RecyclerView 原理分析

一、前言

RecyclerView 是谷歌官方出的一个用于大量数据展示的新控件，可以用来代替传统的 ListView，更加强大和灵活。

支持 RecyclerView 高效运行的主要六大类：

Adapter：为每一项 Item 创建视图

ViewHolder：承载 Item 视图的子布局

LayoutManager：负责 Item 视图的布局的显示管理

ItemDecoration：给每一项 Item 视图添加子 View，例如可以进行画分隔线之类

ItemAnimator：负责处理数据添加或者删除时候的动画效果

Recycler：负责 RecyclerView 中子 View 的回收与复用

Adapter

适配器的作用都是类似的，用于提供每个 item 视图，并返回给 RecyclerView 作为其子布局添加到内部。

但是，与 ListView 不同的是，ListView 的适配器是直接返回一个 View，将这个 View 加入到 ListView 内部。而 RecyclerView 是返回一个 ViewHolder 并且不是直接将这个 holder 加入到视图内部，而是加入到一个缓存区域，在视图需要的时候去缓存区域找到 holder 再间接的找到 holder 包裹的 View。

二、ViewHolder

我们写 ListView 的时候一般也会自己实现一个 ViewHolder，它会持有一个 View，避免不必要的 findViewById()，来提高效率。我们使用 RecyclerView 的时候必须创建一个继承自 RecyclerView.ViewHolder 的类。这主要是因为 RecyclerView 内部的缓存结构并不是像 ListView 那样去缓存一个 View，而是直接缓存一个 ViewHolder，在 ViewHolder 的内部又持有了一个 View。既然是缓存一个 ViewHolder，那么当然就必须所有的 ViewHolder 都继承同一个类才能做到了。

三、LayoutManager

顾名思义，LayoutManager 是布局的管理者。实际上，RecyclerView 就是将 onMeasure()、onLayout() 交给了 LayoutManager 去处理，因此如果给

RecyclerView 设置不同的 LayoutManager 就可以达到不同的显示效果。

四、onMeasure()

RecyclerView 的 onMeasure()最终都会调用：mLayout.onMeasure(mRecycler, mState, widthSpec, heightSpec);，这里的 mLayout 就是 LayoutManager，最终还是调用了 RecyclerView 自己的方法对布局进行了测量。

五、onLayoutChildren()

RecyclerView 的 onLayout()中会调用：mLayout.onLayoutChildren(mRecycler, mState);，这个方法在 LayoutManager 中是空实现：

```
public void onLayoutChildren(Recycler recycler, State state) {
        Log.e(TAG, "You must override onLayoutChildren(Recycler recycler, State state) ");
    }
```

所以 LayoutManager 的子类都应该实现 onLayoutChildren()，这里就将 layout()的工作交给了 LayoutManager 的实现类，来完成对子 View 的布局。

六、ItemDecoration

ItemDecoration 是为了显示每个 item 之间分隔样式的。它的本质实际上就是一个 Drawable。当 RecyclerView 执行到 onDraw()方法的时候，就会调用到他的 onDraw()，这时，如果你重写了这个方法，就相当于是直接在 RecyclerView 上画了一个 Drawable 表现的东西。而最后，在他的内部还有一个叫 getItemOffsets()的方法，从字面就可以理解，他是用来偏移每个 item 视图的。当我们在每个 item 视图之间强行插入绘画了一段 Drawable，那么如果再照着原本的逻辑去绘 item 视图，就会覆盖掉 Decoration 了，所以需要 getItemOffsets()这个方法，让每个 item 往后面偏移一点，不要覆盖到之前画上的分隔样式了。

七、ItemAnimator

每一个 item 在特定情况下都会执行的动画。说是特定情况，其实就是在视图发生改变，我们手动调用 notifyxxxx()的时候。通常这个时候我们会要传一个下标，那么从这个标记开始一直到结束，所有 item 视图都会被执行一次这个

动画。

八、Recycler

```
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;

    private RecycledViewPool mRecyclerPool;

    private ViewCacheExtension mViewCacheExtension;
    ......
}
```

RecyclerView 拥有四级缓存：

1、屏幕内缓存 ：指在屏幕中显示的 ViewHolder，这些 ViewHolder 会缓存在 mAttachedScrap、mChangedScrap 中 。

mChangedScrap 表示数据已经改变的 ViewHolder 列表

mAttachedScrap 未与 RecyclerView 分离的 ViewHolder 列表

2、屏幕外缓存：当列表滑动出了屏幕时，ViewHolder 会被缓存在 mCachedViews，其大小由 mViewCacheMax 决定，默认 DEFAULT_CACHE_SIZE 为 2，可通过 Recyclerview.setItemViewCacheSize()动态设置。

3、自定义缓存：可以自己实现 ViewCacheExtension 类实现自定义缓存，可通过 Recyclerview.setViewCacheExtension()设置。通常我们也不会去设置他，系统已经预先提供了两级缓存了，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，才会用到他。

4、缓存池 ：ViewHolder 首先会缓存在 mCachedViews 中，当超过了 2 个（默认为 2），就会添加到 mRecyclerPool 中。mRecyclerPool 会根据 ViewType 把 ViewHolder 分别存储在不同的集合中，每个集合最多缓存 5 个 ViewHolder。

九、缓存策略：

在 LayoutManager 执行 layoutChildren()中获取子 View 的时候，会调用 RecyclerView 的 getViewForPosition()：

```
View getViewForPosition(int position, boolean dryRun) {
    if (position < 0 || position >= mState.getItemCount()) {
        throw new IndexOutOfBoundsException("Invalid item position " + position
                + "(" + position + "). Item count:" + mState.getItemCount());
    }
    boolean fromScrap = false;
    ViewHolder holder = null;
    // 0) If there is a changed scrap, try to find from there
    if (mState.isPreLayout()) {
        // 从 mChangedScrap 找
        holder = getChangedScrapViewForPosition(position);
        fromScrap = holder != null;
    }
    // 1) Find from scrap by position
    if (holder == null) {
        // 通过 position 从 mAttachedScrap 找，找不到再通过 position 从 mCachedViews 找
        holder = getScrapViewForPosition(position, INVALID_TYPE, dryRun);
        if (holder != null) {
            if (!validateViewHolderForOffsetPosition(holder)) {
                // recycle this scrap
                if (!dryRun) {
                    // we would like to recycle this but need to make sure it is not used by
                    // animation logic etc.
                    holder.addFlags(ViewHolder.FLAG_INVALID);
                    if (holder.isScrap()) {
                        removeDetachedView(holder.itemView, false);
                        holder.unScrap();
                    } else if (holder.wasReturnedFromScrap()) {
                        holder.clearReturnedFromScrapFlag();
                    }
                    recycleViewHolderInternal(holder);
                }
                holder = null;
            } else {
                fromScrap = true;
            }
        }
```

```
            }
            if (holder == null) {
                final int offsetPosition = mAdapterHelper.findPositionOffset(position);
                if (offsetPosition < 0 || offsetPosition >= mAdapter.getItemCount()) {
                    throw new IndexOutOfBoundsException("Inconsistency detected. Invalid
item "
                            + "position " + position + "(offset:" + offsetPosition + ")."
                            + "state:" + mState.getItemCount());
                }

                final int type = mAdapter.getItemViewType(offsetPosition);
                // 2) Find from scrap via stable ids, if exists
                if (mAdapter.hasStableIds()) {
                    // 通过 id 从 mAttachedScrap 找，找不到再通过 id 从 mCachedViews
找
                    holder = getScrapViewForId(mAdapter.getItemId(offsetPosition), type,
dryRun);
                    if (holder != null) {
                        // update position
                        holder.mPosition = offsetPosition;
                        fromScrap = true;
                    }
                }
                if (holder == null && mViewCacheExtension != null) {
                    // 从 mViewCacheExtension 找
                    // We are NOT sending the offsetPosition because LayoutManager does
not
                    // know it.
                    final View view = mViewCacheExtension
                            .getViewForPositionAndType(this, position, type);
                    if (view != null) {
                        holder = getChildViewHolder(view);
                        if (holder == null) {
                            throw new
IllegalArgumentException("getViewForPositionAndType returned"
                                    + " a view which does not have a ViewHolder");
                        } else if (holder.shouldIgnore()) {
                            throw new
IllegalArgumentException("getViewForPositionAndType returned"
                                    + " a view that is ignored. You must call stopIgnoring
before"
                                    + " returning this view.");
                        }
                    }
```

```
                }

                if (holder == null) { // fallback to recycler
                    // 从 mRecyclerPool 找
                    // try recycler.
                    // Head to the shared pool.
                    if (DEBUG) {
                        Log.d(TAG, "getViewForPosition(" + position + ") fetching from
shared "
                                + "pool");
                    }
                    holder = getRecycledViewPool().getRecycledView(type);
                    if (holder != null) {
                        holder.resetInternal();
                        if (FORCE_INVALIDATE_DISPLAY_LIST) {
                            invalidateDisplayListInt(holder);
                        }
                    }
                }
                if (holder == null) {
                    // 从缓存中找不到，创建新的 ViewHolder
                    holder = mAdapter.createViewHolder(RecyclerView.this, type);
                    if (DEBUG) {
                        Log.d(TAG, "getViewForPosition created new ViewHolder");
                    }
                }
            }
            ......
        }
```

Recyclerview 在获取 ViewHolder 时按四级缓存的顺序查找，如果没找到就创建。通过了解 RecyclerView 的四级缓存，我们可以知道，RecyclerView 最多可以缓存 N（屏幕最多可显示的 item 数）+ 2（屏幕外的缓存）+ 5*M（M 代表 M 个 ViewType，缓存池的缓存）。还需要注意的是，RecyclerViewPool 可以被多个 RecyclerView 共享。