# RecyclerView笔记

# RecyclerView的回收复用缓存机制详解

## 概述

  RecyclerView作为官方指定的高效、高拓展性的列表控件，做了很好的封装，灵活好用，深受我们喜欢。官方对它的介绍：为大量数据提供有限展示窗口的灵活视图。要想在有限的手机内存中展示大量的数据，并且保证不会OOM，它是怎么做到的呢？

我们在adapter的onCreateViewHolder()和onBindViewHolder()分别打印了log，其中，onCreateViewHolder()会在创建一个新view的时候调用，onBindViewHolder()会在已存在view，绑定数据的时候调用。所以，如果是新创建的view，会调用onCreateViewHolder()来创建view，调用onBindViewHolder()来绑定数据；如果是复用的view，则不会调用onCreateViewHolder()创建方法，只会调用onBindViewHolder()绑定数据。

private int sum = 0;  
  
@NonNull  
@Override  
public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {  
 Log.e("LinearVerticalAdapter", "onCreateViewHolder == " + sum);  
 sum += 1;  
 ······  
 return null;  
}  
  
@Override  
public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, int position) {  
 Log.e("LinearVerticalAdapter", "onBindViewHolder");  
 ······  
}

因为RecyclerView能够自动回收复用，这必须有强大的缓存机制支撑，RecyclerView的缓存机制是RecyclerView的核心部分。这里围绕RecyclerView的缓存机制来谈一谈，RecyclerView的回收复用机制是怎么样的？

为了方便下面文章的理解，我们先了解几个方法的含义：

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 对应Flag | 含义 | 出现场景 |
| isInvalid() | FLAG\_INVALID | ViewHolder的数据是无效的 | 1.调用adapter的setAdapter()  2.adapter调用了  notifyDataSetChanged();  3.调用RecyclerView的  invalidateItemDecorations()。 |
| isRemoved() | FLAG\_REMOVED | ViewHolder已经被移除，源数据被移除了部分数据 | adapter调用了notifyItemRemoved() |
| isUpdated() | FLAG\_UPDATE | item的ViewHolder数据信息过时了，需要重新绑定数据 | 1.上述isInvalid()的三种情况都会；  2.调用adapter的  onBindViewHolder()；  3.调用了adapter的  notifyItemChanged()。 |
| isBound() | FLAG\_BOUND | ViewHolder已经绑定了某个位置的item上，数据是有效的 | 调用了onBindViewHolder()方法 |

## Recycler的几级缓存

  RecyclerView不需要像ListView那样if(contentView==null) {}else{}处理复用的逻辑，它回收复用是由Recycler来负责的，Recycler是负责管理scrapped(废弃)或者detached(分离)的视图(ViewHolder)以便重复使用。要想了解RecyclerView的回收复用原理，那么首先了解Recycler的几个结构体：

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*;  
  
 RecycledViewPool mRecyclerPool;  
  
 private ViewCacheExtension mViewCacheExtension;  
  
 static final int *DEFAULT\_CACHE\_SIZE* = 2;  
}

Recycler中设置了四层缓存池，按照使用的优先级顺序依次是Scrap、CacheView、ViewCacheExtension、RecycledViewPool；其中Scrap包括mAttachedScrap和mChangedScrap，ViewCacheExtension是默认没有实现的，它RecyclerView留给开发者拓展的回收池。

● mAttachedScrap： 不参与滑动时的回收复用，只保存重新布局时从RecyclerView分离的item的无效、未移除、未更新的holder。因为RecyclerView在onLayout的时候，会先把children全部移除掉，再重新添加进入，mAttachedScrap临时保存这些holder复用。

● mChangedScrap： mChangedScrap和mAttachedScrap类似，不参与滑动时的回收复用，只是用作临时保存的变量，它只会负责保存重新布局时发生变化的item的无效、未移除的holder，那么会重走adapter绑定数据的方法。

● mCachedViews ： 用于保存最新被移除(remove)的ViewHolder，已经和RecyclerView分离的视图；它的作用是滚动的回收复用时如果需要新的ViewHolder时，精准匹配(根据position/id判断)是不是原来被移除的那个item；如果是，则直接返回ViewHolder使用，不需要重新绑定数据；如果不是则不返回，再去mRecyclerPool中找holder实例返回，并重新绑定数据。这一级的缓存是有容量限制的，最大数量为2。

● mViewCacheExtension： RecyclerView给开发者预留的缓存池，开发者可以自己拓展回收池，一般不会用到，用RecyclerView系统自带的已经足够了。

● mRecyclerPool： 是一个终极回收站，真正存放着被标识废弃(其他池都不愿意回收)的ViewHolder的缓存池，如果上述mAttachedScrap、mChangedScrap、mCachedViews、mViewCacheExtension都找不到ViewHolder的情况下，就会从mRecyclerPool返回一个废弃的ViewHolder实例，但是这里的ViewHolder是已经被抹除数据的，没有任何绑定的痕迹，需要重新绑定数据。它是根据itemType来存储的，是以SparseArray嵌套一个ArraryList的形式保存ViewHolder的。

接着我们来详细分析一下各个缓存池：

2.1 缓存池一 (Scrap)

Scrap是RecyclerView最轻量的缓存，包括mAttachedScrap和mChangedScrap，它不参与列表滚动时的回收复用，作为重新布局时的临时缓存，它的作用是，缓存当界面重新布局前和界面重新布局后都出现的ViewHolder，这些ViewHolder是无效、未移除、未标记的。在这些无效、未移除、未标记的ViewHolder之中，mAttachedScrap负责保存其中没有改变的ViewHolder；剩下的由mChangedScrap负责保存。mAttachedScrap和mChangedScrap也只是分工合作保存不同ViewHolder而已。

注意：Scrap只是作为布局的临时缓存，它和滑动时的缓存没有任何关系，它的detach和atach只是临时存在于布局过程中。布局结束时，Scrap列表应该是空的，缓存的数据要么重新布局出来，要么被清空；总之在布局结束后Scrap列表不应该存在任何东西。

我们上图分析：

在一个手机屏幕中，将itemB删除，并且调用notifyItemRemoved()方法，如果item是无效并且被移除的就会回收到其他的缓存，否则都是缓存到Scrap中，那么mAttachedScrap和mChangedScrap会分别存储itemView，itemA没有任何的变化，存储到mAttachedScrap中，itemB虽然被移出了，但是还有效，也被存储到mAttachedScrap中(但是会被标记REMOVED，之后会移除)；itemC和itemD发生了变化，位置往上移动了，会被存储到mChangedScrap中。删除时，ABCD都会进入Scrap中；删除后，ACD都会回来，A没有任何变化，CD只是位置发生了变化，内容没有发生变化。

RecyclerView的局部刷新就是依赖Scrap的临时缓存，当我们通过notifyItemRemoved()，notifyItemChanged()通知item发生变化的时候，通过mAttachedScrap缓存没有发生变化的ViewHolder，其他的则由mChangedScrap缓存，添加itemView的时候快速从里面取出，完成局部刷新。注意，如果我们使用notifyDataSetChanged()来通知RecyclerView刷新，屏幕上的itemView被标记为FLAG\_INVALID并且未被移除，所以不会使用Scrap缓存，而是直接扔到CacheView或者RecycledViewPool池中，回来的时候重新走一次绑定数据。

注意：itemE并没有出现在屏幕中，它不属于Scrap管辖的范围，Scrap只会换在屏幕中已经加载出来的itemView的holder。

2.2 缓存池二 (CacheView)

CacheView用于RecyclerView列表位置产生变动时，对刚刚移出屏幕的view进行回收复用。根据position/id来精准匹配是不是原来的item，如果是则直接返回使用，不需要重新绑定数据；如果不是则去RecycledViewPool中找holder实例返回，并且重新绑定数据。

CacheView的最大容量为2，缓存一个新的ViewHolder时，如果超出了最大限制，那么会将CacheView缓存的第一个数据添加到RecycledViewPool后再移除掉，最后才会将新的ViewHolder添加进来。我们在滑动RecyclerView的时候，Recycler会不断地缓存刚刚移出屏幕不可见的View到CacheView中，CacheView到达上限时又会不断替换CacheView中旧的ViewHolder，将它们扔到RecycledViewPool中。如果一直朝一个方向滚动，CacheView并没有在效率上产生帮助，它只是把后面滑过的ViewHolder缓存起来，如果经常来回滑动，那么从CacheView根据对应位置的item直接复用，不需要重新绑定数据，将会得到很好的利用。

用图来看看CacheView的复用场景：

从图中可以看出，CacheView缓存刚刚变为不可见的view，如果当前View再次进入屏幕中的时候，进行精准匹配，这个itemView还是 之前的itemView，那么会从CacheView中获取ViewHolder进行复用。如果一直向某一个方向滑动，那么CacheView将会不断替换缓存里面的ViewHolder(CacheView最多只能存储2个)，将替换掉的ViewHolder先放到RecycledViewPool中。在CacheView中拿不到复用的ViewHolder，那么最后只能去RecycledViewPool中获取。

2.3 缓存池三 (ViewCacheExtension)

ViewCacheExtension是缓存拓展的帮助类，额外提供了一层缓存池给开发者。开发者视情况而定是否使用ViewCacheExtension增加一层缓存池，Recycler首先去scrap和CacheView中寻找复用view，如果没有就去ViewCacheExtension中寻找View，如果还是没有找到，那么最后去RecycledViewPool寻找复用的View。下面的讲解将会不涉及ViewCacheExtension的知识，大家知道即可。

注意：Recycler并没有将任何的view缓存到ViewCacheExtension中。所以在ViewCacheExtension中并没有缓存任何数据。

2.4 缓存池四 (RecycledViewPool)

在Scrap、CacheView、ViewCacheExtension都不愿意回收的时候，都会丢到RecycledViewPool中回收，所以RecycledViewPool是Recycler的终极回收站。

RecycledViewPool实际上是以SparseArray嵌套一个ArraryList的形式保存ViewHolder的，因为RecycledViewPool保存的ViewHolder是以itemType来区分的。这样方便不同的itemType保存不同的ViewHolder。它在回收的时候只是回收该viewType的ViewHolder对象，并没有保存原来的数据信息，在复用的时候需要重新走onBindViewHolder()方法重新绑定数据。

我们来看看RecycledViewPool的结构：

public static class RecycledViewPool {

private static final int DEFAULT\_MAX\_SCRAP = 5;

static class ScrapData {

final ArrayList<ViewHolder> mScrapHeap = new ArrayList<>();

int mMaxScrap = DEFAULT\_MAX\_SCRAP;

}

SparseArray<ScrapData> mScrap = new SparseArray<>();

}

1

2

3

4

5

6

7

8

可以看出，RecycledViewPool中定义了SparseArray<ScrapData> mScrap，它是一个根据不同itemType来保存静态类ScrapData对象的SparseArray，ScrapData中包含了ArrayList<ViewHolder> mScrapHeap ，mScrapHeap是保存该itemType类型下ViewHolder的ArrayList。

缓存池定义了默认的缓存大小DEFAULT\_MAX\_SCRAP = 5，这个数量不是说整个缓存池只能缓存这多个ViewHolder，而是不同itemType的ViewHolder的list的缓存数量，即mScrap的数量，说明最多只有5组不同类型的mScrapHeap。mMaxScrap = DEFAULT\_MAX\_SCRAP说明每种不同类型的ViewHolder默认保存5个，当然mMaxScrap的值是可以设置的。这样RecycledViewPool就把不同ViewType的ViewHolder按类型分类缓存起来。

其实，Scrap缓存池不参与滚动的回收复用，CacheView缓存池被称为一级缓存，又因为ViewCacheExtension缓存池是给开发者定义的缓存池，一般不用到，所以RecycledViewPool缓存池被称为二级缓存，那么这样来说实际只有两层缓存。

三、源码解析(回收和复用)

  单单看上面的解释可能比较抽象、生硬，不明白这段话所表达的意思。这里我们结合源码来分析一下RecyclerView的回收复用流程，跟着源码走你会明白RecyclerView的缓存整体结构。以LinearLayoutManager为例，在RecyclerView<6>对RecyclerView的布局流程进行了分析，但是没有涉及到RecyclerView的回收复用机制，我们知道RecyclerView的布局和回收复用都是在RecyclerView.LayoutManager处理的。

温馨提示：本文源码基于androidx.recyclerview:recyclerview:1.2.0-alpha01

3.1 回收流程

在LinearLayoutManager中，来到itemView布局入口的方法onLayoutChildren()：

@Override

public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {

if (mPendingSavedState != null || mPendingScrollPosition != RecyclerView.NO\_POSITION) {

if (state.getItemCount() == 0) {

removeAndRecycleAllViews(recycler);//移除所有子View

return;

}

}

ensureLayoutState();

mLayoutState.mRecycle = false;//禁止回收

//颠倒绘制布局

resolveShouldLayoutReverse();

onAnchorReady(recycler, state, mAnchorInfo, firstLayoutDirection);

//暂时分离已经附加的view，即将所有child detach并通过Scrap回收

detachAndScrapAttachedViews(recycler);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

在onLayoutChildren()布局的时候，先根据实际情况是否需要removeAndRecycleAllViews()移除所有的子View，那些ViewHolder不可用；然后通过detachAndScrapAttachedViews()暂时分离已经附加的ItemView，缓存到List中。

试想我们插入了item或者删除了item亦或者打乱了列表的顺序，怎么重新布局这些item呢？如何将屏幕上现有的item布局到新的位置呢？最简单的方法就是把每个item从屏幕中分离下来，保存着，然后按照位置要求重新排列上去。

detachAndScrapAttachedViews()的作用就是把当前屏幕所有的item与屏幕分离，将他们从RecyclerView的布局中拿下来，保存到list中，在重新布局时，再将ViewHolder重新一个个放到新的位置上去。将屏幕上的ViewHolder从RecyclerView的布局中拿下来后，存放在Scrap中，Scrap包括mAttachedScrap和mChangedScrap，它们是一个list，用来保存从RecyclerView布局中拿下来ViewHolder列表，detachAndScrapAttachedViews()只会在onLayoutChildren()中调用，只有在布局的时候，才会把ViewHolder detach掉，然后再add进来重新布局，但是大家需要注意，Scrap只是保存从RecyclerView布局中当前屏幕显示的item的ViewHolder，不参与回收复用，单纯是为了现从RecyclerView中拿下来再重新布局上去。对于没有保存到的item，会放到mCachedViews或者RecycledViewPool缓存中参与回收复用。

public void detachAndScrapAttachedViews(@NonNull Recycler recycler) {

final int childCount = getChildCount();

for (int i = childCount - 1; i >= 0; i--) {

final View v = getChildAt(i);

scrapOrRecycleView(recycler, i, v);

}

}

1

2

3

4

5

6

7

遍历所有view，分离所有已经添加到RecyclerView的itemView，Recycler先废弃它们，然后再在缓存列表中拿出来复用。

private void scrapOrRecycleView(Recycler recycler, int index, View view) {

final ViewHolder viewHolder = getChildViewHolderInt(view);

if (viewHolder.isInvalid() && !viewHolder.isRemoved()

&& !mRecyclerView.mAdapter.hasStableIds()) {

removeViewAt(index);//移除VIew

recycler.recycleViewHolderInternal(viewHolder);//缓存到CacheView或者RecycledViewPool中

} else {

detachViewAt(index);//分离View

recycler.scrapView(view);//scrap缓存

mRecyclerView.mViewInfoStore.onViewDetached(viewHolder);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

进入else分支，可以看到先detachViewAt()分离视图，然后再通过scrapView()缓存到scrap中：

void scrapView(View view) {

final ViewHolder holder = getChildViewHolderInt(view);

if (holder.hasAnyOfTheFlags(ViewHolder.FLAG\_REMOVED | ViewHolder.FLAG\_INVALID)

|| !holder.isUpdated() || canReuseUpdatedViewHolder(holder)) {

holder.setScrapContainer(this, false);

mAttachedScrap.add(holder);//保存到mAttachedScrap中

} else {

if (mChangedScrap == null) {

mChangedScrap = new ArrayList<ViewHolder>();

}

holder.setScrapContainer(this, true);

mChangedScrap.add(holder);//保存到mChangedScrap中

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

进入if()分支的ViewHolder保存到mAttachedScrap中，else分支的保存到mChangedScrap中。

回到scrapOrRecycleView()中，进入if()分支如果viewHolder是无效、未被移除、未被标记的则放到recycleViewHolderInternal()缓存起来，同时removeViewAt()移除了viewHolder，

void recycleViewHolderInternal(ViewHolder holder) {

·····

if (forceRecycle || holder.isRecyclable()) {

if (mViewCacheMax > 0

&& !holder.hasAnyOfTheFlags(ViewHolder.FLAG\_INVALID

| ViewHolder.FLAG\_REMOVED

| ViewHolder.FLAG\_UPDATE

| ViewHolder.FLAG\_ADAPTER\_POSITION\_UNKNOWN)) {

int cachedViewSize = mCachedViews.size();

if (cachedViewSize >= mViewCacheMax && cachedViewSize > 0) {//如果超出容量限制，把第一个移除

recycleCachedViewAt(0);

cachedViewSize--;

}

·····

mCachedViews.add(targetCacheIndex, holder);//mCachedViews回收

cached = true;

}

if (!cached) {

addViewHolderToRecycledViewPool(holder, true);//放到RecycledViewPool回收

recycled = true;

}

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

如果符合条件，会优先缓存到mCachedViews中时，如果超出了mCachedViews的最大限制，通过recycleCachedViewAt()将CacheView缓存的第一个数据添加到终极回收池RecycledViewPool后再移除掉，最后才会add()新的ViewHolder添加到mCachedViews中。

剩下不符合条件的则通过addViewHolderToRecycledViewPool()缓存到RecycledViewPool中。

void addViewHolderToRecycledViewPool(@NonNull ViewHolder holder, boolean dispatchRecycled) {

clearNestedRecyclerViewIfNotNested(holder);

View itemView = holder.itemView;

······

holder.mOwnerRecyclerView = null;

getRecycledViewPool().putRecycledView(holder);//将holder添加到RecycledViewPool中

}

1

2

3

4

5

6

7

还有一个就是在填充布局fill()的时候，它会回收移出屏幕的view到mCachedViews或者RecycledViewPool中：

int fill(RecyclerView.Recycler recycler, LayoutState layoutState,

RecyclerView.State state, boolean stopOnFocusable) {

if (layoutState.mScrollingOffset != LayoutState.SCROLLING\_OFFSET\_NaN) {

recycleByLayoutState(recycler, layoutState);//回收移出屏幕的view

}

}

1

2

3

4

5

6

在recycleByLayoutState()层层追查下去，会来到recycler.recycleView(view)Recycler的公共回收方法中，：

public void recycleView(@NonNull View view) {

ViewHolder holder = getChildViewHolderInt(view);

if (holder.isTmpDetached()) {

removeDetachedView(view, false);

}

recycleViewHolderInternal(holder);

}

1

2

3

4

5

6

7

回收分离的视图到缓存池中，方便以后重新绑定和复用，这里又来到了recycleViewHolderInternal(holder)，和上面的一样，按照优先级缓存 mCachedViews > RecycledViewPool。

那么回收流程就到这里结束了。

3.2 复用流程

itemView的回收流程分析完了，那么这些回收的ViewHolder到底在什么时候，什么地方拿出来使用呢？回到LinearLayoutManager的布局入口的方法onLayoutChildren()：

@Override

public void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state) {

if (mPendingSavedState != null || mPendingScrollPosition != RecyclerView.NO\_POSITION) {

if (state.getItemCount() == 0) {

removeAndRecycleAllViews(recycler);//移除所有子View

return;

}

}

//暂时分离已经附加的view，即将所有child detach并通过Scrap回收

detachAndScrapAttachedViews(recycler);

if (mAnchorInfo.mLayoutFromEnd) {

//描点位置从start位置开始填充ItemView布局

updateLayoutStateToFillStart(mAnchorInfo);

fill(recycler, mLayoutState, state, false);//填充所有itemView

//描点位置从end位置开始填充ItemView布局

updateLayoutStateToFillEnd(mAnchorInfo);

fill(recycler, mLayoutState, state, false);//填充所有itemView

endOffset = mLayoutState.mOffset;

}else {

//描点位置从end位置开始填充ItemView布局

updateLayoutStateToFillEnd(mAnchorInfo);

fill(recycler, mLayoutState, state, false);

//描点位置从start位置开始填充ItemView布局

updateLayoutStateToFillStart(mAnchorInfo);

fill(recycler, mLayoutState, state, false);

startOffset = mLayoutState.mOffset;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

回收view后，紧接着就是填充view了，上面提到，在重新布局的时候会临时将view缓存起来，再一个个把ViewHolder按照正确的位置填充上去。fill()就是填充由layoutState定义的给定布局：

int fill(RecyclerView.Recycler recycler, LayoutState layoutState, RecyclerView.State state, boolean stopOnFocusable) {

recycleByLayoutState(recycler, layoutState);//回收滑出屏幕的view

while ((layoutState.mInfinite || remainingSpace > 0) && layoutState.hasMore(state)) {//一直循环，知道没有数据

layoutChunkResult.resetInternal();

layoutChunk(recycler, state, layoutState, layoutChunkResult);//添加一个child

······

if (layoutChunkResult.mFinished) {//布局结束，退出循环

break;

}

layoutState.mOffset += layoutChunkResult.mConsumed \* layoutState.mLayoutDirection;//根据添加的child高度偏移计算

}

······

return start - layoutState.mAvailable;//返回这次填充的区域大小

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

判断当前可见区域还有没有剩余空间，如果有则填充view上去，核心是通过while()循环执行layoutChunk()填充一个itemView到屏幕， layoutChunk()完成布局工作：

void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State state,

LayoutState layoutState, LayoutChunkResult result) {

View view = layoutState.next(recycler);//获取复用的view

······

}

1

2

3

4

5

该方法通过layoutState.next(recycler)拿到视图，我们看看它是怎么拿到视图的：

View next(RecyclerView.Recycler recycler) {

if (mScrapList != null) {

return nextViewFromScrapList();

}

final View view = recycler.getViewForPosition(mCurrentPosition);

mCurrentPosition += mItemDirection;

return view;

}

@NonNull

public View getViewForPosition(int position) {

return getViewForPosition(position, false);

}

View getViewForPosition(int position, boolean dryRun) {

return tryGetViewHolderForPositionByDeadline(position, dryRun, FOREVER\_NS).itemView;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

tryGetViewHolderForPositionByDeadline()才是获取view的方法，它会根据给出的position/id去scrap、cache、RecycledViewPool、或者创建获取一个ViewHolder：

@Nullable

ViewHolder tryGetViewHolderForPositionByDeadline(int position, boolean dryRun, long deadlineNs) {

ViewHolder holder = null;

// 0) 如果它是改变的废弃的ViewHolder，在scrap的mChangedScrap找

if (mState.isPreLayout()) {

holder = getChangedScrapViewForPosition(position);

fromScrapOrHiddenOrCache = holder != null;

}

// 1)根据position分别在scrap的mAttachedScrap、mChildHelper、mCachedViews中查找

if (holder == null) {

holder = getScrapOrHiddenOrCachedHolderForPosition(position, dryRun);

}

if (holder == null) {

final int type = mAdapter.getItemViewType(offsetPosition);

// 2)根据id在scrap的mAttachedScrap、mCachedViews中查找

if (mAdapter.hasStableIds()) {

holder = getScrapOrCachedViewForId(mAdapter.getItemId(offsetPosition), type, dryRun);

}

if (holder == null && mViewCacheExtension != null) {

//3)在ViewCacheExtension中查找，一般不用到，所以没有缓存

final View view = mViewCacheExtension

.getViewForPositionAndType(this, position, type);

if (view != null) {

holder = getChildViewHolder(view);

}

}

//4)在RecycledViewPool中查找

holder = getRecycledViewPool().getRecycledView(type);

if (holder != null) {

holder.resetInternal();

if (FORCE\_INVALIDATE\_DISPLAY\_LIST) {

invalidateDisplayListInt(holder);

}

}

}

//5)到最后如果还没有找到复用的ViewHolder，则新建一个

holder = mAdapter.createViewHolder(RecyclerView.this, type);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

这个方法确实做了不少事情，分别去scrap、CacheView、ViewCacheExtension、RecycledViewPool中获取ViewHolder，如果没有则创建一个新的ViewHolder返回，我们一步步来分析：

（1）第一步：如果是废弃的发生改变的ViewHolder，则在scrap的mChangedScrap查找视图，通过position和id分别查找；

这个一般在我们调用adapter的notifyItemChanged()方法时，数据发生变化，item缓存在mChangedScrap中，后续拿到的ViewHolder需要重新绑定数据。

ViewHolder getChangedScrapViewForPosition(int position) {

//通过position

for (int i = 0; i < changedScrapSize; i++) {

final ViewHolder holder = mChangedScrap.get(i);

return holder;

}

// 通过id

if (mAdapter.hasStableIds()) {

final long id = mAdapter.getItemId(offsetPosition);

for (int i = 0; i < changedScrapSize; i++) {

final ViewHolder holder = mChangedScrap.get(i);

return holder;

}

}

return null;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

（2）第二步：如果没有找到视图，根据position分别在scrap的mAttachedScrap、mChildHelper、mCachedViews中查找。在getScrapOrHiddenOrCachedHolderForPosition(position, dryRun)这个方法按照以下顺序查找：

首先从mAttachedScrap中查找，精准匹配有效的ViewHolder；

接着在mChildHelper中mHiddenViews查找隐藏的ViewHolder；

最后从我们的一级缓存中mCachedViews查找。

//根据position分别在scrap的mAttachedScrap、mChildHelper、mCachedViews中查找

ViewHolder getScrapOrHiddenOrCachedHolderForPosition(int position, boolean dryRun) {

final int scrapCount = mAttachedScrap.size();

// 首先从mAttachedScrap中查找，精准匹配有效的ViewHolder

for (int i = 0; i < scrapCount; i++) {

final ViewHolder holder = mAttachedScrap.get(i);

return holder;

}

//接着在mChildHelper中mHiddenViews查找隐藏的ViewHolder

if (!dryRun) {

View view = mChildHelper.findHiddenNonRemovedView(position);

if (view != null) {

final ViewHolder vh = getChildViewHolderInt(view);

scrapView(view);

return vh;

}

}

//最后从我们的一级缓存中mCachedViews查找。

final int cacheSize = mCachedViews.size();

for (int i = 0; i < cacheSize; i++) {

final ViewHolder holder = mCachedViews.get(i);

return holder;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

（3）第三步：如果没有找到视图，通过id在scrap的mAttachedScrap、mCachedViews中查找。在getScrapOrCachedViewForId()这个方法按照以下顺序：

首先从mAttachedScrap中查找，精准匹配有效的ViewHolder；

接着从我们的一级缓存中mCachedViews查找；

注意：这一步是跟id来查找的，与上一步根据position查找类似。

ViewHolder getScrapOrCachedViewForId(long id, int type, boolean dryRun) {

//在Scrap的mAttachedScrap中查找

final int count = mAttachedScrap.size();

for (int i = count - 1; i >= 0; i--) {

final ViewHolder holder = mAttachedScrap.get(i);

return holder;

}

//在一级缓存mCachedViews中查找

final int cacheSize = mCachedViews.size();

for (int i = cacheSize - 1; i >= 0; i--) {

final ViewHolder holder = mCachedViews.get(i);

return holder;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

（4）第四步：在mViewCacheExtension中查找，前面提到这个缓存池是由开发者定义的一层缓存策略，Recycler并没有将任何view缓存到这里。这里没有定义过，所有找不到对应的view。

if (holder == null && mViewCacheExtension != null) {

final View view = mViewCacheExtension.getViewForPositionAndType(this, position, type);

if (view != null) {

holder = getChildViewHolder(view);

}

}

1

2

3

4

5

6

（5）第五步：从RecycledViewPool中查找，上面讲过RecycledViewPool是通过itemType把ViewHolder的List缓存到SparseArray中的，在getRecycledViewPool().getRecycledView(type)根据itemType从SparseArray获取ScrapData ，然后再从里面获取ArrayList<ViewHolder>，从而获取到ViewHolder。

@Nullable

public ViewHolder getRecycledView(int viewType) {

final ScrapData scrapData = mScrap.get(viewType);//根据viewType获取对应的ScrapData

if (scrapData != null && !scrapData.mScrapHeap.isEmpty()) {

final ArrayList<ViewHolder> scrapHeap = scrapData.mScrapHeap;

for (int i = scrapHeap.size() - 1; i >= 0; i--) {

if (!scrapHeap.get(i).isAttachedToTransitionOverlay()) {

return scrapHeap.remove(i);

}

}

}

return null;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

（6）第六步：如果还没有获取到ViewHolder，则通过mAdapter.createViewHolder()创建一个新的ViewHolder返回。

//5)到最后如果还没有找到复用的ViewHolder，则新建一个

holder = mAdapter.createViewHolder(RecyclerView.this, type);

1

2

那么复用流程到这里也完毕了。

四、RecyclerVIew的回收复用原理

4.1 RecyclerVIew的回收原理

在RecyclerView重新布局onLayoutChildren()或者填充布局fill()的时候，会先把必要的item与屏幕分离或者移除，并做好标记，保存到list中，在重新布局时，再将ViewHolde拿出来重新一个个放到新的位置上去。

（1）如果是RecyclerView不滚动情况下缓存(比如删除item)，重新布局时，把屏幕上的ViewHolder与屏幕分离下来，存放到Scrap中，即发生改变的ViewHolder缓存到mChangedScrap中，不发生改变的ViewHolder存放到mAttachedScrap中；剩下ViewHolder的会按照mCachedViews>RecycledViewPool的优先级缓存到mCachedViews或者RecycledViewPool中。

（2）如果是RecyclerVIew滚动情况下缓存(比如滑动列表)，在滑动时填充布局，先移除滑出屏幕的item，第一级缓存mCachedViews优先缓存这些ViewHolder，但是mCachedViews最大容量为2，当mCachedViews满了以后，会利用先进先出原则，把旧的ViewHolder存放到RecycledViewPool中后移除掉，腾出空间，再将新的ViewHolder添加到mCachedViews中，最后剩下的ViewHolder都会缓存到终极回收池RecycledViewPool中，它是根据itemType来缓存不同类型的ArrayList<ViewHolder>，最大容量为5。

4.2 RecyclerVIew的复用原理

至此，已经有五个缓存RecyclerView的池子，mChangedScrap、mAttachedScrap、mCachedViews、mViewCacheExtension、mRecyclerPool，除了mViewCacheExtension是系统提供给开发者拓展的没有用到之外，还有四个池子是参与到复用流程中的。

当RecyclerView要拿一个复用的ViewHolder时，如果是预加载，则会先去mChangedScrap中精准查找(分别根据position和id)对应的ViewHolder，如果有就返回，如果没有就再去mAttachedScrap和mCachedViews中精确查找(先position后id)是不是原来的ViewHolder，如果是说明ViewHolder是刚刚被移除的，如果不是，则最终去mRecyclerPool找，如果itemType类型匹配对应的ViewHolder，那么返回实例，让它重新绑定数据，如果mRecyclerPool也没有返回ViewHolder才会调用createViewHolder()重新去创建一个。

这里需要注意：在mChangedScrap、mAttachedScrap、mCachedViews中拿到的ViewHolder都是精准匹配，但是mChangedScrap的是发生了变化的，需要调用onBindViewHolder()重新绑定数据，mAttachedScrap和mCachedViews没有发生变化，是直接使用的，不需要重新绑定数据，而mRecyclerPool中的ViewHolder的内容信息已经被抹除，需要重新绑定数据。所以在RecyclerView来回滚动时，mCachedViews缓存池的使用效率最高。

总的来说：RecyclerView着重在两个场景缓存和回收的优化，一是：在数据更新时，使用Scrap进行局部更新，尽可能复用原来viewHolder，减少绑定数据的工作；二是：在滑动的时候，重复利用原来的ViewHolder，尽可能减少重复创建ViewHolder和绑定数据的工作。最终思想就是，能不创建就不创建，能不重新绑定就不重新绑定，尽可能减少重复不必要的工作。

整个过程大致如下：

# 参考文档

https://blog.csdn.net/m0\_37796683/article/details/103697121

式样

微软雅黑 Light 5号

段落 固定值 23磅

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5