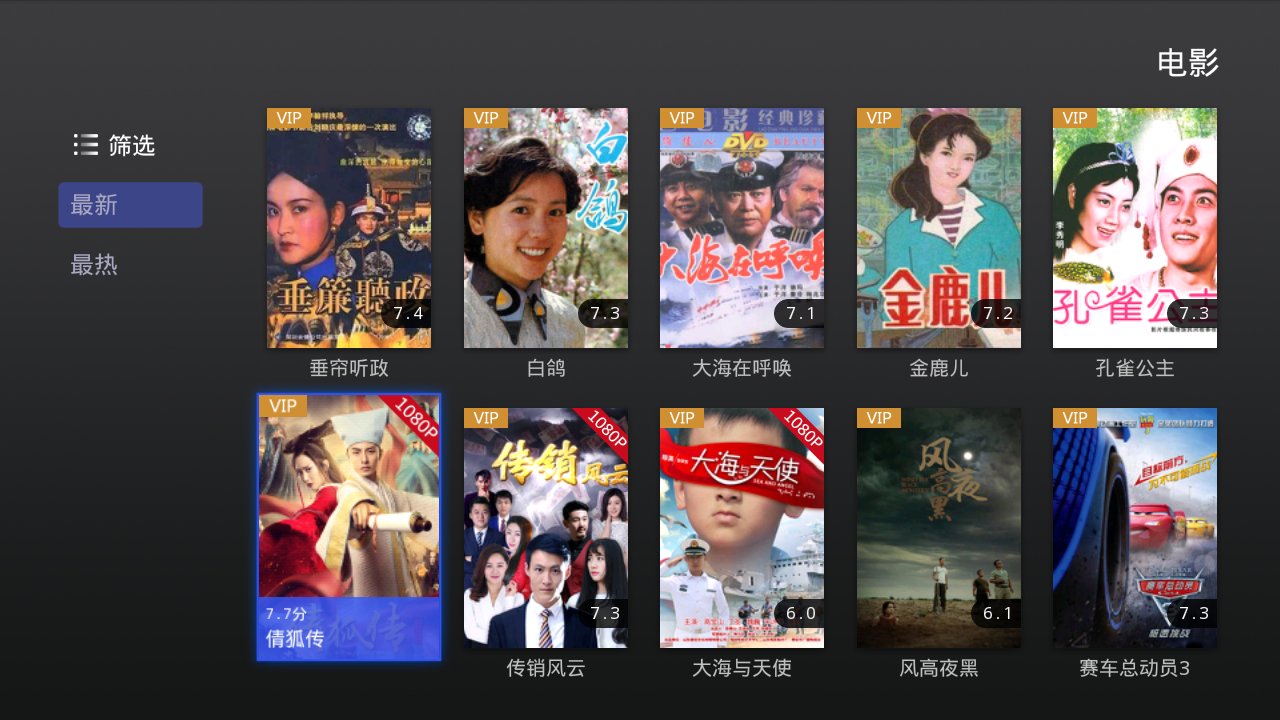
# RecyclerView的回收复用机制

RecyclerView的回收复用机制的内部实现都是由Recycler内部类实现，下面会以频道页的使用场景来讲解RecyclerView的回收复用机制，所以并不会涉及所有的回收复用机制的实现逻辑，但讲解的是经典的在RecyclerView滑动过程中的回收复用的场景。



相应的版本：

RecyclerView: recyclerview-v7-25.1.0.jar

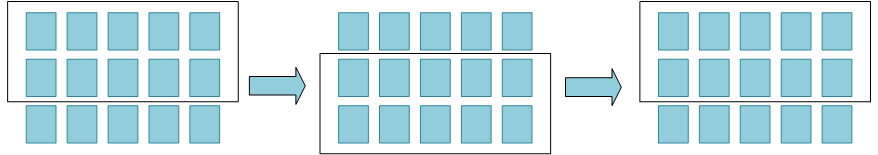
LayoutManager: GridLayoutManager extends LinearLayoutManager (recyclerview-v7-25.1.0.jar)

频道页每行可显示5个卡位，每个卡位的item布局type一致。

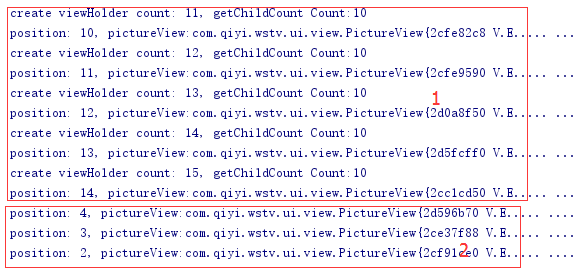
开始分析回收复用机制之前，先提几个问题：

**Q1:如果向下滑动，新一行的5个卡位的显示会去复用缓存的ViewHolder，第一行的5个卡位会移出屏幕被回收，那么在这个过程中，是先进行复用再回收？还是先回收再复用？还是边回收边复用？也就是说，新一行的5个卡位复用的ViewHolder有可能是第一行被回收的5个卡位吗？**

第二个问题之前，先看几张图片：



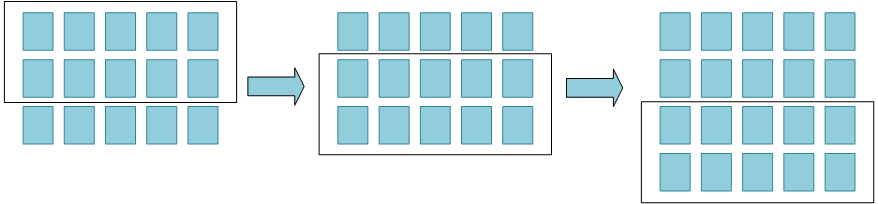
黑框表示屏幕，RecyclerView先向下滑动，第三行卡位显示出来，再向上滑动，第三行移出屏幕，第一行显示出来。我们分别在Adapter的onCreateViewHolder() 和 onBindViewHolder() 里打日志，下面是这个过程的日志：



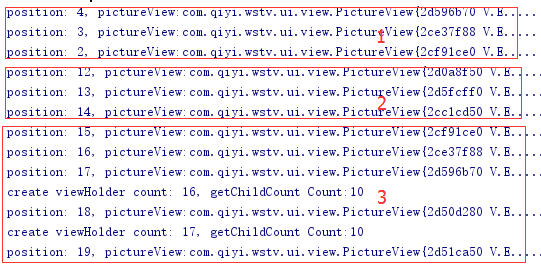
红框1是RecyclerView向下滑动操作的日志，第三行5个卡位的显示都是重新创建的ViewHolder；红框2是再次向上滑动时的日志，第一行5个卡位的重新显示用的ViewHolder都是复用的，因为没有create viewHolder的日志，然后只有后面3个卡位重新绑定数据，调用了onBindViewHolder()；那么问题来了：

**Q2: 在这个过程中，为什么当RecyclerView再次向上滑动重新显示第一行的5个卡位时，只有后面3个卡位触发了onBindViewHolder() 方法，重新绑定数据呢？明明5个卡位都是复用的。**

在上面的操作基础上，我们继续往下操作：



在第二个问题操作的基础上，目前已经创建了15个ViewHolder，此时显示的是第1、2行的卡位，那么继续向下滑动两次，这个过程的日志如下：



红框1是第二个问题操作的日志，在这里截出来只是为了显示接下去的日志是在上面的基础上继续操作的；

红框2就是第一次向下滑时的日志，对比问题2的日志，这次第三行的5个卡位用的ViewHolder也都是复用的，而且也只有后面3个卡位触发了onBindViewHolder() 重新绑定数据；

红框3是第二次向下滑动时的日志，这次第四行的5个卡位，前3个的卡位用的ViewHolder是复用的，后面2个卡位的ViewHolder则是重新创建的，而且5个卡位都调用了onBindViewHolder() 重新绑定数据；

那么，

**Q3：接下去不管是向上滑动还是向下滑动，滑动几次，都不会再有onCreateViewHolder() 的日志了，也就是说RecyclerView总共创建了17个ViewHolder，但有时一行的5个卡位只有3个卡位需要重新绑定数据，有时却又5个卡位都需要重新绑定数据，这是为什么呢？**

如果明白RecyclerView的回收复用机制，那么这三个问题也就都知道原因了；反过来，如果知道这三个问题的原因，那么理解RecyclerView的回收复用机制也就更简单了；所以，带着问题，在特定的场景下去分析源码的话，应该会比较容易。

## 源码分析

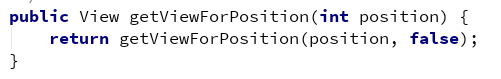
其实，根据问题2的日志，我们就可以回答问题1了。在目前显示1、2行，ViewHolder的个数为10个的基础上，第三行的5个新卡位要显示出来都需要重新创建ViewHolder，也就是说，在这个向下滑动的过程，是5个新卡位的复用机制先进行工作，然后第1行的5个被移出屏幕的卡位再进行回收机制工作。

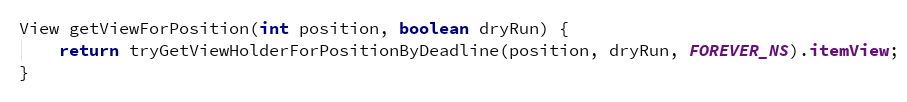
那么，就先来看看复用机制的源码：

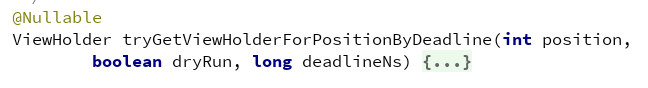
## 复用机制

### getViewForPosition()

这个方法是复用机制的入口，也就是Recycler开放给外部使用复用机制的api，外部调用这个方法就可以返回想要的View，而至于这个View是复用而来的，还是重新创建得来的，就都由Recycler内部实现，对外隐藏。



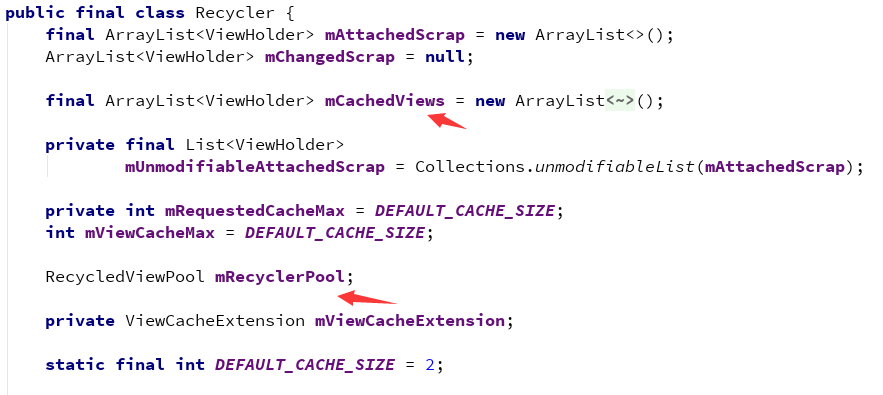




### tryGetViewHolderForPositionByDeadline()

所以，Recycler的复用机制内部实现就在这个方法里。

分析逻辑之前，先看一下Recycler的几个结构体，用来缓存ViewHolder的。



**mAttachedScrap:** 用于缓存显示在屏幕上的item的ViewHolder，场景好像是RecyclerView在onLayout时会先把children都移除掉，再重新添加进去，所以这个List应该是用在布局过程中临时存放children的，反正在RecyclerView滑动过程中不会在这里面来找复用的ViewHolder就是了。

**mChangedScrap：** 这个没理解是干嘛用的，看名字应该跟ViewHolder的数据发生变化时有关吧，在RecyclerView滑动的过程中，也没有发现到这里找复用的ViewHolder，所以这个可以先暂时放一边。

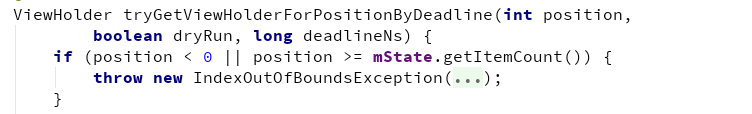
**mCachedViews：**这个就重要得多了，滑动过程中的回收和复用都是先处理的这个List，这个集合里存的ViewHolder的原本数据信息都在，所以可以直接添加到RecyclerView中显示，不需要再次重新onBindViewHolder()。

**mUnmodifiableAttachedScrap：** 不清楚干嘛用的，暂时跳过。

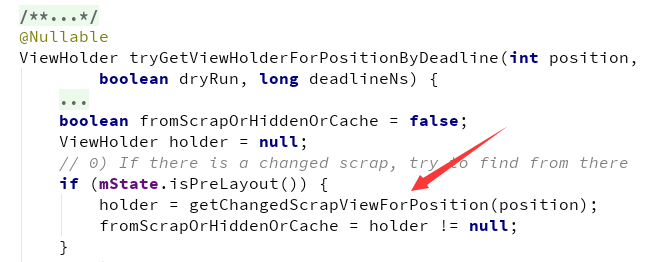
**mRecyclerPool：**这个也很重要，但存在这里的ViewHolder的数据信息会被重置掉，相当于ViewHolder是一个重创新建的一样，所以需要重新调用onBindViewHolder来绑定数据。

**mViewCacheExtension：**这个是留给我们自己扩展的，好像也没怎么用，就暂时不分析了。

那么接下去就看看复用的逻辑：

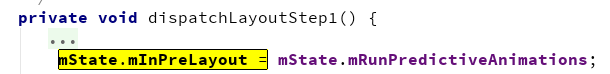


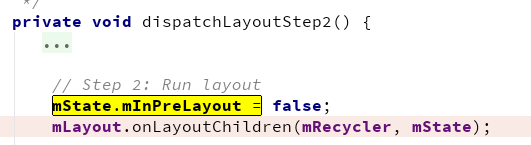
第一步很简单，position如果在item的范围之外的话，那就抛异常吧。继续往下看



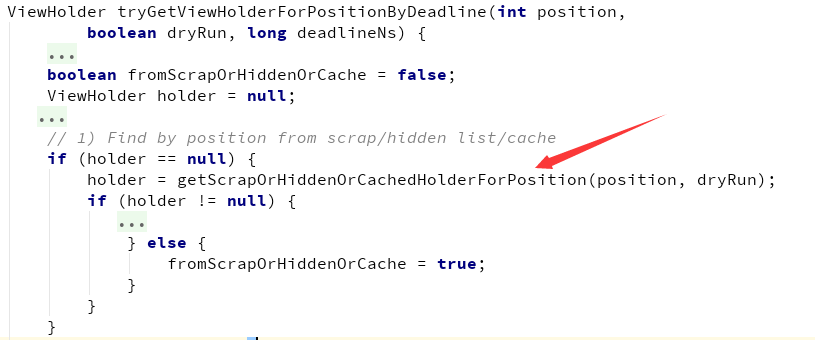
如果是在isPreLayout() 时，那么就去mChangedScrap中找。

那么这个isPreLayout表示的是什么？，有两个赋值的地方。

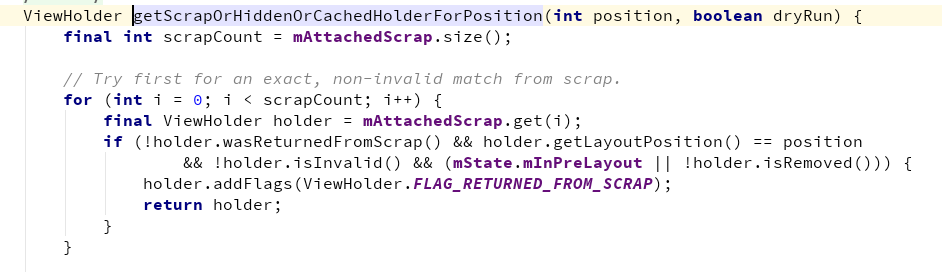




Emmm，看样子，在LayoutManager的onLayoutChildren前就会置为false，不过我还是不懂这个过程是干嘛的，滑动过程中好像mState.mInPreLayou = false，所以并不会来这里，先暂时跳过。继续往下。



跟进这个方法看看



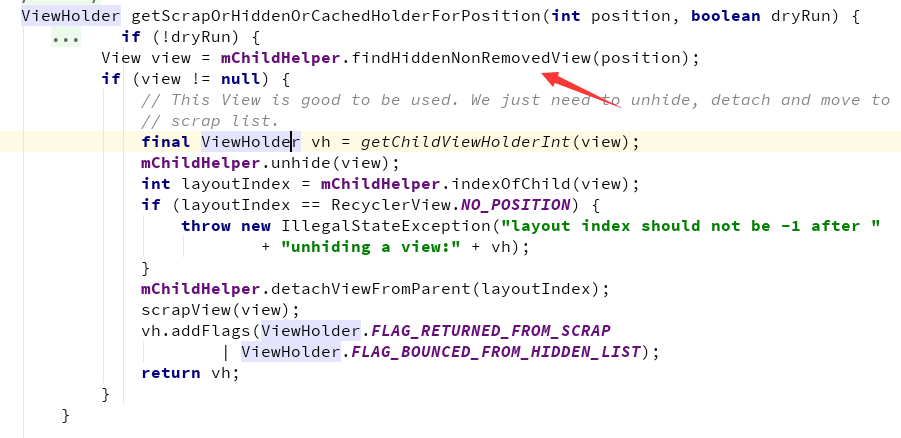
首先，去mAttachedScrap中寻找position一致的viewHolder，需要匹配一些条件，大致是这个viewHolder没有被移除，是有效的之类的条件，满足就返回这个viewHolder。

所以，这里的关键就是要理解这个mAttachedScrap到底是什么，存的是哪些ViewHolder。

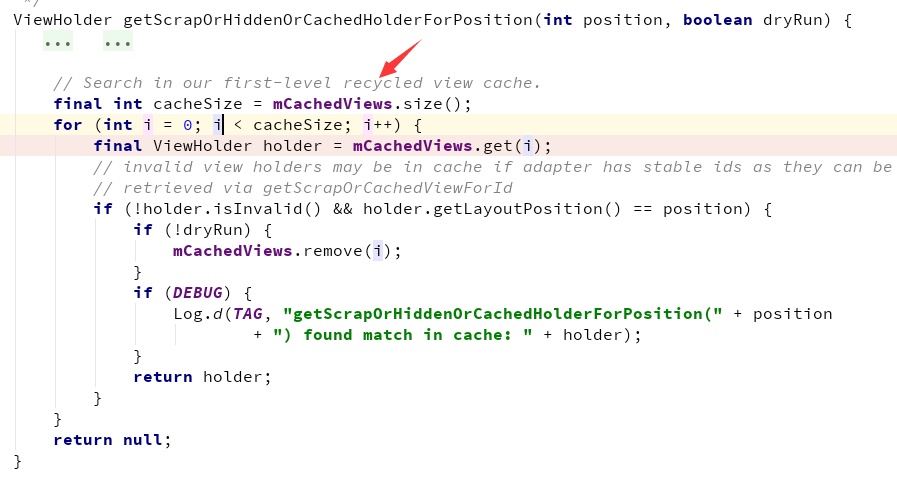
一次遥控器按键的操作，不管有没有发生滑动，都会导致RecyclerView的重新onLayout，那要layout的话，RecyclerView会先把所有children 先remove掉，然后再重新add上去，完成一次layout的过程。那么这暂时性的remove掉的viewHolder要存放在哪呢，就是放在这个mAttachedScrap中了，这就是我的理解了。

所以，感觉这个mAttachedScrap中存放的viewHolder跟回收和复用关系不大。

继续往下看

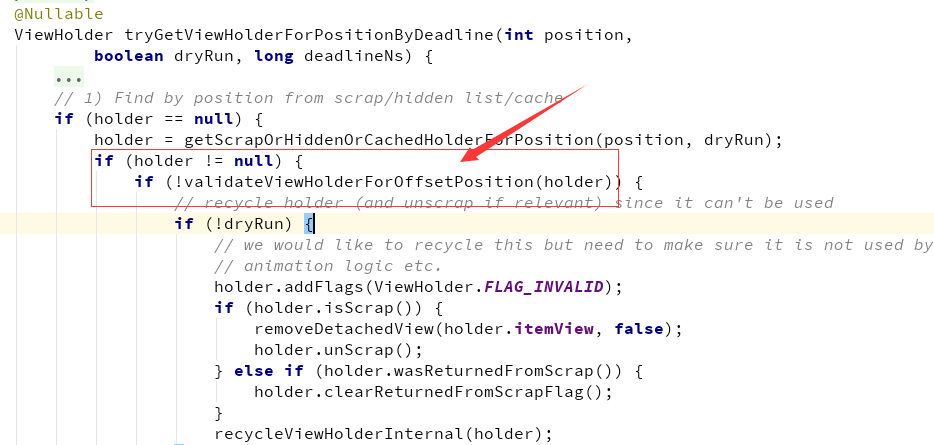


Emmm，这段也还是没看懂，但估计应该需要一些特定的场景下所使用的复用策略吧，看名字，应该跟hidden有关？不懂，跳过这段，应该也没事，滑动过程中的回收复用跟这个应该也关系不大。

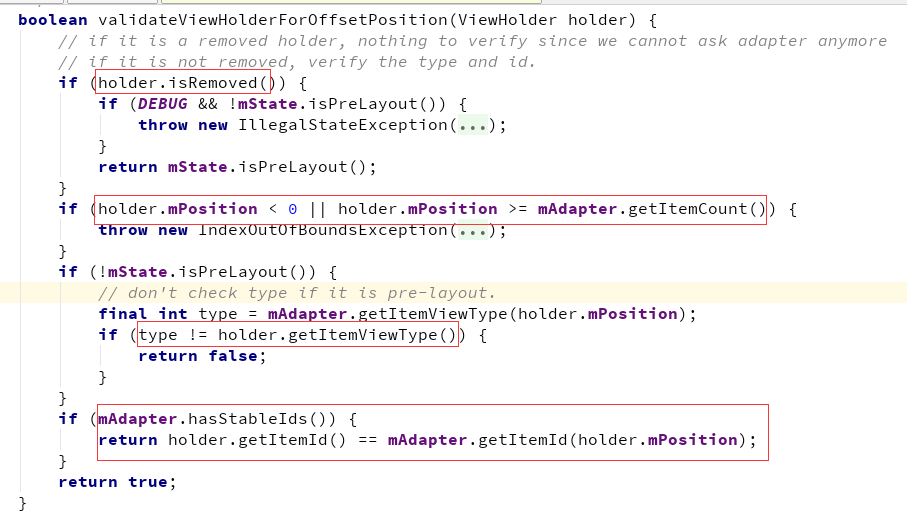


这个就是常说的RecyclerView的一级缓存了，mCachedViews的大小默认为2。遍历mCachedViews，找到position一致的ViewHolder，之前说过，mCachedViews里存放的ViewHolder的数据信息都保存着，所以mCachedViews可以理解成，只有原来的卡位可以重新复用这个ViewHolder，新位置的卡位无法从mCachedViews里拿ViewHolder出来用。

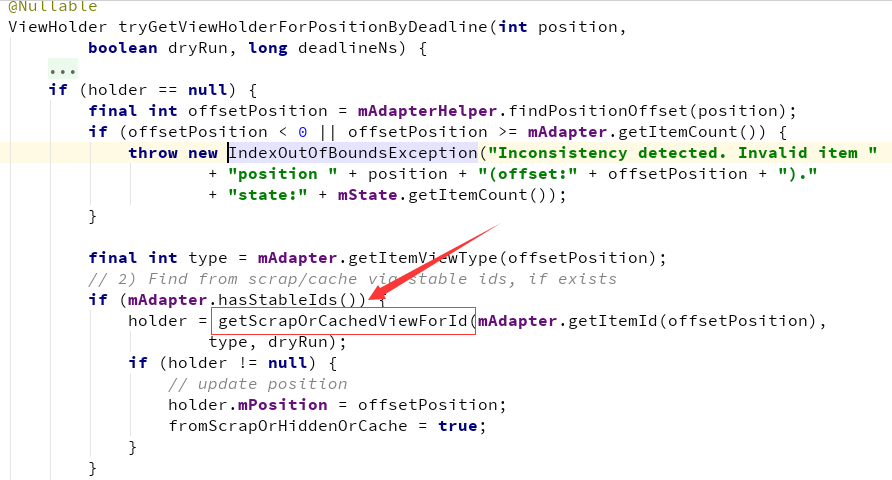
找到viewholder后



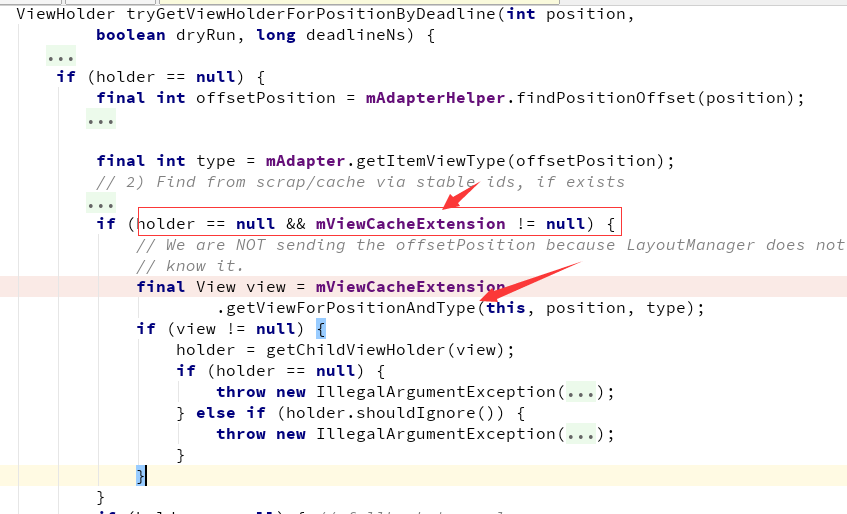
就算position匹配找到了ViewHolder，还需要判断一下这个ViewHolder是否已经被remove掉，type类型一致不一致。



以上是在mCachedViews中寻找，没有找到的话，就继续再找一遍，刚才是通过position来找，那这次就换成id，然后重复上面的步骤再找一遍，如下



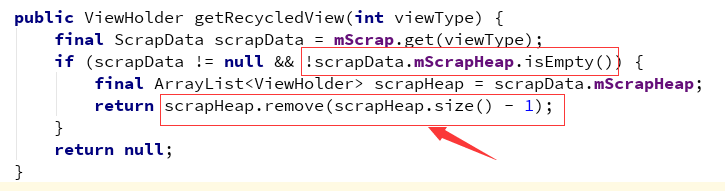
这个其实默认是不会执行的，除非我们重写了Adapter的setHasStableIds()，既然不是常用的场景，那就先略过吧，那就继续往下。



这个就是二级缓存了，RecyclerView提供给我们自定义实现的扩展类，我们可以重写getViewForPositionAndType()方法来实现自己的复用策略。不过，也没用过，那这部分也当作不会执行，略过。继续往下



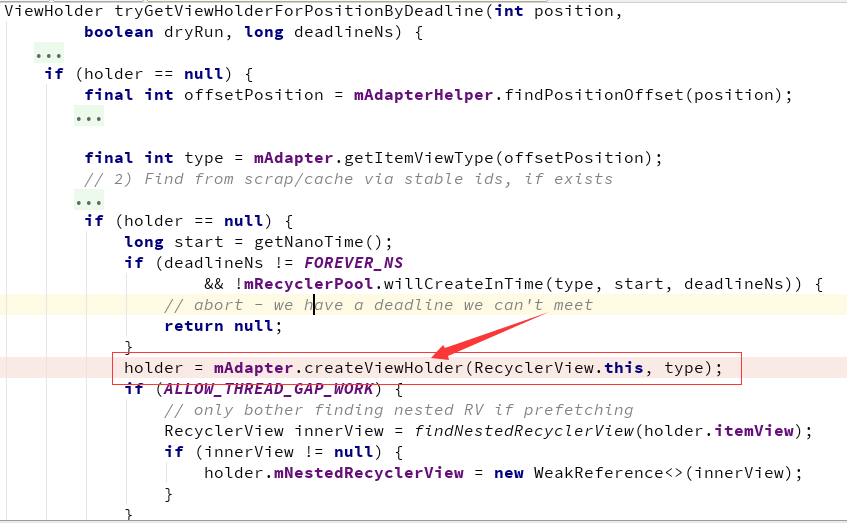
这里是三级缓存了，去RecyclerViewPool里取ViewHolder，ViewPool会根据不同的item type创建不同的List，每个List默认大小为5个。看一下去ViewPool里是怎么找的



之前说过，ViewPool会根据不同的viewType创建不同的集合来存放ViewHolder，那么复用的时候，只要ViewPool里相同的type有ViewHolder缓存的话，就将最后一个拿出来复用，不用像mCachedViews需要各种匹配条件，只要有就可以复用。

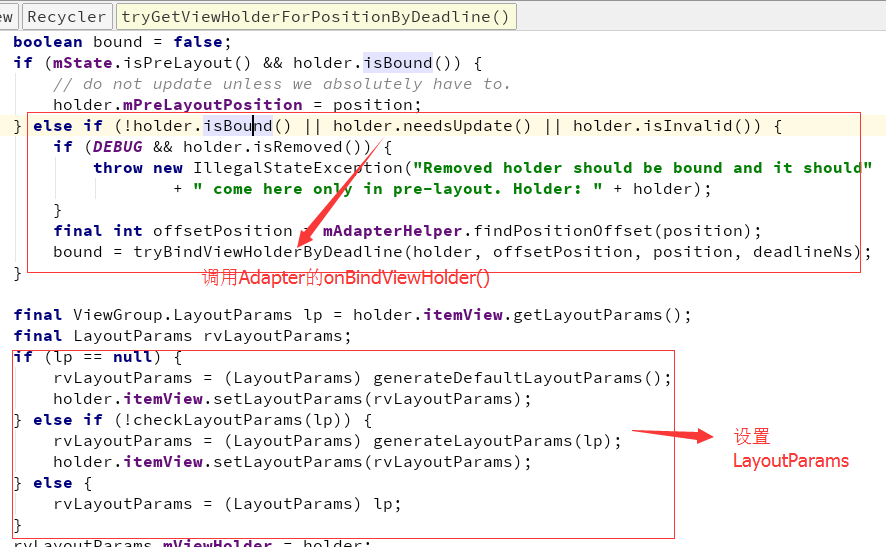
拿到ViewHolder之后，还会再次调用resetInternal() 来重置ViewHolder，这样ViewHolder就可以当作一个全新的ViewHolder来使用了，这也就是为什么从这里拿的ViewHolder都需要重新onBindViewHolder() 了。

那如果在ViewPool里还是没有找到呢，继续往下看



如果ViewPool中都没有找到ViewHolder来使用的话，那就调用Adapter的onCreateViewHolder来创建一个新的ViewHolder使用。

上面一共有很多步骤来找ViewHolder，不管在哪个步骤，只要找到ViewHolder的话，那下面那些步骤就不用管了，然后都要继续往下判断时候需要重新绑定数据，还有检查布局参数是否合法。如下：



到这里，tryGetViewHolderForPositionByDeadline() 这个方法就结束了。这大概就是RecyclerView的复用机制，中间我们跳过很多地方，因为RecyclerView有各种场景可以刷新他的view，比如重新setLayoutManager()，重新setAdapter()，或者notifyDataSetChanged()，或者滑动等等之类的场景，只要重新layout，就会去回收和复用ViewHolder，所以这个复用机制需要考虑到各种各样的场景。

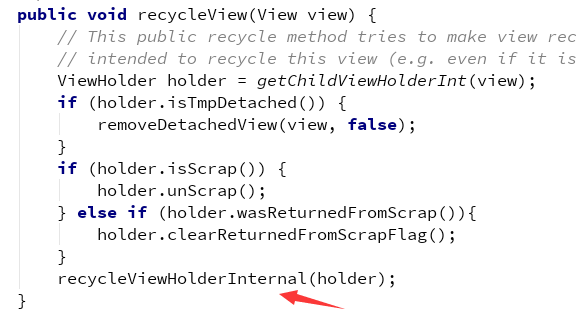
把代码一行行的啃透有点吃力，所以我就只借助RecyclerView的滑动的这种场景来分析它涉及到的回收和复用机制。

下面就分析一下回收机制

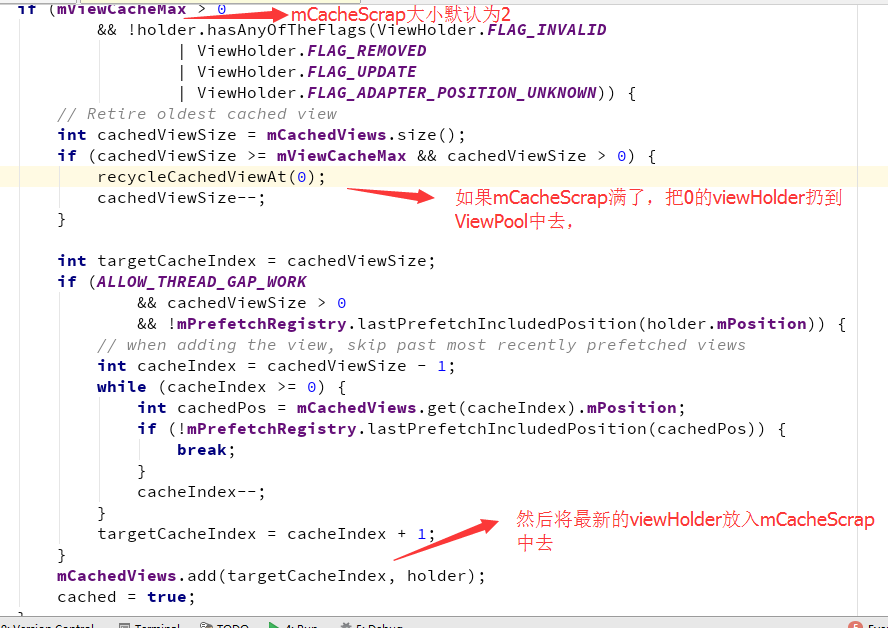
## 回收机制

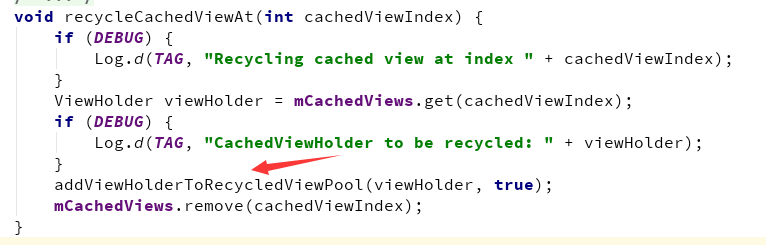
回收机制的入口就有很多了，因为Recycler有各种结构体，比如mAttachedScrap、mCachedViews等等，不同结构体回收的时机都不一样，入口也就多了。

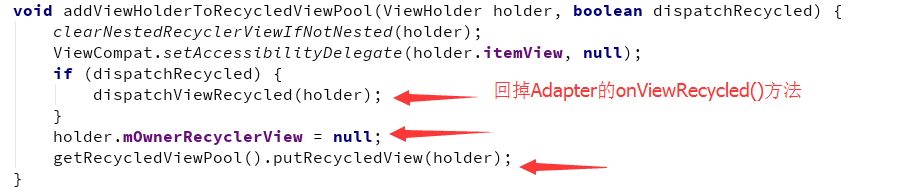
所以，还是基于RecyclerView的滑动场景下，移出屏幕的卡位回收时的入口是：

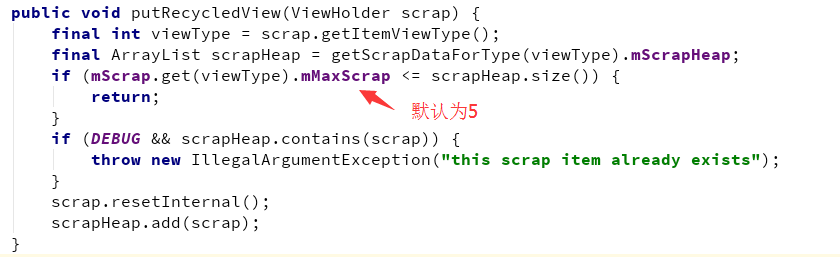












回收的逻辑比较简单，由LayoutManager来遍历移出屏幕的卡位，然后对每个卡位进行回收操作，回收时，都是把ViewHolder放在mCachedViews里面，如果mCachedViews满了，那就在mCachedViews里拿一个ViewHolder扔到ViewPool缓存里，然后mCachedViews就可以空出位置来放新回收的ViewHolder了。

总结一下：

RecyclerView滑动场景下的回收复用涉及到的结构体两个：

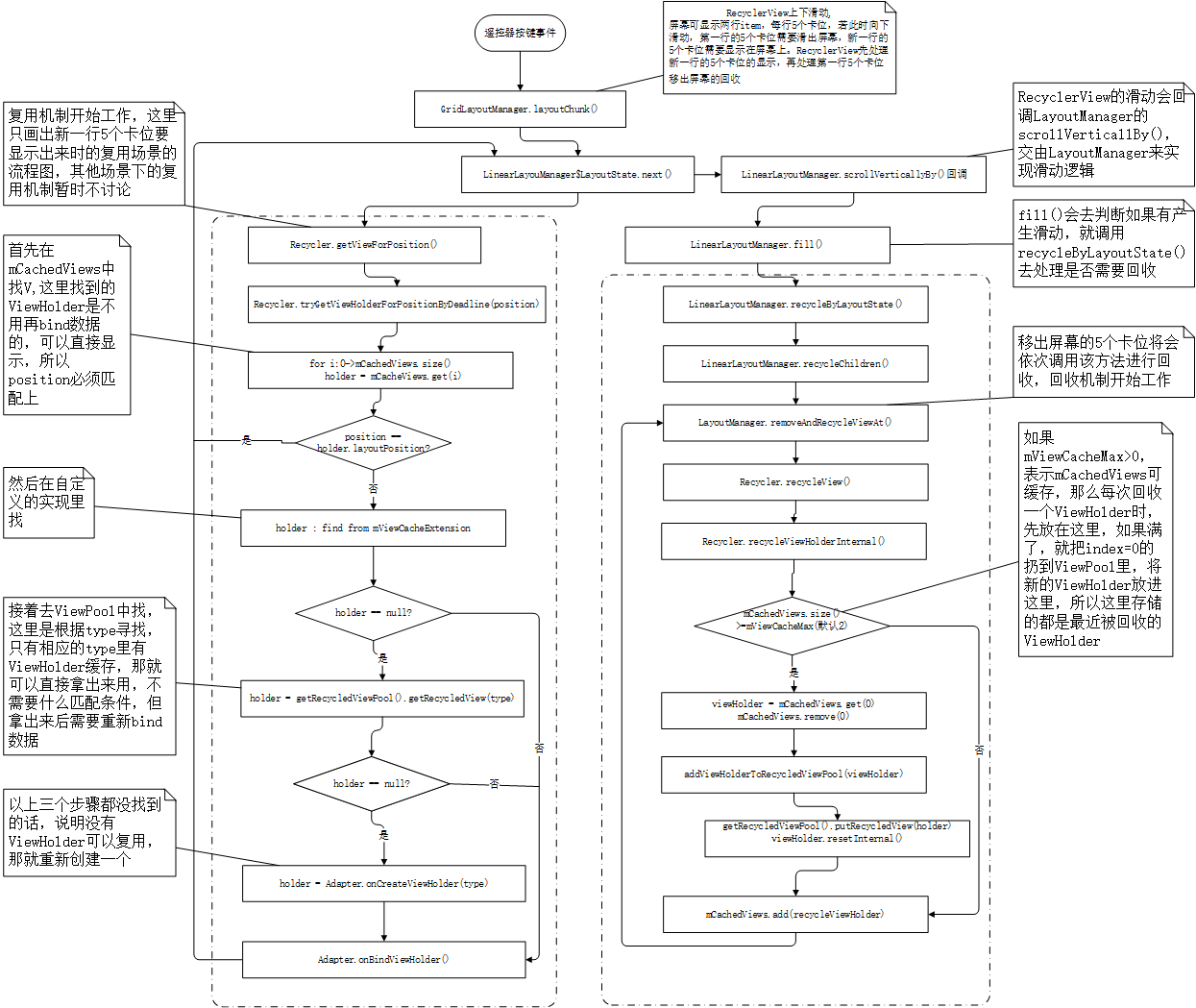
mCachedViews 和 RecyclerViewPool

mCachedViews优先级高于RecyclerViewPool，回收时，最新的ViewHolder都是往mCachedViews里放，如果它满了，那就移出一个扔到ViewPool里。

复用时，也是先到mCachedViews里找ViewHolder，但需要各种匹配条件，概括一下就是只有原来位置的卡位可以复用存在mCachedViews里的ViewHolder，如果mCachedViews里没有，那么才去ViewPool里找。

在ViewPool里的ViewHolder都是跟全新的ViewHolder一样，只要type一样，有找到，就可以拿出来复用，重新绑定下数据即可。

整体的流程图如下：（可放大查看）



最后，解释一下开头的问题

**Q1:如果向下滑动，新一行的5个卡位的显示会去复用缓存的ViewHolder，第一行的5个卡位会移出屏幕被回收，那么在这个过程中，是先进行复用再回收？还是先回收再复用？还是边回收边复用？也就是说，新一行的5个卡位复用的ViewHolder有可能是第一行被回收的5个卡位吗？**

答：先复用再回收，新一行的5个卡位先去目前的mCachedViews和ViewPool的缓存中寻找复用，没有就重新创建，然后移出屏幕的那行的5个卡位再回收缓存到mCachedViews和ViewPool里面，所以新一行5个卡位和复用不可能会用到刚移出屏幕的5个卡位。

**Q2: 在这个过程中，为什么当RecyclerView再次向上滑动重新显示第一行的5个卡位时，只有后面3个卡位触发了onBindViewHolder() 方法，重新绑定数据呢？明明5个卡位都是复用的。**

答：滑动场景下涉及到的回收和复用的结构体是mCachedViews和ViewPool，前者默认大小为2，后者为5。所以，当第三行显示出来后，第一行的5个卡位被回收，回收时先缓存在mCachedViews，满了再移出旧的到ViewPool里，所有5个卡位有2个缓存在mCachedViews里，3个缓存在ViewPool，至于是哪2个缓存在mCachedViews，这是由LayoutManager控制。

频道页使用的是GridLayoutManager，滑动时的回收逻辑则是在父类LinearLayoutManager里实现，回收第一行卡位时是从后往前回收，所以最新的两个卡位是0、1，会放在mCachedViews里，而2、3、4的卡位则放在ViewPool里。

所以，当再次向上滑动时，第一行5个卡位会去两个结构体里找复用，之前说过，mCachedViews里存放的ViewHolder只有原本位置的卡位才能复用，所以0、1两个卡位都可以直接去mCachedViews里拿ViewHolder复用，而且这里的ViewHolder是不用重新绑定数据的，至于2、3、4卡位则去ViewPool里找，刚好ViewPool里缓存着3个ViewHolder，所以第一行的5个卡位都是用的复用的，而从ViewPool里拿的复用需要重新绑定数据，才会这样只有三个卡位需要重新绑定数据。

**Q3：接下去不管是向上滑动还是向下滑动，滑动几次，都不会再有onCreateViewHolder() 的日志了，也就是说RecyclerView总共创建了17个ViewHolder，但有时一行的5个卡位只有3个卡位需要重新绑定数据，有时却又5个卡位都需要重新绑定数据，这是为什么呢？**

答：有时一行只有3个卡位需要重新绑定的原因跟Q2一样，因为mCachedView里正好缓存着当前位置的ViewHolder，本来就是它的ViewHolder当然可以直接拿来用。而至于为什么会创建了17个ViewHolder，那是因为再第四行的卡位要显示出来时，ViewPool里只有3个缓存，而第四行的卡位又用不了mCachedViews里的2个缓存，因为这两个缓存的是6、7卡位的ViewHolder，所以就需要再重新创建2个ViewHodler来给第四行最后的两个卡位使用。