Main Topics Specific

deepwaterooo

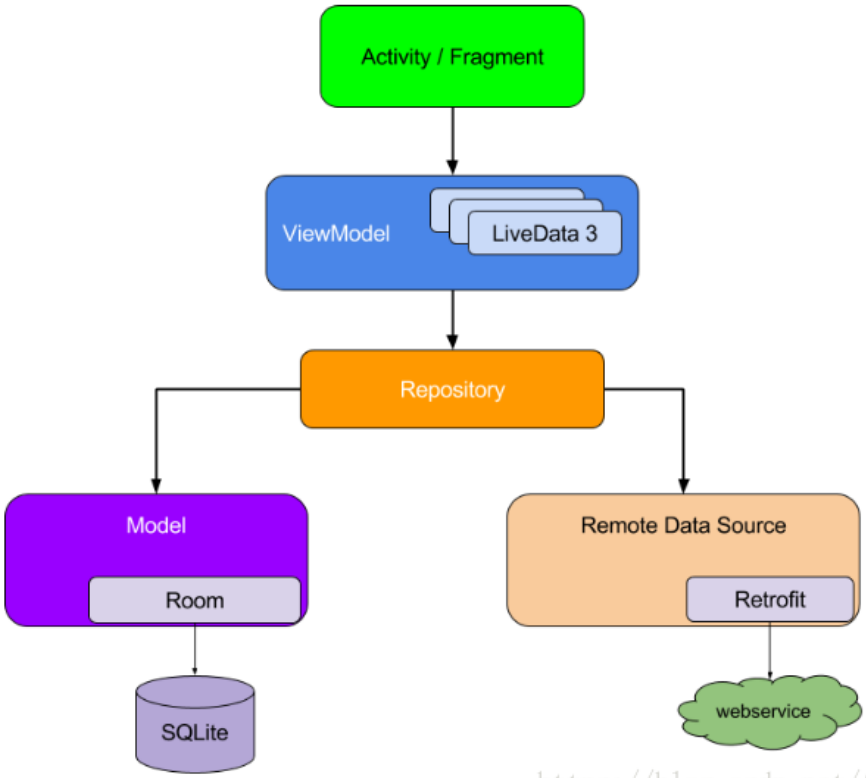
2021 年 12 月 20 日

目录

1 Jetpack library

- https://blog.csdn.net/Alexwll/article/details/83302173
- Android Jetpack 组件的优势：
  - 轻松管理应用程序的生命周期
  - 构建可观察的数据对象，以便在基础数据库更改时通知视图
  - 存储在应用程序轮换中未销毁的 UI 相关数据，在界面重建后恢复数据
  - 轻松的实现 SQLite 数据库
  - 系统自动调度后台任务的执行，优化使用性能

- Android Jetpack 组件推荐的使用项目架构



- 上面架构组件的功能如下：
  - Activity 和 Fragment 负责产品与用户的交互
  - ViewModel 作为数据的存储和驱动
  - Resposity 负责调度数据的获取
  - Room 储存本地序列化的数据
  - Retrofit 获取远程数据的数据

2 JUnit Mockito Unit Testing

- 当您需要更快地运行测试而不需要与在真实设备上运行测试关联的保真度和置信度时，可以使用本地单元测试来评估应用的逻辑。对于这种方法，您通常使用 Robolectric 或模拟框架（如 Mockito）来实现依赖项。
  - 如果您的测试对 Android 框架有依赖性（特别是与框架建立复杂互动的测试），最好使用 Robolectric 添加框架依赖项。
  - 如果您的测试对 Android 框架的依赖性极小，或者如果测试仅取决于您自己的对象，可以使用诸如 Mockito 之类的模拟框架添加模拟依赖项。

```
dependencies {
        // Required -- JUnit 4 framework
        testImplementation 'junit:junit:4.12'
```

```
    // Optional -- Robolectric environment
    testImplementation 'androidx.test:core:1.0.0'
    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:1.10.19'
}
```

- 创建本地单元测试类

```
import com.google.common.truth.Truth.assertThat;
   import org.junit.Test;
   public class EmailValidatorTest {
       @Test
       public void emailValidator_CorrectEmailSimple_ReturnsTrue() {
           assertThat(EmailValidator.isValidEmail("name@email.com")).isTrue();
       }
   }
```

- 添加框架依赖项
  - 如果您的测试与多个 Android 框架依赖项互动，或以复杂的方式与这些依赖项互动，请使用 AndroidX Test 提供的 Robolectric 工件。Robolectric 在本地 JVM 或真实设备上执行真实的 Android 框架代码和原生框架代码的虚假对象。
  - 以下示例展示了如何创建使用 Robolectric 的单元测试：

```
app/build.gradle
    android {
        // ...
        testOptions {
            unitTests.includeAndroidResources = true
        }
    }
```

-

```
    import android.content.Context;
    import androidx.test.core.app.ApplicationProvider;
    import org.junit.Test;
    import static com.google.common.truth.Truth.assertThat;
    public class UnitTestSampleJava {
        private static final String FAKE_STRING = "HELLO_WORLD";
        private Context context = ApplicationProvider.getApplicationContext();
        @Test
        public void readStringFromContext_LocalizedString() {
            // Given a Context object retrieved from Robolectric...
            ClassUnderTest myObjectUnderTest = new ClassUnderTest(context);
            // ...when the string is returned from the object under test...
            String result = myObjectUnderTest.getHelloWorldString();
            // ...then the result should be the expected one.
            assertThat(result).isEqualTo(FAKE_STRING);
        }
    }
```

- 稍微复杂一点儿的

```
    import android.content.Context;
    import org.junit.Test;
    import org.junit.runner.RunWith;
    import org.mockito.Mock;
    import org.mockito.junit.MockitoJUnitRunner;
    import static com.google.common.truth.Truth.assertThat;
    import static org.mockito.Mockito.when;
    @RunWith(MockitoJUnitRunner.class)
    public class UnitTestSample {
        private static final String FAKE_STRING = "HELLO WORLD";
```

```
    @Mock
    Context mockContext;
    @Test
    public void readStringFromContext_LocalizedString() {
        // Given a mocked Context injected into the object under test...
        when(mockContext.getString(R.string.hello_world))
                .thenReturn(FAKE_STRING);
        ClassUnderTest myObjectUnderTest = new ClassUnderTest(mockContext);
        // ...when the string is returned from the object under test...
        String result = myObjectUnderTest.getHelloWorldString();
        // ...then the result should be the expected one.
        assertThat(result, is(FAKE_STRING));
    }
}
```

3 Retrofit2 Json Parsing (data downloaded from internet?)

3.1 常见的网络框架

- HttpURLConnection: 在 Android 2.2 版本之前，HttpClient 拥有较少的 bug，因此使用它是 *** 的选择。而在 Android 2.3 版本及以后，HttpURLConnection 则是 *** 的选择。它的 API 简单，体积较小，因而非常适用于 Android 项目。压缩和缓存机制可以有效地减少网络访问的流量，在提升速度和省电方面也起到了较大的作用。对于新的应用程序应该更加偏向于使用 HttpURLConnection，因为在以后的工作当中我们也会将更多的时间放在优化 HttpURLConnection 上面。特点：比较轻便，灵活，易于扩展，在 3.0 后以及 4.0 中都进行了改善，如对 HTTPS 的支持，在 4.0 中，还增加了对缓存的支持。
- HttpClient：高效稳定，但是维护成本高昂，故 android 开发团队不愿意在维护该库而是转投更为轻便的
- okHttp：okhttp 是一个 Java 的 HTTP+SPDY 客户端开发包，同时也支持 Android。需要 Android 2.3 以上。特点：OKHttp 是 Android 版 Http 客户端。非常高效，支持 SPDY、连接池、GZIP 和 HTTP 缓存。默认情况下，OKHttp 会自动处理常见的网络问题，像二次连接、SSL 的握手问题。如果你的应用程序中集成了 OKHttp，Retrofit 默认会使用 OKHttp 处理其他网络层请求。从 Android4.4 开始 HttpURLConnection 的底层实现采用的是 okHttp。
- volley：早期使用 HttpClient，后来使用 HttpURLConnection，是谷歌 2013 年推出的网络请求框架，非常适合去进行数据量不大，但通信频繁的网络操作，而对于大数据量的网络操作，比如说下载文件等，Volley 的表现就会非常糟糕。
- xutils：缓存网络请求数据
- Retrofit：和 Volley 框架的请求方式很相似，底层网络请求采用 okhttp（效率高，android4.4 底层采用 okhttp），采用注解方式来指定请求方式和 url 地址，减少了代码量。

3.2 **Retrofit** 库的核心实现原理是什么**?** 如果让你实现这个库的某些核心功能，你会考虑怎么去实现**?**

- Retrofit 主要是在 create 方法中采用动态代理模式 (通过访问代理对象的方式来间接访问目标对象) 实现接口方法，这个过程构建了一个 ServiceMethod 对象，根据方法注解获取请求方式，参数类型和参数注解拼接请求的链接，当一切都准备好之后会把数据添加到 Retrofit 的 RequestBuilder 中。然后当我们主动发起网络请求的时候会调用 okhttp 发起网络请求，okhttp 的配置包括请求方式，URL 等在 Retrofit 的 RequestBuilder 的 build() 方法中实现，并发起真正的网络请求。
- 你从这个库中学到什么有价值的或者说可借鉴的设计思想?

- 内部使用了优秀的架构设计和大量的设计模式，在我分析过 Retrofit 最新版的源码和大量优秀的 Retrofit 源码分析文章后，我发现，要想真正理解 Retrofit 内部的核心源码流程和设计思想，首先，需要对它使用到的九大设计模式有一定的了解，下面我简单说一说:
- 1、创建 Retrofit 实例: 使用 建造者模式 通过内部 Builder 类建立了一个 Retroift 实例。网络请求工厂使用了 工厂模式 方法。
- 2、创建网络请求接口的实例:
  - 首先，使用 外观模式 统一调用创建网络请求接口实例和网络请求参数配置的方法。然后，使用 动态代理 动态地去创建网络请求接口实例。
  - 接着，使用了 建造者模式 & 单例模式 创建了 serviceMethod 对象。
  - 再者，使用了 策略模式 对 serviceMethod 对象进行网络请求参数配置，即通过解析网络请求接口方法的参数、返回值和注解类型，从 Retrofit 对象中获取对应的网络的 url 地址、网络请求执行器、网络请求适配器和数据转换器。
  - 最后，使用了 装饰者模式 ExecuteCallBack 为 serviceMethod 对象加入线程切换的操作，便于接受数据后通过 Handler 从子线程切换到主线程从而对返回数据结果进行处理。
- 3、发送网络请求: 在异步请求时，通过静态 delegate 代理对网络请求接口的方法中的每个参数使用对应的 ParameterHanlder 进行解析。
- 4、解析数据
- 5、切换线程: 使用了 适配器模式 通过检测不同的 Platform 使用不同的回调执行器，然后使用回调执行器切换线程，这里同样是使用了装饰模式。
- 6、处理结果

4 SQLite Datebase

5 RecyclerView

5.1 RecyclerView 与 ListView

- RecyclerView 与 ListView 的不同点，主要在于以下几个特性：
  - Adapter 中的 ViewHolder 模式，ListView 没有严格的 ViewHolder 设计模式。但是在 RecyclerView 中，Adapter 必须按照 ViewHolder 模式实现至少一个 ViewHolder。
  - 定制 Item 视图，ListView 只能实现垂直线性排列的列表视图。RecyclerView 可以通过设置 RecyclerView.LayoutManager 来定制不同风格的视图，比如水平滚动列表或者不规则的瀑布流列表。
  - Item 动画，在 ListView 中没有提供任何方法或者接口，以实现 Item 的增删动画。RecyclerView 可以通过设置 RecyclerView.ItemAnimator 来为 Item 增加动画效果。
  - 设置数据源，在 LisView 中针对不同数据封装了各种类型的 Adapter，比如用来处理数组的 ArrayAdapter 和用来展示 Database 结果的 CursorAdapter。而 RecyclerView 中必须自定义实现 RecyclerView.Adapter 并为其提供数据集合。
  - 设置 Item 分割线，在 ListView 中可以通过设置 android:divider 属性来为两个 Item 间设置分割线。而 RecyclerView 必须使用 RecyclerView.ItemDecoration，这种实现方式更灵活，样式也更加丰富。

  - 设置 Item 点击事件，在 ListView 中存在 AdapterView.OnItemClickListener 接口，用来绑定 Item 的点击事件。而 RecyclerView 并没有提供这样的接口，但是它提供了另外一个接口 RcyclerView.OnItemTouchListener，用来响应 Item 的触摸事件。

- RecyclerView 的优点
  - RecyclerView 封装了 viewholder 的回收复用，也就是说 RecyclerView 标准化了 ViewHolder，编写 Adapter 面向的是 ViewHolder 而不再是 View 了，复用的逻辑被封装了，写起来更加简单。
  - 直接省去了 listview 中 convertView.setTag(holder) 和 convertView.getTag() 这些繁琐的步骤。
  - 提供了一种插拔式的体验，高度的解耦，异常的灵活，针对一个 Item 的显示 RecyclerView 专门抽取出了相应的类，来控制 Item 的显示，使其的扩展性非常强。
  - 设置布局管理器以控制 Item 的布局方式，横向、竖向以及瀑布流方式
    * 例如：你想控制横向或者纵向滑动列表效果可以通过 LinearLayoutManager 这个类来进行控制 (与 GridView 效果对应的是 GridLayoutManager, 与瀑布流对应的还 StaggeredGridLayoutManager 等)。也就是说 RecyclerView 不再拘泥于 ListView 的线性展示方式，它也可以实现 GridView 的效果等多种效果。
  - 可设置 Item 的间隔样式（可绘制）
  - 通过继承 RecyclerView 的 ItemDecoration 这个类，然后针对自己的业务需求去书写代码。
  - 可以控制 Item 增删的动画，可以通过 ItemAnimator 这个类进行控制，当然针对增删的动画，RecyclerView 有其自己默认的实现。

5.2 如果你想使用 **RecyclerView**，需要做以下操作：

- 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

5.3 主要部件

5.3.1 Adapter

- 在 RecylerView 中，Adapter 扮演着两个角色：一是根据不同 viewType 创建与之相应的的 itemView，二是访问数据集合并将数据绑定到正确的 View 上。这就需要我们实现以下两个函数：

```
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。
```

- 另外我们还需要重写另一个方法，像 ListView-Adapter 那样，同样地告诉 RecyclerView-Adapter 列表 Items 的总数：

```
public int getItemCount(); // 返回该 Adapter 所持有的 Itme 数量
```

5.3.2 ViewHolder

- ViewHolder 描述 RecylerView 中某个位置的 itemView 和元数据信息，属于 Adapter 的一部分，其实现类通常用于保存 findViewById 的结果。主要元素组成有：

```
public static abstract class ViewHolder {
    View itemView;// itemView
    int mPosition;// 位置
    int mOldPosition;// 上一次的位置
    long mItemId;// itemId
    int mItemViewType;// itemViewType
    int mPreLayoutPosition;
    int mFlags;// ViewHolder 的状态标志
    int mIsRecyclableCount;
    Recycler mScrapContainer;// 若非空，表明当前 ViewHolder 对应的 itemView 可以复用
}
```

- 关于 ViewHolder，这里主要介绍 mFlags：
  - FLAG——ViewHolder 已经绑定到某个位置，mPosition、mItemId、mItemViewType 都有效
  - FLAG——ViewHolder 绑定的 View 对应的数据过时需要重新绑定，mPosition、mItemId 还是一致的
  - FLAG——ViewHolder 绑定的 View 对应的数据无效，需要完全重新绑定不同的数据
  - FLAG——ViewHolder 对应的数据已经从数据集移除
  - FLAG——ViewHolder 不能复用
  - FLAG——这个状态的 ViewHolder 会加到 scrap list 被复用。
  - FLAG——ViewHolder 内容发生变化，通常用于表明有 ItemAnimator 动画
  - FLAG——ViewHolder 完全由 LayoutManager 管理，不能复用
  - FLAG——ViewHolder 从父 RecyclerView 临时分离的标志，便于后续移除或添加回来
  - FLAG——ViewHolder 不知道对应的 Adapter 的位置，直到绑定到一个新位置
  - FLAG——方法 addChangePayload(null) 调用时设置

5.3.3 LayoutManager

- LayoutManager 主要作用是，测量和摆放 RecyclerView 中 itemView，以及当 itemView 对用户不可见时循环复用处理。
- 布局管理器：RecyclerView.LayoutManager
- 上述代码中 mLayoutManager 对象是布局管理器，RecyclerView 提供了三种布局管理器：
  - LinerLayoutManager（线性）：以垂直或者水平列表方式展示 Item
  - GridLayoutManager （网格）：以网格方式展示 Item
  - StaggeredGridLayoutManager（瀑布流）：以瀑布流方式展示 Item

```
// LinearLayoutManager 是用来做列表布局，也就是单列的列表
LinearLayoutManager linearLayoutManager = new LinearLayoutManager(this);
// 设置为垂直布局，默认是垂直的（垂直：LinearLayoutManager.VERTICAL，水平：LinearLayoutManager.HORIZONTAL）
linearLayoutManager.setOrientation(LinearLayoutManager.HORIZONTAL);
// 设置布局管理器
rvView.setLayoutManager(linearLayoutManager);

// 设置网格布局
GridLayoutManager gridLayoutManager = new GridLayoutManager(this, 4); //  第二个参数，行数或是列数，瀑布流布局类似
```

```
// 设置布局管理器
rvView.setLayoutManager(gridLayoutManager);

// 设置竖直瀑布流布局
StaggeredGridLayoutManager staggeredGridLayoutManager = new StaggeredGridLayoutManager(4, StaggeredGridLayoutManager.VERTIC
// 设置布局管理器
rvView.setLayoutManager(staggeredGridLayoutManager);
```

5.3.4 ItemDecoration: 间隔样式：RecyclerView.ItemDecoration

- 当我们想在某些 item 上加一些特殊的 UI 时，往往都是在 itemView 中先布局好，然后通过设置可见性来决定哪些位置显示不显示。RecyclerView 将 itemView 和装饰 UI 分隔开来，装饰 UI 即 ItemDecoration，主要用于绘制 item 间的分割线、高亮或者 margin 等。其源码如下：

```
public static abstract class ItemDecoration {
    // itemView 绘制之前绘制，通常这部分 UI 会被 itemView 盖住
    public void onDraw(Canvas c, RecyclerView parent, State state) {
        onDraw(c, parent);
    }
    // itemView 绘制之后绘制，这部分 UI 盖在 itemView 上面
    public void onDrawOver(Canvas c, RecyclerView parent, State state) {
        onDrawOver(c, parent);
    }
    // 设置 itemView 上下左右的间距
    public void getItemOffsets(Rect outRect, View view, RecyclerView parent, State state) {
        getItemOffsets(outRect, ((LayoutParams) view.getLayoutParams()).getViewLayoutPosition(),
                parent);
    }
}
```

- 用于绘制 itemView 之间的一些特殊 UI，比如在 itemView 之前设置空白区域、画线等。
- RecyclerView 将 itemView 和装饰 UI 分隔开来，装饰 UI 即 ItemDecoration ，主要用于绘制 item 间的分割线、高亮或者 margin 等
- 通过 recyclerView.addItemDecoration(new DividerDecoration(this)) 对 item 添加装饰；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。

```
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
// 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
public void getItemOffsets(Rect outRect, int itemPosition, RecyclerView parent)
```

自定义间隔样式需要继承 RecyclerView.ItemDecoration 类，该类是个抽象类，主要有三个方法

```
onDraw(Canvas c, RecyclerView parent, State state) // 在 Item 绘制之前被调用，该方法主要用于绘制间隔样式
onDrawOver(Canvas c, RecyclerView parent, State state) // 在 Item 绘制之前被调用，该方法主要用于绘制间隔样式
// 设置 item 的偏移量，偏移的部分用于填充间隔样式，在 RecyclerView 的 onMesure() 中会调用该方法
getItemOffsets(Rect outRect, View view, RecyclerView parent, State state)
```

- onDraw() 和 onDrawOver() 这两个方法都是用于绘制间隔样式，我们只需要复写其中一个方法即可。直接来看一下自定义的间隔样式的实现，参考官方实例, 然后在代码中设置 RecyclerView 的间隔样式

5.3.5 ItemAnimator

- 过去 AdapterView 的 item 项操作往往是没有动画的。现在 RecyclerView 的 ItemAnimator 使得 item 的动画实现变得简单而样式丰富，我们可以自定义 item 项不同操作（如添加，删除）的动画效果。

- RecyclerView 做了一个 notifyItemChanged() 的操作，功能都顺利实现，问题当前 Item 闪烁，QA 甚至为此提了 Bug。闪烁主要由于 RecyclerView 使用的默认的动画导致的，所以解决的方法就是修改默认的动画。

1. 问题解决

(a) 更新部分 item

- （1）个别更新

```
imgAdapter.notifyItemChanged(i); // 只更新修改的 item
```

- （2）删除某个

```
selectedImgs.remove(position);
notifyItemRemoved(position);
notifyItemRangeChanged(0,selectedImgs.size());
```

(b) 屏蔽动画方法

- DefaultItemAnimator 继承自 SimpleItemAnimator，里面有个方法是：

```
    /**
     * Sets whether this ItemAnimator supports animations of item change events.
     * If you set this property to false, actions on the data set which change the
     * contents of items will not be animated. What those animations do is left
     * up to the discretion of the ItemAnimator subclass, in its
     * {@link #animateChange(ViewHolder, ViewHolder, int, int, int, int)} implementation.
     * The value of this property is true by default.
     *
     */
    public void setSupportsChangeAnimations(boolean supportsChangeAnimations) {
        mSupportsChangeAnimations = supportsChangeAnimations;
    }
```

- 只要设置为 false，就可以不显示动画了，也就解决了闪烁问题。关键代码：

```
((SimpleItemAnimator)recyclerView.getItemAnimator()).setSupportsChangeAnimations(false);
```

(c) 设置动画执行时间为 0 来解决闪烁问题

```
recyclerView.getItemAnimator().setChangeDuration(0);// 通过设置动画执行时间为 0 来解决闪烁问题
```

(d) 修改默认的动画方法

```
// 1. 定义动画类
public class NoAlphaItemAnimator extends SimpleItemAnimator {
}
// 2. 将 DefaultItemAnimator 类里的代码全部 copy 到自己写的动画类中，然后做一些修改。
// 3. 首先找到 private void animateChangeImpl(final ChangeInfo changeInfo) {} 方法。
// 4. 找到方法里这两句代码：
//    4.1 去掉 alpha(0)
oldViewAnim.alpha(0).setListener(new VpaListenerAdapter() {...}).start();
oldViewAnim.setListener(new VpaListenerAdapter() {...}).start();
//    4.2 去掉 alpha(1)
newViewAnimation.translationX(0).translationY(0).setDuration(getChangeDuration()).
                    alpha(1).setListener(new VpaListenerAdapter() {...}).start();
newViewAnimation.translationX(0).translationY(0).setDuration(getChangeDuration()).
                    setListener(new VpaListenerAdapter() {...}).start();
// 5. 最后使用修改后的动画
recyclerView.setItemAnimator(new NoAlphaItemAnimator());
```

5.3.6 刷新方法

除了 adapter.notifyDataSetChanged() 这个方法之外，新的 Adapter 还提供了其他的方法，如下：

```
// 刷新所有
public final void notifyDataSetChanged();
// position 数据发生了改变，那调用这个方法，就会回调对应 position 的 onBindViewHolder() 方法了
public final void notifyItemChanged(int position);
// 刷新从 positionStart 开始 itemCount 数量的 item 了（这里的刷新指回调 onBindViewHolder() 方法）
public final void notifyItemRangeChanged(int positionStart, int itemCount);

// 在第 position 位置被插入了一条数据的时候可以使用这个方法刷新，注意这个方法调用后会有插入的动画，这个动画可以使用默认的，也可以自己
public final void notifyItemInserted(int position);
// 从 fromPosition 移动到 toPosition 为止的时候可以使用这个方法刷新
public final void notifyItemMoved(int fromPosition, int toPosition);
// 批量添加
public final void notifyItemRangeInserted(int positionStart, int itemCount);
// 第 position 个被删除的时候刷新，同样会有动画
public final void notifyItemRemoved(int position);
// 批量删除
public final void notifyItemRangeRemoved(int positionStart, int itemCount);
```

5.3.7 Recycler

- Recycler 用于管理已经废弃或与 RecyclerView 分离的（scrapped or detached）item view，便于重用。Scrapped view 指依附于 RecyclerView，但被标记为可移除或可复用的 view。
- LayoutManager 获取 Adapter 某一项的 View 时会使用 Recycler。当复用的 View 有效（clean）时，View 能直接被复用，反之若 View 失效（dirty）时，需要重新绑定 View。对于有效的 View，如果不主动调用 request layout，则不需要重新测量大小就能复用。在分析 Recycler 的复用原理之前，我们先了解下如下两个类：

5.3.8 RecycledViewPool

- RecyclerViewPool 用于多个 RecyclerView 之间共享 View。只需要创建一个 RecyclerViewPool 实例，然后调用 RecyclerView 的 setRecycledViewPool(RecycledViewPool) 方法即可。RecyclerView 默认会创建一个 RecyclerViewPool 实例。
- 首先我们看一下 ReyclerPool 的结构。

```
public static class RecycledViewPool {
    private static final int DEFAULT_MAX_SCRAP = 5;
    static class ScrapData {
        ArrayList<ViewHolder> mScrapHeap = new ArrayList<>();
        int mMaxScrap = DEFAULT_MAX_SCRAP;
        long mCreateRunningAverageNs = 0;
        long mBindRunningAverageNs = 0;
    }
    SparseArray<ScrapData> mScrap = new SparseArray<>();
}
```

- 可以看到 RecyclerPool 内部其实是一个 SparseArray，可想而知，key 就是我们的 ViewType，而 Value 是 ArrayList<ViewHolder>。
- 我们来看一下 RecyclerPool 的 put 方法。

```
public void putRecycledView(ViewHolder scrap) {
    final int viewType = scrap.getItemViewType();
    final ArrayList<ViewHolder> scrapHeap = getScrapDataForType(viewType).mScrapHeap;
    if (mScrap.get(viewType).mMaxScrap <= scrapHeap.size())
        return;
    if (DEBUG && scrapHeap.contains(scrap))
        throw new IllegalArgumentException("this scrap item already exists");
    //重置 ViewHolder
    scrap.resetInternal();
    scrapHeap.add(scrap);
}
```

- 其中 resetInternal 方法值得我们注意。

```
void resetInternal() {
    mFlags = 0;
    mPosition = NO_POSITION;
    mOldPosition = NO_POSITION;
    mItemId = NO_ID;
    mPreLayoutPosition = NO_POSITION;
    mIsRecyclableCount = 0;
    mShadowedHolder = null;
    mShadowingHolder = null;
    clearPayload();
    mWasImportantForAccessibilityBeforeHidden = ViewCompat.IMPORTANT_FOR_ACCESSIBILITY_AUTO;
    mPendingAccessibilityState = PENDING_ACCESSIBILITY_STATE_NOT_SET;
    clearNestedRecyclerViewIfNotNested(this);
}
```

- 可以看到所有被 put 进入 RecyclerPool 中的 ViewHolder 都会被重置，这也就意味着 RecyclerPool 中的 ViewHolder 再被复用的时候是需要重新 Bind 的。这一点就可以区分和 CacheViews 中缓存的区别。

RecyclerView				
	是否需要回调createView	是否需要回调bindView	生命周期	备注
mAttachedScrap	否	否	onLayout函数周期内	用于屏幕内ItemView快速重用
mCacheViews	否	否	与mAdapter一致，当mAdapter被更换时，mCacheViews即被缓存至mRecyclerPool	默认上限为2，即缓存屏幕外2个ItemView
mViewCacheExtension				不直接使用，需要用户在定制，默认不实现
mRecyclerPool	否	是	与自身生命周期一致，不再被引用时即被释放	默认上限为5，技术上可以实现所有RecyclerViewPool共用同一个

```
public static class RecycledViewPool {
    private SparseArray<ArrayList<ViewHolder>> mScrap = new SparseArray<ArrayList<ViewHolder>>(); //  SparseArray 应用的实例
    private SparseIntArray mMaxScrap = new SparseIntArray();
    private int mAttachCount = 0;
    private static final int DEFAULT_MAX_SCRAP = 5;

    public void clear() {
        mScrap.clear();
    }
    public void setMaxRecycledViews(int viewType, int max) {
        mMaxScrap.put(viewType, max);
        final ArrayList<ViewHolder> scrapHeap = mScrap.get(viewType);
        if (scrapHeap != null)
            while (scrapHeap.size() > max)
```

```
            scrapHeap.remove(scrapHeap.size() - 1);
    }
    public ViewHolder getRecycledView(int viewType) {
        final ArrayList<ViewHolder> scrapHeap = mScrap.get(viewType);
        if (scrapHeap != null && !scrapHeap.isEmpty()) {
            final int index = scrapHeap.size() - 1;
            final ViewHolder scrap = scrapHeap.get(index);
            scrapHeap.remove(index);
            return scrap;
        }
        return null;
    }
    int size() {
        int count = 0;
        for (int i = 0; i < mScrap.size(); i ++) {
            ArrayList<ViewHolder> viewHolders = mScrap.valueAt(i);
            if (viewHolders != null)
                count += viewHolders.size();
        }
        return count;
    }
    public void putRecycledView(ViewHolder scrap) {
        final int viewType = scrap.getItemViewType();
        final ArrayList scrapHeap = getScrapHeapForType(viewType);
        if (mMaxScrap.get(viewType) <= scrapHeap.size())
            return;
        if (DEBUG && scrapHeap.contains(scrap))
            throw new IllegalArgumentException("this scrap item already exists");
        scrap.resetInternal();
        scrapHeap.add(scrap);
    }
    void attach(Adapter adapter) {
        mAttachCount++;
    }
    void detach() {
        mAttachCount--;
    }
    /**
     * Detaches the old adapter and attaches the new one.
     * <p>
     * RecycledViewPool will clear its cache if it has only one adapter attached and the new
     * adapter uses a different ViewHolder than the oldAdapter.
     *
     * @param oldAdapter The previous adapter instance. Will be detached.
     * @param newAdapter The new adapter instance. Will be attached.
     * @param compatibleWithPrevious True if both oldAdapter and newAdapter are using the same
     *                               ViewHolder and view types.
     */
    void onAdapterChanged(Adapter oldAdapter, Adapter newAdapter, boolean compatibleWithPrevious) {
        if (oldAdapter != null)
            detach();
        if (!compatibleWithPrevious && mAttachCount == 0)
            clear();
        if (newAdapter != null)
            attach(newAdapter);
    }
    private ArrayList<ViewHolder> getScrapHeapForType(int viewType) {
        ArrayList<ViewHolder> scrap = mScrap.get(viewType);
        if (scrap == null) {
            scrap = new ArrayList<ViewHolder>();
            mScrap.put(viewType, scrap);
            if (mMaxScrap.indexOfKey(viewType) < 0)
                mMaxScrap.put(viewType, DEFAULT_MAX_SCRAP);
        }
        return scrap;
    }
}
```

- 通过源码我们可以看出 mScrap 是一个 <viewType, List> 的映射，mMaxScrap 是一个 <viewType, maxNum> 的映射，这两个成员变量代表可复用 View 池的基本信息。调用 setMaxRecycledViews(int viewType, int max) 时，当用于复用的 mScrap 中 viewType 对应的 ViewHolder 个数超过 maxNum 时，会从列表末尾开始丢弃超过的部分。调用 getRecycledView(int view-

Type) 方法时从 mScrap 中移除并返回 viewType 对应的 List 的末尾项。

5.3.9 ViewCacheExtension

- ViewCacheExtension 是一个由开发者控制的可以作为 View 缓存的帮助类。调用 Recycler.getViewFo 方法获取 View 时，Recycler 先检查 attached scrap 和一级缓存，如果没有则检查 ViewCache-Extension.getViewForPositionAndType(Recycler, int, int)，如果没有则检查 RecyclerView-Pool。注意：Recycler 不会在这个类中做缓存 View 的操作，是否缓存 View 完全由开发者控制。

```
public abstract static class ViewCacheExtension {
    abstract public View getViewForPositionAndType(Recycler recycler, int position, int type);
}
```

- 现在大家熟悉了 RecyclerViewPool 和 ViewCacheExtension 的作用后，下面开始介绍 Recycler。
- 如下是 Recycler 的几个关键成员变量和方法：

```
private ArrayList<ViewHolder> mAttachedScrap;
private ArrayList<ViewHolder> mChangedScrap; // 与 RecyclerView 分离的 ViewHolder 列表。
private ArrayList<ViewHolder> mCachedViews; // ViewHolder 缓存列表。
private ViewCacheExtension mViewCacheExtension; // 开发者控制的 ViewHolder 缓存
private RecycledViewPool mRecyclerPool; // 提供复用 ViewHolder 池。
public void bindViewToPosition(View view, int position); // 将某个 View 绑定到 Adapter 的某个位置。
public View getViewForPosition(int position);
```

- 获取某个位置需要展示的 View，先检查是否有可复用的 View，没有则创建新 View 并返回。具体过程为：
  - 检查 mChangedScrap，若匹配到则返回相应 holder
  - 检查 mAttachedScrap，若匹配到且 holder 有效则返回相应 holder
  - 检查 mViewCacheExtension，若匹配到则返回相应 holder
  - 检查 mRecyclerPool，若匹配到则返回相应 holder
  - 否则执行 Adapter.createViewHolder()，新建 holder 实例
  - 返回 holder.itemView
- 注：以上每步匹配过程都可以匹配 position 或 itemId（如果有 stableId）。

5.4 RecyclerView 的缓存机制

- https://www.jianshu.com/p/3e9aa4bdaefd 希望能更深入地理解一下缓存机制
- ListView 的缓存机制：
  - listView 是继承于 AbsListView ，RecycleBin 是 AbsListView 的内部类，其作用是通过两级缓存来缓存 view。
  - 一级缓存：mActiveViews
    - 第一级缓存，这些 View 是布局过程开始时屏幕上的 view，layout 开始时这个数组被填充，layout 结束，mActiveViews 中的 View 移动到 mScrapView，意义在于快速重用屏幕上可见的列表项 ItemView，而不需要重新 createView 和 bindView。
  - 二级缓存：mScrapView

    * 第二级缓存，mScrapView 是多个 List 组成的数据，数组的长度为 viewTypeCount，每个 List 缓存不同类型 Item 布局的 View，其意义在于缓存离开屏幕的 ItemView，目的是让即将进入屏幕的 itemView 重用，当 mAdapter 被更换时，mScrapViews 则被清空。

- RecyclerView 也有一个类专门来管理缓存，不过与 ListView 不同的是，RecylerView 缓存的是 ViewHolder，而且实现的是四级缓存。
  - 一级：Scrap
    * 对应 ListView 的一级缓存，快速重用屏幕上可见的 ViewHolder。
    * 简而言之，屏幕内显示的。
  - 二级：Cached
    * 对应 ListView 的二级缓存，如果仍依赖于 RecyclerView(比如已经滑出可视范围，但还没有被移除掉)，但已经被标记移除的 ItemView 集合被添加到 mAttachedScrap 中。然后如果 mAttachedScrap 中不再依赖时会被加入到 mCachedViews 中，默认缓存 2 个 ItemView，RecycleView 从这里获取的缓存时，如果数据源不变的情况下，无需重新 bindView。
    * 简而言之，linearlayoutmanager 来说 cached 缓存默认大小为 2，起到的作用就是 rv 滑动时刚被移出屏幕的 viewholer 的收容所。
  - 三级：CacheExtension
    * 第三级缓存，其实是一个抽象静态类，用于充当附加的缓存池，当 RecyclerView 从 mCacheViews 找不到需要的 View 时，将会从 ViewCacheExtension 中寻找。不过这个缓存是由开发者维护的，如果没有设置它，则不会启用。通常我们也不会设置它，除非有特殊需求，比如要在调用系统的缓存池之前，返回一个特定的视图，才会用到它。
    * 简而言之，这是一个自定义的缓存，没错 rv 是可以自定义缓存行为的。
    * 目前来说这还只是个空实现而已，从这点来看其实 rv 所说的四级缓存本质上还只是三级缓存。
  - 四级：RecycledViewPool（最强大）
    * 它是一个缓存池，实现上，是通过一个默认为 5 大小的 ArrayList 实现的。这一点，同 ListView 的 RecyclerBin 这个类一样。每一个 ArrayList 又都是放在一个 Map 里面的，SparseArray 用两个数组用来替代 Map。
    * 把所有的 ArrayList 放在一个 Map 里面，这也是 RecyclerView 最大的亮点，这样根据 itemType 来取不同的缓存 Holder，每一个 Holder 都有对应的缓存，而只需要为这些不同 RecyclerView 设置同一个 Pool 就可以了。
    * 简而言之，pool 一般会和 cached 配合使用，这么来说，cached 存不下的会被保存到 pool 中毕竟 cached 默认容量大小只有 2，但是 pool 容量也是有限的当保存满之后再有 viewholder 到来的话就只能会无情抛弃掉，它也有一个默认的容量大小

```
privatestaticfinalintDEFAULT_MAX_SCRAP = 5;
```

- Scrap: 在屏幕内可视的 Item。
- Cache: 在屏幕外的 Item
- ViewCacheExtension : 用户自定义的缓存策略
- RecycledViewPool : 被废弃的 itemview, 脏数据, 需要重新 onBindViewHolder.

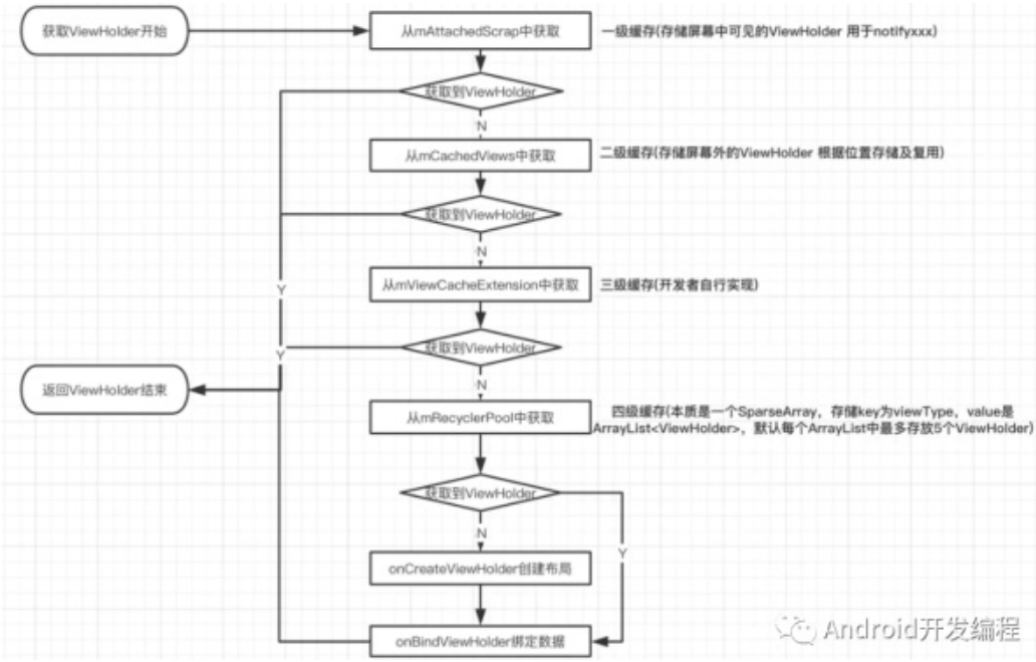
- Recycler 缓存 ViewHolder 对象有 4 个等级，优先级从高到底依次为：
  - 1、ArrayList mAttachedScrap — 缓存屏幕中可见范围的 ViewHolder
  - 2、ArrayList mCachedViews —- 缓存滑动时即将与 RecyclerView 分离的 ViewHolder，按子 View 的 position 或 id 缓存，默认最多存放 2 个
  - 3、ViewCacheExtension mViewCacheExtension — 开发者自行实现的缓存
  - 4、RecycledViewPool mRecyclerPool — ViewHolder 缓存池，本质上是一个 SparseArray，其中 key 是 ViewType(int 类型)，value 存放的是 ArrayList< ViewHolder>，默认每个 ArrayList 中最多存放 5 个 ViewHolder。
- RecyclerView 滑动时会触发 onTouchEvent#onMove，回收及复用 ViewHolder 在这里就会开始。我们知道设置 RecyclerView 时需要设置 LayoutManager,LayoutManager 负责 RecyclerView 的布局，包含对 ItemView 的获取与复用。以 LinearLayoutManager 为例，当 RecyclerView 重新布局时会依次执行下面几个方法：
  - onLayoutChildren()：对 RecyclerView 进行布局的入口方法
  - fill(): 负责对剩余空间不断地填充，调用的方法是 layoutChunk()
  - layoutChunk()：负责填充 View, 该 View 最终是通过在缓存类 Recycler 中找到合适的 View 的
- 上述的整个调用链:onLayoutChildren()->fill()->layoutChunk()->next()->getViewForPosition(),getV 即是是从 RecyclerView 的回收机制实现类 Recycler 中获取合适的 View，
- 下面主要就来从看这个 Recycler#getViewForPosition() 的实现。

```
@NonNull
public View getViewForPosition(int position) {
    return getViewForPosition(position, false);
}
View getViewForPosition(int position, boolean dryRun) {
    return tryGetViewHolderForPositionByDeadline(position, dryRun, FOREVER_NS).itemView;
}
```

- 他们都会执行 tryGetViewHolderForPositionByDeadline 函数，继续跟进去：

```
ViewHolder tryGetViewHolderForPositionByDeadline(int position, boolean dryRun, long deadlineNs) {
    // ... 省略
    boolean fromScrapOrHiddenOrCache = false;
    ViewHolder holder = null;
    // 预布局 属于特殊情况 从 mChangedScrap 中获取 ViewHolder
    if (mState.isPreLayout()) {
        holder = getChangedScrapViewForPosition(position);
        fromScrapOrHiddenOrCache = holder != null;
    }
    if (holder == null) {
        // 1、尝试从 mAttachedScrap 中获取 ViewHolder, 此时获取的是屏幕中可见范围中的 ViewHolder
        // 2、mAttachedScrap 缓存中没有的话，继续从 mCachedViews 尝试获取 ViewHolder
        holder = getScrapOrHiddenOrCachedHolderForPosition(position, dryRun);
        //  ... 省略
    }
    if (holder == null) {
        final int offsetPosition = mAdapterHelper.findPositionOffset(position);
        //  ... 省略
        final int type = mAdapter.getItemViewType(offsetPosition);
        // 如果 Adapter 中声明了 Id，尝试从 id 中获取，这里不属于缓存
        if (mAdapter.hasStableIds()) {
            holder = getScrapOrCachedViewForId(mAdapter.getItemId(offsetPosition), type, dryRun);
        }
        if (holder == null && mViewCacheExtension != null) {
            //  3、从自定义缓存 mViewCacheExtension 中尝试获取 ViewHolder，该缓存需要开发者实现
            final View view = mViewCacheExtension.getViewForPositionAndType(this, position, type);
```

```
            if (view != null)
                holder = getChildViewHolder(view);
        }
        if (holder == null) { //  fallback to pool
            // 4、从缓存池 mRecyclerPool 中尝试获取 ViewHolder
            holder = getRecycledViewPool().getRecycledView(type);
            if (holder != null) {
                // 如果获取成功，会重置 ViewHolder 状态，所以需要重新执行 Adapter#onBindViewHolder 绑定数据
                holder.resetInternal();
                if (FORCE_INVALIDATE_DISPLAY_LIST)
                    invalidateDisplayListInt(holder);
            }
        }
        if (holder == null) {
            // ... 省略
            // 5、若以上缓存中都没有找到对应的 ViewHolder，最终会调用 Adapter 中的 onCreateViewHolder 创建一个
            holder = mAdapter.createViewHolder(RecyclerView.this, type);
        }
    }
    boolean bound = false;
    if (mState.isPreLayout() && holder.isBound()) {
        holder.mPreLayoutPosition = position;
    } else if (!holder.isBound() || holder.needsUpdate() || holder.isInvalid()) {
        final int offsetPosition = mAdapterHelper.findPositionOffset(position);
        // 6、如果需要绑定数据，会调用 Adapter#onBindViewHolder 来绑定数据
        bound = tryBindViewHolderByDeadline(holder, offsetPosition, position, deadlineNs);
    }
    // ... 省略
    return holder;
}
```



- 通过 mAttachedScrap、mCachedViews 及 mViewCacheExtension 获取的 ViewHolder 不需要重新创建布局及绑定数据; 通过缓存池 mRecyclerPool 获取的 ViewHolder 不需要重新创建布局，但是需要重新绑定数据; 如果上述缓存中都没有获取到目标 ViewHolder，那么就会回调 Adapter#onCreateViewHolder 创建布局，以及回调 Adapter#onBindViewHolder 来绑定数据
- RecyclerView 滑动场景下的回收复用涉及到的结构体两个：
  - mCachedViews 和 RecyclerViewPool

- mCachedViews 优先级高于 RecyclerViewPool，回收时，最新的 ViewHolder 都是往 mCachedViews 里放，如果它满了，那就移出一个扔到 ViewPool 里好空出位置来缓存最新的 ViewHolder。
- 复用时，也是先到 mCachedViews 里找 ViewHolder，但需要各种匹配条件，概括一下就是只有原来位置的卡位可以复用存在 mCachedViews 里的 ViewHolder，如果 mCachedViews 里没有，那么才去 ViewPool 里找。
- 在 ViewPool 里的 ViewHolder 都是跟全新的 ViewHolder 一样，只要 type 一样，有找到，就可以拿出来复用，重新绑定下数据即可。

5.5 你可能不知道的 **RecyclerView** 性能优化策略

- 1. 不要在 onBindViewHolder 中设置监听器，在 onCreateViewHolder 中设置监听器.
- 2. 做如下设置

```
LinearLayoutManager.setInitialPrefetchitemCount();
```

- 用户滑动到横向滑动的 item RecyclerView 的时候，由于需要创建更复杂的 RecyclerView 以及多个子 view, 可能会导致页面卡顿
- 由于 RenderThread 的存在，RecyclerView 会进行 prefetch
- LinearLayoutManager.setInitialPrefetchItemCount(横向列表初次显示可见的 item 个数)
  - 只有 LinearLayoutManager 有这个 API
  - 只有潜逃在内部的 RecyclerView 才会生效.
- 3. 当数据容量固定的时候，设置

```
RecyclerView.setHasFixedSize(true);
// 伪代码
void onContentsChanged() {
    if (mHasFixedSize)
        layoutChildren();
    else requestLayout();
}
```

- 什么时候用？
  - 如果 Adapter 的数据变化不会导致 RecyclerView 的大小变化就可以用
- 4. 多个 RectclerView 共用 RecycledViewPool.
- 5. 使用 DiffUtil