什么是RecyclerView

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

用于向大型数据集提供有限窗口的灵活的视图。首先它是一个空间容器

**public class** RecyclerView **extends** ViewGroup **implements** ScrollingView, NestedScrollingChild {

}

onLayout

@Override  
**protected void** onLayout(**boolean** changed, **int** l, **int** t, **int** r, **int** b) {  
 TraceCompat.*beginSection*(***TRACE\_ON\_LAYOUT\_TAG***);  
 dispatchLayout();  
 TraceCompat.*endSection*();  
 **mFirstLayoutComplete** = **true**;  
}

dispatchLayout

**void** dispatchLayout() {  
 **if** (**mAdapter** == **null**) {  
 Log.*e*(***TAG***, **"No adapter attached; skipping layout"**);  
 *// leave the state in START* **return**;  
 }  
 **if** (**mLayout** == **null**) {  
 Log.*e*(***TAG***, **"No layout manager attached; skipping layout"**);  
 *// leave the state in START* **return**;  
 }  
 **mState**.**mIsMeasuring** = **false**;  
 **if** (**mState**.**mLayoutStep** == State.***STEP\_START***) {  
 dispatchLayoutStep1();  
 **mLayout**.setExactMeasureSpecsFrom(**this**);  
 dispatchLayoutStep2();  
 } **else if** (**mAdapterHelper**.hasUpdates() || **mLayout**.getWidth() != getWidth() ||  
 **mLayout**.getHeight() != getHeight()) {  
 *// First 2 steps are done in onMeasure but looks like we have to run again due to  
 // changed size.* **mLayout**.setExactMeasureSpecsFrom(**this**);  
 dispatchLayoutStep2();  
 } **else** {  
 *// always make sure we sync them (to ensure mode is exact)* **mLayout**.setExactMeasureSpecsFrom(**this**);  
 }  
 dispatchLayoutStep3();  
}

dispatchLayoutStep1

/\*\*

\* 第一步布局子控件，处理 adapter的更新操作，决定执行哪种动画效果，为当前展示的几个子视图保存信息，如果需要的话，预先布局并保存他们的信息

\*/

**private void** dispatchLayoutStep1() {  
 **mState**.assertLayoutStep(State.***STEP\_START***);  
 **mState**.**mIsMeasuring** = **false**;  
 eatRequestLayout();  
 **mViewInfoStore**.clear();  
 onEnterLayoutOrScroll();  
 processAdapterUpdatesAndSetAnimationFlags();  
 saveFocusInfo();  
 **mState**.**mTrackOldChangeHolders** = **mState**.**mRunSimpleAnimations** && **mItemsChanged**;  
 **mItemsAddedOrRemoved** = **mItemsChanged** = **false**;  
 **mState**.**mInPreLayout** = **mState**.**mRunPredictiveAnimations**;  
 **mState**.**mItemCount** = **mAdapter**.getItemCount();  
 findMinMaxChildLayoutPositions(**mMinMaxLayoutPositions**);  
  
 **if** (**mState**.**mRunSimpleAnimations**) {  
 *// Step 0: Find out where all non-removed items are, pre-layout* **int** count = **mChildHelper**.getChildCount();  
 **for** (**int** i = 0; i < count; ++i) {  
 **final** ViewHolder holder = *getChildViewHolderInt*(**mChildHelper**.getChildAt(i));  
 **if** (holder.shouldIgnore() || (holder.isInvalid() && !**mAdapter**.hasStableIds())) {  
 **continue**;  
 }  
 **final** ItemHolderInfo animationInfo = **mItemAnimator** .recordPreLayoutInformation(**mState**, holder,  
 ItemAnimator.*buildAdapterChangeFlagsForAnimations*(holder),  
 holder.getUnmodifiedPayloads());  
 **mViewInfoStore**.addToPreLayout(holder, animationInfo);  
 **if** (**mState**.**mTrackOldChangeHolders** && holder.isUpdated() && !holder.isRemoved()  
 && !holder.shouldIgnore() && !holder.isInvalid()) {  
 **long** key = getChangedHolderKey(holder);  
 *// This is NOT the only place where a ViewHolder is added to old change holders  
 // list. There is another case where:  
 // \* A VH is currently hidden but not deleted  
 // \* The hidden item is changed in the adapter  
 // \* Layout manager decides to layout the item in the pre-Layout pass (step1)  
 // When this case is detected, RV will un-hide that view and add to the old  
 // change holders list.* **mViewInfoStore**.addToOldChangeHolders(key, holder);  
 }  
 }  
 }  
 **if** (**mState**.**mRunPredictiveAnimations**) {  
 *// Step 1: run prelayout: This will use the old positions of items. The layout manager  
 // is expected to layout everything, even removed items (though not to add removed  
 // items back to the container). This gives the pre-layout position of APPEARING views  
 // which come into existence as part of the real layout.  
  
 // Save old positions so that LayoutManager can run its mapping logic.* saveOldPositions();  
 **final boolean** didStructureChange = **mState**.**mStructureChanged**;  
 **mState**.**mStructureChanged** = **false**;  
 *// temporarily disable flag because we are asking for previous layout* **mLayout**.onLayoutChildren(**mRecycler**, **mState**);  
 **mState**.**mStructureChanged** = didStructureChange;  
  
 **for** (**int** i = 0; i < **mChildHelper**.getChildCount(); ++i) {  
 **final** View child = **mChildHelper**.getChildAt(i);  
 **final** ViewHolder viewHolder = *getChildViewHolderInt*(child);  
 **if** (viewHolder.shouldIgnore()) {  
 **continue**;  
 }  
 **if** (!**mViewInfoStore**.isInPreLayout(viewHolder)) {  
 **int** flags = ItemAnimator.*buildAdapterChangeFlagsForAnimations*(viewHolder);  
 **boolean** wasHidden = viewHolder  
 .hasAnyOfTheFlags(ViewHolder.***FLAG\_BOUNCED\_FROM\_HIDDEN\_LIST***);  
 **if** (!wasHidden) {  
 flags |= ItemAnimator.***FLAG\_APPEARED\_IN\_PRE\_LAYOUT***;  
 }  
 **final** ItemHolderInfo animationInfo = **mItemAnimator**.recordPreLayoutInformation(  
 **mState**, viewHolder, flags, viewHolder.getUnmodifiedPayloads());  
 **if** (wasHidden) {  
 recordAnimationInfoIfBouncedHiddenView(viewHolder, animationInfo);  
 } **else** {  
 **mViewInfoStore**.addToAppearedInPreLayoutHolders(viewHolder, animationInfo);  
 }  
 }  
 }  
 *// we don't process disappearing list because they may re-appear in post layout pass.* clearOldPositions();  
 } **else** {  
 clearOldPositions();  
 }  
 onExitLayoutOrScroll();  
 resumeRequestLayout(**false**);  
 **mState**.**mLayoutStep** = State.***STEP\_LAYOUT***;  
}

processAdapterUpdateAndSetAnimationFlags

/\*\*

\* 消费 adapter 的更新事件，判断我们需要哪种类型的动画；

\* 在 onMeasure 和 disatchLayout方法中调用

\* 这个方法 可以 处理 预布局状态

\*/

**private void** processAdapterUpdatesAndSetAnimationFlags() {  
 **if** (**mDataSetHasChangedAfterLayout**) {  
 *// Processing these items have no value since data set changed unexpectedly.  
 // Instead, we just reset it.* **mAdapterHelper**.reset();  
 **mLayout**.onItemsChanged(**this**);  
 }  
 *// simple animations are a subset of advanced animations (which will cause a  
 // pre-layout step)  
 // If layout supports predictive animations, pre-process to decide if we want to run them* **if** (predictiveItemAnimationsEnabled()) {  
 **mAdapterHelper**.preProcess();  
 } **else** {  
 **mAdapterHelper**.consumeUpdatesInOnePass();  
 }  
 **boolean** animationTypeSupported = **mItemsAddedOrRemoved** || **mItemsChanged**;  
 **mState**.**mRunSimpleAnimations** = **mFirstLayoutComplete** && **mItemAnimator** != **null** && (**mDataSetHasChangedAfterLayout** || animationTypeSupported  
 || **mLayout**.**mRequestedSimpleAnimations**)  
 && (!**mDataSetHasChangedAfterLayout** || **mAdapter**.hasStableIds());  
 **mState**.**mRunPredictiveAnimations** = **mState**.**mRunSimpleAnimations** && animationTypeSupported  
 && !**mDataSetHasChangedAfterLayout** && predictiveItemAnimationsEnabled();  
}

参考

<https://chacojack.github.io/2016/08/20/RecyclerView%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-%E4%B8%80-%E6%95%B4%E4%BD%93%E8%AE%BE%E8%AE%A1/>

<https://github.com/MindorksOpenSource/SnapHelperExample>