTE);

startTimeTask();

grayGuard();

}

为进一步保证闹钟服务的存活，同加上灰色保活（利用系统的漏洞启动前台Service）。代码如下：

private void grayGuard() {

if (Build.VERSION.SDK\_INT < 18) {

//API < 18 ，此方法能有效隐藏Notification上的图标

startForeground(GRAY\_SERVICE\_ID, new Notification());

} else {

Intent innerIntent = new Intent(this, DaemonInnerService.class);

startService(innerIntent);

startForeground(GRAY\_SERVICE\_ID, new Notification());

}

//发送唤醒广播来促使挂掉的UI进程重新启动起来

AlarmManager alarmManager = (AlarmManager) getSystemService(Context.ALARM\_SERVICE);

Intent alarmIntent = new Intent();

alarmIntent.setAction(WakeReceiver.GRAY\_WAKE\_ACTION);

PendingIntent operation = PendingIntent.getBroadcast(this,

WAKE\_REQUEST\_CODE, alarmIntent, PendingIntent.FLAG\_UPDATE\_CURRENT);

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.KITKAT) {

alarmManager.setWindow(AlarmManager.RTC\_WAKEUP,

System.currentTimeMillis(), ALARM\_INTERVAL, operation);

}else {

alarmManager.setInexactRepeating(AlarmManager.RTC\_WAKEUP,

System.currentTimeMillis(), ALARM\_INTERVAL, operation);

}

}

/\*\*

\* 给 API >= 18 的平台上用的灰色保活手段

\*/

public static class DaemonInnerService extends Service {

@Override

public void onCreate() {

Log.i(LOG\_TAG, "InnerService -> onCreate");

super.onCreate();

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

Log.i(LOG\_TAG, "InnerService -> onStartCommand");

startForeground(GRAY\_SERVICE\_ID, new Notification());

//stopForeground(true);

stopSelf();

return super.onStartCommand(intent, flags, startId);

}

@Override

public IBinder onBind(Intent intent) {

throw new UnsupportedOperationException("Not yet implemented");

}

@Override

public void onDestroy() {

Log.i(LOG\_TAG, "InnerService -> onDestroy");

super.onDestroy();

}

}

上面操作尽可能提高闹钟服务的存活。但是在5.0以上的手机，利用系统的自带的Clean功能的时候，还是会将闹钟服务彻底的干掉。为了解决5.0以上的问题，这里引入5.0以上的新特性 JobScheduler。

## 5.0以上的JobScheduler

在这里利用5.0以上的JobScheduler创建一个定时的任务，定时检测闹钟服务是否存在，没在存在则重新启动闹钟服务。（这里我设置每一分钟检测一次闹钟服务）,在进入应用程序的时候检测当前系统是否是5.0以上，如果是则启动JobScheduler这个服务。代码如下：

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.LOLLIPOP) {

mJobScheduler = (JobScheduler) getSystemService(Context.JOB\_SCHEDULER\_SERVICE);

JobInfo.Builder builder = new JobInfo.Builder(JOB\_ID,

new ComponentName(getPackageName(), JobSchedulerService.class.getName()));

builder.setPeriodic(60 \* 1000); //每隔60秒运行一次

builder.setRequiresCharging(true);

builder.setPersisted(true); //设置设备重启后，是否重新执行任务

builder.setRequiresDeviceIdle(true);

if (mJobScheduler.schedule(builder.build()) <= 0) {

//If something goes wrong

}

}

其中的builder.setPersisted(true); 方法是设备重启后，是否重新执行任务，在这测过是可以重新启动任务的。上面的操作进一步保证了闹钟服务被Kill掉后，重新启动服务。但是在6.0以上引入了Doze模式，当6.0以上的手机进入这个模式后，便会使JobScheduler停止工作。

## 6.0以上Doze模式的处理

为了让JobScheduler可以在6.0以上进入Doze模式工作，这里针对6.0以上的Doze模式做特殊的处理-忽略电池的优化。

在Manifest.xml中加入权限。

<uses-permission android:name="android.permission.REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS"/>

在设置闹钟的时候，判断系统是否是6.0以上，如果是,则判断是否忽略电池的优化。判断是否忽略电池优化代码如下：

@TargetApi(Build.VERSION\_CODES.M)

public static boolean isIgnoringBatteryOptimizations(Activity activity){

String packageName = activity.getPackageName();

PowerManager pm = (PowerManager) activity

.getSystemService(Context.POWER\_SERVICE);

if (pm.isIgnoringBatteryOptimizations(packageName)) {

return true;

}else {

return false;

}

}

如果没有忽略电池优化的时候，弹出提醒对话框，提示用户进行忽略电池优化操作。代码如下：

/\*\*

\* 针对N以上的Doze模式

\* @param activity

\*/

public static void isIgnoreBatteryOption(Activity activity) {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.M) {

try {

Intent intent = new Intent();

String packageName = activity.getPackageName();

PowerManager pm = (PowerManager) activity.getSystemService(Context.POWER\_SERVICE);

if (!pm.isIgnoringBatteryOptimizations(packageName)) {

// intent.setAction(Settings.ACTION\_IGNORE\_BATTERY\_OPTIMIZATION\_SETTINGS);

intent.setAction(Settings.ACTION\_REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS);

intent.setData(Uri.parse("package:" + packageName));

activity.startActivityForResult(intent, REQUEST\_IGNORE\_BATTERY\_CODE);

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

在界面重写onActivityResult方法来捕获用户的选择。如，代码如下：

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

if (resultCode == RESULT\_OK) {

if (requestCode == BatteryUtils.REQUEST\_IGNORE\_BATTERY\_CODE){

//TODO something

}

}else if (resultCode == RESULT\_CANCELED){

if (requestCode == BatteryUtils.REQUEST\_IGNORE\_BATTERY\_CODE){

ToastUtils.show(getActivity(), "请开启忽略电池优化~");

}

}

}

## 补充

当应用程序被Kill掉，但是闹钟的服务没有被Kill掉的，这时候又设置了闹钟。这就意味着设置的闹钟没有放到闹钟服务那里。所以这种情况，设置的闹钟会失效。为了解决这种情况，利用AIDL（闹钟服务在另一个进程的需要进程间通信）调用闹钟服务的重新设置闹钟方法重设闹钟。在应用程序的onCreat()方法启动闹钟服务，然后再绑定闹钟服务。

private void initAlarmService() {

startService(new Intent(this, DaemonService.class));//启动闹钟服务

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.LOLLIPOP) {

//JobScheduler

...

}

//绑定闹钟服务

Intent intent = new Intent(this, DaemonService.class);

intent.setAction("android.intent.action.DaemonService");

bindService(intent, mConnection, Context.BIND\_AUTO\_CREATE);

}

在onDestroy()方法，调用闹钟服务的重设闹钟方法。代码如下：

@Override

protected void onDestroy() {

super.onDestroy();

try {//判断是否有闹钟，没有则关闭闹钟服务

String alarm = localPreferencesHelper.getString(LocalPreferencesHelper.ALARM\_CLOCK);

if (daemonService != -1 && mIRemoteService != null) {

// android.os.Process.killProcess(daemonService);

mIRemoteService.resetAlarm();

}

if (!alarm.equals("[]")) {

if (daemonService != -1) {

startService(new Intent(this, DaemonService.class));

}

} else {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.LOLLIPOP) {

mJobScheduler.cancel(JOB\_ID);

}

}

unbindService(mConnection); //解除绑定服务。

} catch (Exception e) {

}

}

这里说明一下，当服务启动并且被绑定的情况下，unbindService是不会停止服务的。最后,以上并不代表所有的Android手机的闹钟都可以用，这只是尽最大的可能保证大部分的手机。

# 日期、时间、时间戳转换

/\*\*

\* 日期扩展类

\*/

public class DateUtil {

/\*\*

\* 时间戳Str 转为 自定义时间样式

\*/

public static String stampStrToStrReg(String time, String reg) {

String result = "";

try {

long l = Long.parseLong(time);

Date data = new Date(l);

SimpleDateFormat sf = new SimpleDateFormat(reg);

result = sf.format(data);

} catch (Exception e) {

e.printStackTrace();

}

return result;

}

/\*\*

\* 自定义时间样式 转为 Date

\*/

public static Date strRegToDate(String time, String reg) {

Date date = new Date(0);

SimpleDateFormat sf = new SimpleDateFormat(reg);

try {

date = sf.parse(time);

} catch (ParseException e) {

e.printStackTrace();

}

return date;

}

/\*\*

\* 自定义时间样式 转为 时间戳long

\*/

public static long strRegToStampLong(String time, String reg) {

return strRegToDate(time, reg).getTime();

}

/\*\*

\* 自定义时间样式 转为 时间戳Str

\*/

public static String strRegToStampStr(String time, String reg) {

String result = "";

SimpleDateFormat sf = new SimpleDateFormat(reg);

try {

Date date = sf.parse(time);

result = date.getTime() + "";

} catch (ParseException e) {

e.printStackTrace();

}

return result;

}

/\*\*

\* 自定义时间样式 转换 自定义时间样式

\*/

public static String StrRegToStrReg(String reg1, String reg2, String time) {

SimpleDateFormat sf2 = new SimpleDateFormat(reg2);

String result = sf2.format(new Date(strRegToStampLong(time, reg1)));

return result;

}

//endregion

//region get currentTime

/\*\*

\* 当前时间 转为 yyyy-MM-dd HH:mm:ss

\*/

public static String getCurrent\_yMdHmsE() {

String result = "";

try {

SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss EEEE");

result = sf.format(new Date());

} catch (Exception e) {

e.printStackTrace();

}

return result;

}

/\*\*

\* 当前时间 转为 时间戳Str

\*/

public static String getCurrentStampStr() {

// return String.valueOf(new Date().getTime());

return String.valueOf(System.currentTimeMillis());

}

/\*\*

\* yyyy/MM/dd 转为 时间戳Long

\*/

public static long yMdToStampLong2(String time) {

return strRegToStampLong(time, "yyyy/MM/dd");

}

}

# 日期选择器

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:orientation=*"vertical"* >

<DatePicker

android:id=*"@+id/datepicker"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:calendarViewShown=*"false"* />

<!-- 用于设置是否显示calendar view -->

<TimePicker

android:id=*"@+id/timepicker"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"* />

</LinearLayout>

**public** **class** **DateTimePickerUtil** **implements** OnDateChangedListener, OnTimeChangedListener{

**private** **Context** context;

**private** **DatePicker** datePicker;

**private** **TimePicker** timePicker;

**private** **Calendar** calendar;

**private** **AlertDialog** dialog;

**private** **String** initDateTime;

**private** **String** dateTime;//返回的日期时间

**public** **DateTimePickerUtil**(**Context** context, **String** initDateTime) {

**this**.context = context;

**this**.initDateTime = initDateTime;

}

/\*\*

\* 弹出日期时间选择框方法

\* **@param** inputDate :需要设置的日期时间文本编辑框

\*/

**public** **AlertDialog** **showDialogForTv**(**final** **TextView** inputtv) {

//初始化时间日期选择框布局

**LinearLayout** dateTimeLayout = (**LinearLayout**) **LayoutInflater**.*from*(context).inflate(**R**.**layout**.**common\_datetime**, **null**);

datePicker = (**DatePicker**) dateTimeLayout.findViewById(**R**.**id**.**datepicker**);

timePicker = (**TimePicker**) dateTimeLayout.findViewById(**R**.**id**.**timepicker**);

//初始化日期和时间选择器

initThem(datePicker, timePicker);

//实例dialog

dialog = **new** **AlertDialog**.Builder(context)//build链式

.setTitle(initDateTime)//设置自定义或者当前时间的String

.setView(dateTimeLayout)//dilog设置自定义日期时间选择器

.setPositiveButton("设置", **new** OnClickListener() {//自定义按钮，将设置的时间返回到的textview

@Override

**public** **void** **onClick**(DialogInterface dialog, **int** which) {

inputtv.setText(dateTime);//设置文本框的时间文字

}

})

.setNegativeButton("取消", **null**)//自定义按钮，隐藏dialog

.show();//默认show出来

//防止刚打开dialog没有修改时，dateTime为""，所以先调一次

onDateChanged(**null**, 0, 0, 0);

**return** dialog;

}

/\*\*

\* 初始化日期和时间选择器

\* \*/

**private** **void** **initThem**(**DatePicker** datePicker, **TimePicker** timePicker) {

calendar = **Calendar**.*getInstance*();//获取当前日历

**if** (initDateTime != **null** && !initDateTime.equals("")) {//如果有自定义初始化日期时间

calendar = **this**.getCalendarByInintData(initDateTime);//将日历改为自定义的初始日期时间

}**else** {//没有初始化日期时间，自己获取initDateTime

**SimpleDateFormat** sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm"); //设置日期格式

initDateTime = sdf.format(calendar.getTime());//获取当前时间的String

}

//使用操作好的calendar初始datapicker，并设置日期变化监听

//DatePicker.init(int year, int month, int day, OnDateChangedListener onListener)

datePicker.init(calendar.get(**Calendar**.**YEAR**),

calendar.get(**Calendar**.**MONTH**),

calendar.get(**Calendar**.**DAY\_OF\_MONTH**),

**this**);

//初始timePicker

timePicker.setCurrentHour(calendar.get(**Calendar**.**HOUR\_OF\_DAY**));

timePicker.setCurrentMinute(calendar.get(**Calendar**.**MINUTE**));

timePicker.setIs24HourView(**true**);

//timePicker设置时间变化监听

timePicker.setOnTimeChangedListener(**this**);

}

@Override

**public** **void** **onTimeChanged**(**TimePicker** view, **int** hourOfDay, **int** minute) {

onDateChanged(**null**, 0, 0, 0);//时间改变直接调用日期改变逻辑

}

@Override

**public** **void** **onDateChanged**(**DatePicker** view, **int** year, **int** monthOfYear,**int** dayOfMonth) {

**SimpleDateFormat** sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm"); //设置日期格式

//使用calendar.set()，只是为了使用calendar.getTime()，方便sdf.format()，得到时间String

calendar.set(datePicker.getYear(), datePicker.getMonth(),datePicker.getDayOfMonth(), timePicker.getCurrentHour(),timePicker.getCurrentMinute());

dateTime = sdf.format(calendar.getTime());//获取当前时间的String，赋值给dateTime，返回到textview使用

dialog.setTitle(dateTime);//同步更新dialog标题

}

/\*\*

\* 实现将初始日期时间YYYY-MM-DD hh:mm 拆分成年 月 日 时 分,并赋值给calendar

\* **@param** initDateTime 初始日期时间值 字符串型

\* **@return** Calendar

\*/

**private** **Calendar** **getCalendarByInintData**(**String** initDateTime){

**String** yearStr = *spliteString*(initDateTime, "-", "index", "front"); // 年

**String** mmddhhmm = *spliteString*(initDateTime, "-", "index", "back"); // 月日时分

**String** monthStr = *spliteString*(mmddhhmm, "-", "index", "front"); // 月

**String** ddhhmm = *spliteString*(mmddhhmm, "-", "index", "back"); // 日时分

**String** dayStr = *spliteString*(ddhhmm, " ", "index", "front"); // 日

**String** hhmm = *spliteString*(ddhhmm, " ", "index", "back"); // 时分

**String** hourStr = *spliteString*(hhmm, ":", "index", "front"); // 时

**String** minuteStr = *spliteString*(hhmm, ":", "index", "back"); // 分

**int** currentYear = **Integer**.*valueOf*(yearStr.trim()).intValue();

**int** currentMonth = **Integer**.*valueOf*(monthStr.trim()).intValue() - 1;//月份的起始值为0而不是1,所以要-1

**int** currentDay = **Integer**.*valueOf*(dayStr.trim()).intValue();

**int** currentHour = **Integer**.*valueOf*(hourStr.trim()).intValue();

**int** currentMinute = **Integer**.*valueOf*(minuteStr.trim()).intValue();

calendar.set(currentYear, currentMonth, currentDay, currentHour, currentMinute);

**return** calendar;

}

/\*\*

\* 截取子串

\* **@param** srcStr 源串

\* **@param** pattern 匹配模式

\* **@param** indexOrLast 取匹配的字符串第一次出现还是最后一次出现的位置

\* **@param** frontOrBack 取前还是后

\*/

**public** **static** **String** **spliteString**(**String** srcStr, **String** pattern, **String** indexOrLast, **String** frontOrBack) {

**String** result = "";

**int** loc = -1;

**if** (indexOrLast.equalsIgnoreCase("index")) {

loc = srcStr.indexOf(pattern); // 取得字符串第一次出现的位置

} **else** {

loc = srcStr.lastIndexOf(pattern); // 倒着取，取得字符串第一次出现的位置

}

**if** (frontOrBack.equalsIgnoreCase("front")) {

**if** (loc != -1)

result = srcStr.substring(0, loc); // 截取子串

} **else** {

**if** (loc != -1)

result = srcStr.substring(loc + 1, srcStr.length()); // 截取子串

}

**return** result;

}

}

**DateTimePickerUtil** dateTimePickerUtil = **new** DateTimePickerUtil(context, initDateTime);

dateTimePickerUtil.showDialogForTv(tv);

## 常用方法

**1.Calendar**

**Calendar** calendar = **Calendar**.getInstance(); **//系统时间**

calendar.set(year, month, day, hourOfDay, minute); **//自定义时间**

**SimpleDateFormat** sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm"); **//设置日期格式**

**Date** date = calendar.getTime(); **//从calendar中获取date**

dateTime = sdf.format(date); **//获取当前时间的String**

calendar.get(Calendar.XXX) **//从calendar中获取年月日时分**

**给datePicker、timePicker设置用的Calendar常量如下，其他还有DAY\_OF\_WEEK、DAY\_OF\_YEAR、WEEK\_OF\_YEAR**

datePicker.init(calendar.get(Calendar.YEAR),

calendar.get(Calendar.MONTH),

calendar.get(Calendar.DAY\_OF\_MONTH),

this);

timePicker.setCurrentHour(calendar.get(Calendar.HOUR\_OF\_DAY));

timePicker.setCurrentMinute(calendar.get(Calendar.MINUTE));

**2.DatePicker：**

1. public CalendarView getCalendarView(); 　　 //获取CalendarView
2. public boolean getCalendarViewShown(); 　 //获取CalendarView是否显示
3. public int getDayOfMonth(); 　　　　　　　　 //获取当前日期的日
4. public long getMaxDate(); 　　  　　 　　　　 //获取最大日期
5. public long getMinDate();　　    　　　　　　 //获取最小日期
6. public int getMonth(); 　　　　   　　　　　   //获取当前日期的月
7. public boolean getSpinnersShown(); 　　      //获取Spinners是否显示
8. public int getYear(); 　　　　　　　　　　　　 //获取当前日期的年
9. public void init(int year, int month, int day, OnDateChangedListener onListener) //初始化日期,设置日期变化监听
10. public void setCalendarViewShown(boolean shown);　　　 //设置是否显示CalendarView
11. public void setMaxDate(long maxDate); 　　　　　　 //设置最大日期
12. public void setMinDate(long minDate); 　　　　　　 //设置最小日期
13. public void setSpinnersShown(boolean shown); 　 //设置是否显示Spinners
14. public void updateDate(int year,int month,int dayOfMonth); //更新当前日期

**3.TimePicker**

在Android中，TimePicker用来实现时间输入设置，可以选择12或24小时模式。TimePicker的常用方法有以下一些：

1. public Integer getCurrentHour(); 　　　　　　 //获取当前时间的小时
2. public Integer getCurrentMinute(); 　　　　  //获取当前时间的分钟
3. public boolean is24HourView(); 　　　　　　  //获取是否为24小时模式
4. public void setCurrentHour(Integer currentHour);             //设置当前时间的小时
5. public void setCurrentMinute(Integer currentMinute);       //设置当前时间的分钟
6. public void setIs24HourView(Boolean is24HourView);       //设置24小时模式

**4.DatePickerDialog**

　　在Android中，DatePickerDialog用来显示日期对话框。DatePickerDialog的常用方法有以下一些：

1. public DatePicker getDatePicker();                                 //获取DatePicker中的日期值
2. public void onClick(DialogInterface dialog,int which);       //响应对话框中的点击事件
3. public void onDateChanged(DatePicker view,int year,int month,int day);       //响应日期改变事件
4. public void updateDate(int year,int monthOfYear,int dayOfMonth);             //更新当前日期

**5.TimePickerDialog**

　　在Android中，TimePickerDialog用来显示时间对话框。TimePickerDialog的常用方法有以下一些：

1. public void onClick(DialogInterface dialog,int which);                         //响应对话框中的点击事件
2. public void onTimeChanged(TimePicker view,int hourOfDay,int minute);     //响应时间改变事件
3. public void updateTime(int hourOfDay,int minuteOfHour);                     //更新当前时间

**6.AnalogClock**

　　在Android中，AnalogClock用于显示指针式时钟，该时钟仅有时钟和分钟两个指针。

**7.DigitalClock**

在Android中，DigitalClock用来显示数字式时钟，显示格式为HH:MM:SS AM/PM。

# 时间显示控件TextClock

## 注意

最低支持版本17，在红米1（4.2.1）上有点问题。低版本用差性能的DigitalClock

设置formatxxHour不是将格式设为12或者24小时制，是在系统时间设置为12或24小时制时，采用formatxxHour设置的格式。你甚至可以在系统设置24小时制时，设置tc以12小时制显示，及format24Hour="yyyy-MM-dd hh:mm:ss EEEE"

**TextClock** tcClock = (**TextClock**) findViewById(**R**.**id**.**tc**);

tcClock .is24HourModeEnabled()//判断系统时间设置是否为24小时制

## XML中

<TextClock

android:id="@+id/tc"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:format24Hour="yyyy-MM-dd HH:mm:ss EEEE" <!-- hh表示12小时制格式 -->

android:format12Hour="yyyy-MM-dd HH:mm:ss EEEE"

android:textColor="#fff"

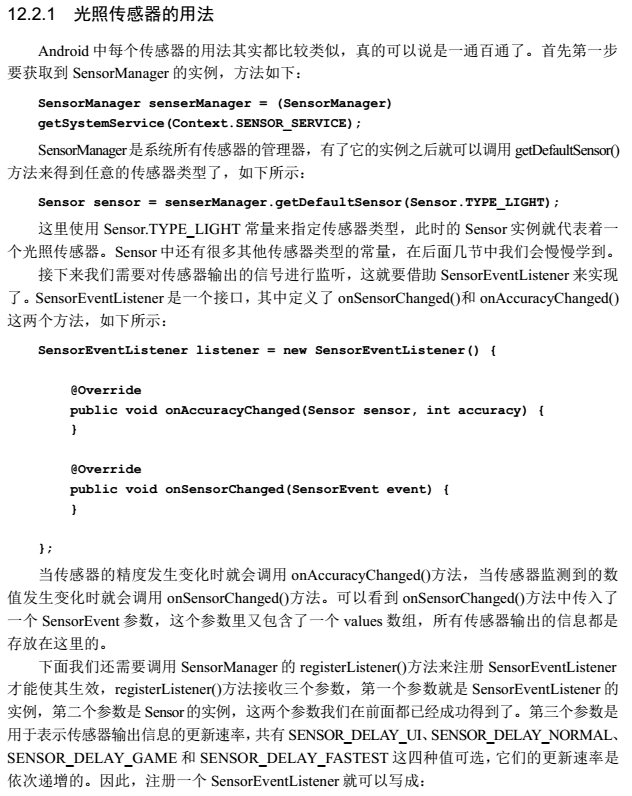
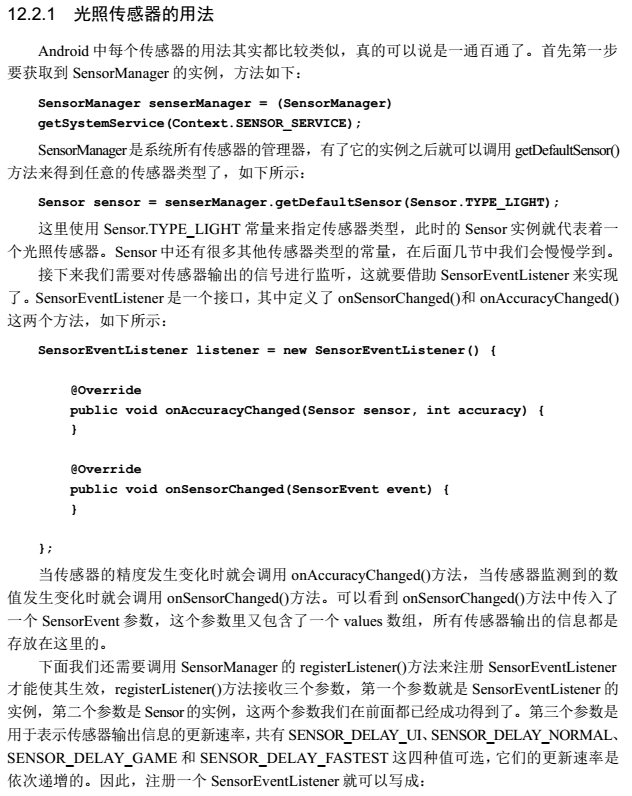
android:textSize="30sp" />

## 代码中

tcClock.setFormat12Hour("yyyy-MM-dd HH:mm:ss EEEE");

tcClock.setFormat24Hour("yyyy-MM-dd HH:mm:ss EEEE");

# 传感器Sensor



**public** **class** **MainActivity** **extends** **Activity** {

**private** **SensorManager** sensorManager;//传感器管理器

**private** **Sensor** lightSensor, acceleraterSensor, magnetic\_fieldSensor;//传感器

**private** **LightListener** lightListener;//光线监听

**private** **Magne\_acce\_Listener** magne\_acce\_Listener;//加速度、磁场监听

**private** **TextView** tv, tv2, tv3;

**private** **ImageView** imgOri;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

tv = (**TextView**) findViewById(**R**.**id**.**tv**);

tv2 = (**TextView**) findViewById(**R**.**id**.**tv2**);

tv3 = (**TextView**) findViewById(**R**.**id**.**tv3**);

imgOri = (**ImageView**) findViewById(**R**.**id**.**img\_bg**);

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

lightSensor = sensorManager.getDefaultSensor(**Sensor**.**TYPE\_LIGHT**);

lightListener = **new** LightListener();

sensorManager.registerListener(lightListener, lightSensor, **SensorManager**.**SENSOR\_DELAY\_NORMAL**);

acceleraterSensor = sensorManager.getDefaultSensor(**Sensor**.**TYPE\_ACCELEROMETER**);

magnetic\_fieldSensor = sensorManager.getDefaultSensor(**Sensor**.**TYPE\_MAGNETIC\_FIELD**);

magne\_acce\_Listener = **new** Magne\_acce\_Listener();

sensorManager.registerListener(magne\_acce\_Listener, acceleraterSensor, **SensorManager**.**SENSOR\_DELAY\_NORMAL**);

sensorManager.registerListener(magne\_acce\_Listener, magnetic\_fieldSensor, **SensorManager**.**SENSOR\_DELAY\_NORMAL**);

}

**class** **LightListener** **implements** SensorEventListener {

@Override

**public** **void** **onSensorChanged**(**SensorEvent** event) {

**float** value = event.values[0];

tv.setText(value + "lx");

}

@Override

**public** **void** **onAccuracyChanged**(**Sensor** sensor, **int** accuracy) {

}

};

**class** **Magne\_acce\_Listener** **implements** SensorEventListener {

**float**[] acceValues = **new** **float**[3];

**float**[] magneValues = **new** **float**[3];

**float** lastRotateDregree;

@Override

**public** **void** **onSensorChanged**(**SensorEvent** event) {

**if** (event.sensor.getType() == **Sensor**.**TYPE\_ACCELEROMETER**) {//加速度

acceValues = event.values.clone();

**float** xvalue = **Math**.abs(acceValues[0]);

**float** yvalue = **Math**.abs(acceValues[1]);

**float** zvalue = **Math**.abs(acceValues[2]);

tv2.setText("x:" + xvalue + "---y:" + yvalue + "---z:" + zvalue);

**if** (xvalue > 15 || yvalue > 15 || zvalue > 15) {

**Toast**.makeText(**MainActivity**.**this**, "摇一摇", **Toast**.**LENGTH\_SHORT**).show();

}

}

**if** (event.sensor.getType() == **Sensor**.**TYPE\_MAGNETIC\_FIELD**) {//磁场

magneValues = event.values.clone();

}

**float**[] R = **new** **float**[9];

**float**[] oriValues = **new** **float**[3];

sensorManager.getRotationMatrix(R, **null**, acceValues, magneValues);//包含反向的矩阵

sensorManager.getOrientation(R, oriValues);//从矩阵中获取方向

**float** xvalue = (**float**) **Math**.toDegrees(oriValues[0]);

**float** yvalue = (**float**) **Math**.toDegrees(oriValues[1]);

**float** zvalue = (**float**) **Math**.toDegrees(oriValues[2]);

tv3.setText("x:" + xvalue + "---y:" + yvalue + "---z:" + zvalue);

**float** rotateDregree = -(**float**) **Math**.toDegrees(oriValues[0]);

**if** (**Math**.abs(rotateDregree - lastRotateDregree) > 2) {

**RotateAnimation** animation = **new** RotateAnimation(lastRotateDregree, rotateDregree,

**Animation**.**RELATIVE\_TO\_SELF**, 0.5f, **Animation**.**RELATIVE\_TO\_SELF**, 0.5f);

animation.setFillAfter(**true**);

imgOri.setAnimation(animation);

lastRotateDregree = rotateDregree;

}

}

@Override

**public** **void** **onAccuracyChanged**(**Sensor** sensor, **int** accuracy) {

}

};

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

sensorManager.unregisterListener(lightListener);

sensorManager.unregisterListener(magne\_acce\_Listener);

}

}

# Dialog

## ProgressDialog

**// 方式一：new Dialog ，可以条形，可以圆形**

**ProgressDialog** dialog = **new** ProgressDialog(**this**);

dialog.setTitle("title");

dialog.setMessage("message");

// 设置提示的title的图标，默认是没有的，如果没有设置title的话只设置Icon是不会显示图标的

dialog.setIcon(**R**.**drawable**.**ic\_launcher**);

dialog.**setButton**("button", mOnClickListener);

dialog.setProgressStyle(**ProgressDialog**.**STYLE\_HORIZONTAL**);// 设置进度条的形式为条形

dialog.setCancelable(**false**);

dialog.setOnDismissListener(mOnDismissListener);

dialog.setCanceledOnTouchOutside(**false**);

dialog.setMax(200);//设置进度最大为200

dialog.setIndeterminate(**true**);//设置为true时，进度条为不明确的状态，更新进度无效

dialog.show();

**new** Thread(**new** Runnable() {

@Override

**public** **void** **run**() {

**for** (**int** i = 0; i <= 100; i++) {

**try** {

**Thread**.*sleep*(200);

// 更新进度条的进度,可以在子线程中更新进度条进度 ,Indeterminate必须为false

dialog.incrementProgressBy(2); //增加指定进度，不能用i

// dialog.incrementSecondaryProgressBy(10)//二级进度条更新方式

} **catch** (**Exception** e) {

}

}

dialog.dismiss();// 在进度条走完时删除Dialog

}

}).start();

**// 方式二：使用静态方式创建并显示，这种进度条只能是圆形条,设置title和Message提示内容**

**ProgressDialog** dialog2 = **ProgressDialog**.*show*(**this**, null, "正在登陆中");

**// 方式三：boolean indeterminate设置是否是不明确的状态**

**ProgressDialog** dialog3 = **ProgressDialog** .*show*(**this**, "提示", "正在登陆中", **false**);

**// 方式四 ：boolean cancelable 设置是否进度条是可以取消的**

**ProgressDialog** dialog4 = **ProgressDialog**.*show*(**this**, "提示", "正在登陆中", **false**, **true**);

**// 方式五：DialogInterface.OnCancelListener cancelListener用于监听进度条被取消**

**ProgressDialog** dialog5 = **ProgressDialog**.*show*(**this**, "提示", "msg", **true**, **true**, **new** OnCancelListener() {

@Override

**public** **void** **onCancel**(DialogInterface dialog) {

**Toast**.*makeText*(**MainActivity**.**this**, "进度条被取消", **Toast**.**LENGTH\_LONG**).show();

}

});

### ProgressDialog和AlertDialog必须先show才能setContentView

不然会报android.util.AndroidRuntimeException: requestFeature() must be called before adding content，原因是：

**ProgressDialog的show()的源码：**

public static ProgressDialog show(Context context, CharSequence title,

CharSequence message, boolean indeterminate,

boolean cancelable, OnCancelListener cancelListener) {

ProgressDialog dialog = new ProgressDialog(context);

dialog.setTitle(title);

dialog.setMessage(message);

dialog.setIndeterminate(indeterminate);

dialog.setCancelable(cancelable);

dialog.setOnCancelListener(cancelListener);

dialog.show();

return dialog;

}

**AlertDialog的show()的源码：**

public AlertDialog show() {

AlertDialog dialog = create();

dialog.show();

return dialog;

}

## dismiss和cancel的区别

dismiss和cancle都会有回调 可以通过setDismissListenner来获取。

区别在于 cancle会先发起回调 然后 再执行dismiss.

所以就是cancel的listenner叫做 preDismissListenner比较容易理解

而dimiss的listenner叫做 postDismissListenner

所以如果仅仅是取消到dialog的显示 二者没有什么区别。

## Dialog遇到横竖屏切换

### 基于Dialog实现的对话框

**public** **class** **MainActivity** **extends** **Activity** {

**private** **Button** clk;

**private** **Dialog** dialog;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

clk = (**Button**) findViewById(**R**.**id**.**clk**);

dialog = **new** Dialog(**this**);

dialog.setContentView(**R**.**layout**.**dialog**);

clk.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

dialog.show();

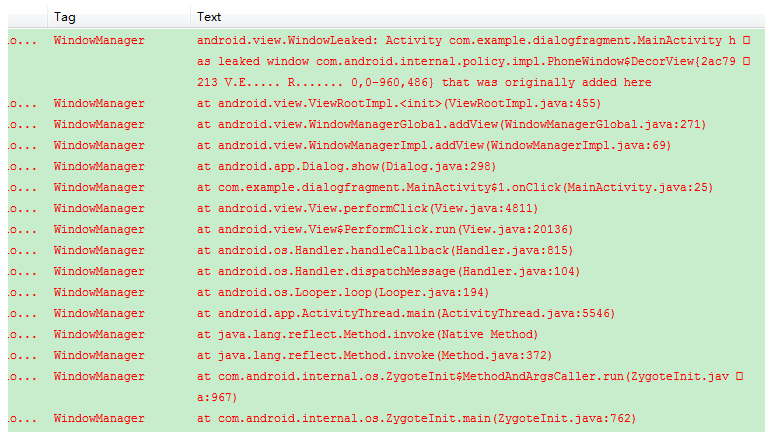
}

});

}

}

当我们点击按钮时，会弹出对话框，当我们旋转屏幕后，Activity重新创建，整个Activity的界面没有问题，而对话框消失了。除此之外，其实还有一个问题，就是在logcat中会看到异常信息：Android..leaked .. window，这是因为在Activity结束之前，Android要求所有的Dialog必须要关闭。我们旋屏后，Activity会被重建，而上面的代码逻辑并没有考虑到对话框的状态以及是否已关闭。



**于是，将上述代码改为：**

**public** **class** **MainActivity** **extends** **Activity** {

**private** **Button** clk;

**private** **Dialog** dialog;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**。。。**

//用户恢复对话框的状态

**if** (savedInstanceState != **null** && savedInstanceState.getBoolean("dialogIsShow")) {

dialog.show();

}

}

/\*\*

\* 用于保存对话框的状态以便恢复

\*/

@Override

**protected** **void** **onSaveInstanceState**(**Bundle** outState) {

**if** (dialog != **null** && dialog.isShowing()) {

outState.putBoolean("dialogIsShow", **true**);

}

**super**.onSaveInstanceState(outState);

}

/\*\*

\* 在Activity销毁之前，确保对话框以关闭

\*/

@Override

**protected** **void** **onDestroy**() {

**super**.onDestroy();

**if** (dialog != **null** && dialog.isShowing()) {

dialog.dismiss();

}

}

}

### 基于DialogFragment的对话框

**public** **class** **MainActivity** **extends** **FragmentActivity** {

**private** **Button** clk;

@Override

**protected** **void** **onCreate**(**Bundle** savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(**R**.**layout**.**activity\_main**);

clk = (**Button**) findViewById(**R**.**id**.**clk**);

clk.setOnClickListener(**new** OnClickListener() {

@Override

**public** **void** **onClick**(**View** v) {

**MyDialogFragment** mdf = **new** MyDialogFragment();

**FragmentTransaction** ft = getSupportFragmentManager().beginTransaction();

ft.setTransition(**FragmentTransaction**.**TRANSIT\_FRAGMENT\_FADE**);

mdf.show(ft, "df");

}

});

}

}

这两段代码可以实现第一种方式的同样功能，此处我们并没有去关心对话框的重建，以及Activity销毁前对话框是否已关闭，这一切都是由FragmentManager来管理。其实DialogFragment还拥有fragment的优点，即可以在一个Activity内部实现回退（因为FragmentManager会管理一个回退栈）

## 使dialog居于所有的dialog最上层

**//小米、魅族手机设置需开启悬浮窗权限,所以用TYPE\_TOAST**

String whatPhone = WhatPhoneUtil.whatPhone()；

if (whatPhone.equals(WhatPhoneUtil.SYS\_MIUI) || whatPhone.equals(WhatPhoneUtil.SYS\_FLYME)) {

//TYPE\_TOAST在最新的Android 7.1.1上已经被Google制裁了，只允许添加一个，并且在API 25之后会直接崩溃

//mDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_TOAST);

} else {

mDialog.getWindow().setType(WindowManager.LayoutParams.TYPE\_SYSTEM\_ALERT);

}

## setCancelable和setCanceledOnTouchOutside使用顺序

setCancelable()设置是否可点击返回按钮取消，true表示可以

setCanceledOnTouchOutside()设置是否可点击dialog外部取消，true表示可以

两者使用顺序需注意，先分析源码

protected boolean mCancelable = true;

public void setCancelable(boolean flag) {

mCancelable = flag;

updateWindowForCancelable();

}

public void setCanceledOnTouchOutside(boolean cancel) {

if (cancel && !mCancelable) {

mCancelable = true;

}

mWindow.setCloseOnTouchOutside(cancel);

}

//点击外部判断逻辑

public boolean onTouchEvent(MotionEvent event) {

if (mCancelable && mShowing && mWindow.shouldCloseOnTouch(mContext, event)) {

cancel();

return true;

}

return false;

}

//返回按钮判断逻辑

public void onBackPressed() {

if (mCancelable) {

cancel();

}

}

public void setCloseOnTouchOutside(boolean close) {

mCloseOnTouchOutside = close;

mSetCloseOnTouchOutside = true;

}  
public boolean shouldCloseOnTouch(Context context, MotionEvent event) {

if (mCloseOnTouchOutside && event.getAction() == MotionEvent.ACTION\_DOWN

&& isOutOfBounds(context, event) && peekDecorView() != null) {

return true;

}

return false;

}

* 先使用setCanceledOnTouchOutside(true)再使用setCancelable(false)，外部和返回键 都不可取消
* 先使用setCancelable(false)再使用setCanceledOnTouchOutside(true)，外部和返回键 都可取消
* 先使用setCanceledOnTouchOutside(false)再使用setCancelable(true)，返回键可取消 外部不可取消
* 先使用setCancelable(true)再使用setCanceledOnTouchOutside(false)，返回键可取消 外部不可取消

综上：

* 要实现：可以点击外部取消，不可以返回键取消，做不到
* 要实现：可以返回键取消，不可以点击外部取消，使用setCanceledOnTouchOutside(false)；setCancelable(true)；
* 要实现都可取消和都不可取消，同时设置两个参数的值为true和false即可官方
* setCanceledOnTouchOutside(true)后如何监听Dialog消失

private boolean isOutOfBounds(Context context, MotionEvent event) {

final int x = (int) event.getX();

final int y = (int) event.getY();

final int slop = ViewConfiguration.get(context).getScaledWindowTouchSlop();

final View decorView = getDecorView();

return (x < -slop) || (y < -slop) || (x > (decorView.getWidth()+slop)) || (y > (decorView.getHeight()+slop));

}

所以可以看出setCanceledOnTouchOutside(true)的监听其实是通过isOutOfBounds(context, event) 的，所以我们要监听点击其他区域关闭dialog的方法就要重写dialog的onTouchEvent方法，然后将isOutOfBounds中的代码加上，代码如下：

class myDialog extends Dialog{

//其他代码略·······

//触摸对话框其他区域的监听

private void onTouchOutside(){}

@Override

public boolean onTouchEvent(MotionEvent event) {

//触摸外部弹窗

if (isOutOfBounds(getContext(), event)) {

onTouchOutside();

}

return super.onTouchEvent(event);

}

private boolean isOutOfBounds(Context context, MotionEvent event) {

final int x = (int) event.getX();

final int y = (int) event.getY();

final int slop = ViewConfiguration.get(context).getScaledWindowTouchSlop();

final View decorView = getWindow().getDecorView();

return (x < -slop) || (y < -slop) || (x > (decorView.getWidth() + slop)) || (y > (decorView.getHeight() + slop));

}

}

## 自定义Style实现不同的ui需求

<!--dialog框背景透明，可自定义dialog框（比如圆角），没有默认Title占位置，且界面有蒙层-->

<style name="ComWinTransBackHalfDia" parent="@android:style/Theme.Dialog">

<!--wind无标题占位置-->

<item name="android:windowNoTitle">true</item>

<!--wind的边框-->

<item name="android:windowFrame">@null</item>

<!--设置为true可能导致activity透明-->

<item name="android:windowIsTranslucent">false</item>

<!--wind背景透明，不设置它有默认背景的-->

<item name="android:windowBackground">@android:color/transparent</item>

<!--是否有蒙层，true表示有-->

<item name="android:backgroundDimEnabled">true</item>

<!--蒙层透光度, 值越小蒙层越透明-->

<item name="android:backgroundDimAmount">0.6</item>

</style>

<!--继承与WinTransBackHalfDia，界面没有蒙层-->

<style name="TransparentDialog" parent="@style/WinTransBackHalfDia">

<!--是否有蒙层，false表示没有-->

<item name="android:backgroundDimEnabled">false</item>

</style>

dialog显示和隐藏的动画

<!--字面意思看起来是透明，其实是从右边划入划出-->

<item name="android:windowAnimationStyle">@android:style/Animation.Translucent</item>

<!--Dialog默认的动画效果，从中间淡入淡出-->

<item [name="windowAnimationStyle">@android:style/Animation.Dialog</item>](mailto:name=\"windowAnimationStyle\">@style/Animation.Dialog</item>)

## 封装工具类

public class CommonDialogTwoBtn extends Dialog {  
 public CommonDialogTwoBtn(Context context) {  
 super(context);  
 }  
 public CommonDialogTwoBtn(Context context, boolean cancelable, OnCancelListener cancelListener) {  
 super(context, cancelable, cancelListener);  
 }  
 public CommonDialogTwoBtn(Context context, int theme) {  
 super(context, theme);  
 }  
 */\*\*  
 \* 每次使用必须new一个builder，因为初始化界面在builder构造函数里  
 \*/* public static class Builder {  
 private CommonDialogTwoBtn mDialog;  
 private Context context;  
 private View mView;//dialog视图  
 private TextView mTextView\_title;//标题  
 private TextView mTextView\_content;//内容  
 private TextView mBottom;//按钮  
 private TextView mBottom2;//按钮  
 private boolean canBack = true;//是否可以返回键取消，默认可以  
 private boolean canTouch = true;//是否可以点击外部取消，默认可以  
 public Builder(Context context) {  
 this.context = context;  
 initView();  
 }  
 */\*\*  
 \* 初始化视图  
 \*/* private void initView() {  
 LayoutInflater mInflater = (LayoutInflater) context.getSystemService(Context.*LAYOUT\_INFLATER\_SERVICE*);  
 mView = mInflater.inflate(R.layout.*dialog\_common\_two\_btn*, null);  
 mTextView\_title = mView.findViewById(R.id.*tv\_title*);  
 mTextView\_content = mView.findViewById(R.id.*tv\_content*);  
 mBottom = mView.findViewById(R.id.*tv\_bottom\_btn*);  
 mBottom2 = mView.findViewById(R.id.*tv\_bottom\_btn2*);  
 }  
 */\*\*  
 \* 设置标题  
 \*/* public Builder setTitle(String title) {  
 mTextView\_title.setText(title);  
 return this;  
 }  
 */\*\*  
 \* 设置内容  
 \*/* public Builder setContent(String content){  
 mTextView\_content.setText(content);  
 return this;  
 }  
 */\*\*  
 \* 设置按钮  
 \*/* public Builder setButton(String text, View.OnClickListener listener) {  
 mBottom.setText(text);  
 mBottom.setOnClickListener(listener);  
 return this;  
 }  
 */\*\*  
 \* 设置按钮2  
 \*/* public Builder setButton2(String text, View.OnClickListener listener) {  
 mBottom2.setText(text);  
 mBottom2.setOnClickListener(listener);  
 return this;  
 }  
 */\*\*  
 \* 设置不可以返回键+out取消  
 \*/* public Builder noCancleAll() {  
 canBack = false;  
 canTouch = false;  
 return this;  
 }  
 */\*\*  
 \* 设置不可以点击外部取消，可以点击返回按钮取消  
 \*/* public Builder noCancleTouchButCanBackPress() {  
 canBack = true;  
 canTouch = false;  
 return this;  
 }  
 */\*\*  
 \* 关闭dialog  
 \*/* public void closeDialog() {  
 if (mDialog != null) {  
 mDialog.dismiss();  
 }  
 }  
 public CommonDialogTwoBtn create() {  
 mDialog = new CommonDialogTwoBtn(context, R.style.*ComWinTransBackHalfDia*);  
 mDialog.setContentView(mView);  
 //设置dialog的大小  
 android.view.WindowManager.LayoutParams lp = mDialog.getWindow().getAttributes();  
 lp.width = lp.*WRAP\_CONTENT*;//dp  
 lp.height = lp.*WRAP\_CONTENT*;  
 mDialog.getWindow().setAttributes(lp);  
 mDialog.setCancelable(canBack);  
 mDialog.setCanceledOnTouchOutside(canTouch);  
 return mDialog;  
 }  
 public boolean isShowing() {  
 if (mDialog != null && mDialog.isShowing()) {  
 return true;  
 }  
 return false;  
 }  
 }  
}

# popupWindow

## 设置基本的view

**public** **class** **MPoPuWindow** **extends** **PopupWindow** **implements** OnClickListener {

**public** **Context** mContext;

**private** Type type;

**public** **Activity** mActivity;

**private** **File** file;

**private** **Uri** ImgUri;

**private** **TextView** mTakePhoto, mAlbumPhoto, mCancel;

**public** **MPoPuWindow**(**Context** context, **Activity** mActivity) {

initView(context);

**this**.mActivity = mActivity;

}

**private** **void** **initView**(**Context** mContext) {

**this**.mContext = mContext;

**View** v = **LayoutInflater**.*from*(mContext).inflate(**R**.**layout**.**activity\_popu**, **null**);

setContentView(v);

mTakePhoto = (**TextView**) v.findViewById(**R**.**id**.**photo\_take**);

mAlbumPhoto = (**TextView**) v.findViewById(**R**.**id**.**photo\_album**);

mCancel = (**TextView**) v.findViewById(**R**.**id**.**photo\_cancel**);

mTakePhoto.setOnClickListener(**this**);

mAlbumPhoto.setOnClickListener(**this**);

mCancel.setOnClickListener(**this**);

**// 设置SelectPicPopupWindow弹出窗体的宽**

setWidth(**ViewGroup**.**LayoutParams**.**MATCH\_PARENT**);

**// 设置SelectPicPopupWindow弹出窗体的高**

setHeight(**ScreenUtils**.*getScreenHeight*(mContext));

**// 设置SelectPicPopupWindow弹出窗体可点**

setTouchable(**true**);

setFocusable(**true**);

setOutsideTouchable(**true**);

**// 刷新状态**

update();

**// 设置SelectPicPopupWindow弹出窗体动画效果**

**this**.setAnimationStyle(**R**.**style**.**popuwindow\_from\_bottom**);

**// 实例化一个ColorDrawable颜色为半透明**

**ColorDrawable** dw = **new** ColorDrawable(0x50000000);

**// 设置SelectPicPopupWindow弹出窗体的背景**

setBackgroundDrawable(dw);

**//设置touch事件，点击外部消失**

setTouchInterceptor(**new** OnTouchListener() {

@Override

**public** **boolean** **onTouch**(**View** arg0, **MotionEvent** arg1) {

**if** (arg1.getAction() == **MotionEvent**.**ACTION\_OUTSIDE**) {

dismiss();

**return** **true**;

}

**return** **false**;

}

});

}

/\*\*

\* 显示popwindow

\*/

**public** **void** **showPopupWindow**(**View** parent) {

**if** (!isShowing()) {

showAtLocation(parent, **Gravity**.**NO\_GRAVITY**, 0, **ScreenUtils**.*getScreenHeight*(mContext));

} **else** {

**//dismiss();**

}

}

## 设置动画效果

setAnimationStyle(**R**.**style**.**dir\_popupWindow\_anim**);

<style name="dir\_popupWindow\_anim">

<item name="android:windowEnterAnimation">@anim/slide\_up</item>

<item name="android:windowExitAnimation">@anim/slide\_down</item>

</style>

<?xml version="1.0" encoding="utf-8"?>

<set xmlns:android="http://schemas.android.com/apk/res/android" >

<translate

android:duration="200"

android:fromXDelta="0"

android:fromYDelta="100%"

android:toXDelta="0"

android:toYDelta="0" />

</set>

## **放置位置**

**正上方**

**int**[] location = **new** **int**[2];

button.getLocationOnScreen(location);

mDirPopupWindow.showAtLocation(button, **Gravity**.**NO\_GRAVITY**,

location[0] + button.getWidth() / 2 - mDirPopupWindow.getWidth() / 2,

location[1] - mDirPopupWindow.getHeight());

**正左方**

**int**[] location = **new** **int**[2];

button.getLocationOnScreen(location);

mDirPopupWindow.showAtLocation(button, **Gravity**.**NO\_GRAVITY**,

location[0] - mDirPopupWindow.getWidth(),

location[1] + button.getHeight() / 2 - mDirPopupWindow.getHeight() / 2);

**正右方**

**int**[] location = **new** **int**[2];

button.getLocationOnScreen(location);

mDirPopupWindow.showAtLocation(button, **Gravity**.**NO\_GRAVITY**,

location[0] + button.getWidth(),

location[1] + button.getHeight() / 2 - mDirPopupWindow.getHeight() / 2);

**正下方**

**int**[] location = **new** **int**[2];

button.getLocationOnScreen(location);

mDirPopupWindow.showAtLocation(button, **Gravity**.**NO\_GRAVITY**,

location[0] + button.getWidth() / 2 - mDirPopupWindow.getWidth() / 2,

location[1] + button.getHeight());

**showAsDropDown处于父控件的正下方**

当父控件在最下方时，处于父控件的正上方**mDirPopupWindow.showAsDropDown(button, 0, 0);**

## 某个特定位置显隐动画

Show

<?xml version="1.0" encoding="utf-8"?>  
<set xmlns:android="http://schemas.android.com/apk/res/android"  
 android:duration="300">  
 <scale  
 android:fromXScale="1.0"  
 android:fromYScale="0.0"  
 android:pivotX="50%"  
 android:pivotY="0"  
 android:toXScale="1.0"  
 android:toYScale="1.0" />  
</set>

**Hidden**

<?xml version="1.0" encoding="utf-8"?>  
<set xmlns:android="http://schemas.android.com/apk/res/android"  
 android:duration="300">  
 <scale  
 android:fromXScale="1.0"  
 android:fromYScale="1.0"  
 android:pivotX="50%"  
 android:pivotY="0"  
 android:toXScale="1.0"  
 android:toYScale="0.0" />  
</set>

## 7.0以上适配

在android7.1以上，如果不主动约束PopuWindow的大小，比如，设置布局大小为 MATCH\_PARENT,那么PopuWindow会变得尽可能大

if (Build.VERSION.SDK\_INT >= 25) {

Rect visibleFrame = new Rect();

parentStandard.getGlobalVisibleRect(visibleFrame);

int height = parentStandard.getResources().getDisplayMetrics().heightPixels - visibleFrame.bottom;

mPopupWindow.setHeight(height);

}

在android7.0以上，showAsDropDown有bug，需要重新计算后用showAtLocation替代

if (Build.VERSION.SDK\_INT >= 24) {

int[] location = new int[2];

v.getLocationInWindow(location);

int y = location[1];

int yLocation = y + v.getHeight();

mPopupWindow.showAtLocation(v, Gravity.NO\_GRAVITY, 0, yLocation);

} else {

mPopupWindow.showAsDropDown(parentStandard);

}

在7.0及以下，不会自动限制在屏幕内，7.1以上会。为了适配7.0，以显示在控件下方为例，需要计算最低y轴高度

//7.0手机没有自动适配到屏幕内，强行假设有5个item的大小来适配  
int[] location = new int[2];  
v.getLocationInWindow(location);  
int y = location[1];  
int yLocation = y + v.getHeight();  
//7.0以下手机没有自动适配到屏幕内，强行适配低定位高度  
if (Build.VERSION.*SDK\_INT* <= 24) {  
 popupWindow.getHeight();  
 if (yLocation > 1100) yLocation = 1100;  
}  
popupWindow.showAtLocation(v, Gravity.*NO\_GRAVITY*, x, yLocation);

## BasePopup库

文章地址：[https://github.com/razerdp/Article/blob/master/xxx.md](https://github.com/razerdp/Article/blob/master/%E4%BA%B2%EF%BC%8C%E8%BF%98%E5%9C%A8%E4%B8%BAPopupWindow%E7%83%A6%E6%81%BC%E5%90%97.md)

git地址：<https://github.com/razerdp/BasePopup>

引用：implementation 'com.github.razerdp:BasePopup:2.1.8'

### 使用

1.编写pop的xml文件

像平时定制View布局文件一样定制PopupWindow布局

<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="center"  
 android:background="#63CDBD"  
 android:orientation="vertical">  
  
 <TextView  
 android:id="@+id/tx\_1"  
 style="@style/wrap\_redfe\_14"  
 android:padding="16dp"  
 android:text="test1" />  
</LinearLayout>

2.创建自己的Popup类并继承BasePopupWindow

public class PopupTest1 extends BasePopupWindow {  
 public PopupTest1(Context context) {  
 super(context);  
 }  
 // 必须实现，这里返回您的contentView  
 // 为了让库更加准确的做出适配，强烈建议使用createPopupById()进行inflate  
 @Override  
 public View onCreateContentView() {  
 return createPopupById(R.layout.*popup\_test1*);  
 }  
 // 以下为可选代码（非必须实现）  
 // 返回作用于PopupWindow的show和dismiss动画，本库提供了默认的几款动画，这里可以自由实现  
 @Override  
 protected Animation onCreateShowAnimation() {  
 return getDefaultScaleAnimation(true);  
 }  
 @Override  
 protected Animation onCreateDismissAnimation() {  
 return getDefaultScaleAnimation(false);  
 }  
}

3.show

展示PopupWindow的方法有三种，分别是showPopupWindow()和showPopupWindow(View anchor)、showPopupWindow(int x, int y)

new DemoPopup(getContext()).showPopupWindow();

//new DemoPopup(getContext()).showPopupWindow(v);

//new DemoPopup(getContext()).showPopupWindow(x, y);

这三个方法有不同的含义：

* showPopupWindow()：无参传入，此时PopupWindow参考对象为屏幕（或者说整个DecorView），Gravity的表现就像在FrameLayout里面的Gravity表现一样，表示其处于屏幕的哪个方位
* showPopupWindow(View anchor)：传入AnchorView，此时PopupWindow参考对象为传入的anchorView，Gravity的表现则意味着这个PopupWindow应该处于目标AnchorView的哪个方位
* showPopupWindow(int x, int y)：传入位置信息，此时PopupWindow将会在指定位置弹出。

### 平移动画

#### 工具类

public class PopupAniUtil {  
 */\*\*  
 \* 初始化平移动画  
 \*/* public static void initTranslateAnimate(BasePopupWindow basePopupWindow, boolean isHasAnchorView) {  
 int gravity = basePopupWindow.getPopupGravity();  
 float in\_fromX = 0;  
 float in\_toX = 0;  
 float in\_fromY = 0;  
 float in\_toY = 0;  
  
 float exit\_fromX = 0;  
 float exit\_toX = 0;  
 float exit\_fromY = 0;  
 float exit\_toY = 0;  
  
 switch (gravity & Gravity.*HORIZONTAL\_GRAVITY\_MASK*) {  
 case Gravity.*LEFT*:  
 case Gravity.*START*:  
 in\_fromX = isHasAnchorView ? 1f : -1f;  
 exit\_toX = isHasAnchorView ? 1f : -1f;  
 break;  
 case Gravity.*RIGHT*:  
 case Gravity.*END*:  
 in\_fromX = isHasAnchorView ? -1f : 1f;  
 exit\_toX = isHasAnchorView ? -1f : 1f;  
 break;  
 case Gravity.*CENTER\_HORIZONTAL*:  
 break;  
 default:  
 break;  
 }  
 switch (gravity & Gravity.*VERTICAL\_GRAVITY\_MASK*) {  
 case Gravity.*TOP*:  
 in\_fromY = isHasAnchorView ? 1f : -1f;  
 exit\_toY = isHasAnchorView ? 1f : -1f;  
 break;  
 case Gravity.*BOTTOM*:  
 in\_fromY = isHasAnchorView ? -1f : 1f;  
 exit\_toY = isHasAnchorView ? -1f : 1f;  
 break;  
 case Gravity.*CENTER\_VERTICAL*:  
 break;  
 default:  
 break;  
 }  
 basePopupWindow.setShowAnimation(*createTranslateAnimate*(in\_fromX, in\_toX, in\_fromY, in\_toY));  
 basePopupWindow.setDismissAnimation(*createTranslateAnimate*(exit\_fromX, exit\_toX, exit\_fromY, exit\_toY));  
 }  
  
 */\*\*  
 \* 初始化平移动画for快速  
 \*/* public static Animation initTranslateAnimateQuick(int gravity, boolean isHasAnchorView, boolean isIn) {  
 float in\_fromX = 0;  
 float in\_toX = 0;  
 float in\_fromY = 0;  
 float in\_toY = 0;  
  
 float exit\_fromX = 0;  
 float exit\_toX = 0;  
 float exit\_fromY = 0;  
 float exit\_toY = 0;  
  
 switch (gravity & Gravity.*HORIZONTAL\_GRAVITY\_MASK*) {  
 case Gravity.*LEFT*:  
 case Gravity.*START*:  
 in\_fromX = isHasAnchorView ? 1f : -1f;  
 exit\_toX = isHasAnchorView ? 1f : -1f;  
 break;  
 case Gravity.*RIGHT*:  
 case Gravity.*END*:  
 in\_fromX = isHasAnchorView ? -1f : 1f;  
 exit\_toX = isHasAnchorView ? -1f : 1f;  
 break;  
 case Gravity.*CENTER\_HORIZONTAL*:  
 break;  
 default:  
 break;  
 }  
 switch (gravity & Gravity.*VERTICAL\_GRAVITY\_MASK*) {  
 case Gravity.*TOP*:  
 in\_fromY = isHasAnchorView ? 1f : -1f;  
 exit\_toY = isHasAnchorView ? 1f : -1f;  
 break;  
 case Gravity.*BOTTOM*:  
 in\_fromY = isHasAnchorView ? -1f : 1f;  
 exit\_toY = isHasAnchorView ? -1f : 1f;  
 break;  
 case Gravity.*CENTER\_VERTICAL*:  
 break;  
 default:  
 break;  
 }  
 return isIn ? *createTranslateAnimate*(in\_fromX, in\_toX, in\_fromY, in\_toY)

: *createTranslateAnimate*(exit\_fromX, exit\_toX, exit\_fromY, exit\_toY);  
 }  
  
 private static Animation createTranslateAnimate(float fromX, float toX, float fromY, float toY) {  
 Animation result = new TranslateAnimation(Animation.*RELATIVE\_TO\_PARENT*,  
 fromX,  
 Animation.*RELATIVE\_TO\_PARENT*,  
 toX,  
 Animation.*RELATIVE\_TO\_PARENT*,  
 fromY,  
 Animation.*RELATIVE\_TO\_PARENT*,  
 toY);  
 result.setDuration(500);  
 return result;  
 }  
}

#### 自定义PopUpwindow重写show方法

public class PopupTest1 extends BasePopupWindow {  
 private Context context;  
 public PopupTest1(Context context) {  
 super(context);  
 this.context = context;  
 }  
 @Override  
 public View onCreateContentView() {  
 View view = createPopupById(R.layout.*popup\_test1*);  
 initView(view);  
 return view;  
 }  
 private void initView(View view) {  
 TextView textView1 = view.findViewById(R.id.*tx\_1*);  
 textView1.setOnClickListener(v -> {  
 Toast.*makeText*(context, "clicked", Toast.*LENGTH\_LONG*).show();  
 dismiss();  
 });  
 }  
 @Override  
 public void showPopupWindow() {  
 PopupAniUtil.*initTranslateAnimate*(this, false);  
 super.showPopupWindow();  
 }  
 @Override  
 public void showPopupWindow(View anchorView) {  
 PopupAniUtil.*initTranslateAnimate*(this, true);  
 super.showPopupWindow(anchorView);  
 }  
}

#### 快速集成

val gravity = Gravity.*BOTTOM* or Gravity.*CENTER\_HORIZONTAL*pop = QuickPopupBuilder.with(bActivity)  
 .contentView(R.layout.*popup\_test1*)  
 .config(QuickPopupConfig()  
 .gravity(gravity)  
 .withShowAnimation(PopupAniUtil.initTranslateAnimateQuick(gravity, false, true))  
 .withDismissAnimation(PopupAniUtil.initTranslateAnimateQuick(gravity, false, false))  
 .withClick(R.id.*tx\_1*) **{** Toast.makeText(bActivity, "clicked", Toast.*LENGTH\_LONG*).show()  
 pop.dismiss()  
 **}**)  
 .show()