链滴

BaseRecyclerViewAdapterHelper 源码分析（1）

作者：hiquanta

原文链接：https://ld246.com/article/1491527811083

来源网站：链滴

BaseViewHolder源码学习

接下来假设一个场景来分析，假设我们要给一个TextView控件设置一段文字进行显示，一般我们会使该方法：

```
holder.setText(R.id.xxx,"hello world");
```java

源码：
```java
public BaseViewHolder setText(int viewId, CharSequence value) {
    TextView view = getView(viewId);
  view.setText(value);
 return this;}
```

可以看到，源码里第一步会调用getView(viewId)方法，我们来看下该方法的代码

```
@SuppressWarnings("unchecked")
public <T extends View> T getView(int viewId) {
    View view = views.get(viewId);
 if (view == null) {
        view = convertView.findViewById(viewId);
  views.put(viewId, view);
  }
    return (T) view;
}
```

代码中，他会先到views中根据view 的id进行查找，看是否前面已经实时化该view，如果实例化来，直接拿来用就好。

如果view==null 我们会通过convertView去调用我们再熟悉不过的findViewById进行创建，然后把建的view实例缓存到views中方便下次使用，上面我们说到convertView是viewholder的ItemView也就是每个item的总布局。其实是在初始化BaseViewHolder的时候传进来的,源码所示:

```
public BaseViewHolder(final View view) {
    super(view);
 this.views = new SparseArray();
 this.childClickViewIds = new LinkedHashSet<>();
 this.itemChildLongClickViewIds = new LinkedHashSet<>();
 this.nestViews = new HashSet<>();
  convertView = view;

}
```

而BseVIewHolder的创建又是从我们的BaseQuickAdapter里面执行的，下一篇会分析BaseQuickAdpter的代码。

剩下的事情就相当简单了，如果从views缓存队列中找到了我们需要的view，直接调用setText设置文，没找到就先创建，然后压入缓存，然后再调用。其余的api调用方法执行的逻辑基本一样。

BaseViewHolder里还有三个比较重要的成员

- HashSet nestViews;
- LinkedHashSet childClickViewIds;

- LinkedHashSet itemChildLongClickViewIds;

这是跟点击事件相关的，后面会单独出一篇来进行学习。

BaseQuickAdapter之生命周期分析

BaseQuickAdapter实例化第一步当然是调用我们的构造方法：

```
public BaseQuickAdapter(int layoutResId, List<T> data) {
    this.mData = data == null ? new ArrayList<T>() : data;
 if (layoutResId != 0) {
        this.mLayoutResId = layoutResId;
  }
}
public BaseQuickAdapter(List<T> data) {
    this(0, data);
}

public BaseQuickAdapter(int layoutResId) {
    this(layoutResId, null);
}
```

可以看到，加入你不传入data，内部会先为我们创建一个空的list，

然后在构造时记录我们的布局资源id。

那么，布局资源id会在什么时候用呢，当然是在onCreateViewHolder 方法中创建viewholder的时，那么我没继续看onCreateViewHolder 方法的代码。而调用onCreteViewHolder前会先调用getIteViewType方法。

```
@Override
public int getItemViewType(int position) {
    if (getEmptyViewCount() == 1) {
        boolean header = mHeadAndEmptyEnable && getHeaderLayoutCount() != 0;
 switch (position) {
            case 0:
                if (header) {
                    return HEADER_VIEW;
  } else {
                    return EMPTY_VIEW;
  }
            case 1:
                if (header) {
                    return EMPTY_VIEW;
  } else {
                    return FOOTER_VIEW;
  }
            case 2:
                return FOOTER_VIEW;
 default:
                return EMPTY_VIEW;
  }
    }
    autoLoadMore(position);
 int numHeaders = getHeaderLayoutCount();
```

```
 if (position < numHeaders) {
        return HEADER_VIEW;
  } else {
        int adjPosition = position - numHeaders;
 int adapterCount = mData.size();
 if (adjPosition < adapterCount) {
            return getDefItemViewType(adjPosition);
  } else {
            adjPosition = adjPosition - adapterCount;
 int numFooters = getFooterLayoutCount();
 if (adjPosition < numFooters) {
                return FOOTER_VIEW;
  } else {
                return LOADING_VIEW;
  }
        }
    }
}
```

从代码可以看到，他是根据我们data的index值以及我们是否开启空视图之类的数据来决定在onCreatViewHolder中应该返回什么类型的viewHolder。

我们继续看onCreateViewHolder的代码：

```
@Override
public K onCreateViewHolder(ViewGroup parent, int viewType) {
    K baseViewHolder = null;
 this.mContext = parent.getContext();
 this.mLayoutInflater = LayoutInflater.from(mContext);
 switch (viewType) {
        case LOADING_VIEW:
            baseViewHolder = getLoadingView(parent);
 break; case HEADER_VIEW:
            baseViewHolder = createBaseViewHolder(mHeaderLayout);
 break; case EMPTY_VIEW:
            baseViewHolder = createBaseViewHolder(mEmptyLayout);
 break; case FOOTER_VIEW:
            baseViewHolder = createBaseViewHolder(mFooterLayout);
 break; default:
            baseViewHolder = onCreateDefViewHolder(parent, viewType);
  bindViewClickListener(baseViewHolder);
  }
    baseViewHolder.setAdapter(this);
 return baseViewHolder;

}
```

但是，在上面的代码中我们并没有看到有使用我们的 mLayoutResId 资源id的代码。我们可以看到，witch里面前几个都是创建空视图啊，头部视图，底部视图之类的。default这个onCreateDefViewHoder估计是我们需要找的代码，我们来看下这个方法：

```
protected K onCreateDefViewHolder(ViewGroup parent, int viewType) {
    return createBaseViewHolder(parent, mLayoutResId);
}
```

里面调用来createBaseViewHolder方法，我们继续看：

```
protected K createBaseViewHolder(ViewGroup parent, int layoutResId) {
    return createBaseViewHolder(getItemView(layoutResId, parent));
}
```

这里面先调用getItemView

```
protected View getItemView(int layoutResId, ViewGroup parent) {
    return mLayoutInflater.inflate(layoutResId, parent, false);
}
```

里面代码相当简单，就是根据布局资源id渲染我们的view然后返回。

然后才继续调用 createBaseViewHolder

```
protected K createBaseViewHolder(View view) {
    Class temp = getClass();
  Class z = null;
 while (z == null && null != temp) {
        z = getInstancedGenericKClass(temp);
  temp = temp.getSuperclass();
  }
    K k = createGenericKInstance(z, view);
 return null != k ? k : (K) new BaseViewHolder(view);
}
```

里面的代码主要是判断你是否在adapter中使用的BaseViewHolder的子类，大概意思是这样的，

temp=getClass()会返回我们当前的adapter的类型，然后其传入temp调用getInstancedGenericKClass

```
private Class getInstancedGenericKClass(Class z) {
    Type type = z.getGenericSuperclass();
 if (type instanceof ParameterizedType) {
        Type[] types = ((ParameterizedType) type).getActualTypeArguments();
 for (Type temp : types) {
            if (temp instanceof Class) {
                Class tempClass = (Class) temp;
                //判断tempClass是否是BaseViewHolder类型或者其子类型或者接口
 if (BaseViewHolder.class.isAssignableFrom(tempClass)) {
                    return tempClass;
  }
            }
        }
    }
    return null;
}
```

该方法最终会判断当前的类是否是BaseViewHolder的子类或者接口，如果是返回，不是返回null。过下面的三目运算

```
return null != k ? k : (K) new BaseViewHolder(view);
```

如果你不实用集成自BaseViewHolder的子类时，基本调用的是

```
(K) new BaseViewHolder(view)
```

在我们上一篇分析BaseViewHolder中有一个convertView字段的值及来自这里。

那么好，当viewholder被创建后，接下来就是数据绑定了，我们看一下源代码：

```
@Override
public void onBindViewHolder(K holder, int positions) {
    int viewType = holder.getItemViewType();

 switch (viewType) {
        case 0:

            convert(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
 break; case LOADING_VIEW:
            mLoadMoreView.convert(holder);
 break; case HEADER_VIEW:
            break;
 case EMPTY_VIEW:
            break;
 case FOOTER_VIEW:
            break;
 default:
            convert(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
 break;  }
}
```

大家从上面会看到很熟悉的一个函数

```
convert(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
```

为什么0是我们正常要显示的数据呢?

```
default:
    baseViewHolder = onCreateDefViewHolder(parent, viewType);
  bindViewClickListener(baseViewHolder);
```

从onCreateViewHolder中可以看到在default情况下就是viewType=0，该值来自adapter的getItem
iewType方法

```
public int getItemViewType(int position) {
    ```
}
```

当既不是头部视图，也不是尾部视图，空视图时，他就会调用

getDefItemViewType(adjPosition);

```
protected int getDefItemViewType(int position) {
    return super.getItemViewType(position);
}
```

可以看到，他返回的就是0，所以viewType等于0就是正常加载数据的viewHolder，然后我们将holde
和经过计算
```

mData.get(holder.getLayoutPosition() - getHeaderLayoutCount())
后得到的该positions下的data传入convert函数，我们就可以拿着holder和data进行数据的绑定操作了

```
@Override
public void onBindViewHolder(K holder, int positions) {
    ```
}
```

从创建adapter到根据不同的情况创建不同的viewholder到绑定数据的流程。

**BaseQuickAdapter之预加载实现**

autoLoadMore(int position) 见名知意，是与自动加载更多相关的。我们先看下该函数的代码实现

```
private void autoLoadMore(int position) {
    if (getLoadMoreViewCount() == 0) {
        return;
  }
    if (position < getItemCount() - mAutoLoadMoreSize) {
        return;
  }
    if (mLoadMoreView.getLoadMoreStatus() != LoadMoreView.STATUS_DEFAULT) {
        return;
  }
    mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_LOADING);
 if (!mLoading) {
        mLoading = true;
 if (getRecyclerView() != null) {
            getRecyclerView().post(new Runnable() {
                @Override
  public void run() {
                    mRequestLoadMoreListener.onLoadMoreRequested();
  }
            });
  } else {
            mRequestLoadMoreListener.onLoadMoreRequested();
  }
    }
}
```

第一句先判断getLoadMoreViewCount判断是否==0.其实他并不是简单的判断是是否有加载更多视的数量。进入方法里：

```
public int getLoadMoreViewCount() {
    if (mRequestLoadMoreListener == null || !mLoadMoreEnable) {
        return 0;
  }
    if (!mNextLoadEnable && mLoadMoreView.isLoadEndMoreGone()) {
        return 0;
  }
    if (mData.size() == 0) {
        return 0;
  }
    return 1;
}
```
```

我们可以看到，他的返回结果跟很多因素有关，从代码很容易看出:

return 0的情况:

1、你没有设置mRequestLoadMoreListener 或者没有开启mLoadMoreEnable开关;

2、mNextLoadEnable = false ， mNextLoadEnable 在加载更多结束时，你调用

loadMoreEnd(boolean gone) 时会置为false。且 mLoadMoreView 是处于gone状态的。

3、当数据源大小为0时接下来 autoLoadMore方法中的第二句代码很重要

```
if (position < getItemCount() - mAutoLoadMoreSize) {
    return;
}
```

理解起来大概是这样的，mAutoLoadMoreSize是标识开启自动加载更多的一个数量阀值。这个返回巧妙。

假设你的data.size =20 ,mAutoLoadMoreSize =10，当前position=9, 按照理解，这个pisition=9个临界值，因为我们设置了剩余数量<10个时自动加载更多，此时计算9<20-10，position等于9，明后面还有10个数据没渲染，当position=10时(未加载数据还剩9个，此时应该预加载更多)，10<20-0，不成立，代码继续往下走，执行

```
if (mLoadMoreView.getLoadMoreStatus() != LoadMoreView.STATUS_DEFAULT) {
    return;
}
```

我们为什么还要做这个判断的。假如不做这个判断。直接执行下面的代码。

```
mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_LOADING);
if (!mLoading) {
    mLoading = true;
 if (getRecyclerView() != null) {
        getRecyclerView().post(new Runnable() {
            @Override
  public void run() {
                mRequestLoadMoreListener.onLoadMoreRequested();
  }
        });
  } else {
        mRequestLoadMoreListener.onLoadMoreRequested();
  }
}
```

那么会出现很好玩的 现象，当你快速上滑时，由于position>=10后满足条件，执行加载更多的回调position=11时也会执行，以此类推，那么你将收到多次加载更多的回调。所以我们需要判断此时是当前的加载更能多回调已完成，保证每次到达阀值后只调用一次加载更多回调方法。

理解了这个方法之后，我们接下来看下该方法在哪里被调用呢?

```
@Override
public int getItemViewType(int position) {
    if (getEmptyViewCount() == 1) {
```

```
        boolean header = mHeadAndEmptyEnable && getHeaderLayoutCount() != 0;
 switch (position) {
            case 0:
                if (header) {
                    return HEADER_VIEW;
 } else {
                    return EMPTY_VIEW;
 }
            case 1:
                if (header) {
                    return EMPTY_VIEW;
 } else {
                    return FOOTER_VIEW;
 }
            case 2:
                return FOOTER_VIEW;
 default:
                return EMPTY_VIEW;
 }
   }
   autoLoadMore(position);
 int numHeaders = getHeaderLayoutCount();
 if (position < numHeaders) {
        return HEADER_VIEW;
 } else {
        int adjPosition = position - numHeaders;
 int adapterCount = mData.size();
 if (adjPosition < adapterCount) {
            return getDefItemViewType(adjPosition);
 } else {
            adjPosition = adjPosition - adapterCount;
 int numFooters = getFooterLayoutCount();
 if (adjPosition < numFooters) {
                return FOOTER_VIEW;
 } else {
                return LOADING_VIEW;
 }
        }
    }
}
```

在 getItemViewType方法中，为什么在这个方法里面呢，因为根据recyclerView.Adapter的执行逻，在渲染一个新的itemview时，会先调用getItemViewType询问我需要加载什么类型的ViewHolder在这里调用能更早的调用我们的加载更多的方法，当前，你在onBindViewHolder数据绑定方法中调也可以实现这个功能。接下来就很好理解了，当RecyclerView在渲染一个新的itemView时，就会调下

autoLoadMore(position);判断是不是需要调用加载更多回调，需要就调用，有关预加载的分析就OK啦

BaseQuickAdapter上拉加载实现

首先我们先了解几个有关加载更多功能的方法

第一步：打开上拉加载的开关

```
public void setEnableLoadMore(boolean enable) {
    int oldLoadMoreCount = getLoadMoreViewCount();
  mLoadMoreEnable = enable;
 int newLoadMoreCount = getLoadMoreViewCount();

 if (oldLoadMoreCount == 1) {
        if (newLoadMoreCount == 0) {
            notifyItemRemoved(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount(
);
  }
    } else {
        if (newLoadMoreCount == 1) {
            mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_DEFAULT);
  notifyItemInserted(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
  }
    }
}
```

通过上面方法打开我们的上拉加载的开关。首先我们先看下以下两个变量的意思。

1、oldLoadMoreCount 代表在改变这个开关时我们是否处于显示上拉加载的view的状态，1表示处该状态。

2、newLoadMoreCount 代表我们当前是否可以开启上拉加载功能，同样，1表示可以。

这段代码很有意思

```
if (oldLoadMoreCount == 1) {
    if (newLoadMoreCount == 0) {
        notifyItemRemoved(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
  }
}
```

我们为什么要做插入这段代码呢，他的作用其实是这样的：加入当前处于显示加载更多view的状态，时你想关闭该开关，那我们第一件事要做什么呢，当然是移除加载更多view 了，这段代码的作用就这个。

反过来，我们现在要开启上拉加载。走的是这段代码

```
else {
    if (newLoadMoreCount == 1) {
        mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_DEFAULT);
  notifyItemInserted(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
  }
}
```

因为们的的loadMoreView一直是处于最底部的一个view，所以我们通过调用

notifyItemInserted(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());

告诉recycerlView将loadViewMore显示出来。

当我们同过上拉加载加载新的数据完成后，我们需要告诉BaseQuickAdapter你可以恢复正常状态了此时我们将用到以下方法：

```
@Override
  public void onLoadMoreRequested() {
        mSwipeRefreshLayout.setEnabled(false);
 if (pullToRefreshAdapter.getData().size() < PAGE_SIZE) {
            pullToRefreshAdapter.loadMoreEnd(true);
  } else {
            if (mCurrentCounter >= TOTAL_COUNTER) {
//                  pullToRefreshAdapter.loadMoreEnd();//default visible
  pullToRefreshAdapter.loadMoreEnd(mLoadMoreEndGone);//true is gone,false is visible
  } else {
                if (isErr) {
                    pullToRefreshAdapter.addData(DataServer.getSampleData(PAGE_SIZE));
  mCurrentCounter = pullToRefreshAdapter.getData().size();
  pullToRefreshAdapter.loadMoreComplete();
  } else {
                    isErr = true;
  Toast.makeText(PullToRefreshUseActivity.this, R.string.network_err, Toast.LENGTH_LONG).sh
w();
  pullToRefreshAdapter.loadMoreFail();

  }
            }
            mSwipeRefreshLayout.setEnabled(true);
  }
    }
```

//加载完成第一个if是防止我们错误的调用该方法。可以看到，方法内部帮我们调用了更新数据源的法。而且是局部更新。

```
public void loadMoreComplete() {
    if (getLoadMoreViewCount() == 0) {
        return;
  }
    mLoading = false;
  mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_DEFAULT);
  notifyItemChanged(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
}
```

我们看到这么一句恢复我们的loadMoreView为默认值，我们可以跟进去看下一下

```
mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_DEFAULT);
```

他内部是重置了loadMoreStatus这个字段

```
public void setLoadMoreStatus(int loadMoreStatus) {
    this.mLoadMoreStatus = loadMoreStatus;
}
```

而这个字段是在什么时候用到呢，LoadMoreView的代码很少，可以看到

```
public void convert(BaseViewHolder holder) {
    switch (mLoadMoreStatus) {
        case STATUS_LOADING:
            visibleLoading(holder, true);
  visibleLoadFail(holder, false);
```

```
 visibleLoadEnd(holder, false);
 break; case STATUS_FAIL:
            visibleLoading(holder, false);
  visibleLoadFail(holder, true);
  visibleLoadEnd(holder, false);
 break; case STATUS_END:
            visibleLoading(holder, false);
  visibleLoadFail(holder, false);
  visibleLoadEnd(holder, true);
 break; case STATUS_DEFAULT:
            visibleLoading(holder, false);
  visibleLoadFail(holder, false);
  visibleLoadEnd(holder, false);
 break;  }
}
```

他是在一个convert方法中根据mLoadMoreStatus来改变loadMoreView的显示和隐藏，convert方大家应该很熟悉，参数holder其实就是我们的loadMoreView本身，那么loadMoreView.convert在被调用呢。

其实是在我们绑定数据时，如果判断当前viewholder时loadMore类型，就会调用。

```
@Override
public void onBindViewHolder(K holder, int positions) {
    int viewType = holder.getItemViewType();

 switch (viewType) {
        case 0:

            convert(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
 break; case LOADING_VIEW:
            mLoadMoreView.convert(holder);
 break; case HEADER_VIEW:
            break;
 case EMPTY_VIEW:
            break;
 case FOOTER_VIEW:
            break;
 default:
            convert(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
 break;  }
}
```

很好理解，我们的loadMoreView是一直存在的。作为我们recyclerView的最后一个item，当加载到后一个item的时候，他就调用loadView的convert方法，方法内部根据我们当前是否应该显示loadVi w来做相应的操作。这样我们就理解了loadMore的隐藏和显示的逻辑了。后面还有两个方法也很好解，请看。

加载失败调用，可能你有需求在加载失败后要显示一个加载失败的view提示用户，而不是直接关闭loa MoreView。此时你可以调用该方法。

loadMoreFail

```
/**
 * Refresh failed */public void loadMoreFail() {
```

```
    if (getLoadMoreViewCount() == 0) {
        return;
    }
    mLoading = false;
    mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_FAIL);
    notifyItemChanged(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
}
```

loadMoreEnd

```
/**
 * Refresh end, no more data * * @param gone if true gone the load more view
 */public void loadMoreEnd(boolean gone) {
    if (getLoadMoreViewCount() == 0) {
        return;
    }
    mLoading = false;
    mNextLoadEnable = false;
    mLoadMoreView.setLoadMoreEndGone(gone);
    if (gone) {
        notifyItemRemoved(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
    } else {
        mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_END);
        notifyItemChanged(getHeaderLayoutCount() + mData.size() + getFooterLayoutCount());
    }
}
```

之后就到我们关心的回调部分了。首先我们需要设置我们的回调监听器

```
public void setOnLoadMoreListener(RequestLoadMoreListener requestLoadMoreListener, Rec
clerView recyclerView) {
    openLoadMore(requestLoadMoreListener);
    if (getRecyclerView() == null) {
        setRecyclerView(recyclerView);
    }
}
```

在设置监听器的时候，代码也帮我们做了一个字段的赋值操作。默认开启上拉加载，mLoading是表当前是否处于上拉加载中。

接下来你可能要问，那这个morequestLoadMoreListener在什么时候被调用呢，其实这个也设计的较好，上一章我们分析了预加载功能，其实就在上次介绍的代码中。

```
private void autoLoadMore(int position) {
    if (getLoadMoreViewCount() == 0) {
        return;
    }
    if (position < getItemCount() - mAutoLoadMoreSize) {
        return;
    }
    if (mLoadMoreView.getLoadMoreStatus() != LoadMoreView.STATUS_DEFAULT) {
        return;
    }
    mLoadMoreView.setLoadMoreStatus(LoadMoreView.STATUS_LOADING);
    if (!mLoading) {
```

```
        mLoading = true;
 if (getRecyclerView() != null) {
            getRecyclerView().post(new Runnable() {
                @Override
  public void run() {
                    mRequestLoadMoreListener.onLoadMoreRequested();
  }
            });
  } else {
            mRequestLoadMoreListener.onLoadMoreRequested();
  }
    }
}
```

如果你想关闭预加载，当时是mAutoLoadMoreSize =0 ,此时要调用最后一句代码，条件就变成了

position

思路大概是这样的：

在关闭预加载功能时：如果加载到最后一个item，首先会调用getItemViewType询问即将加载的view older是什么类型。我们前面说过了，loadMoreView是一直作为最后一个viewHolder存在的。此时果符合显示loadMoreView条件，那么就设置loadMoreView的状态，调用我们的函数。在执行到onB ndViewHolder生命周期方法时，会根据我们设置的值显示或者隐藏loadMoreView视图，也就是说 onLoadMoreRequested方法的回调在显示loadMoreView之前就被调用了。时间是很短的，用户基 察觉不出来。

本次分析就到这里，接下来会继续分析其余的代码。如果有需要想一起分析哪部份代码，也可以直接 言，我会调整顺序。