RecyclerView 必知必会

导语

RecyclerView 是 Android 5.0 提出的新 UI 控件，可以用来代替传统的 ListView。

Bugly 之前也发过一篇相关文章，讲解了 RecyclerView 与 ListView 在缓存机制上的一些区别：
Android ListView 与 RecyclerView 对比浅析—缓存机制
https://zhuanlan.zhihu.com/p/23339185

今天精神哥来给大家详细介绍关于 RecyclerView，你需要了解的方方面面。

本文来自腾讯 天天 P 图团队——damonxia(夏正冬)，Android 工程师

前言

下文中 Demo 的源代码地址：RecyclerViewDemo。
https://github.com/xiazdong/RecyclerViewDemo

- Demo1: RecyclerView 添加 HeaderView 和 FooterView，ItemDecoration 范例。
- Demo2: ListView 实现局部刷新。
- Demo3: RecyclerView 实现拖拽、侧滑删除。
- Demo4: RecyclerView 闪屏问题。
- Demo5: RecyclerView 实现 setEmptyView()。
- Demo6: RecyclerView 实现万能适配器，瀑布流布局，嵌套滑动机制。

基本概念

RecyclerView 是 Android 5.0 提出的新 UI 控件，位于 support-v7 包中，可以通过在 build.gradle 中添加 compile 'com.android.support:recyclerview-v7:24.2.1'导入。

RecyclerView 的官方定义如下：

A flexible view for providing a limited window into a large data set.

从定义可以看出，flexible（可扩展性）是 RecyclerView 的特点。不过我们发现和 ListView 有点像，本文后面会介绍 RecyclerView 和 ListView 的区别。

为什么会出现 RecyclerView?

RecyclerView 并不会完全替代 ListView（这点从 ListView 没有被标记为@Deprecated 可以看出），两者的使用场景不一样。但是 RecyclerView 的出现会让很多开源项目被废弃，例如横

向滚动的 ListView，横向滚动的 GridView，瀑布流控件，因为 RecyclerView 能够实现所有这些功能。

比如有一个需求是屏幕竖着的时候的显示形式是 ListView，屏幕横着的时候的显示形式是 2 列的 GridView，此时如果用 RecyclerView，则通过设置 LayoutManager 一行代码实现替换。

ListView vs RecyclerView

ListView 相比 RecyclerView，有一些优点：

- addHeaderView(), addFooterView()添加头视图和尾视图。
- 通过"android:divider"设置自定义分割线。
- setOnItemClickListener()和 setOnItemLongClickListener()设置点击事件和长按事件。

这些功能在 RecyclerView 中都没有直接的接口，要自己实现（虽然实现起来很简单），因此如果只是实现简单的显示功能，ListView 无疑更简单。

RecyclerView 相比 ListView，有一些明显的优点：

- 默认已经实现了 View 的复用，不需要类似 if(convertView == null)的实现，而且回收机制更加完善。
- 默认支持局部刷新。
- 容易实现添加 item、删除 item 的动画效果。
- 容易实现拖拽、侧滑删除等功能。

RecyclerView 是一个插件式的实现，对各个功能进行解耦，从而扩展性比较好。

ListView 实现局部刷新

我们都知道 ListView 通过 adapter.notifyDataSetChanged()实现 ListView 的更新，这种更新方法的缺点是全局更新，即对每个 Item View 都进行重绘。但事实上很多时候，我们只是更新了其中一个 Item 的数据，其他 Item 其实可以不需要重绘。

这里给出 ListView 实现局部更新的方法：

```
public void updateItemView(ListView listview, int position, Data data){
    int firstPos = listview.getFirstVisiblePosition();
    int lastPos = listview.getLastVisiblePosition();
    if(position >= firstPos && position <= lastPos){
        // 可见才更新，不可见则在 getView()时更新
        // listview.getChildAt(i)获得的是当前可见的第 i 个 item 的 view
        View view = listview.getChildAt(position - firstPos);
        VH vh = (VH)view.getTag();
        vh.text.setText(data.text);
    }
}
```

可以看出，我们通过 ListView 的 getChildAt()来获得需要更新的 View，然后通过 getTag()获得 ViewHolder，从而实现更新。

标准用法

RecyclerView 的标准实现步骤如下：

- 创建 Adapter：创建一个继承 RecyclerView.Adapter<VH>的 Adapter 类（VH 是 ViewHolder 的类名），记为 NormalAdapter。
- 创建 ViewHolder：在 NormalAdapter 中创建一个继承 RecyclerView.ViewHolder 的静态内部类，记为 VH。ViewHolder 的实现和 ListView 的 ViewHolder 实现几乎一样。
- 在 NormalAdapter 中实现：
- VH onCreateViewHolder(ViewGroup parent, int viewType): 映射 Item Layout Id，创建 VH 并返回。
- void onBindViewHolder(VH holder, int position): 为 holder 设置指定数据。
- int getItemCount(): 返回 Item 的个数。

可以看出，RecyclerView 将 ListView 中 getView()的功能拆分成了 onCreateViewHolder()和 onBindViewHolder()。

基本的 Adapter 实现如下：

```
public class NormalAdapter extends RecyclerView.Adapter<NormalAdapter.VH>{
    private List<String> mDatas;
    public NormalAdapter(List<String> data) {
        this.mDatas = data;
    }

    @Override
    public void onBindViewHolder(VH holder, int position) {
        holder.title.setText(mDatas.get(position));
        holder.itemView.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //item 点击事件
            }
        });
    }

    @Override
    public int getItemCount() {
        return mDatas.size();
    }

    @Override
    public VH onCreateViewHolder(ViewGroup parent, int viewType) {
        View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_1, parent,
```

```
false);
            return new VH(v);
        }

        public static class VH extends RecyclerView.ViewHolder{
            public final TextView title;
            public VH(View v) {
                super(v);
                title = (TextView) v.findViewById(R.id.title);
            }
        }
 }
```

创建完 Adapter，接着对 RecyclerView 进行设置，一般来说，需要为 RecyclerView 进行四大设置，也就是后文说的四大组成：Adapter(必选),Layout Manager(必选),Item Decoration(可选，默认为空), Item Animator(可选，默认为 DefaultItemAnimator)。

需要注意的是在 onCreateViewHolder()中，映射 Layout 必须为
View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_1, parent, false);
而不能是：
View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_1, null);

如果要实现 ListView 的效果，只需要设置 Adapter 和 Layout Manager，如下：

```
List<String> data = initData();
RecyclerView rv = (RecyclerView) findViewById(R.id.rv);
rv.setLayoutManager(new LinearLayoutManager(this));
rv.setAdapter(new NormalAdapter(data));
```

ListView 只提供了 notifyDataSetChanged()更新整个视图，这是很不合理的。RecyclerView 提供了 notifyItemInserted(),notifyItemRemoved(),notifyItemChanged()等 API 更新单个或某个范围的 Item 视图。

四大组成

RecyclerView 的四大组成是：

- Adapter：为 Item 提供数据。
- Layout Manager：Item 的布局。
- Item Animator：添加、删除 Item 动画。
- Item Decoration：Item 之间的 Divider。

Adapter

Adapter 的使用方式前面已经介绍了，功能就是为 RecyclerView 提供数据，这里主要介绍万能适配器的实现。其实万能适配器的概念在 ListView 就已经存在了，即 base-adapter-helper。
https://github.com/JoanZapata/base-adapter-helper

这里我们只针对 RecyclerView，聊聊万能适配器出现的原因。为了创建一个 RecyclerView 的 Adapter，每次我们都需要去做重复劳动，包括重写 onCreateViewHolder(),getItemCount()、创建 ViewHolder，并且实现过程大同小异，因此万能适配器出现了，他能通过以下方式快捷地创建一个 Adapter：

```
mAdapter = new QuickAdapter<String>(data) {
        @Override
        public int getLayoutId(int viewType) {
            return R.layout.item;
        }

        @Override
        public void convert(VH holder, String data, int position) {
            holder.setText(R.id.text, data);
            // holder.itemView.setOnClickListener(); 此处还可以添加点击事件
        }
 };
```

是不是很方便。当然复杂情况也可以轻松解决。

```
 mAdapter = new QuickAdapter<Model>(data) {
        @Override
        public int getLayoutId(int viewType) {
            switch(viewType){
                case TYPE_1:
                    return R.layout.item_1;
                case TYPE_2:
                    return R.layout.item_2;
            }
        }

        public int getItemViewType(int position) {
            if(position % 2 == 0){
                return TYPE_1;
            } else{
                return TYPE_2;
            }
        }

        @Override
        public void convert(VH holder, Model data, int position) {
            int type = getItemViewType(position);
            switch(type){
                case TYPE_1:
                    holder.setText(R.id.text, data.text);
                    break;
```

```
                case TYPE_2:
                    holder.setImage(R.id.image, data.image);
                    break;
            }
        }
    };
```

这里讲解下万能适配器的实现思路。

我们通过

public abstract class QuickAdapter<T> extends RecyclerView.Adapter<QuickAdapter.VH>定义万能适配器 QuickAdapter 类，T 是列表数据中每个元素的类型，QuickAdapter.VH 是 QuickAdapter 的 ViewHolder 实现类，称为万能 ViewHolder。

首先介绍 QuickAdapter.VH 的实现：

```
static class VH extends RecyclerView.ViewHolder{
        private SparseArray<View> mViews;
        private View mConvertView;

        private VH(View v){
            super(v);
            mConvertView = v;
            mViews = new SparseArray<>();
        }

        public static VH get(ViewGroup parent, int layoutId){
            View convertView = LayoutInflater.from(parent.getContext()).inflate(layoutId, parent, false);
            return new VH(convertView);
        }

        public <T extends View> T getView(int id){
            View v = mViews.get(id);
            if(v == null){
                v = mConvertView.findViewById(id);
                mViews.put(id, v);
            }
            return (T)v;
        }

        public void setText(int id, String value){
            TextView view = getView(id);
            view.setText(value);
        }
    }
```

其中的关键点在于通过 SparseArray<View>存储 item view 的控件，getView(int id)的功能就是通过 id 获得对应的 View（首先在 mViews 中查询是否存在，如果没有，那么 findViewById()并放入 mViews 中，避免下次再执行 findViewById()）。

QuickAdapter 的实现如下：

```
public abstract class QuickAdapter<T> extends RecyclerView.Adapter<QuickAdapter.VH>{

    private List<T> mDatas;

    public QuickAdapter(List<T> datas){
        this.mDatas = datas;
    }

    public abstract int getLayoutId(int viewType);

    @Override
    public VH onCreateViewHolder(ViewGroup parent, int viewType) {
        return VH.get(parent,getLayoutId(viewType));
    }

    @Override
    public void onBindViewHolder(VH holder, int position) {
        convert(holder, mDatas.get(position), position);
    }

    @Override
    public int getItemCount() {
        return mDatas.size();
    }

    public abstract void convert(VH holder, T data, int position);

    static class VH extends RecyclerView.ViewHolder{...}
 }
```

其中：

- getLayoutId(int viewType)是根据 viewType 返回布局 ID。
- convert()做具体的 bind 操作。

就这样，万能适配器实现完成了。

Item Decoration

RecyclerView 通过 addItemDecoration()方法添加 item 之间的分割线。Android 并没有提供实现好的 Divider，因此任何分割线样式都需要自己实现。

方法是：创建一个类并继承 RecyclerView.ItemDecoration，重写以下两个方法：

- onDraw(): 绘制分割线。
- getItemOffsets(): 设置分割线的宽、高。

Google 在 sample 中给了一个参考的实现类：DividerItemDecoration，这里我们通过分析这个例子来看如何自定义 Item Decoration。

首先看构造函数，构造函数中获得系统属性 android:listDivider，该属性是一个 Drawable 对象。

因此如果要设置，则需要在 value/styles.xml 中设置：

```
<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="android:listDivider">@drawable/item_divider</item>
</style>
```

接着来看 getItemOffsets()的实现：

```
public void getItemOffsets(Rect outRect, int position, RecyclerView parent) {
    if (mOrientation == VERTICAL_LIST) {
        outRect.set(0, 0, 0, mDivider.getIntrinsicHeight());
    } else {
        outRect.set(0, 0, mDivider.getIntrinsicWidth(), 0);
    }
}
```

这里只看 mOrientation == VERTICAL_LIST 的情况，outRect 是当前 item 四周的间距，类似 margin 属性，现在设置了该 item 下间距为 mDivider.getIntrinsicHeight()。

那么 getItemOffsets()是怎么被调用的呢？

RecyclerView 继承了 ViewGroup，并重写了 measureChild()，该方法在 onMeasure()中被调用，用来计算每个 child 的大小，计算每个 child 大小的时候就需要加上 getItemOffsets()设置的外间距：

```
public void measureChild(View child, int widthUsed, int heightUsed){
    // 调用 getItemOffsets()获得 Rect 对象
    final Rect insets = mRecyclerView.getItemDecorInsetsForChild(child);
    widthUsed += insets.left + insets.right;
    heightUsed += insets.top + insets.bottom;
    //...
}
```

这里我们只考虑 mOrientation == VERTICAL_LIST 的情况，DividerItemDecoration 的 onDraw()实际上调用了 drawVertical()：

```
public void drawVertical(Canvas c, RecyclerView parent) {
    final int left = parent.getPaddingLeft();
    final int right = parent.getWidth() - parent.getPaddingRight();
    final int childCount = parent.getChildCount();
    /**
     * 画每个 item 的分割线
```

```
         */
        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 +
Math.round(ViewCompat.getTranslationY(child));
            final int bottom = top + mDivider.getIntrinsicHeight();
            mDivider.setBounds(left, top, right, bottom);/*规定好左上角和右下角*/
            mDivider.draw(c);
        }
    }
```

那么 onDraw()是怎么被调用的呢？还有 ItemDecoration 还有一个方法 onDrawOver()，该方法也可以被重写，那么 onDraw()和 onDrawOver()之间有什么关系呢？

我们来看下面的代码：

```
    class RecyclerView extends ViewGroup{
        public void draw(Canvas c) {
            // 调用 View 的 draw()，该方法会先调用 onDraw()，再调用 dispatchDraw()绘制
children
            super.draw(c);

            final int count = mItemDecorations.size();
            for (int i = 0; i < count; i++) {
                mItemDecorations.get(i).onDrawOver(c, this, mState);
            }
            ...
        }
        public void onDraw(Canvas c) {
            super.onDraw(c);
            final int count = mItemDecorations.size();
            for (int i = 0; i < count; i++) {
                mItemDecorations.get(i).onDraw(c, this, mState);
            }
        }
    }
```

根据 View 的绘制流程，首先调用 RecyclerView 重写的 draw()方法，随后 super.draw()即调用 View 的 draw()，该方法会先调用 onDraw()（这个方法在 RecyclerView 重写了），再调用 dispatchDraw()绘制 children。因此：ItemDecoration 的 onDraw()在绘制 Item 之前调用，ItemDecoration 的 onDrawOver()在绘制 Item 之后调用。

当然，如果只需要实现 Item 之间相隔一定距离，那么只需要为 Item 的布局设置 margin 即可，没必要自己实现 ItemDecoration 这么麻烦。

Layout Manager

LayoutManager 负责 RecyclerView 的布局，其中包含了 Item View 的获取与回收。这里我们简单分析 LinearLayoutManager 的实现。

对于 LinearLayoutManager 来说，比较重要的几个方法有：

- onLayoutChildren(): 对 RecyclerView 进行布局的入口方法。
- fill(): 负责填充 RecyclerView。
- scrollVerticallyBy():根据手指的移动滑动一定距离，并调用 fill()填充。
- canScrollVertically()或 canScrollHorizontally(): 判断是否支持纵向滑动或横向滑动。

onLayoutChildren()的核心实现如下：

```
public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {
    detachAndScrapAttachedViews(recycler); //将原来所有的 Item View 全部放到 Recycler 的 Scrap Heap 或 Recycle Pool
    fill(recycler, mLayoutState, state, false); //填充现在所有的 Item View
}
```

RecyclerView 的回收机制有个重要的概念，即将回收站分为 Scrap Heap 和 Recycle Pool，其中 Scrap Heap 的元素可以被直接复用，而不需要调用 onBindViewHolder()。detachAndScrapAttachedViews()会根据情况，将原来的 Item View 放入 Scrap Heap 或 Recycle Pool，从而在复用时提升效率。

fill()是对剩余空间不断地调用 layoutChunk()，直到填充完为止。layoutChunk()的核心实现如下：

```
public void layoutChunk() {
    View view = layoutState.next(recycler); //调用了 getViewForPosition()
    addView(view);   //加入 View
    measureChildWithMargins(view, 0, 0); //计算 View 的大小
    layoutDecoratedWithMargins(view, left, top, right, bottom); //布局 View
}
```

其中 next()调用了 getViewForPosition(currentPosition)，该方法是从 RecyclerView 的回收机制实现类 Recycler 中获取合适的 View，在后文的回收机制中会介绍该方法的具体实现。

如果要自定义 LayoutManager，可以参考：

- 创建一个 RecyclerView LayoutManager – Part 1
- 创建一个 RecyclerView LayoutManager – Part 2
- 创建一个 RecyclerView LayoutManager – Part 3

Item Animator

RecyclerView 能够通过 mRecyclerView.setItemAnimator(ItemAnimator animator)设置添加、删除、移动、改变的动画效果。RecyclerView 提供了默认的 ItemAnimator 实现类：

DefaultItemAnimator。这里我们通过分析 DefaultItemAnimator 的源码来介绍如何自定义

Item Animator。

DefaultItemAnimator 继承自 SimpleItemAnimator，SimpleItemAnimator 继承自 ItemAnimator。

首先我们介绍 ItemAnimator 类的几个重要方法：

- animateAppearance()：当 ViewHolder 出现在屏幕上时被调用（可能是 add 或 move）。
- animateDisappearance()：当 ViewHolder 消失在屏幕上时被调用（可能是 remove 或 move）。
- animatePersistence()：在没调用 notifyItemChanged()和 notifyDataSetChanged()的情况下布局发生改变时被调用。
- animateChange()：在显式调用 notifyItemChanged()或 notifyDataSetChanged()时被调用。
- runPendingAnimations()：RecyclerView 动画的执行方式并不是立即执行，而是每帧执行一次，比如两帧之间添加了多个 Item，则会将这些将要执行的动画 Pending 住，保存在成员变量中，等到下一帧一起执行。该方法执行的前提是前面 animateXxx()返回 true。
- isRunning()：是否有动画要执行或正在执行。
- dispatchAnimationsFinished()：当全部动画执行完毕时被调用。

上面用斜体字标识的方法比较难懂，不过没关系，因为 Android 提供了 SimpleItemAnimator 类（继承自 ItemAnimator），该类提供了一系列更易懂的 API，在自定义 Item Animator 时只需要继承 SimpleItemAnimator 即可：

- animateAdd(ViewHolder holder)：当 Item 添加时被调用。
- animateMove(ViewHolder holder, int fromX, int fromY, int toX, int toY)：当 Item 移动时被调用。
- animateRemove(ViewHolder holder)：当 Item 删除时被调用。
- animateChange(ViewHolder oldHolder, ViewHolder newHolder, int fromLeft, int fromTop, int toLeft, int toTop)：当显式调用 notifyItemChanged()或 notifyDataSetChanged()时被调用。

对于以上四个方法，注意两点：

- 当 Xxx 动画开始执行前（在 runPendingAnimations()中）需要调用 dispatchXxxStarting(holder)，执行完后需要调用 dispatchXxxFinished(holder)。
- 这些方法的内部实际上并不是书写执行动画的代码，而是将需要执行动画的 Item 全部存入成员变量中，并且返回值为 true，然后在 runPendingAnimations()中一并执行。

DefaultItemAnimator 类是 RecyclerView 提供的默认动画类。我们通过阅读该类源码学习如何自定义 Item Animator。我们先看 DefaultItemAnimator 的成员变量：

```
private ArrayList<ViewHolder> mPendingAdditions = new ArrayList<>();//存放下一帧要执行的一系列 add 动画
ArrayList<ArrayList<ViewHolder>> mAdditionsList = new ArrayList<>();//存放正在执行的一批 add 动画
ArrayList<ViewHolder> mAddAnimations = new ArrayList<>(); //存放当前正在执行的 add 动画
```

```
private ArrayList<ViewHolder> mPendingRemovals = new ArrayList<>();
ArrayList<ViewHolder> mRemoveAnimations = new ArrayList<>();

private ArrayList<MoveInfo> mPendingMoves = new ArrayList<>();
ArrayList<ArrayList<MoveInfo>> mMovesList = new ArrayList<>();
ArrayList<ViewHolder> mMoveAnimations = new ArrayList<>();

private ArrayList<ChangeInfo> mPendingChanges = new ArrayList<>();
ArrayList<ArrayList<ChangeInfo>> mChangesList = new ArrayList<>();
ArrayList<ViewHolder> mChangeAnimations = new ArrayList<>();
```

DefaultItemAnimator 实现了 SimpleItemAnimator 的 animateAdd()方法，该方法只是将该 item 添加到 mPendingAdditions 中，等到 runPendingAnimations()中执行。

```
public boolean animateAdd(final ViewHolder holder) {
    resetAnimation(holder);   //重置清空所有动画
    ViewCompat.setAlpha(holder.itemView, 0); //将要做动画的 View 先变成透明
    mPendingAdditions.add(holder);
    return true;
}
```

接着看 runPendingAnimations()的实现，该方法是执行 remove,move,change,add 动画，执行顺序为：remove 动画最先执行，随后 move 和 change 并行执行，最后是 add 动画。为了简化，我们将 remove,move,change 动画执行过程省略，只看执行 add 动画的过程，如下：

```
public void runPendingAnimations() {
    //1、判断是否有动画要执行，即各个动画的成员变量里是否有值。
    //2、执行 remove 动画
    //3、执行 move 动画
    //4、执行 change 动画，与 move 动画并行执行
    //5、执行 add 动画
    if (additionsPending) {
        final ArrayList<ViewHolder> additions = new ArrayList<>();
        additions.addAll(mPendingAdditions);
        mAdditionsList.add(additions);
        mPendingAdditions.clear();
        Runnable adder = new Runnable() {
            @Override
            public void run() {
                for (ViewHolder holder : additions) {
                    animateAddImpl(holder);   //***** 执行动画的方法 *****
                }
```

```
                additions.clear();
                mAdditionsList.remove(additions);
            }
        };
        if (removalsPending || movesPending || changesPending) {
            long removeDuration = removalsPending ? getRemoveDuration() : 0;
            long moveDuration = movesPending ? getMoveDuration() : 0;
            long changeDuration = changesPending ? getChangeDuration() : 0;
            long totalDelay = removeDuration + Math.max(moveDuration,
changeDuration);
            View view = additions.get(0).itemView;
            ViewCompat.postOnAnimationDelayed(view, adder, totalDelay); //等 remove，
move，change 动画全部做完后，开始执行 add 动画
        }
    }
}
```

为了防止在执行 add 动画时外面有新的 add 动画添加到 mPendingAdditions 中，从而导致执行 add 动画错乱，这里将 mPendingAdditions 的内容移动到局部变量 additions 中，然后遍历 additions 执行动画。

在 runPendingAnimations()中，animateAddImpl()是执行 add 动画的具体方法，其实就是将 itemView 的透明度从 0 变到 1（在 animateAdd()中已经将 view 的透明度变为 0），实现如下：

```
void animateAddImpl(final ViewHolder holder) {
    final View view = holder.itemView;
    final ViewPropertyAnimatorCompat animation = ViewCompat.animate(view);
    mAddAnimations.add(holder);
    animation.alpha(1).setDuration(getAddDuration()).
            setListener(new VpaListenerAdapter() {
                @Override
                public void onAnimationStart(View view) {
                    dispatchAddStarting(holder);  //在开始 add 动画前调用
                }
                @Override
                public void onAnimationCancel(View view) {
                    ViewCompat.setAlpha(view, 1);
                }

                @Override
                public void onAnimationEnd(View view) {
                    animation.setListener(null);
                    dispatchAddFinished(holder); //在结束 add 动画后调用
```

```
                    mAddAnimations.remove(holder);
                    if (!isRunning()) {
                        dispatchAnimationsFinished(); //结束所有动画后调用
                    }
                }
            }).start();
    }
```

从 DefaultItemAnimator 类的实现来看，发现自定义 Item Animator 好麻烦，需要继承 SimpleItemAnimator 类，然后实现一堆方法。别急，recyclerview-animators 解救你，原因如下：
https://github.com/wasabeef/recyclerview-animators

首先，recyclerview-animators 提供了一系列的 Animator，
比如 FadeInAnimator,ScaleInAnimator。其次，如果该库中没有你满意的动画，该库提供了 BaseItemAnimator 类，该类继承自 SimpleItemAnimator，进一步封装了自定义 Item Animator 的代码，使得自定义 Item Animator 更方便，你只需要关注动画本身。如果要实现 DefaultItemAnimator 的代码，只需要以下实现：

```
public class DefaultItemAnimator extends BaseItemAnimator {

    public DefaultItemAnimator() {
    }

    public DefaultItemAnimator(Interpolator interpolator) {
        mInterpolator = interpolator;
    }

    @Override protected void animateRemoveImpl(final RecyclerView.ViewHolder holder) {
        ViewCompat.animate(holder.itemView)
                .alpha(0)
                .setDuration(getRemoveDuration())
                .setListener(new DefaultRemoveVpaListener(holder))
                .setStartDelay(getRemoveDelay(holder))
                .start();
    }

    @Override protected void preAnimateAddImpl(RecyclerView.ViewHolder holder) {
        ViewCompat.setAlpha(holder.itemView, 0); //透明度先变为 0
    }

    @Override protected void animateAddImpl(final RecyclerView.ViewHolder holder) {
        ViewCompat.animate(holder.itemView)
                .alpha(1)
```

```
                .setDuration(getAddDuration())
                .setListener(new DefaultAddVpaListener(holder))
                .setStartDelay(getAddDelay(holder))
                .start();
    }
}
```

是不是比继承 SimpleItemAnimator 方便多了。

对于 RecyclerView 的 Item Animator，有一个常见的坑就是"闪屏问题"。这个问题的描述是：当 Item 视图中有图片和文字，当更新文字并调用 notifyItemChanged()时，文字改变的同时图片会闪一下。这个问题的原因是当调用 notifyItemChanged()时，会调用 DefaultItemAnimator 的 animateChangeImpl()执行 change 动画，该动画会使得 Item 的透明度从 0 变为 1，从而造成闪屏。

解决办法很简单，在 rv.setAdapter()之前
调用((SimpleItemAnimator)rv.getItemAnimator()).setSupportsChangeAnimations(false)
禁用 change 动画。

拓展 RecyclerView

添加 setOnItemClickListener 接口

RecyclerView 默认没有像 ListView 一样提供 setOnItemClickListener()接口，而 RecyclerView 无法添加 onItemClickListener 最佳的高效解决方案这篇文章给出了通过 recyclerView.addOnItemTouchListener(...)添加点击事件的方法，但我认为根本没有必要费这么大劲对外暴露这个接口，因为我们完全可以把点击事件的实现写在 Adapter 的 onBindViewHolder()中，不暴露出来。具体方法就是通过：
https://blog.csdn.net/liaoinstan/article/details/51200600

```
public void onBindViewHolder(VH holder, int position) {
    holder.itemView.setOnClickListener(...);
}
```

添加 HeaderView 和 FooterView

RecyclerView 默认没有提供类似 addHeaderView()和 addFooterView()的 API，因此这里介绍如何优雅地实现这两个接口。

如果你已经实现了一个 Adapter，现在想为这个 Adapter 添加 addHeaderView()和 addFooterView()接口，则需要在 Adapter 中添加几个 Item Type，然后修改 getItemViewType(),onCreateViewHolder(),onBindViewHolder(),getItemCount()等方法，并添加 switch 语句进行判断。那么如何在不破坏原有 Adapter 实现的情况下完成呢？

这里引入装饰器（Decorator）设计模式，该设计模式通过组合的方式，在不破话原有类代码的情况下，对原有类的功能进行扩展。

这恰恰满足了我们的需求。我们只需要通过以下方式为原有的 Adapter（这里命名为 NormalAdapter）添加 addHeaderView()和 addFooterView()接口：

```
NormalAdapter adapter = new NormalAdapter(data);
NormalAdapterWrapper newAdapter = new NormalAdapterWrapper(adapter);
View headerView = LayoutInflater.from(this).inflate(R.layout.item_header, mRecyclerView, false);
View footerView = LayoutInflater.from(this).inflate(R.layout.item_footer, mRecyclerView, false);
newAdapter.addFooterView(footerView);
newAdapter.addHeaderView(headerView);
mRecyclerView.setAdapter(newAdapter);
```

是不是看起来特别优雅。具体实现思路其实很简单，创建一个继承 RecyclerView.Adapter<RecyclerView.ViewHolder>的类，并重写常见的方法，然后通过引入 ITEM TYPE 的方式实现：

```
public class NormalAdapterWrapper
extends RecyclerView.Adapter<RecyclerView.ViewHolder>{

    enum ITEM_TYPE{
        HEADER,
        FOOTER,
        NORMAL
    }

    private NormalAdapter mAdapter;
    private View mHeaderView;
    private View mFooterView;

    public NormalAdapterWrapper(NormalAdapter adapter){
        mAdapter = adapter;
    }

    @Override
    public int getItemViewType(int position) {
        if(position == 0){
            return ITEM_TYPE.HEADER.ordinal();
        } else if(position == mAdapter.getItemCount() + 1){
            return ITEM_TYPE.FOOTER.ordinal();
        } else{
```

```
                return ITEM_TYPE.NORMAL.ordinal();
            }
        }

        @Override
        public int getItemCount() {
            return mAdapter.getItemCount() + 2;
        }

        @Override
        public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
            if(position == 0){
                return;
            } else if(position == mAdapter.getItemCount() + 1){
                return;
            } else{
                mAdapter.onBindViewHolder(((NormalAdapter.VH)holder), position - 1);
            }
        }

        @Override
        public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
{
            if(viewType == ITEM_TYPE.HEADER.ordinal()){
                return new RecyclerView.ViewHolder(mHeaderView) {};
            } else if(viewType == ITEM_TYPE.FOOTER.ordinal()){
                return new RecyclerView.ViewHolder(mFooterView) {};
            } else{
                return mAdapter.onCreateViewHolder(parent,viewType);
            }
        }

        public void addHeaderView(View view){
            this.mHeaderView = view;
        }
        public void addFooterView(View view){
            this.mFooterView = view;
        }
    }
```

添加 setEmptyView

ListView 提供了 setEmptyView()设置 Adapter 数据为空时的 View 视图。RecyclerView 虽然没提供直接的 API，但是也可以很简单地实现。

- 创建一个继承 RecyclerView 的类，记为 EmptyRecyclerView。
- 通过 getRootView().addView(emptyView)将空数据时显示的 View 添加到当前 View 的层次结构中。
- 通过 AdapterDataObserver 监听 RecyclerView 的数据变化，如果 adapter 为空，那么隐藏 RecyclerView，显示 EmptyView。

具体实现如下：

```
public class EmptyRecyclerView extends RecyclerView{

    private View mEmptyView;

    private AdapterDataObserver mObserver = new AdapterDataObserver() {
        @Override
        public void onChanged() {
            Adapter adapter = getAdapter();
            if(adapter.getItemCount() == 0){
                mEmptyView.setVisibility(VISIBLE);
                EmptyRecyclerView.this.setVisibility(GONE);
            } else{
                mEmptyView.setVisibility(GONE);
                EmptyRecyclerView.this.setVisibility(VISIBLE);
            }
        }

        public void onItemRangeChanged(int positionStart, int itemCount) {onChanged();}
        public void onItemRangeMoved(int fromPosition, int toPosition, int itemCount)
{onChanged();}
        public void onItemRangeRemoved(int positionStart, int itemCount) {onChanged();}
        public void onItemRangeInserted(int positionStart, int itemCount) {onChanged();}
        public void onItemRangeChanged(int positionStart, int itemCount, Object payload)
{onChanged();}
    };

    public EmptyRecyclerView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
    }

    public void setEmptyView(View view){
        this.mEmptyView = view;
        ((ViewGroup)this.getRootView()).addView(mEmptyView); //加入主界面布局
    }

    public void setAdapter(RecyclerView.Adapter adapter){
```

```
            super.setAdapter(adapter);
            adapter.registerAdapterDataObserver(mObserver);
            mObserver.onChanged();
        }
    }
```

拖拽、侧滑删除

Android 提供了 ItemTouchHelper 类，使得 RecyclerView 能够轻易地实现滑动和拖拽，此处我们要实现上下拖拽和侧滑删除。首先创建一个继承自 ItemTouchHelper.Callback 的类，并重写以下方法：

- getMovementFlags(): 设置支持的拖拽和滑动的方向，此处我们支持的拖拽方向为上下，滑动方向为从左到右和从右到左，内部通过 makeMovementFlags()设置。
- onMove(): 拖拽时回调。
- onSwiped(): 滑动时回调。
- onSelectedChanged(): 状态变化时回调，一共有三个状态，分别是 ACTION_STATE_IDLE(空闲状态)，ACTION_STATE_SWIPE(滑动状态)，ACTION_STATE_DRAG(拖拽状态)。此方法中可以做一些状态变化时的处理，比如拖拽的时候修改背景色。
- clearView(): 用户交互结束时回调。此方法可以做一些状态的清空，比如拖拽结束后还原背景色。
- isLongPressDragEnabled(): 是否支持长按拖拽，默认为 true。如果不想支持长按拖拽，则重写并返回 false。

具体实现如下：

```
 public class SimpleItemTouchCallback extends ItemTouchHelper.Callback {

        private NormalAdapter mAdapter;
        private List<ObjectModel> mData;
        public SimpleItemTouchCallback(NormalAdapter adapter, List<ObjectModel> data){
            mAdapter = adapter;
            mData = data;
        }

        @Override
        public int getMovementFlags(RecyclerView recyclerView, RecyclerView.ViewHolder
viewHolder) {
            int dragFlag = ItemTouchHelper.UP | ItemTouchHelper.DOWN; //s 上下拖拽
            int swipeFlag = ItemTouchHelper.START | ItemTouchHelper.END; //左->右和右->
左滑动
            return makeMovementFlags(dragFlag,swipeFlag);
        }
```

```
        @Override
        public boolean onMove(RecyclerView recyclerView, RecyclerView.ViewHolder
viewHolder, RecyclerView.ViewHolder target) {
            int from = viewHolder.getAdapterPosition();
            int to = target.getAdapterPosition();
            Collections.swap(mData, from, to);
            mAdapter.notifyItemMoved(from, to);
            return true;
        }

        @Override
        public void onSwiped(RecyclerView.ViewHolder viewHolder, int direction) {
            int pos = viewHolder.getAdapterPosition();
            mData.remove(pos);
            mAdapter.notifyItemRemoved(pos);
        }

        @Override
        public void onSelectedChanged(RecyclerView.ViewHolder viewHolder, int actionState)
{
            super.onSelectedChanged(viewHolder, actionState);
            if(actionState != ItemTouchHelper.ACTION_STATE_IDLE){
                NormalAdapter.VH holder = (NormalAdapter.VH)viewHolder;
                holder.itemView.setBackgroundColor(0xffbcbcbc); //设置拖拽和侧滑时的背
景色
            }
        }

        @Override
        public void clearView(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder)
{
            super.clearView(recyclerView, viewHolder);
            NormalAdapter.VH holder = (NormalAdapter.VH)viewHolder;
            holder.itemView.setBackgroundColor(0xffeeeeee); //背景色还原
        }
    }
```

然后通过以下代码为 RecyclerView 设置该滑动、拖拽功能：

```
    ItemTouchHelper helper = new ItemTouchHelper(new SimpleItemTouchCallback(adapter,
data));
    helper.attachToRecyclerView(recyclerview);
```

前面拖拽的触发方式只有长按，如果想支持触摸 Item 中的某个 View 实现拖拽，则核心方法

为 helper.startDrag(holder)。首先定义接口：

```
interface OnStartDragListener{
    void startDrag(RecyclerView.ViewHolder holder);
}
```

然后让 Activity 实现该接口：

```
public MainActivity extends Activity implements OnStartDragListener{
    ...
    public void startDrag(RecyclerView.ViewHolder holder) {
        mHelper.startDrag(holder);
    }
}
```

如果要对 ViewHolder 的 text 对象支持触摸拖拽，则在 Adapter 中的 onBindViewHolder()中添加：

```
holder.text.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if(event.getAction() == MotionEvent.ACTION_DOWN){
            mListener.startDrag(holder);
        }
        return false;
    }
});
```

其中 mListener 是在创建 Adapter 时将实现 OnStartDragListener 接口的 Activity 对象作为参数传进来。

回收机制

ListView 回收机制

ListView 为了保证 Item View 的复用，实现了一套回收机制，该回收机制的实现类是 RecycleBin，他实现了两级缓存：

- View[] mActiveViews: 缓存屏幕上的 View，在该缓存里的 View 不需要调用 getView()。
- ArrayList<View>[] mScrapViews;: 每个 Item Type 对应一个列表作为回收站，缓存由于滚动而消失的 View，此处的 View 如果被复用，会以参数的形式传给 getView()。

接下来我们通过源码分析 ListView 是如何与 RecycleBin 交互的。其实 ListView 和 RecyclerView 的 layout 过程大同小异，ListView 的布局函数是 layoutChildren()，实现如下：

```
 void layoutChildren(){
      //1. 如果数据被改变了，则将所有 Item View 回收至 scrapView
   // （而 RecyclerView 会根据情况放入 Scrap Heap 或 RecyclePool）；否则回收至
mActiveViews
      if (dataChanged) {
          for (int i = 0; i < childCount; i++) {
              recycleBin.addScrapView(getChildAt(i), firstPosition+i);
          }
      } else {
          recycleBin.fillActiveViews(childCount, firstPosition);
      }
      //2. 填充
      switch(){
          case LAYOUT_XXX:
              fillXxx();
              break;
          case LAYOUT_XXX:
              fillXxx();
              break;
      }
      //3. 回收多余的 activeView
      mRecycler.scrapActiveViews();
 }
```

其中 fillXxx()实现了对 Item View 进行填充，该方法内部调用了 makeAndAddView()，实现如下：

```
 View makeAndAddView(){
      if (!mDataChanged) {
          child = mRecycler.getActiveView(position);
          if (child != null) {
              return child;
          }
      }
      child = obtainView(position, mIsScrap);
      return child;
 }
```

其中，getActiveView()是从 mActiveViews 中获取合适的 View，如果获取到了，则直接返回，而不调用 obtainView()，这也印证了如果从 mActiveViews 获取到了可复用的 View，则不需要调用 getView()。

obtainView()是从 mScrapViews 中获取合适的 View，然后以参数形式传给了 getView()，实

现如下：

```
 View obtainView(int position){
        final View scrapView = mRecycler.getScrapView(position);  //从 RecycleBin 中获取复用的 View
        final View child = mAdapter.getView(position, scrapView, this);
 }
```

接下去我们介绍 getScrapView(position)的实现，该方法通过 position 得到 Item Type，然后根据 Item Type 从 mScrapViews 获取可复用的 View，如果获取不到，则返回 null，具体实现如下：

```
 class RecycleBin{
        private View[] mActiveViews;       //存储屏幕上的 View
        private ArrayList<View>[] mScrapViews;   //每个 item type 对应一个 ArrayList
        private int mViewTypeCount;                  //item type 的个数
        private ArrayList<View> mCurrentScrap;   //mScrapViews[0]

        View getScrapView(int position) {
            final int whichScrap = mAdapter.getItemViewType(position);
            if (whichScrap < 0) {
                return null;
            }
            if (mViewTypeCount == 1) {
                return retrieveFromScrap(mCurrentScrap, position);
            } else if (whichScrap < mScrapViews.length) {
                return retrieveFromScrap(mScrapViews[whichScrap], position);
            }
            return null;
        }
        private View retrieveFromScrap(ArrayList<View> scrapViews, int position){
            int size = scrapViews.size();
            if(size > 0){
                return scrapView.remove(scrapViews.size() - 1);  //从回收列表中取出最后一个元素复用
            } else{
                return null;
            }
        }
 }
```

RecyclerView 回收机制

RecyclerView 和 ListView 的回收机制非常相似，但是 ListView 是以 View 作为单位进行回收，

RecyclerView 是以 ViewHolder 作为单位进行回收。Recycler 是 RecyclerView 回收机制的实现类，他实现了四级缓存：

- mAttachedScrap: 缓存在屏幕上的 ViewHolder。
- mCachedViews: 缓存屏幕外的 ViewHolder，默认为 2 个。ListView 对于屏幕外的缓存都会调用 getView()。
- mViewCacheExtensions: 需要用户定制，默认不实现。
- mRecyclerPool: 缓存池，多个 RecyclerView 共用。

在上文 Layout Manager 中已经介绍了 RecyclerView 的 layout 过程，但是一笔带过了 getViewForPosition()，因此此处介绍该方法的实现。

```
 View getViewForPosition(int position, boolean dryRun){
      if(holder == null){
           //从 mAttachedScrap,mCachedViews 获取 ViewHolder
           holder = getScrapViewForPosition(position,INVALID,dryRun); //此处获得的 View 不需要 bind
      }
      final int type = mAdapter.getItemViewType(offsetPosition);
      if (mAdapter.hasStableIds()) { //默认为 false
           holder = getScrapViewForId(mAdapter.getItemId(offsetPosition), type, dryRun);
      }
      if(holder == null && mViewCacheExtension != null){
           final View view = mViewCacheExtension.getViewForPositionAndType(this, position, type); //从
           if(view != null){
                holder = getChildViewHolder(view);
           }
      }
      if(holder == null){
           holder = getRecycledViewPool().getRecycledView(type);
      }
      if(holder == null){   //没有缓存，则创建
           holder = mAdapter.createViewHolder(RecyclerView.this, type); // 调用 onCreateViewHolder()
      }
      if(!holder.isBound() || holder.needsUpdate() || holder.isInvalid()){
           mAdapter.bindViewHolder(holder, offsetPosition);
      }
      return holder.itemView;
 }
```

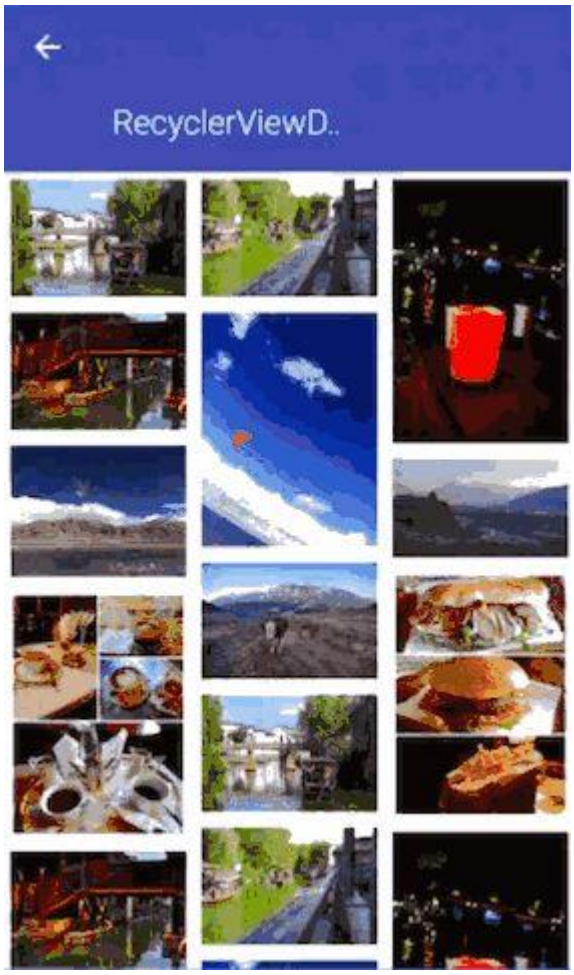
从上述实现可以看出，依次从 mAttachedScrap, mCachedViews, mViewCacheExtension, mRecyclerPool 寻找可复用的 ViewHolder，如果是从 mAttachedScrap 或 mCachedViews 中

获取的 ViewHolder，则不会调用 onBindViewHolder()，mAttachedScrap 和 mCachedViews 也就是我们所说的 Scrap Heap；而如果从 mViewCacheExtension 或 mRecyclerPool 中获取的 ViewHolder，则会调用 onBindViewHolder()。

RecyclerView 局部刷新的实现原理也是基于 RecyclerView 的回收机制，即能直接复用的 ViewHolder 就不调用 onBindViewHolder()。

嵌套滑动机制

Android 5.0 推出了嵌套滑动机制，在之前，一旦子 View 处理了触摸事件，父 View 就没有机会再处理这次的触摸事件，而嵌套滑动机制解决了这个问题，能够实现如下效果：



为了支持嵌套滑动，子 View 必须实现 NestedScrollingChild 接口，父 View 必须实现 NestedScrollingParent 接口，而 RecyclerView 实现了 NestedScrollingChild 接口，而 CoordinatorLayout 实现了 NestedScrollingParent 接口，上图是实现 CoordinatorLayout 嵌套 RecyclerView 的效果。

为了实现上图的效果，需要用到的组件有：

- CoordinatorLayout：布局根元素。

- AppBarLayout: 包裹的内容作为应用的 Bar。
- CollapsingToolbarLayout: 实现可折叠的 ToolBar。
- ToolBar: 代替 ActionBar。

实现中需要注意的点有:

- 我们为 ToolBar 的 app:layout_collapseMode 设置为 pin，表示折叠之后固定在顶端，而为 ImageView 的 app:layout_collapseMode 设置为 parallax，表示视差模式，即渐变的效果。
- 为了让 RecyclerView 支持嵌套滑动，还需要为它设置 app:layout_behavior="@string/appbar_scrolling_view_behavior"。
- 为 CollapsingToolbarLayout 设置 app:layout_scrollFlags="scroll|exitUntilCollapsed"，其中 scroll 表示滚动出屏幕，exitUntilCollapsed 表示退出后折叠。

具体实现参见 Demo6。

回顾

回顾整篇文章，发现我们已经实现了 RecyclerView 的很多扩展功能，包括: 打造万能适配器、添加 Item 事件、添加头视图和尾视图、设置空布局、侧滑拖拽。BaseRecyclerViewAdapterHelper 是一个比较火的 RecyclerView 扩展库，仔细一看发现，这里面 80%的功能在我们这篇文章中都实现了。
http://www.recyclerview.org/

扩展阅读

Google I/O 2016: RecyclerView Ins and Outs
RecyclerView 优秀文章集
https://github.com/CymChad/CymChad.github.io
https://github.com/wasabeef/recyclerview-animators
https://zhuanlan.zhihu.com/p/24807254