转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81613313

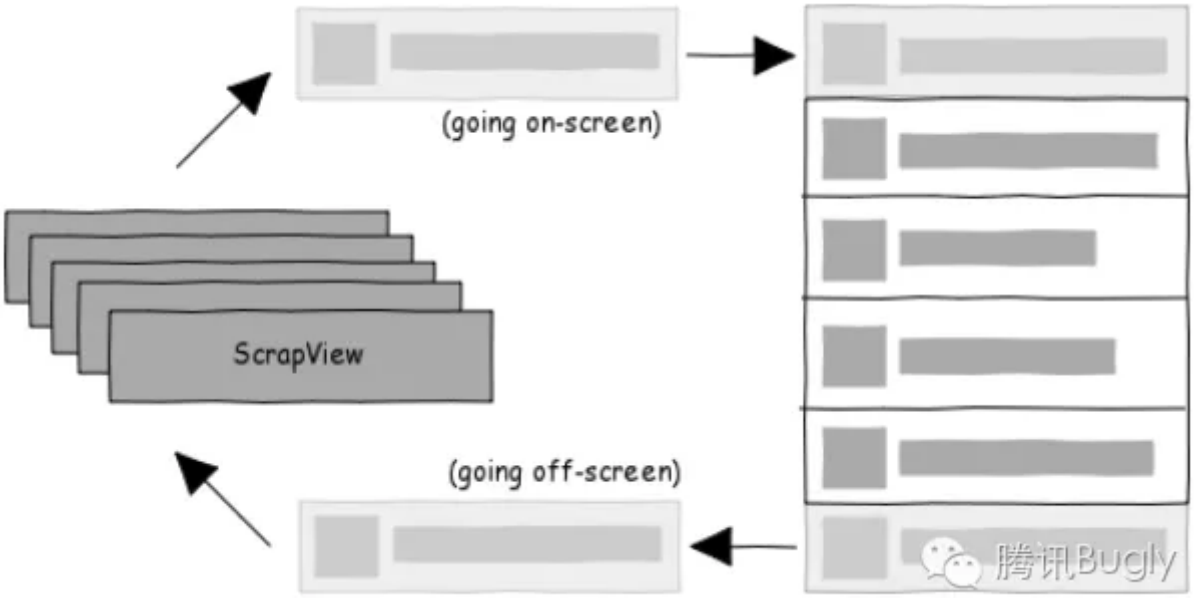
> 1. ListView涉及到RecycleBin和缓存流程
> 2. RecyclerView涉及到Recycler、定向刷新、布局流程、局部刷新的原理。

有帮助的话，请点个赞，万分感谢！

RecyclerView和ListView原理(49题)

最后修改版本：2018/8/23-1(15:59)

1、RecyclerView和ListView缓存原理



> 1. 离屏的ItemView即被回收至缓存
> 2. 入屏的ItemView则会优先从缓存中获取
> 3. 只是ListView与RecyclerView的实现细节有差异

ListView(15题)

2、AbsListView

> 1. ListView和GridView都继承自AbsListView
> 2. ListView/GridView的缓存原理都处于AbsListView中

3、AbsListView的测量过程

```
@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    if (mSelector == null) {
        useDefaultSelector();
    }
   //计算padding
    final Rect listPadding = mListPadding;
    listPadding.left = mSelectionLeftPadding + mPaddingLeft;
    listPadding.top = mSelectionTopPadding + mPaddingTop;
    listPadding.right = mSelectionRightPadding + mPaddingRight;
    listPadding.bottom = mSelectionBottomPadding + mPaddingBottom;
    // 如果transcriptMode是TRANSCRIPT_MODE_NORMAL，
    //当Adapter中的数据集改变之后，其子类会自动滚动到底部
    if (mTranscriptMode == TRANSCRIPT_MODE_NORMAL) {
        final int childCount = getChildCount();
        final int listBottom = getHeight() - getPaddingBottom();
        final View lastChild = getChildAt(childCount - 1);
        final int lastBottom = lastChild != null ? lastChild.getBottom() : listBottom;
        mForceTranscriptScroll = mFirstPosition + childCount >= mLastHandledItemCount &&
                lastBottom <= listBottom;
    }
}
```

> 1. padding计算
> 2. 当Adapter中的数据集改变之后，其子类会自动滚动到底部

4、AbsListView的布局过程

```
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    super.onLayout(changed, l, t, r, b);

    xxx
    final int childCount = getChildCount();
    if (changed) {
        for (int i = 0; i < childCount; i++) {
            getChildAt(i).forceLayout();
        }
      // 1-重新测量Child,mRecycler其实就是RecycleBin---用于管理回收View的回收类
        mRecycler.markChildrenDirty();
    }
    // 2-给Child布局
    layoutChildren();
    xxx
}
```

> 1. 子类不允许覆盖onLayout方法
> 2. 子类如ListView需要重写layoutChildren()方法
> 3. mRecycler.markChildrenDirty(): 对Child进行重新测量
> 4. layoutChildren(): 给Child布局

RecycleBin

5、RecycleBin是什么?

> 1. 用于帮助在layout过程中View的复用。
> 2. 包含两种层级的缓存：ActiveView和ScarpViews
> 3. ActiveViews：布局开始时处于屏幕上的View---会在离屏时被降级为ScrapViews
> 4. ScrapViews：Adpater用于避免不必要的分配View而使用的老旧的View
> 5. 缓存的操作一般为4种：初始化、存、取、清空缓存。

6、RecycleBin

```
class RecycleBin {

    // 可见的View数组
    private View[] mActiveViews = new View[0];
    // 存储在可见View数组中第一个View的位置
    private int mFirstActivePosition;


    // 不可见的View集合的数组：每种类型的Item都用一个集合存储、未排序、用于重用
    private ArrayList<View>[] mScrapViews;
    // View的类型(Type)的数量
    private int mViewTypeCount;
    // mScrapViews数组中第一个元素(集合)；或者说View Type = 1的集合
    private ArrayList<View> mCurrentScrap;

    xxx
}
```

> 1. mActiveViews: 可见的View数组
> 2. mScrapViews: 不可见的View集合的数组

7、ViewType和ViewTypeCount是什么?

> 1. 当ListView需要实现复杂列表时，比如根据Type从而显示的样式不同。需要View Type进行区分。
> 2. ViewTypeCount：有几种不同的View

```
// View的类型-int值.必须从0开始依次递增.
private static final int TYPE_TITLE = 0;
private static final int TYPE_CONTENT = 1;
// ListView创建View
@Override
public View getView(int position, View convertView, ViewGroup viewGroup) {

    switch (getItemViewType(position)){
        case TYPE_TITLE:
             // 第一种布局
            break;
        case TYPE_CONTENT:
             // 第二种布局
            break;
    }
    return convertView;
}
// 根据数据列表中的数据返回当前位置所属的Type，由开发者自定义。
@Override
public int getItemViewType(int position) {
    if(TextUtils.isEmpty(mData.get(position).getCode())){
        return TYPE_TITLE;
    }else{
        return TYPE_CONTENT;
    }
}
```

8、RecycleBin-初始化缓存

```
//AbsListView.java的内部类：RecycleBin---初始化缓存
public void setViewTypeCount(int viewTypeCount) {
    if (viewTypeCount < 1) {
        throw new IllegalArgumentException("Can't have a viewTypeCount < 1");
    }
    // 1-根据ViewTypeCount初始化数组： 不可见View集合的数组
    ArrayList<View>[] scrapViews = new ArrayList[viewTypeCount];
    for (int i = 0; i < viewTypeCount; i++) {
        scrapViews[i] = new ArrayList<View>();
    }
    // 2-初始化RecycleBin的数组
    mViewTypeCount = viewTypeCount;
    mCurrentScrap = scrapViews[0];
    mScrapViews = scrapViews;
}
```

> setViewTypeCount在设置Adpater时进行调用。

```
//使用: ListView.java
@Override
public void setAdapter(ListAdapter adapter) {
    // 1-清空RecycleBin
    mRecycler.clear();
    // 2-设置新的Adapter
    super.setAdapter(adapter);
    // 3-设置View的类型数量
    mRecycler.setViewTypeCount(mAdapter.getViewTypeCount());
    // 4-请求重新布局
    requestLayout();
}
```

> 1. 缓存初始化就是创建 `ScrapViews` 的过程。
> 2. ListView设置Adpater时会对缓存进行清空，并且进行缓存初始化。

9、RecycleBin-存缓存

```
/**====================================
 * //AbsListView.java
 * 存储屏幕上可见的View---将所有子View保存到ActiveViews中
 *====================================*/
void fillActiveViews(int childCount, int firstActivePosition) {
    // 1、扩容检查
    if (mActiveViews.length < childCount) {
        mActiveViews = new View[childCount];
    }
    // 2、存储在ActiveViews中第一个View的position
    mFirstActivePosition = firstActivePosition;

    // 3、遍历子View---将非头部非尾部的View放置到ActiveViews数组中
    final View[] activeViews = mActiveViews;
    for (int i = 0; i < childCount; i++) {
        View child = getChildAt(i);
        // 存储
        if (lp != null && lp.viewType != ITEM_VIEW_TYPE_HEADER_OR_FOOTER) {
            activeViews[i] = child;
        }
    }
}

/**====================================
 * //AbsListView.java
 * 将View添加到ScrapViews中---对不在屏幕中的View进行缓存
 *====================================*/
void addScrapView(View scrap, int position) {
    xxx
    // 不直接缓存具有transient state的View，用transient数组进行缓存
    final boolean scrapHasTransientState = scrap.hasTransientState();
    if (scrapHasTransientState) {
        //
    } else {
        // 根据Type存储到ScrapViews中
        mScrapViews[viewType].add(scrap);
    }
}
```

10、RecycleBin-取缓存

```
// 获取缓存的View(不可见)
View getScrapView(int position) {
    // 1、拿到当前位置View的type
    final int whichScrap = mAdapter.getItemViewType(position);
    // 2、type的种类为1，直接从第一个数组中取。否则从对应的type数组中取出
    if (mViewTypeCount == 1) {
        //type的种类为1，直接从第一个数组中取
        return retrieveFromScrap(mCurrentScrap, position);
    } else if (whichScrap < mScrapViews.length) {
        //直接从对应的type数组中取
        return retrieveFromScrap(mScrapViews[whichScrap], position);
    }
    return null;
}

// 获取可见的View
View getActiveView(int position) {
    int index = position - mFirstActivePosition;
    final View[] activeViews = mActiveViews;
    if (index >=0 && index < activeViews.length) {
        // 1、返回ActiveView数组中的View
        final View match = activeViews[index];
        //获取之后将数组置空，便于虚拟机回收
        activeViews[index] = null;
        return match;
    }
    return null;
}
```

11、RecycleBin-清除缓存

```
// 清除指定的ScrapView
private void clearScrap(final ArrayList<View> scrap) {
    final int scrapCount = scrap.size();
    for (int j = 0; j < scrapCount; j++) {
        removeDetachedView(scrap.remove(scrapCount - 1 - j), false);
    }
}

// 清空所有缓存数组
void clear() {
    if (mViewTypeCount == 1) {
        final ArrayList<View> scrap = mCurrentScrap;
        clearScrap(scrap);
    } else {
        final int typeCount = mViewTypeCount;
        for (int i = 0; i < typeCount; i++) {
            final ArrayList<View> scrap = mScrapViews[i];
            clearScrap(scrap);
        }
    }
    clearTransientStateViews();
}
```

12、RecycleBin-markChildrenDirty

```
// 执行所有缓存数组中View的forceLayout
public void markChildrenDirty() {
    // 1、ScrapView全部执行forceLayout
    for (int i = 0; i < typeCount; i++) {
        final ArrayList<View> scrap = mScrapViews[i];
        final int scrapCount = scrap.size();
        for (int j = 0; j < scrapCount; j++) {
            scrap.get(j).forceLayout();
        }
    }
    // 2、TransientStateViews都执行forceLayout
    for (int i = 0; i < count; i++) {
        mTransientStateViews.valueAt(i).forceLayout();
    }
    // 3、TransientStateViewsById都执行forceLayout
    for (int i = 0; i < count; i++) {
        mTransientStateViewsById.valueAt(i).forceLayout();
    }
}
```

缓存流程

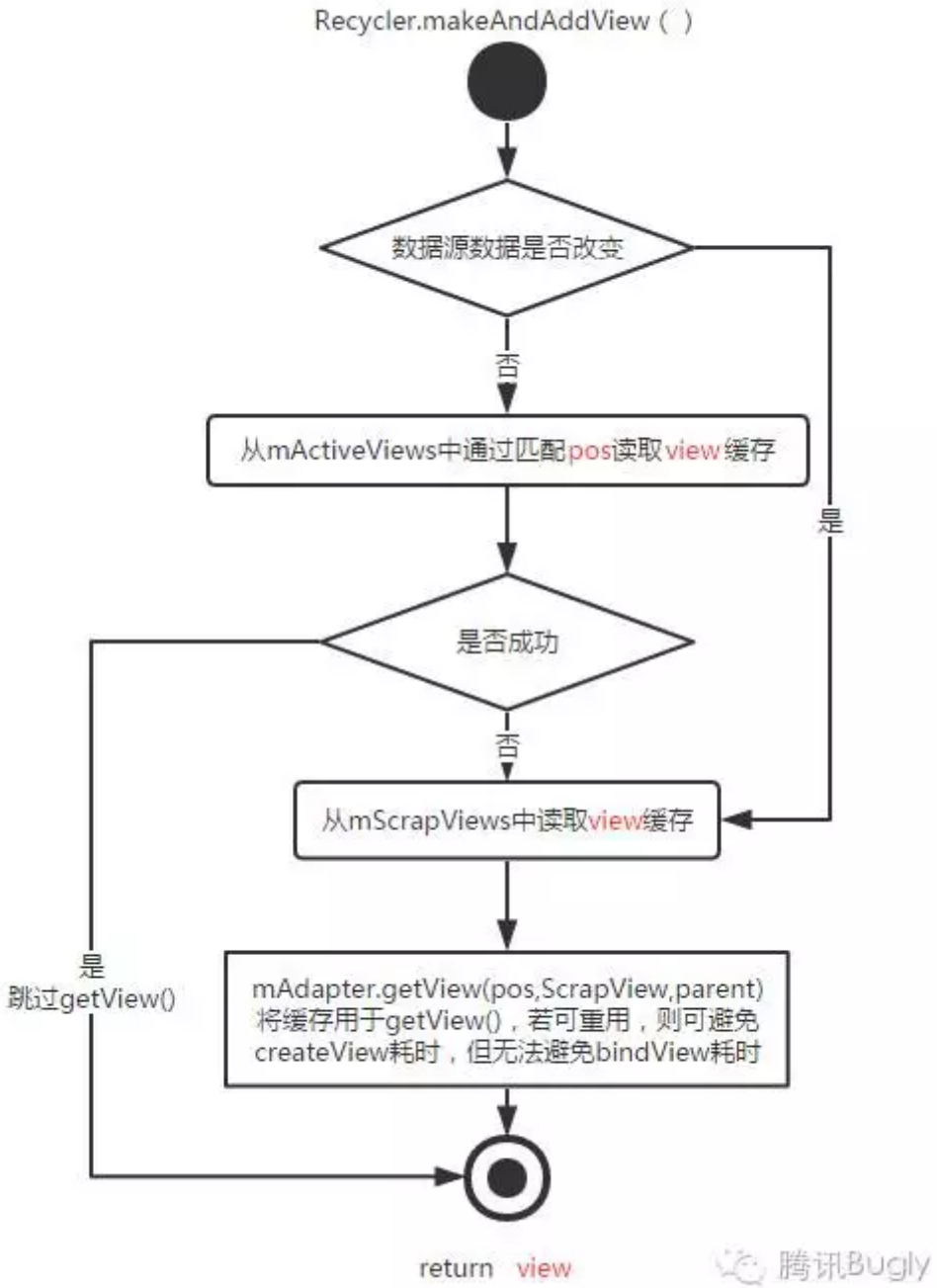
13、ListView有两级缓存

ListView				
	是否需要回调createView	是否需要回调bindView	生命周期	备注
mActiveViews	否	否	onLayout函数周期内	用于屏幕内ItemView快速重用
mScrapViews	否	是	与mAdapter一致，当mAdapter被更换时，mScrapViews即被清空	

> 1. 第一层 `ActiveViews` ： 用于屏幕内ItemView的快速重用
> 2. 第二层 `ScrapViews` ： 不可见View

```
// 1-mActiveViews是View数组
private View[] mActiveViews = new View[0];
// 2-mScrapViews是ArrayList
private ArrayList<View>[] mScrapViews;
```

14、ListView缓存流程



1. 有新View需要显式的时候，先去ActiveViews中获取。存在就直接取出并且复用。
2. 不存在，会从ScrapView中去获取。存在就会复用，避免执行createView但是会执行bindView。
3. ScrapViews中也不存在，就会inflate View，然后bindView。

15、ListView布局流程

```
// ListView.java-布局Children
@Override
protected void layoutChildren() {
    // 1-从ListView当前顶部往下填充
    fillFromTop(childrenTop);
    xxx
}
// ListView.java-从顶至下填充
private View fillFromTop(int nextTop) {
    //xxx
    return fillDown(mFirstPosition, nextTop);
}
// ListView.java-从pos填充到屏幕可见区域的底部
private View fillDown(int pos, int nextTop) {
    // 1、创建View填充ListView(屏幕可见区域的最上端填充到底部，或者Item已经都创建完毕)
    while (nextTop < end && pos < mItemCount) {
        boolean selected = pos == mSelectedPosition;
        // 2、创建并且获取到ChildView
        View child = makeAndAddView(pos, nextTop, true, mListPadding.left, selected);
        //xxx
    }
    return selectedView;
}
// ListView.java-获取到Child View
private View makeAndAddView(int position, int y, boolean flow, int childrenLeft, boolean se
    // 1、数据集没有变化，从ActiveView中获取
    if (!mDataChanged) {
        final View activeView = mRecycler.getActiveView(position);
        return activeView;
    }else{
        // 2、数据集中有变化
        final View child = obtainView(position, mIsScrap);
        return child;
    }
}

// AbsListView.java-从缓存中获取到View
View obtainView(int position, boolean[] outMetadata) {
    // 1、获取到ScrapView(不可见的View)
    final View scrapView = mRecycler.getScrapView(position);
    // 2、获取到ChildView(createView和bindView)
    final View child = mAdapter.getView(position, scrapView, this);
    if (scrapView != null) {
        if (child != scrapView) {
            // 3、将新View(复用后的View)添加到ScrapView中
            mRecycler.addScrapView(scrapView, position);
        }
    }
    return child;
}
```

1. onLayout(): 调用layoutChildren

2. layoutChildren(): fillFromTop
3. fillFromTop(): fillDown
4. makeAndAddView(): obtainView

RecyclerView(28题)

1、RecyclerView特点

1. 多样式：可以对数据的展示进行定制，可以是列表\网格\瀑布流，还可以自定义样式.
2. 定向刷新：可以对指定的Item数据进行刷新
3. 刷新动画：RecyclerView支持对Item的刷新添加动画
4. 添加装饰：相对于ListView以及GridView的单一的分割线，RecyclerView可以自定义添加分割样式

2、RecyclerView的6大组成

RecyclerView内部类	
LayoutManager	负责Item的布局和显示
ItemDecoration	给Item添加修饰的View
Adapter	为Item创建视图
ViewHolder	承载Item的布局
ItemAnimator	负责处理数据添加或者删除时的动画效果
【Cache】	Recycler/RecycledViewPool/ViewCacheExtension

缓存机制

四级缓存

3、RecyclerView具有四级缓存

1. 屏幕内缓存： 在屏幕中显示的ViewHolder。缓存到mChangedScrap和mAttachedScrap中。
2. 屏幕外缓存：列表滑动出屏幕时，ViewHolder会被缓存。缓存到mCachedViews中。
3. 自定义缓存：自己实现ViewCacheExtension类来实现自定义缓存。
4. 缓存池：屏幕外缓存的mCachedViews已满时，会将ViewHolder缓存到RecycledViewPool中。

4、屏幕内缓存

在屏幕中显示的ViewHolder，会进行缓存。

> 1. mChangedScrap: 缓存数据已经改变的ViewHolder。
> 2. mAttachedScrap: 缓存所有-Attached-Scrapped-View，连接着的-废弃的-View。

5、屏幕外缓存

> 列表滑动出屏幕时，ViewHolder会被缓存。
>
> 1. mCachedViews: 进行屏幕外缓存，默认大小为2。大小由 `mViewCacheMax` 决定。DEFAULT_CACHE_SIZE = 2。
> 2. Recyclerview.setItemViewCacheSize(), 可以设置屏幕外缓存的大小。

6、自定义缓存

> 1. 可以自己实现ViewCacheExtension类实现自定义缓存
> 2. 可以通过Recyclerview.setViewCacheExtension()设置。

7、缓存池

> ViewHolder在首先会缓存在 mCachedViews中，当超过了个数(默认为2)， 就会添加到RecycledViewPool 中。
>
> 1. RecycledViewPool: 会根据每个ViewType把ViewHolder分别存储在不同的列表中。
> 2. 每个ViewType最多缓存DEFAULT_MAX_SCRAP个ViewHolder(默认是5个)

8、RecyclerView的最多缓存多少个?

> 在共享缓存池的情况下： N + 2 + 5*ViewType
>
> 1. 屏幕内最多可以显示的Item数：N
> 2. 屏幕外缓存：2个
> 3. 缓存池1-被多个RecyclerView共享：5 * ViewType数量
> 4. 缓存池2-没有被共享：线性布局-1个 * ViewType数量；网格布局-1个 * 行数

缓存池

9、缓存池的使用

```
// 1、获取到第一个RV的缓存池
RecycledViewPool pool = mRecyclerView1.getRecycledViewPool();
// 2、第二个RV使用第一个RV的缓存池
mRecyclerView2.setRecycledViewPool(pool);
// 3、第三个RV使用第一个RV的缓存池
mRecyclerView3.setRecycledViewPool(pool);

// 可以设置缓存池大小(指定某个ViewType)
pool.setMaxRecycledViews(ITEM_VIEW_TYPE, maxNumber);

// 可以手动清除pool
pool.clear();
```

1. 不需要自己去new 一个 RecycledViewPool
2. 必须确保共享的RecyclerView的Adapter是同一个，或ViewType 不会冲突。(RecycledViewPool是依据ItemViewType来索引ViewHolder)
3. 在合适的时机，RecycledViewPool会自动清除掉所持有的ViewHolder对象引用，不用担心内存泄漏问题。也可以手动调用clear()

Recycler

10、RV的内部类Recycler是什么？有什么作用?

1. Recylcer用于管理"Scrpped View"和"Detached View"
2. Scrapped View: 废弃的View，也就是仍然和RV连接着的，但是已经被标记为“删除的”或者“重用的”
3. Detached View: 已经移除的View。

```
/**
 * Recylcer用于管理"Scrpped View"和"Detached item"
 */
public final class Recycler {
    // 1、连接着RV的所有的Scraped View
    final ArrayList<RecyclerView.ViewHolder> mAttachedScrap = new ArrayList();
    // 2、数据源更改过的Attached View(连接着RV的View)
    ArrayList<RecyclerView.ViewHolder> mChangedScrap = null;
    // 3、缓存的所有View(包括可见和不可见的ViewHolder)
    final ArrayList<RecyclerView.ViewHolder> mCachedViews = new ArrayList();

    private final List<RecyclerView.ViewHolder> mUnmodifiableAttachedScrap;
    // 4、RecycledViewPool：ViewHolder的缓存池，如果多个RV之间用setRecycledViewPool设置同一个
    RecyclerView.RecycledViewPool mRecyclerPool;
    /**==========================================
     * 5、缓存的扩展，可以对指定的position跟Type缓存
     *  * 通过实现方法getViewForPositionAndType()来实现自己的缓存。
     *==========================================================*/

    private RecyclerView.ViewCacheExtension mViewCacheExtension;

    static final int DEFAULT_CACHE_SIZE = 2; //默认缓存数量
    private int mRequestedCacheMax = DEFAULT_CACHE_SIZE; //设置的最大缓存数量，默认为2。
    int mViewCacheMax = DEFAULT_CACHE_SIZE; //View的缓存的最大数量，默认为2。
}
```

11、RV的缓存相对于ListView缓存的创新之处?

1. `RV` 给ViewHolder增加了一个 `UpdateOp` 标志。
2. 通过 `UpdateOp` 标志可以进行定向刷新指定的Item，并通过 `Payload` 参数对 `Item` 进行局部刷新。
3. 如果数据源经常变动，RecyclerView是最好的选择。

初始化

12、RecycledViewPool的作用

> 1. RecycledViewPool类是用来缓存Item用，是一个ViewHolder的缓存池
> 2. 如果多个RecyclerView之间用setRecycledViewPool(RecycledViewPool)设置同一个RecycledViewPool，他们就可以共享Item。
> 3. RecycledViewPool的内部维护了一个Map，里面以不同的viewType为Key存储了各自对应的ViewHolder集合。
> 4. 可以通过提供的方法来修改内部缓存的Viewholder。

13、Recycler会对RecylcedViewPool初始化

```
void setRecycledViewPool(RecyclerView.RecycledViewPool pool) {
    if (this.mRecyclerPool != null) {
        this.mRecyclerPool.detach();
    }

    this.mRecyclerPool = pool;
    if (this.mRecyclerPool != null && RecyclerView.this.getAdapter() != null) {
        this.mRecyclerPool.attach();
    }

}
```

> 1. 如果原来有RecyclerPool则进行detach工作，然后使用新的pool

存缓存

14、Recycler的recycleView()进行缓存

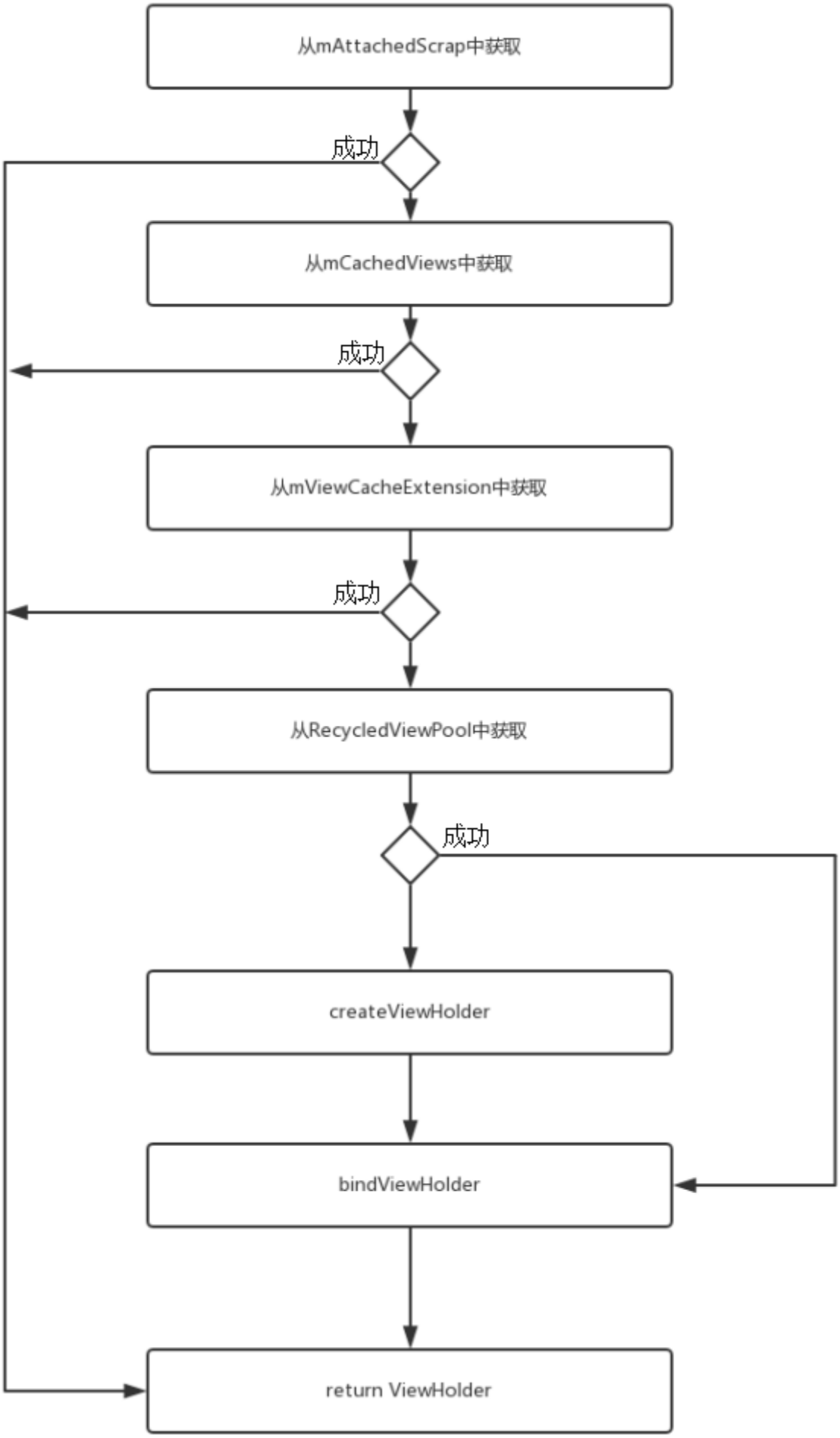
```
/**==============================================
 * 回收不可见的View：
 *    1. 采用LFU算法，最少使用策略，会去覆盖掉最少是用的缓存
 *    2. 会添加缓存到mCachedViews中(缓存的所有View---包括可见和不可见的ViewHolder)
 *    3. 特定的View回访金RecyclerViewPool
 *==============================================*/
public void recycleView(View view) {
    //这边传递过来的是一个View，然后通过View获取ViewHolder
    ViewHolder holder = getChildViewHolderInt(view);
    xxx
    //开始缓存
    recycleViewHolderInternal(holder);
}

void recycleViewHolderInternal(ViewHolder holder) {
    // 1. 缓存的时候不能覆盖最近经常被使用到缓存
    if (!mPrefetchRegistry.lastPrefetchIncludedPosition(cachedPos)) {
        break;
    }

    // 2. 添加缓存
    mCachedViews.add(targetCacheIndex, holder);
    cached = true;
    // 3. 如果没有缓存的话就添加进RecycledViewPool
    if (!cached) {
        addViewHolderToRecycledViewPool(holder, true);
        recycled = true;
    }
}
```

取出缓存

15、RecyclerView的四级缓存


从mAttachedScrap中获取
成功
从mCachedViews中获取
成功
从mViewCacheExtension中获取
成功
从RecycledViewPool中获取
成功
createViewHolder
bindViewHolder
return ViewHolder

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

16、Recycler取出缓存

```
// 1、获取到Position上的View
View getViewForPosition(int position, boolean dryRun) {
    return this.tryGetViewHolderForPositionByDeadline(position, dryRun, xxx).itemView;
}

@Nullable
RecyclerView.ViewHolder tryGetViewHolderForPositionByDeadline(int position, boolean dryRun)
    RecyclerView.ViewHolder holder = null;
    /**====================================
     * 1、如果是提前layout，会先从 mChangedScrap集合中取（数据源更改过的Attached View）
     *====================================*/
    if (RecyclerView.this.mState.isPreLayout()) {
        // mChangedScrap.get(offsetPosition)
        holder = this.getChangedScrapViewForPosition(position);
        xxx
    }

    /**====================================
     * 2、会先从 mAttachedScrap集合中取（所有Attached Scrapped View），再从mCachedViews中寻找
     *====================================*/
    if (holder == null) {
        // mAttachedScrap.get(cacheSize)->mCachedViews.get(i)
        holder = this.getScrapOrHiddenOrCachedHolderForPosition(position, dryRun);
        xxxx
    }
    /**====================================
     * 3、会先从 mAttachedScrap集合中取，再从mCachedViews中寻找。
     *====================================*/
    if (holder == null) {
        // mAttachedScrap.get(cacheSize)->mCachedViews.get(i)
        holder = this.getScrapOrCachedViewForId(RecyclerView.this.mAdapter.getItemId(offset
    }

    /**====================================
     * 4、从mViewCacheExtension中获取
     *====================================*/
    if (holder == null && this.mViewCacheExtension != null) {
        View view = this.mViewCacheExtension.getViewForPositionAndType(this, position, type
        if (view != null) {
            holder = RecyclerView.this.getChildViewHolder(view);
        }
    }
    /**====================================
     * 5、从RecycledViewPool中获取ViewHolder
     *====================================*/
    if (holder == null) {
        holder = this.getRecycledViewPool().getRecycledView(type);
    }

    /**====================================================
     * 6、均获取失败，会先create ViewHolder，再bind ViewHolder
     *====================================================*/
    if (holder == null) {
        // 调用RV的Adapter的方法创建ViewHolder
```

```
            holder = RecyclerView.this.mAdapter.createViewHolder(RecyclerView.this, type);
            // 绑定ViewHolder：  RecyclerView.this.mAdapter.bindViewHolder(holder, offsetPositio
            this.tryBindViewHolderByDeadline(holder, type, position, deadlineNs);
        }
        // 设置布局参数
        android.view.ViewGroup.LayoutParams lp = holder.itemView.getLayoutParams();
        RecyclerView.LayoutParams rvLayoutParams = generateLayoutParams(lp);

        return holder;
    }
```

清除缓存

17、Recycler清除所有缓存

```
// RecyclerView.java内部类：Recycler
public void clear() {
    // 1、mAttachedScrap(所有Attached Scrapped View)都清除
    this.mAttachedScrap.clear();
    // 2、mCachedViews(缓存的所有可见和不可见View)全部清除
    this.recycleAndClearCachedViews();
}
void recycleAndClearCachedViews() {
    // 3、清除mCachedViews
    this.mCachedViews.clear();
    xxx
}
```

观察者模式

18、RecyclerView的Observer(观察者)

> 1)-RV没有采用系统的Observer，而是用RecyclerViewDataObserver继承自自定义的AdapterDataObserver。
> 2)-AdapterDataObserver如下：

```
public abstract static class AdapterDataObserver {
    // 改变：范围内item
    public void onItemRangeChanged(int positionStart, int itemCount) {
    }
    // 插入：范围内item
    public void onItemRangeInserted(int positionStart, int itemCount) {
    }
    // 删除：范围内item
    public void onItemRangeRemoved(int positionStart, int itemCount) {
    }
    // 移动：范围内item
    public void onItemRangeMoved(int fromPosition, int toPosition, int itemCount) {
    }
}
```

> 3)-RecyclerViewDataObserver: 继承并实现了AdapterDataObserver的所有方法。

19、RecyclerView的Observable(被观察者)

> 1. RecyclerView实际采用了系统的Observable。
> 2. AdapterDataObservable继承自Observable。

```
static class AdapterDataObservable extends Observable<RecyclerView.AdapterDataObserver> {
    // 获取到【观察者】，并且调用其onChanged方法
    public void notifyChanged() {
        for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onChanged();
        }

    }
    // 通知观察者，范围刷新Item
    public void notifyItemRangeChanged(int positionStart, int itemCount, @Nullable Object p
        for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onItemRangeChanged(p
        }

    }
    // 通知观察者，范围插入Item
    public void notifyItemRangeInserted(int positionStart, int itemCount) {
        for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onItemRangeInserted(
        }

    }
    // 通知观察者，范围删除Item
    public void notifyItemRangeRemoved(int positionStart, int itemCount) {
        for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onItemRangeRemoved(p
        }

    }
    // 通知观察者，范围移动Item
    public void notifyItemMoved(int fromPosition, int toPosition) {
        for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onItemRangeMoved(fro
        }
    }
}
```

setAdapter

20、setAdapter内部的流程是什么?

> 1. 设置新的Adapter
> 2. 解除原来Adapter中所有的观察者(RecycleView内部的Observer)。
> 3. 通过新Adapyter去注册观察者。这样Adapter数据改变时，RecyclerView能收到通知。

```
// ReyclerView.java
public class RecyclerView extends ViewGroup implements ScrollingView, NestedScrollingChild {
    // 1、每个RecyclerView对象中都有一个Observer(观察者)
    private final RecyclerViewDataObserver mObserver = new RecyclerViewDataObserver();

    /**============================================================
     * 2、设置新的Adapter
     *   1. 当Adapter改变时，所有存在的View都会回收到RecycledViewPool。
     *      如果Pool只有一个adapter，Pool会直接清空。
     *   2. requestLayout：  请求重新测量、布局
     *============================================================*/
    public void setAdapter(Adapter adapter) {
        // 1.
        setAdapterInternal(adapter, false, true);
        // 2. 请求重新测量、布局
        requestLayout();
    }

    /**============================================================
     * 3、用新的Adpater替换当前Adapter。并且触发监听器。
     *      1. 所有存在的View都会回收到RecycledViewPool。
     *      2. 如果Pool只有一个adapter，Pool会直接清空。
     *============================================================*/
    private void setAdapterInternal(Adapter adapter, boolean compatibleWithPrevious,
                                    boolean removeAndRecycleViews) {
        /**============================
         * 1、当前Adapter存在时。
         *   1. 解注册(移除掉该Adapter的观察者)
         *   2. 触发Adpater的回调方法：
         *==============================*/
        if (mAdapter != null) {
            // 1. 解注册(移除那些观察者)
            mAdapter.unregisterAdapterDataObserver(mObserver);
            // 2. 触发Adpater的回调方法：onDetachedFromRecyclerView
            mAdapter.onDetachedFromRecyclerView(this);
        }
        /**============================
         * 2、移除并且回收所有存在的View
         *==============================*/
        if (!compatibleWithPrevious || removeAndRecycleViews) {
            // 1. 停止Item动画
            if (mItemAnimator != null) {
                mItemAnimator.endAnimations();
            }
            if (mLayout != null) {
                // 2. 移除并回收所有Views
                mLayout.removeAndRecycleAllViews(mRecycler);
                // 3. 移除并回收所有Scrapped Views.
                mLayout.removeAndRecycleScrapInt(mRecycler);
            }
            // 4. 清除Recycler
            mRecycler.clear();
        }
        // 3、重置AdapterHelper(该类用于帮助Adapter更新数据)
```

```
        mAdapterHelper.reset();
        final Adapter oldAdapter = mAdapter;
        mAdapter = adapter;
        /**==========================
         * 4、在新Adapter注册RecyclerView内部的观察者上。(Adapter是被观察者)
         *   1. 注册观察者
         *   2. 触发Adpater的回调方法
         *============================*/
        if (adapter != null) {
            // 1. 进行注册监听
            adapter.registerAdapterDataObserver(mObserver);
            // 2. 触发Apdater的onAttachedToRecyclerView方法，这是空实现
            adapter.onAttachedToRecyclerView(this);
        }
        if (mLayout != null) {
            mLayout.onAdapterChanged(oldAdapter, mAdapter);
        }
        // 5、所有存在的View都会回收到RecycledViewPool。如果Pool只有一个adapter，Pool会直接清空。
        mRecycler.onAdapterChanged(oldAdapter, mAdapter, compatibleWithPrevious);
    }
}

// ReyclerView.java
public static abstract class Adapter<VH extends ViewHolder> {
    private final AdapterDataObservable mObservable = new AdapterDataObservable();
    /**=======================================
     * 1. 注册一个观察者。在数据发生变化时，Adapter会去通知这些观察者。
     *=========================================*/
    public void registerAdapterDataObserver(AdapterDataObserver observer) {
        mObservable.registerObserver(observer);
    }
}
```

定向刷新

21、AdapterHelper的作用

> 1. 用于帮助Adapter更新数组
> 2. 类似于ChildHelper帮助LayoutManager进行布局

22、RecyclerView的定向刷新

> 调用 `mAdapter.notifyItemChanged(int position);` 进行定向刷新

```
// RecyclerView.Adapter
public final void notifyItemChanged(int position) {
    // 被观察者调用方法
    this.mObservable.notifyItemRangeChanged(position, 1);
}
// RecyclerView.AdapterDataObservable
public void notifyItemRangeChanged(int positionStart, int itemCount, @Nullable Object payload)
    for(int i = this.mObservers.size() - 1; i >= 0; --i) {
            // 通知观察者进行定向刷新
            ((RecyclerView.AdapterDataObserver)this.mObservers.get(i)).onItemRangeChanged(posit
    }
}
// RecyclerView.AdapterDataObserver
public void onItemRangeChanged(int positionStart, int itemCount, @Nullable Object payload) {
    this.onItemRangeChanged(positionStart, itemCount);
}

// RecyclerView.RecyclerViewDataObserver
public void onItemRangeChanged(int positionStart, int itemCount, Object payload) {
    // 1、通过AdapterHelper给需要更新的Item添加更新标记，并且记录范围。
    if (RecyclerView.this.mAdapterHelper.onItemRangeChanged(positionStart, itemCount, payload))
        // 2、执行动画，并且requestLayout
        this.triggerUpdateProcessor();
    }
}
// AdapterHelper.java
boolean onItemRangeChanged(int positionStart, int itemCount, Object payload) {
    if (itemCount < 1) {
        return false;
    } else {
        // 将需要更新的Item的范围记录下来，并且添加更新标识。
        this.mPendingUpdates.add(this.obtainUpdateOp(UpdateOp.UPDATE, positionStart, itemCount,
        this.mExistingUpdateTypes |= UpdateOp.UPDATE;
        return this.mPendingUpdates.size() == 1;
    }
}

/**=======================================
 * //RecyclerView.RecyclerViewDataObserver
 *  1. 无论是全部刷新，还是指定范围刷新，RecyclerView都需要进行重新measure和layout，不一定会draw
 *  2. 定向刷新本质是对指定范围内的ViewHolder进行数据刷新
 *  3. 根据逻辑推测：应该是在测量、布局阶段进行的定向刷新。
 *=======================================*/
void triggerUpdateProcessor() {
    if (mPostUpdatesOnAnimation && mHasFixedSize && mIsAttached) {
        // 1、有动画，就先进行动画
        ViewCompat.postOnAnimation(RecyclerView.this, mUpdateChildViewsRunnable);
    } else {
        mAdapterUpdateDuringMeasure = true;
        // 2、重新测量
        requestLayout();
    }
}
```

1. Adapter的notifyItemChanged，通过观察者模式最终交给 `RecyclerView` 进行处理。
2. 会通过AdapterHelper的onItemRangeChanged给需要更新的Item添加更新标记，并且记录范围。

23、定向刷新是如何给Item添加标记和记录范围的?

1. AdapterHelper内部有一个 `UpdateOp` 的列表。
2. 会将更新状态 `UpdateOp.UPDATE` 保存到UpdateOp中
3. 会将需要更新的范围起点 `positionStart` 和item数量 `itemCount` 也保存到UpdateOp中。
4. 将UpdateOp存储到AdapterHelper内部的列表中后，后续在 `layout` 时，会根据这些信息，对ViewHolder的数据进行更新(重新执行BindViewHolder)

```
// UpdateOp列表：用于延迟更新
ArrayList<UpdateOp> mPendingUpdates = new ArrayList<UpdateOp>();
// 将UpdateOp添加到列表中(包含更新标志，和更新范围)
mPendingUpdates.add(this.obtainUpdateOp(UpdateOp.UPDATE, positionStart, itemCount, payload));
```

24、AdapterHelper和RecyclerView的关联是如何建立的?

1. RecyclerView创建时会在内部创建AdapterHelper
2. 因此AdapterHelper内部辅助更新数据的列表等内容，就属于该唯一的RecyclerView

```
public RecyclerView(Context context, @Nullable AttributeSet attrs, int defStyle) {
    initAdapterManager();
    //xxx
}
void initAdapterManager() {
    mAdapterHelper = new AdapterHelper(new Callback()){...}
}
```

布局

布局流程

25、RecyclerView的布局流程。

```
// RecyclerView.java---onLayout进行布局
@Override
protected void onLayout(boolean changed, int l, int t, int r, int b) {
    xxx
    dispatchLayout();
}

// RecyclerView.java
void dispatchLayout() {
    if (mAdapter == null) {
        return;
    }
    if (mLayout == null) {
        return;
    }
    // 1. RV布局处于Start阶段，进行dispatchLayoutStep1和dispatchLayoutStep2两步
    if (mState.mLayoutStep == RecyclerView.State.STEP_START) {
        // 第一步
        dispatchLayoutStep1();
        mLayout.setExactMeasureSpecsFrom(this);
        // 第二步
        dispatchLayoutStep2();
    } else if (mAdapterHelper.hasUpdates() || mLayout.getWidth() != getWidth() ||
            mLayout.getHeight() != getHeight()) {
        // 2. 因为size的改变，不得不进行第二步：dispatchLayoutStep2
        mLayout.setExactMeasureSpecsFrom(this);
        dispatchLayoutStep2();
    }
    // 3. 第三步
    dispatchLayoutStep3();
}

/**============================================================================
 * // RecyclerView.java
 * 布局第一步完成如下4个工作：
 * 1. 处理adapter的更新
 * 2. 决定哪个animation需要运行
 * 3. 存储关于当前View的信息
 * 4. 如果有必要，会进行预言性的布局，并且保存相关信息。
 *============================================================================*/
private void dispatchLayoutStep1() {
    // 1. 处理Adapter数据更新的问题；  计算需要运行的动画类型
    processAdapterUpdatesAndSetAnimationFlags();
    // 2. 存储关于当前View的信息。
    //  xxx
    // 3.  如果有必要，会进行预言性的布局，并且保存相关信息。
    //  xxx
    mState.mLayoutStep = RecyclerView.State.STEP_LAYOUT;
}

/**============================================================
 * // RecyclerView.java
 * 布局第(二)步完成如下工作：
 *   1. 进行View的实际布局，该步骤可能会被多次调用(在必要的时候，如测量)
```

```
 *   2. 设置状态为STEP_ANIMATIONS，用于第三步的动画。
 *==============================================================*/
private void dispatchLayoutStep2() {
    // 0. 回调接口
    mAdapterHelper.consumeUpdatesInOnePass();
    // 1. Children的市局布局
    mLayout.onLayoutChildren(mRecycler, mState);
    // 2. 设置布局步骤为：STEP_ANIMATIONS
    mState.mRunSimpleAnimations = mState.mRunSimpleAnimations && mItemAnimator != null;
    // 动画阶段
    mState.mLayoutStep = RecyclerView.State.STEP_ANIMATIONS;
    onExitLayoutOrScroll();
    resumeRequestLayout(false);
}

/**================================================
 * // RecyclerView.java
 * 布局第(三)步完成如下工作：
 *   1. 保存关于Views的所有信息
 *   2. 触发动画
 *   3. 做必要的清理操作
 *================================================*/
private void dispatchLayoutStep3() {mState.mLayoutStep = RecyclerView.State.STEP_START;
    if (mState.mRunSimpleAnimations) {
        // 0. 处理Change动画。倒序遍历，因为改变动画可能会移除目标View Holder。
        for (int i = mChildHelper.getChildCount() - 1; i >= 0; i--) {
            RecyclerView.ViewHolder holder = getChildViewHolderInt(mChildHelper.getChildAt(i));
            // 处理动画
            animateChange(oldChangeViewHolder, holder, preInfo, postInfo, xxx);
        }
        // 1|2. 处理View信息列表和触发动画
        mViewInfoStore.process(mViewInfoProcessCallback);
    }
    resumeRequestLayout(false);
    // 3. 清理工作
    mViewInfoStore.clear();
}

/**==========================================
 *  // RecyclerView.LayoutManager
 *  定义了需要子类实现的公共方法：
 *      onLayoutChildren:  布局Child Views
 *==========================================*/
public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {
    Log.e(TAG, "You must override onLayoutChildren(Recycler recycler, State state) ");
}
// LinearLayoutManager.java
@Override
public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {
    /**============================================================
     * 布局算法：
     * 1. 通过检查Children和其他变量，找到锚点坐标和一个锚点Item的position
     * 2. 从bottom填充到start
     * 3. 从top填充到end
     * 4. scroll to fulfill requirements like stack from bottom.
```

```
     *============================================================*/
    // 开始布局
    fill(recycler, mLayoutState, state, false);
}
// LinearLayoutManager.java
int fill(RecyclerView.Recycler recycler, LayoutState layoutState,) {
    xxx
    while ((layoutState.mInfinite || remainingSpace > 0) && layoutState.hasMore(state)) {
        //循环中调用了此方法
        layoutChunk(recycler, state, layoutState, layoutChunkResult);
    }
}

void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State state, xxx) {
    /**==========================================
     *  1、从4级缓存中获取到View
     *  //recycler.getViewForPosition(mCurrentPosition);
     *==========================================*/
    View view = layoutState.next(recycler);
    // 2、 将View添加到RecyclerView中
    addView(view);
    // 3、对view进行测量(包括margin)
    measureChildWithMargins(view, 0, 0);
    // 4、设置view的left/top/bottom/right ， 并且也会考虑上padding
    int left, top, right, bottom;
    if (mOrientation == VERTICAL) {
        if (isLayoutRTL()) {
            right = getWidth() - getPaddingRight();
            left = right - mOrientationHelper.getDecoratedMeasurementInOther(view);
        } else {
            left = getPaddingLeft();
            right = left + mOrientationHelper.getDecoratedMeasurementInOther(view);
        }
        if (layoutState.mLayoutDirection == LayoutState.LAYOUT_START) {
            bottom = layoutState.mOffset;
            top = layoutState.mOffset - result.mConsumed;
        } else {
            top = layoutState.mOffset;
            bottom = layoutState.mOffset + result.mConsumed;
        }
    } else {
        top = getPaddingTop();
        bottom = top + mOrientationHelper.getDecoratedMeasurementInOther(view);
        if (layoutState.mLayoutDirection == LayoutState.LAYOUT_START) {
            right = layoutState.mOffset;
            left = layoutState.mOffset - result.mConsumed;
        } else {
            left = layoutState.mOffset;
            right = layoutState.mOffset + result.mConsumed;
        }
    }
    // 5、对view进行布局(考虑上装饰品和margin)
    layoutDecoratedWithMargins(view, left, top, right, bottom);
}
```

1. onLayout()：布局的入口，会执行dispatchLayout()
2. dispatchLayout(): 根据RecyclerView的布局步骤，选择执行步骤1,2,3。
3. dispatchLayoutStep1(): STEP_START时调用，(1)处理adapter的更新 (2) 决定哪个animation需要运行 (3)保存View相关信息。
4. dispatchLayoutStep2(): 进行View的实际布局, 可能会被多次调用。状态为STEP_START，或者Adapter数据更新，或者Layout的size发生改变时，调用该步骤。
5. dispatchLayoutStep3(): 主要是触发动画，该步骤必定执行。
6. onLayoutChildren(): 主要在dispatchLayoutStep2中调用。需要LayoutManager的子类进行实现。
7. LinearLayoutManager.onLayoutChildren(): 调用fill()
8. fill(): 遍历循环，调用layoutChunk对子View进行添加、测量、布局
9. layoutChunk(): (1)从四级缓存中获取到View (2)addView (3)测量(包括amrgin) (4)布局，会涉及到margin和装饰品

定向刷新流程

26、RecyclerView布局过程中处理定向刷新的流程

1. 开始点也是从onLayout->disptachLayout->dispatchLayout1
2. dispatchLayout1中会处理Adapter的数据更新。
3. 本质是：Adapter进行notifyDataChanged设置更新标识(UPDATE)，并且保存更新的范围(startposition，itemcount)，requestLayout进行重新测量布局时，根据这些信息，对ViewHolder进行数据更新。
4. dispatchLayoutStep1(): 内部调用processAdapterUpdatesAndSetAnimationFlags()
5. processAdapterUpdatesAndSetAnimationFlags(): 会调用AdapterHelper的consumeUpdatesInOnePass()
6. consumeUpdatesInOnePass(): 内部会切换到AdapterHelper创建时锁传入的Callback中
7. 对于Update操作，最终会层层调用到RecyclerView的viewRangeUpdate()方法中。
8. viewRangeUpdate(): 给ViewHolder设置FLAG_UPDATE和payload。
9. 最后在布局阶段，调用getViewForPosition()从Recylcer中取出View时，因为FLAG为FLAG_UPDATE，因此会执行 `mAdapter.bindViewHolder()` 完成ViewHolder数据的刷新。

```
/**============================================================
 * // RecyclerView.java
 * 布局第一步完成如下4个工作：
 * 1. 处理adapter的更新
 * 2. 决定哪个animation需要运行
 * 3. 存储关于当前View的信息
 * 4. 如果有必要，会进行预言性的布局，并且保存相关信息。
 *============================================================================*/
private void dispatchLayoutStep1() {
    // 1. 处理Adapter数据更新的问题；  计算需要运行的动画类型
    processAdapterUpdatesAndSetAnimationFlags();
    // 2. 存储关于当前View的信息。
    //  xxx
    // 3.  如果有必要，会进行预言性的布局，并且保存相关信息。
    //  xxx
    mState.mLayoutStep = RecyclerView.State.STEP_LAYOUT;
}

/**================================================
 * // RecyclerView.java
 *  1. 处理Adapter的更新
 *  2. 计算出想要运行的动画类型
 *  * 该方法会在onMeasure和dispatchLayout中运行。
 *================================================*/
private void processAdapterUpdatesAndSetAnimationFlags() {
    if (mDataSetHasChangedAfterLayout) {
        // 1、将没有数值的所有Item，reset
        mAdapterHelper.reset();
        // 2、留给子类的空实现，用于在Item改变时进行通知回调
        mLayout.onItemsChanged(this);
    }
    // 3、更新ViewHolder
    mAdapterHelper.consumeUpdatesInOnePass();
    // 4、动画相关操作
    // xxx
}

/**================================================
 * //AdapterHelper.java
 *   1. 根据UpdateOp的cmd是update、add、remove、move进行处理
 *   2. mCallback是在AdapterHelper进行初始化时赋值的
 *================================================*/
void consumeUpdatesInOnePass() {
    for (int i = 0; i < mPendingUpdates.size(); i++) {
        AdapterHelper.UpdateOp op = mPendingUpdates.get(i);
        switch (op.cmd) {
            case AdapterHelper.UpdateOp.ADD:
                mCallback.onDispatchSecondPass(op);
                break;
            case AdapterHelper.UpdateOp.REMOVE:
                mCallback.onDispatchSecondPass(op);
                break;
            case AdapterHelper.UpdateOp.UPDATE:
                // 更新数据
```

```
                mCallback.onDispatchSecondPass(op);
                break;
            case AdapterHelper.UpdateOp.MOVE:
                mCallback.onDispatchSecondPass(op);
                break;
        }
    }
}
// AdapterHelper.java
AdapterHelper(AdapterHelper.Callback callback, boolean disableRecycler) {
    mCallback = callback;
    //xxx
}
// RecyclerView.java---在initAdapterManager中创建AdapterHelper
void initAdapterManager() {
    mAdapterHelper = new AdapterHelper(new AdapterHelper.Callback() {
        // 1、调用dispatchUpdate
        @Override
        public void onDispatchSecondPass(AdapterHelper.UpdateOp op) {
            dispatchUpdate(op);
        }
        // 2、交给LayoutManager进行处理
        void dispatchUpdate(AdapterHelper.UpdateOp op) {
            switch (op.cmd) {
                case AdapterHelper.UpdateOp.ADD:
                    mLayout.onItemsAdded(RecyclerView.this, op.positionStart, op.itemCount)
                    mCallback.offsetPositionsForAdd(op.positionStart, op.itemCount);
                    break;
                case AdapterHelper.UpdateOp.REMOVE:
                    mLayout.onItemsRemoved(RecyclerView.this, op.positionStart, op.itemCour
                    mCallback.offsetPositionsForRemovingInvisible(op.positionStart, op.item
                    break;
                case AdapterHelper.UpdateOp.UPDATE:
                    // 3、 层层深入，会发现是空实现，用于子LayoutManager进行实现，用于通知回调
                    mLayout.onItemsUpdated(RecyclerView.this, op.positionStart, op.itemCour
                            op.payload);
                    // 4、 ViewHolder的实际更新(从op中获取到开始位置，以及数量)
                    mCallback.markViewHoldersUpdated(op.positionStart, op.itemCount, op.pay
                    break;
                case AdapterHelper.UpdateOp.MOVE:
                    mLayout.onItemsMoved(RecyclerView.this, op.positionStart, op.itemCount,
                    mCallback.offsetPositionsForMove(op.positionStart, op.itemCount);
                    break;
            }
        }
        // 5、ViewHolder的实际更新
        @Override
        public void markViewHoldersUpdated(int positionStart, int itemCount, Object payloa
            viewRangeUpdate(positionStart, itemCount, payload);
            mItemsChanged = true;
        }

        // xxx
    });
}
```

```
    /**==================================
     * // RecyclerView.LayoutManager
     *   1. 调用onItemsUpdated，交给LayoutManager的子类去实现，用于进行回调
     *==================================*/
    public void onItemsUpdated(RecyclerView recyclerView, int positionStart, xxx) {
        onItemsUpdated(recyclerView, positionStart, itemCount);
    }
    public void onItemsUpdated(RecyclerView recyclerView, int positionStart, int itemCount) {
        // 空实现
    }

    /**==================================================
     * //RecyclerView.java
     *  ViewHolder的实际更新，positionStart + itemCount
     *==================================================*/
    void viewRangeUpdate(int positionStart, int itemCount, Object payload) {
        for (int i = 0; i < childCount; i++) {
            // 1、获取到childView，再通过该childView获取到对应的ViewHolder
            final View child = mChildHelper.getUnfilteredChildAt(i);
            final RecyclerView.ViewHolder holder = getChildViewHolderInt(child);
            if (holder.mPosition >= positionStart && holder.mPosition < positionEnd) {
                // 2、添加Flag，用于从缓存中取出该ViewHolder时，进行刷新(唯一标识)
                holder.addFlags(RecyclerView.ViewHolder.FLAG_UPDATE);
                // 3、添加payload参数，用于ViewHolder的局部刷新
                holder.addChangePayload(payload);
            }
        }
        // 4、 更新Recycler缓存中的标记位，便于数据源不改变的时候直接复用。
        mRecycler.viewRangeUpdate(positionStart, itemCount);
    }
```

27、RecyclerView如何根据ViewHolder的Flag进行数据更新?

> 1. 在layoutChunk阶段，会去获取position对应的View
> 2. 在内部会根据holder.needsUpdate()去重新执行bindViewHolder方法，完成数据的更新。

```
View getViewForPosition(int position, boolean dryRun) {
    // xxx
    // 1. 判断ViewHolder的Flag为FLAG_UPDATE
    if (holder.needsUpdate() || holder.isInvalid()) {
        // 2. 需要执行bindViewHolder方法
        mAdapter.bindViewHolder(holder, offsetPosition);
    }
    return holder.itemView;
}
```

局部刷新

28、局部刷新是什么?

1. 某个Item中，只有一小部分数据需要改变，而不是刷新这个Item。使用该功能需要两步骤。
2. 第一步：需要调用Adapter的具有payload参数的方法 `notifyItemChanged(int position, Object payload)`
3. 第二步：实现RecyclerView.Adapter中具有参数payload的onBindViewHolder()

```
int position = 1;
// 1. 没有Payload参数-不支持局部刷新【常规】
adapter.notifyItemChanged(position);
// 2. 有Payload参数-支持局部刷新
adapter.notifyItemChanged(position, "payload");


// 自定义Adapter中：常规的bindView
@Override
public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
    // 正常的设置数据
    holder.title.setText(mDatas.get(position).getTitle()+"");
    holder.id.setText(mDatas.get(position).getId()+"");
    xxx
}

// 自定义Adapter中：使用payload进行局部刷新
@Override
public void onBindViewHolder(RecyclerView.ViewHolder holder, int position, List payloads) {
    // 1. 没有payload还是执行常规的内容
    if(payloads.isEmpty()){
        onBindViewHolder(holder, position);
    }else{
        // 2. 存在payload，paoload内容没啥实际用途，只刷新了Title
        holder.title.setText(mDatas.get(position).getTitle()+"");
        //没有刷新ID
        //holder.id.setText(mDatas.get(position).getId()+"");
    }
}
```

RecyclerView和ListView对比(2题)

1、RecyclerView和ListView的缓存层级不同

1. ListView是两层
2. RecyclerView是四层
3. RecyclerView支持多个离屏ItemView缓存
4. RecyclerView支持开发者自定义缓存处理逻辑
5. RecyclerView支持所有RV公用同一个RecycklerViewPool(缓存池)

2、RecyclerView和AbsListView的区别和联系(8)?

1. 缓存机制不同：RV四级缓存; AbsListView两级缓存。

2. 定向刷新：只有RV支持
3. 局部刷新：只有RV支持
4. 刷新动画：只有RV支持
5. 分割线：RV自定义样式; AbsListView样式单一
6. 布局方式：RV自定义样式；列表/网格，切换麻烦
7. 头尾添加：RV不支持；AbsListView支持
8. Item点击：RV不支持；AbsListView支持

面试题：考考你(4题)

1、RecyclerView如何复用Item?

1. 会通过Recycler.recycleView()方法将ViewHolder进行缓存。
2. 获取缓存的时候，会调用Recycler.getViewForPosition()从四级缓存中获取View。
3. 当RecycledViewPool中获取到View时，会去调用 `bindViewHolder()` 去绑定数据。
4. `ViewHolder的Flag处于更新标识` 时，也会去调用 `bindViewHolder()` 去绑定数据。

2、RecyclerView如何定向刷新Item?

本质是：对指定的Item的ViewHolder刷新数据。

1. 通过Adapter的nofity系列方法，最终会交给AdapterHelper进行数据定向刷新。
2. 本质在 `UpdateOp` 中存储 `更新标识和更新范围` ，并将 `UpdateOp` 存储到 `AdapterHelper` 内部的UpdateOp列表中。
3. 在RecyclerView的布局阶段，onLayout->disptachLayout->dispatchLayout1中会去更新ViewHolder的Flag为 `FLAG_UPDATE` 。
4. 在布局的第二阶段disptachLayout2()->layoutChildren->layoutChunk，中会通过Recylcer去获取缓存的ViewHolder，因为其Flag为更新标识，因此会执行 `bindViewHolder` 进行数据刷新。

3、RecyclerView如何实现局部刷新Item？大致原理?

局部刷新某个ViewHolder的某个控件。

1. 需要调用Adapter具有payload参数的方法notifyItemChanged(int position, Object payload)
2. 需要实现RecyclerView.Adapter中具有参数payload的onBindViewHolder()
3. 原理是将ViewHolderFlag设置为FLAG_STATE，并且添加payload
4. 在布局流程中，会去调用具有payload参数的onBinderViewHolder

4、DiffUtil: 如何找到新旧集合的差别?

知识储备

DiffUtil

1、DiffUtil的作用

> 比较两个集合的区别

2、DiffUtil是如何找到新旧集合的差别?

SortedList

1、SortedList的作用

> 数据去重

参考资料

1. ListView的getItemViewType和getViewTypeCount
2. Android ListView工作原理完全解析，带你从源码的角度彻底理解
3. android列表View,ListView源码分析
4. 深入理解Android中的缓存机制(二)RecyclerView跟ListView缓存机制对比
5. 关于Recyclerview的缓存机制的理解
6. RecycledViewPool的使用