链滴

BaseRecyclerViewAdapterHelper 源码分析：分组、多布局、折叠

作者：hiquanta

原文链接：https://ld246.com/article/1491549193811

来源网站：链滴

● BaseMultiItemQuickAdapter 多布局

首先我们要实现多类型布局，我们的adapter不再是继承自BaseQuickAdapter类，而是继承自其的类

BaseMultiItemQuickAdapter。而且数据源类型需要继承自MultiItemEntity。

MultiItemEntity是一个接口，代码很少：

```
public interface MultiItemEntity {

    int getItemType();

}
```

其主要用意是我们的数据源继承MultiItemEntity，这样子，我们可以在数据源中动态的返回一个int型的数值（代表某一类型的item），大家可以想一下，我们在渲染viewholder的时候，如果想实现类型的viewholder，而viewholder的类型展示又跟所需绑定的数据息息相关，那么如果我们在数据中提供一种确定viewholder类型的能力，理论上是不是就可以达到我们要的效果了？我们带着这样的个假设继续往下看。

而这个BaseMultiItemQuickAdapter 是何许人也，是如何实现多类型布局的呢？我们来看下源码：

```
package com.chad.library.adapter.base;

import android.support.annotation.LayoutRes;
import android.util.SparseArray;
import android.view.ViewGroup;

import com.chad.library.adapter.base.entity.MultiItemEntity;

import java.util.List;

/**
 * https://github.com/CymChad/BaseRecyclerViewAdapterHelper */public abstract class Base
ultiItemQuickAdapter<T extends MultiItemEntity, K extends BaseViewHolder> extends BaseQ
ickAdapter<T, K> {

    /**
 * layouts indexed with their types */  private SparseArray layouts;

 private static final int DEFAULT_VIEW_TYPE = -0xff;

  /**
 * Same as QuickAdapter#QuickAdapter(Context,int) but with * some initialization data. * * @
aram data A new list is created out of this one to avoid mutable list
 */  public BaseMultiItemQuickAdapter( List<T> data) {
        super( data);
  }

    @Override
  protected int getDefItemViewType(int position) {
        Object item = mData.get(position);
 if (item instanceof MultiItemEntity) {
```

```
            return ((MultiItemEntity)item).getItemType();
        }
        return DEFAULT_VIEW_TYPE;
    }

    protected void setDefaultViewTypeLayout(@LayoutRes int layoutResId) {
        addItemType(DEFAULT_VIEW_TYPE, layoutResId);
    }

    @Override
    protected K onCreateDefViewHolder(ViewGroup parent, int viewType) {
        return createBaseViewHolder(parent, getLayoutId(viewType));
    }

    private int getLayoutId(int viewType) {
        return layouts.get(viewType);
    }

    protected void addItemType(int type, @LayoutRes int layoutResId) {
        if (layouts == null) {
            layouts = new SparseArray<>();
        }
        layouts.put(type, layoutResId);
    }

}
```

源码不多：

字段解析：

1、存储我们的布局资源的ids

private SparseArray layouts;

2、缺省的布局类型值，当使用多布局时，在渲染viewholder的时候类型址不是我们配置的类型值中就会使用这个。

private static final int DEFAULT_VIEW_TYPE = -0xff;

接下来，我们以一个BaseMultiItemQuickAdapter的创建过程来分析代码：

之前我们分析了BaseQuickAdapter的代码，其执行过程是一样的，我们实现多布局功能的切入口无是

1、在执行getItemViewType时的能够根据我们的数据源返回对应的布局类型值。

2、在onCreateDefViewHolder 能够正确拿到类型值进行viewholder的渲染。

3、我们在onBindViewHolder中根据传递给我们的数据源中接口定义的getItemViewType方法返回类型值来确定当前的viewholder是什么类型的，需要绑定什么数据。

（注：之前分析了adapter的加载数据时的生命周期方法：getItemViewType->onCreateDefViewHolder->onBindViewHolder，如果不大清楚可以看下前面的文章）

所以，我们在BaseMultiItemQuickAdapter 里面重写了getDefItemViewType方法，为什么时重写

getDefItemViewType方法而不是getItemViewType方法呢？这可不是我糊弄你，因为我们在BaseQuckAdapter里面重写了getItemViewType方法，而在getItemViewType方法里调用了getDefItemVieType方法来回去类型值，该方法也在之前的分析BaseQuickAdapter源码的文章中分析了的。

重写之后做了什么呢？看代码：

```
@Override
protected int getDefItemViewType(int position) {
    Object item = mData.get(position);
 if (item instanceof MultiItemEntity) {
        return ((MultiItemEntity)item).getItemType();
  }
    return DEFAULT_VIEW_TYPE;
}
```

很简单，因为我们的数据源实现了MultiItemEntity接口。直接判断该position的数据是不是实现了

MultiItemEntity接口，是调用接口的getItemType方法返回类型值，不是返回默认类型值。

第一步返回类型值的代码改造完成了，接下来第二部就是根据类型值渲染viewholder。BaseMultiIteQuickAdapter直接重写了onCreateDefViewHolder 方法来实现该扩展：

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

代码很简单，从我们存储布局缓存的字段中根据viewType返回对象的布局资源的ids。

所以BaseMultiItemQuickAdapter 还给我们包装了一个addItemType方法：

```
protected void addItemType(int type, @LayoutRes int layoutResId) {
    if (layouts == null) {
        layouts = new SparseArray<>();
  }
    layouts.put(type, layoutResId);
}
```

该方法很简单，就是将不同的布局资源的ids和对应的类型值存储起来。

所以我们的创建多布局的时候，需要的构造函数中调用addItemType来添加不同的布局资源

最后一步，绑定数据;一般绑定数据实在onBindViewHolder中实现的，而我们的BaseRecyclerViewAapterHelper对其进行了包装，提供了一个convert方法，所以我们只需要的convert方法中根据数据数据节点的类型值判断绑定的是那个布局的数据即可。

总结：理解了adapter加载数据的生命周期方法的执行顺序很重要（getItemViewType->onCreateDeViewHolder->onBindViewHolder）。

只要控制viewType的返回、viewholder的渲染。viewholder数据的绑定即可。

BaseSectionQuickAdapter 分组

首先今天的学习我们还是按照前面的学习思路，根据getItemViewType->onCreateDefViewHolder-onBindViewHolder，即从确认viewholder类型->根据类型值创建viewholder->根据数据源类型绑数据到viewholder上。

第一步：我们看一下BaseSectionQuickAdapter这个类的定义

public abstract class BaseSectionQuickAdapter extends BaseQuickAdapter {

跟前面分析的多类型BaseMultiItemQuickAdapter差不多，只是我们的数据源需要继承自SetionEntit。那么这个SetionEntity做了什么事呢，我们来看下源码：

```
package com.chad.library.adapter.base.entity;

/**
 * https://github.com/CymChad/BaseRecyclerViewAdapterHelper */public abstract class SectinEntity<T> {
    public boolean isHeader;
 public T t;
 public String header;

 public SectionEntity(boolean isHeader, String header) {
        this.isHeader = isHeader;
 this.header = header;
 this.t = null;
  }

    public SectionEntity(T t) {
        this.isHeader = false;
 this.header = null;
 this.t = t;
  }
}
```

从源码可以看出，他是一个抽象类,可能你会问，为什么要定义成抽象类呢，为什么不定义成接口或者通类呢。

以下理由仅由我意想得出，大家也可以发表下自己的看法：

1、我们定义SectionEntity这个类，目的自然是希望用户的bean都具有某些规范，而我们的BaseSectinQuickAdapter将根据该规范进行数据的处理。虽然使用普通类一样能达到相同的效果，但是不推荐我觉得这让有可能会让用户忽略我们所需要让用户知道的规范。

2、接口类，接口类其实是特殊的抽象类，上次分析的MultiItemEntity为什么又定义成接口类型呢。

```
public interface MultiItemEntity {

    int getItemType();

}
```

根据实际需求而定，因为我们在实现多类型时，只需要用户的数据源提供一个类型值给我们即可，所此时定义成接口类是最为合适的，因为用户数据源只要实现了该接口，他必须实现接口的方法，而我需要的恰恰是在使用时调用该接口即可。

但是在SetionEntity中，我们帮用户多做点事，为其提供两个构造方法，一个时分组头，一个是分组。而此时如果是定义成接口类，是不符合需求的，因为接口类的方法不能有方法体等。

SectionEntity代码分析：从源码可以看出，假如我们当前数据是分组头，那么我们在创建bean时使用

```
public SectionEntity(boolean isHeader, String header) {
    this.isHeader = isHeader;
 this.header = header;
 this.t = null;
}
```

即可，当前定义分组头只有个string类型的分组头名字，你在继承时可以根据实际需求进行扩展，内调用父类的该构造方法即可。

如果时普通的数据bean：调用以下构造方法即可，当然你也可以进行扩展，根据个人需求而定。

```
public SectionEntity(T t) {
    this.isHeader = false;
 this.header = null;
 this.t = t;
}
```

我们来看BaseSectionQuickAdapter的源码：

```
package com.chad.library.adapter.base;

import android.view.ViewGroup;

import com.chad.library.adapter.base.entity.SectionEntity;

import java.util.List;

/**
 * https://github.com/CymChad/BaseRecyclerViewAdapterHelper */public abstract class Base
ectionQuickAdapter<T extends SectionEntity, K extends BaseViewHolder> extends BaseQuic
Adapter<T, K> {

    protected int mSectionHeadResId;
 protected static final int SECTION_HEADER_VIEW = 0x00000444;

  /**
 * Same as QuickAdapter#QuickAdapter(Context,int) but with * some initialization data. * * @
aram sectionHeadResId The section head layout id for each item
 * @param layoutResId The layout resource id of each item.
 * @param data A new list is created out of this one to avoid mutable list
 */  public BaseSectionQuickAdapter( int layoutResId, int sectionHeadResId, List<T> data) {
        super(layoutResId, data);
 this.mSectionHeadResId = sectionHeadResId;
  }

    @Override
  protected int getDefItemViewType(int position) {
        return mData.get(position).isHeader ? SECTION_HEADER_VIEW : 0;
  }

    @Override
  protected K onCreateDefViewHolder(ViewGroup parent, int viewType) {
        if (viewType == SECTION_HEADER_VIEW)
            return createBaseViewHolder(getItemView(mSectionHeadResId, parent));
```

```
 return super.onCreateDefViewHolder(parent, viewType);
  }

    @Override
  public void onBindViewHolder(K holder, int positions) {
        switch (holder.getItemViewType()) {
            case SECTION_HEADER_VIEW:
                setFullSpan(holder);
  convertHead(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
  break; default:
                super.onBindViewHolder(holder, positions);
  break;  }
    }

    protected abstract void convertHead(K helper, T item);

}
```

大家可以看到，源码比较少，跟BaseMultiItemQuickAdapter是一样的思路。

字段解析：

protected int mSectionHeadResId;

mSectionHeadResId用来保存我们分组头的布局资源ids；

protected static final int SECTION_HEADER_VIEW = 0x00000444;

定义了一个默认的分组头类型。思想与实现多类型一致；

我们在实例化BaseSectionQuickAdapter时需要多传递一个分组头的资源ids过来，所以构造方法是样的：

```
/**
 * Same as QuickAdapter#QuickAdapter(Context,int) but with * some initialization data. * * @
aram sectionHeadResId The section head layout id for each item
 * @param layoutResId The layout resource id of each item.
 * @param data A new list is created out of this one to avoid mutable list
 */public BaseSectionQuickAdapter( int layoutResId, int sectionHeadResId, List<T> data) {
    super(layoutResId, data);
  this.mSectionHeadResId = sectionHeadResId;
}
```

构造好之后，我们也是利用来adapter的生命周期方法：

1、重写getDefItemViewType方法，前面也解释过为什么不是重写Recycler.adapter的getItemViewype方法，以为我们的BaseQuickAdapter对其进行来包装。最终在getItemViewType方法中会调用们的getDefItemViewType方法。

重写该方法所做的事不多：

```
@Override
protected int getDefItemViewType(int position) {
    return mData.get(position).isHeader ? SECTION_HEADER_VIEW : 0;
}
```

根据我们当前位置的数据bean，判断当前节点的数据bean是不是分组头bean，如果是，返回SECTIN_HEADER_VIEW告诉BaseQuickAdapter，你要创建的viewholder是分组头类型的viewholder。则返回0（0时RecyclerView.Adapter的缺省值，前面有分析）

接下来，我们也同样是重写了onCreateDefViewHolder方法。

```
@Override
protected K onCreateDefViewHolder(ViewGroup parent, int viewType) {
    if (viewType == SECTION_HEADER_VIEW)
        return createBaseViewHolder(getItemView(mSectionHeadResId, parent));

 return super.onCreateDefViewHolder(parent, viewType);
}
```

根据返回的类型值，如果是SECTION_HEADER_VIEW 那么我们就创建一个分组头viewholder返回。则调用父类的方法按原流程走。

在这里我们还需要重写onBindViewHolder方法，因为我们要多做两件事情：

1、对我们的分组头进行特殊处理；

2、增加一个分组头数据绑定的抽象方法的调用；

```
@Override
public void onBindViewHolder(K holder, int positions) {
    switch (holder.getItemViewType()) {
        case SECTION_HEADER_VIEW:
            setFullSpan(holder);
  convertHead(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));
 break; default:
            super.onBindViewHolder(holder, positions);
 break;  }
}
```

里面有个很有趣的方法。setFullSpan，从字面上理解是设置充满空间，我们来看下代码：

```
/**
 * When set to true, the item will layout using all span area. That means, if orientation * is vert
cal, the view will have full width; if orientation is horizontal, the view will * have full height. * if
he hold view use StaggeredGridLayoutManager they should using all span area * * @param h
lder True if this item should traverse all spans.
 */protected void setFullSpan(RecyclerView.ViewHolder holder) {
    if (holder.itemView.getLayoutParams() instanceof StaggeredGridLayoutManager.LayoutPar
ms) {
        StaggeredGridLayoutManager.LayoutParams params = (StaggeredGridLayoutManager.L
youtParams) holder
                .itemView.getLayoutParams();
  params.setFullSpan(true);
  }
}
```

里面原来是对Layoutmanager为StaggeredGridLayoutManager类型时做特殊处理，大家可以去了下StaggeredGridLayoutManager这种类型的LayoutManager。

最后会调用一个方法

params.setFullSpan(true);

继续看该方法源码：

```
/**
 * When set to true, the item will layout using all span area. That means, if orientation * is vert
cal, the view will have full width; if orientation is horizontal, the view will * have full height. * *
@param fullSpan True if this item should traverse all spans.
 * @see #isFullSpan()
 */public void setFullSpan(boolean fullSpan) {
    mFullSpan = fullSpan;
}
```

该方法是StaggeredGridLayoutManager提供的，英文说明的大意是：

如果你设置true，当前item将使用布局的所有空间。如果是垂直的，会沾满水平方向的宽度空间，如是水平，会占满垂直方向的高度空间。

然后将holder和当前节点的数据bean作为参数调用convertHead函数。

所以当你实现的是带分组头的adapter时，会多出一个数据绑定的回调接口：

protected abstract void convertHead(BaseViewHolder helper, T item);

可能你还会看到以下代码：

convertHead(holder, mData.get(holder.getLayoutPosition() - getHeaderLayoutCount()));

里面有一句holder.getLayoutPosition()。

getLayoutPosition是干什么用的呢，因为RecyclerView的item的布局和渲染其实是交给layoutManger来完成的，所以adapter中的item的位置可能跟data的index匹配不上，而getLayoutPosition将回给我们当前viewholder在recyclerView中的最新位置信息。

总结：分析思路还是老套路，根据一个组件的生命周期及业务流程进行分析，掌握一个控件的执行流是理解一个控件的实现的一个较好的方法，本人是这么认为的，也是这么做的。

Expandable 折叠

BaseRecyclerViewAdapterHelper中有关实现可展开和折叠二级Item或多级Item的源码。在开始学之前，我想先分析下实现的思路，这样对于进行源码的理解效果比较好。

实现伸展and折叠，很多控件都有，网上也有用linearlayout实现的功能很强大、很炫酷的开源项目平时要实现一些伸缩性的自定义控件，我们也可以是用属性动画，或者动态控制控件的Layout属性等可以实现。那么现在我们来想象一下，如果在recyclerview中实现该功能，相对来说能想到的比较合的方式是什么呢?

其实我们可以很好的利用RecyclerView.Adapter给我们提供的如下一些通知数据源更新的方法来实现们的动态伸展and折叠功能。当要伸展时，我们动态将下一级item的数据添加在与adapter绑定的数集合中，然后通知layoutManger更新数据源。当要收缩时，同理，将下一级的item的数据源从与adater绑定的数据集合中移除，然后通知更新。

```
* @see #notifyItemChanged(int)
* @see #notifyItemInserted(int)
* @see #notifyItemRemoved(int)
```

```
* @see #notifyItemRangeChanged(int, int)
* @see #notifyItemRangeInserted(int, int)
* @see #notifyItemRangeRemoved(int, int)
```

思路：

1. 数据bean应该有存储自己数据的字段
2. 数据bean应该有存储下一级item列表的集合类型的字段
3. 数据bean应该有一个字段标识当前item的状态（伸展or收缩）
4. 初始化adapter时只渲染顶级的item
5. 点击item是检测该item是否支持伸缩
6. 支持伸缩：当前状态展开->折叠（将次级list插入adapter绑定的data集合中，刷新数据）；当前态折叠->展开(将次级的list从与adapter绑定的data集合中移除，刷新数据)
7. 插入或移除的位置根据点击的item确定，插入量与移除量根据下一级item数量确定
8. 插入移除过程中可以使用动画效果

思路理清之后我们接下来开始学习源代码：

实现Expandable And collapse 效果我们仍然是使用BaseMultiItemQuickAdapter实现即可

然后我们需要先看两个相关的类：IExpandable接口；AbstractExpandableItem： 对数据bean的再封装，某个bean如果有次级的list 可以实现该抽象类。

```
package com.chad.library.adapter.base.entity;

import java.util.List;

/**
 * implement the interface if the item is expandable * Created by luoxw on 2016/8/8. */public
nterface IExpandable<T> {
    boolean isExpanded();
 void setExpanded(boolean expanded);
  List<T> getSubItems();

  /**
 * Get the level of this item. The level start from 0. * If you don't care about the level, just retu
n a negative. */  int getLevel();
}
```

可以看到，IExpandable 里面定义了四个接口方法：

1. isExpanded判断当前的bean是否已展开
2. setExoanded更新bean的当前状态
3. getSubItems返回下一级的数据集合
4. getLevel 返回当前item属于第几个层级， 第一级from 0

```
package com.chad.library.adapter.base.entity;

import java.util.ArrayList;
import java.util.List;
```

```
/**
 * A helper to implement expandable item.
 * if you don't want to extent a class, you can also implement the interface IExpandable
 * Created by luoxw on 2016/8/9.
 */public abstract class AbstractExpandableItem<T> implements IExpandable<T> {
    protected boolean mExpandable = false;
  protected List<T> mSubItems;

  @Override
  public boolean isExpanded() {
        return mExpandable;
  }

    @Override
  public void setExpanded(boolean expanded) {
        mExpandable = expanded;
  }

    @Override
  public List<T> getSubItems() {
        return mSubItems;
  }

    public boolean hasSubItem() {
        return mSubItems != null && mSubItems.size() > 0;
  }

    public void setSubItems(List<T> list) {
        mSubItems = list;
  }

    public T getSubItem(int position) {
        if (hasSubItem() && position < mSubItems.size()) {
            return mSubItems.get(position);
  } else {
            return null;
  }
    }

    public int getSubItemPosition(T subItem) {
        return mSubItems != null ? mSubItems.indexOf(subItem) : -1;
  }

    public void addSubItem(T subItem) {
        if (mSubItems == null) {
            mSubItems = new ArrayList<>();
  }
        mSubItems.add(subItem);
  }

    public void addSubItem(int position, T subItem) {
        if (mSubItems != null && position >= 0 && position < mSubItems.size()) {
            mSubItems.add(position, subItem);
```

```
  } else {
            addSubItem(subItem);
  }
    }

    public boolean contains(T subItem) {
        return mSubItems != null && mSubItems.contains(subItem);
  }

    public boolean removeSubItem(T subItem) {
        return mSubItems != null && mSubItems.remove(subItem);
  }

    public boolean removeSubItem(int position) {
        if (mSubItems != null && position >= 0 && position < mSubItems.size()) {
            mSubItems.remove(position);
 return true;  }
        return false;
  }
}
```

字段方法解析：

1. mExpandable 保存当前的状态值，默认为false
2. mSubItems 存储数据bean集合

里面还包装了一些常用的方法，这里就不一一解析了。

接下来我们以一个使用demo的实现来进行分析：

ExpandableUseActivity ：

```
private ArrayList generateData() {
    int lv0Count = 9;
 int lv1Count = 3;
 int personCount = 5;

  String[] nameList = {"Bob", "Andy", "Lily", "Brown", "Bruce"};
  Random random = new Random();

  ArrayList res = new ArrayList<>();
 for (int i = 0; i < lv0Count; i++) {
        Level0Item lv0 = new Level0Item("This is " + i + "th item in Level 0", "subtitle of " + i);
 for (int j = 0; j < lv1Count; j++) {
            Level1Item lv1 = new Level1Item("Level 1 item: " + j, "(no animation)");
 for (int k = 0; k < personCount; k++) {
                lv1.addSubItem(new Person(nameList[k], random.nextInt(40)));
  }
            lv0.addSubItem(lv1);
  }
        res.add(lv0);
  }
    return res;
}
```

这段代码的作用是生成一个支持Expandable and collapse 的数据集合，创建一个0级的Level0Item 然后将下一级的Level1Item添加到Level0Item中

```
public class Level0Item extends AbstractExpandableItem implements MultiItemEntity {
    public String title;
 public String subTitle;

 public Level0Item( String title, String subTitle) {
        this.subTitle = subTitle;
 this.title = title;
  }

    @Override
  public int getItemType() {
        return ExpandableItemAdapter.TYPE_LEVEL_0;
  }

    @Override
  public int getLevel() {
        return 0;
  }
}
```

可以看到Level0Item继承了AbstractExpandableItem 并实现MultiItemEntity接口。里面根据实际求定义相应的字段即可。

Level1Item 与Level0Item一样，只是返回的Level =1：

```
public class Level1Item extends AbstractExpandableItem implements MultiItemEntity{
    public String title;
 public String subTitle;

 public Level1Item(String title, String subTitle) {
        this.subTitle = subTitle;
 this.title = title;
  }

    @Override
  public int getItemType() {
        return ExpandableItemAdapter.TYPE_LEVEL_1;
  }

    @Override
  public int getLevel() {
        return 1;
  }
}
```

当如过某一级的item没有下一级的list时，就不需要在实现AbstractExpandableItem了

然后我们的切入点时adapter，因为默认是折叠状态，当我们点击具备展开折叠能力的item时才会触该功能，所以逻辑的控制是在adapter中的。

```
package com.chad.baserecyclerviewadapterhelper.adapter;
```

```
import android.util.Log;
import android.view.View;

import com.chad.baserecyclerviewadapterhelper.R;
import com.chad.baserecyclerviewadapterhelper.entity.Level0Item;
import com.chad.baserecyclerviewadapterhelper.entity.Level1Item;
import com.chad.baserecyclerviewadapterhelper.entity.Person;
import com.chad.library.adapter.base.BaseMultiItemQuickAdapter;
import com.chad.library.adapter.base.BaseViewHolder;
import com.chad.library.adapter.base.entity.MultiItemEntity;

import java.util.List;

/**
 * Created by luoxw on 2016/8/9. */public class ExpandableItemAdapter extends BaseMultiIt
mQuickAdapter, BaseViewHolder> {
    private static final String TAG = ExpandableItemAdapter.class.getSimpleName();

 public static final int TYPE_LEVEL_0 = 0;
 public static final int TYPE_LEVEL_1 = 1;
 public static final int TYPE_PERSON = 2;

  /**
 * Same as QuickAdapter#QuickAdapter(Context,int) but with * some initialization data. * * @
aram data A new list is created out of this one to avoid mutable list
 */  public ExpandableItemAdapter(List data) {
       super(data);
  addItemType(TYPE_LEVEL_0, R.layout.item_expandable_lv0);
  addItemType(TYPE_LEVEL_1, R.layout.item_expandable_lv1);
  addItemType(TYPE_PERSON, R.layout.item_expandable_lv2);
  }

    @Override
  protected void convert(final BaseViewHolder holder, final MultiItemEntity item) {
       switch (holder.getItemViewType()) {
          case TYPE_LEVEL_0:
             switch (holder.getLayoutPosition() %
                    3) {
                case 0:
                   holder.setImageResource(R.id.iv_head, R.mipmap.head_img0);
 break; case 1:
                   holder.setImageResource(R.id.iv_head, R.mipmap.head_img1);
 break; case 2:
                   holder.setImageResource(R.id.iv_head, R.mipmap.head_img2);
 break;  }
             final Level0Item lv0 = (Level0Item)item;
  holder.setText(R.id.title, lv0.title)
                   .setText(R.id.sub_title, lv0.subTitle)
                   .setImageResource(R.id.iv, lv0.isExpanded() ? R.mipmap.arrow_b : R.mipmap.ar
ow_r);
  holder.itemView.setOnClickListener(new View.OnClickListener() {
              @Override
  public void onClick(View v) {
                   int pos = holder.getAdapterPosition();
```

```
 Log.d(TAG, "Level 0 item pos: " + pos);
if (lv0.isExpanded()) {
                    collapse(pos);
 } else {
//                      if (pos % 3 == 0) {
//                          expandAll(pos, false);
//                      } else {
  expand(pos);
//                      }
  }
                }
            });
 break; case TYPE_LEVEL_1:
            final Level1Item lv1 = (Level1Item)item;
  holder.setText(R.id.title, lv1.title)
                    .setText(R.id.sub_title, lv1.subTitle)
                    .setImageResource(R.id.iv, lv1.isExpanded() ? R.mipmap.arrow_b : R.mipmap.ar
ow_r);
  holder.itemView.setOnClickListener(new View.OnClickListener() {
                @Override
  public void onClick(View v) {
                    int pos = holder.getAdapterPosition();
  Log.d(TAG, "Level 1 item pos: " + pos);
 if (lv1.isExpanded()) {
                    collapse(pos, false);
 } else {
                    expand(pos, false);
 }
                }
            });
 break; case TYPE_PERSON:
            final Person person = (Person)item;
  holder.setText(R.id.tv, person.name + " parent pos: " + getParentPosition(person));
 break;  }
   }
}
```

可以看到里面我们先添加3个level的布局资源文件。重点在convert回调方法；

1. 最外层进行viewholder 的类型判断进行数据绑定

2. 添加点击事件的监听

3. 当被点击时，判断当前的levelitem是不是展开的或折叠的，然后根据你的需要调用collapse或者exand进行折叠或展开操作。

重点来的，最终实现展开、折叠功能其实是依赖collapse和expand这些api;那我们来看下这些api到内部是怎么实现的，我们从expand开始。代码中expand(pos);传了一个pos进来，而这个pos就是被击的item在adapter数据集合中的index。

```
/**
 * Expand an expandable item * * @param position position of the item
 * @param animate expand items with animation
 * @param shouldNotify notify the RecyclerView to rebind items, false if you want to do it
 *                     yourself. * @return the number of items that have been added.
```

```
 */public int expand(@IntRange(from = 0) int position, boolean animate, boolean shouldNotif
) {
    position -= getHeaderLayoutCount();

  IExpandable expandable = getExpandableItem(position);
 if (expandable == null) {
        return 0;
  }
    if (!hasSubItems(expandable)) {
        expandable.setExpanded(false);
 return 0;
  }
    int subItemCount = 0;
 if (!expandable.isExpanded()) {
        List list = expandable.getSubItems();
  mData.addAll(position + 1, list);
  subItemCount += recursiveExpand(position + 1, list);

  expandable.setExpanded(true);
  subItemCount += list.size();
  }
    int parentPos = position + getHeaderLayoutCount();
 if (shouldNotify) {
        if (animate) {
            notifyItemChanged(parentPos);
  notifyItemRangeInserted(parentPos + 1, subItemCount);
  } else {
            notifyDataSetChanged();
  }
    }
    return subItemCount;
}
```

可以看到expand是一个方法多态，提供了三种参数类型的调用。支持是否需要动画，是否更新数据。

排除headerview的干扰，获得实际的位置position

```
position -= getHeaderLayoutCount();
```

判断其是否支持展开折叠，是否有下一级items需要展开，没有就直接返回0

```
IExpandable expandable = getExpandableItem(position);
if (expandable == null) {
    return 0;
}
if (!hasSubItems(expandable)) {
    expandable.setExpanded(false);
 return 0;
}
```

下面代码作用：如果处于折叠状态且需要展开，则执行到下面代码，通过getSubItems获得要展开的list，将其添加到mdata中，通过recursiveExpand获得要展开的items的数量

```
int subItemCount = 0;
```

```
if (!expandable.isExpanded()) {
    List list = expandable.getSubItems();
  mData.addAll(position + 1, list);
  subItemCount += recursiveExpand(position + 1, list);

  expandable.setExpanded(true);
  subItemCount += list.size();
}
```

我们可以看到recursiveExpand的源码如下：下面是一个递归调用，一直遍历到最后一层不支持展开叠的item才会回溯回来，遍历过程中可以看到一个判断，if(item.isExpanded) 就是如果下一级的item原来已经是处于展开状态的，此时我们也需要展开他。最终返回的是所需展开的items的数量。

```
/**
 * Get the row id associated with the specified position in the list. * * @param position The po
ition of the item within the adapter's data set whose row id we want.
 * @return The id of the item at the specified position.
 */@Override
public long getItemId(int position) {
    return position;
}

private int recursiveExpand(int position, @NonNull List list) {
    int count = 0;
 int pos = position + list.size() - 1;
 for (int i = list.size() - 1; i >= 0; i--, pos--) {
        if (list.get(i) instanceof IExpandable) {
            IExpandable item = (IExpandable) list.get(i);
 if (item.isExpanded() && hasSubItems(item)) {
                List subList = item.getSubItems();
  mData.addAll(pos + 1, subList);
 int subItemCount = recursiveExpand(pos + 1, subList);
  count += subItemCount;
  }
        }
    }
    return count;

}
```

获得需要展开的items的数量值，也将数据集合添加到了mData中，此时我们通知layoutManager刷数据即可

```
int parentPos = position + getHeaderLayoutCount();
if (shouldNotify) {
    if (animate) {
        notifyItemChanged(parentPos);
  notifyItemRangeInserted(parentPos + 1, subItemCount);
  } else {
        notifyDataSetChanged();
  }
}
```

刷新的时候我们要先确定开始刷新位置，所以需要加上headerview的数量

然后调用如上代码即可。折叠是反向进行的，根据这个思路看就可以了。

总结：折叠->展开：mData添加需展开的数据集，更新数据源；展开->折叠：mData移除需折叠的据集，更新数据源。