一、概述

记得好久以前针对ListView类控件写过一篇打造万能的ListView GridView 适配器，如今RecyclerView异军突起，其Adapter的用法也与ListView类似，那么我们也可以一步一步的为其打造通用的Adapter，使下列用法书写更加简单：

- 简单的数据绑定（单种Item）
- 多种Item Type 数据绑定
- 增加onItemClickListener , onItenLongClickListener
- 优雅的添加分类header

二、使用方式和效果图

在一步一步完成前，我们先看下使用方式和效果图：

（1）简单的数据绑定

首先看我们最常用的单种Item的书写方式：

```
mRecyclerView.setAdapter(new CommonAdapter<String>(this, R.layout.item_list,
mDatas)
{
    @Override
    public void convert(ViewHolder holder, String s)
    {
        holder.setText(R.id.id_item_list_title, s);
    }
});12345678
```

是不是相当方便，在convert方法中完成数据、事件绑定即可。

（2）多种ItemViewType

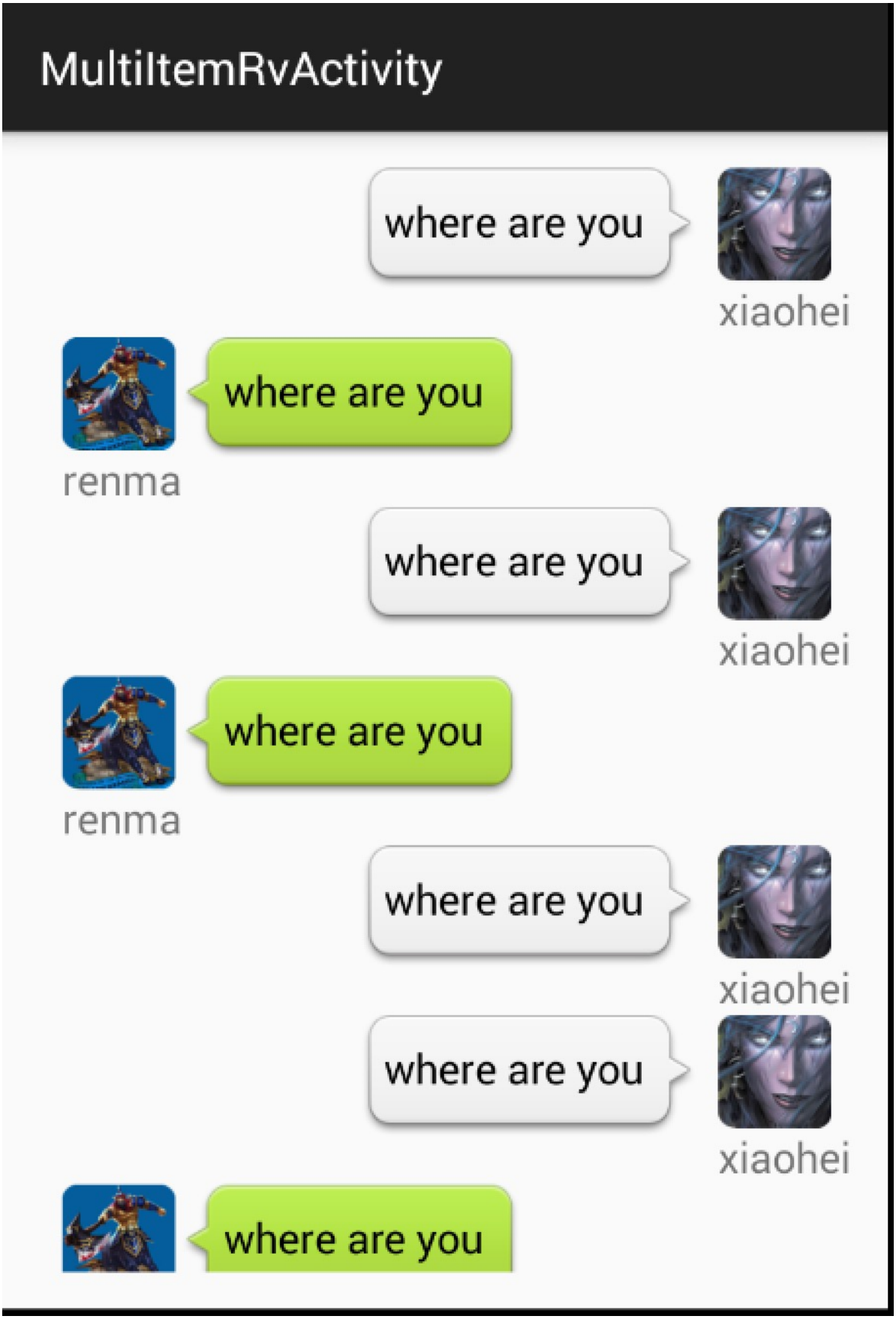
多种ItemViewType，正常考虑下，我们需要根据Item指定ItemType，并且根据ItemType指定相应的布局文件。我们通过`MultiItemTypeSupport`完成指定：

```
MultiItemTypeSupport  multiItemSupport = new MultiItemTypeSupport<ChatMessage>()
{
    @Override
    public int getLayoutId(int itemType)
    {
       //根据itemType返回item布局文件id
    }

    @Override
    public int getItemViewType(int postion, ChatMessage msg)
    {
       //根据当前的bean返回item type
    }
}
123456789101112131415
```

剩下就简单了,将其作为参数传入到`MultiItemCommonAdapter`即可。

```
mRecyclerView.setAdapter(new SectionAdapter<String>(this, mDatas,
multiItemSupport)
{
    @Override
    public void convert(ViewHolder holder, String s)
    {
        holder.setText(R.id.id_item_list_title, s);
    }
});12345678
```

贴个效果图:



(3)添加分类header

其实属于多种ItemViewType的一种了，只是比较常用，我们就简单封装下。

依赖正常考虑下，这种方式需要额外指定header的布局，以及布局中显示标题的TextView了，以及根据Item显示什么样的标题。我们通过`SectionSupport`对象指定：

```
SectionSupport<String> sectionSupport = new SectionSupport<String>()
```

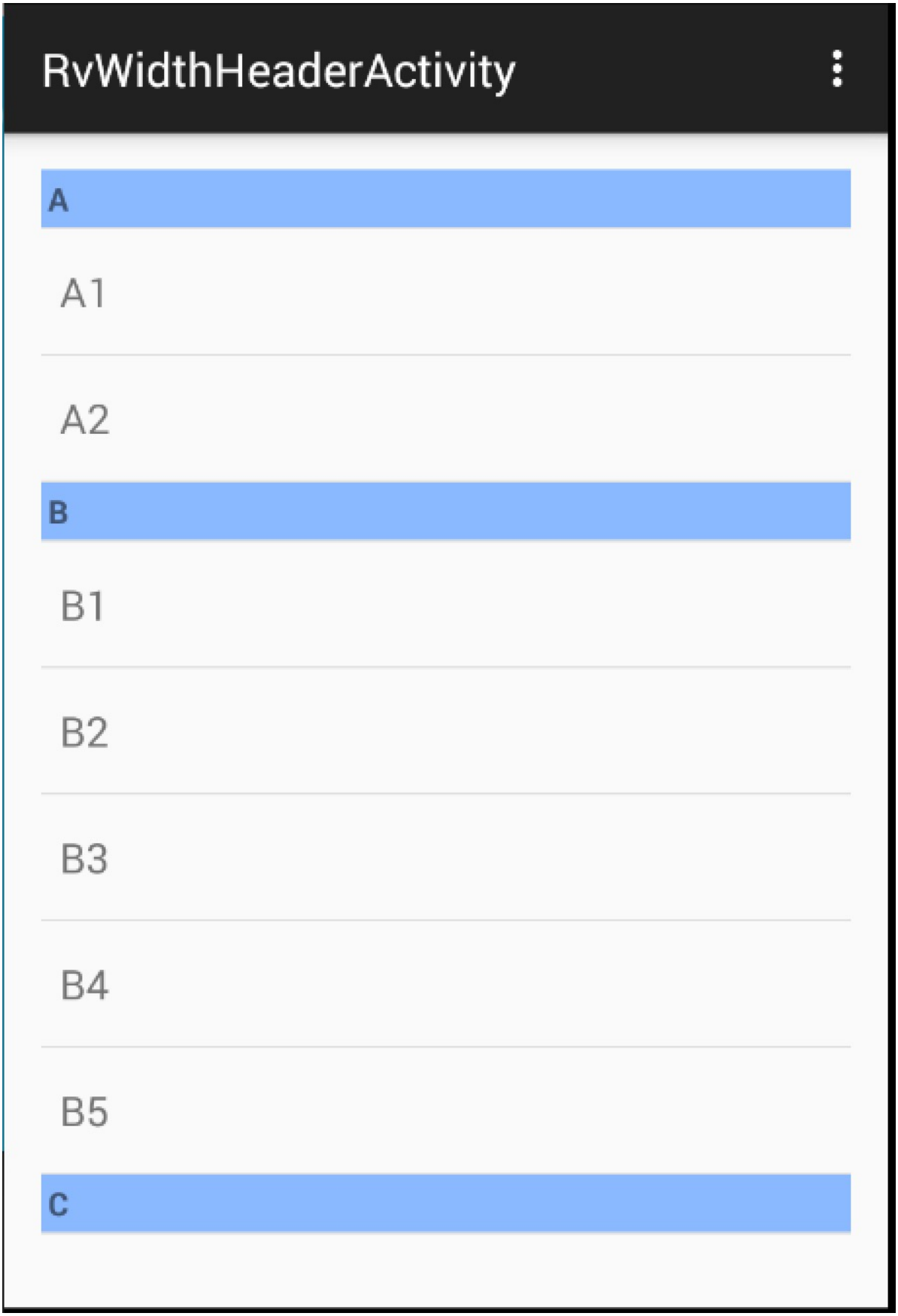
```
{
    @Override
    public int sectionHeaderLayoutId()
    {
        return R.layout.header;
    }

    @Override
    public int sectionTitleTextViewId()
    {
        return R.id.id_header_title;
    }

    @Override
    public String getTitle(String s)
    {
        return s.substring(0, 1);
    }
};12345678910111213141516171819 20
```

3个方法，一个指定header的布局文件，一个指定布局文件中显示title的TextView，最后一个用于指定显示什么样的标题（根据Adapter的Bean）。

接下来就很简单了：

```
mRecyclerView.setAdapter(new SectionAdapter<String>(this, R.layout.item_list,
mDatas, sectionSupport)
{
    @Override
    public void convert(ViewHolder holder, String s)
    {
        holder.setText(R.id.id_item_list_title, s);
    }
});12345678
```

这样就完了，效果图如下：



说了这么多，下面进入正题，看我们如何一步步完成整个封装的过程。

三、通用的ViewHolder

RecyclerView要求必须使用ViewHolder模式，一般我们在使用过程中，都需要去建立一个新的ViewHolder然后作为泛型传入Adapter。那么想要建立通用的Adapter，必须有个通用的ViewHolder。

首先我们确定下ViewHolder的主要的作用，实际上是通过成员变量存储对应的convertView中需要操作的字View，避免每次findViewById，从而提升运行的效率。

那么既然是通用的View，那么对于不同的ItemType肯定没有办法确定创建哪些成员变量View，取而代之的只能是个集合来存储了。

那么代码如下：

```
public class ViewHolder extends RecyclerView.ViewHolder
{
    private SparseArray<View> mViews;
    private View mConvertView;
    private Context mContext;

    public ViewHolder(Context context, View itemView, ViewGroup parent)
    {
        super(itemView);
        mContext = context;
        mConvertView = itemView;
        mViews = new SparseArray<View>();
    }


    public static ViewHolder get(Context context, ViewGroup parent, int
layoutId)
    {

        View itemView = LayoutInflater.from(context).inflate(layoutId, parent,
                false);
        ViewHolder holder = new ViewHolder(context, itemView, parent, position);
        return holder;
    }


    /**
     * 通过viewId获取控件
     *
     * @param viewId
     * @return
     */
    public <T extends View> T getView(int viewId)
    {
        View view = mViews.get(viewId);
        if (view == null)
        {
            view = mConvertView.findViewById(viewId);
            mViews.put(viewId, view);
        }
        return (T) view;
    }
}12345678910111213141516171819202122232425262728293031323334353637383940414 2
```

代码很简单，我们的ViewHolder继承自`RecyclerView.ViewHolder`，内部通过SparseArray来缓存我们itemView内部的子View，从而得到一个通用的ViewHolder。每次需要创建ViewHolder只需要传入我们的layoutId即可。

ok，有了通用的ViewHolder之后，我们的通用的Adapter分分钟就出来了。

四、通用的Adapter

我们的每次使用过程中，针对的数据类型Bean肯定是不同的，那么这里肯定要引入泛型代表我们的Bean，内部通过一个List代表我们的数据，ok，剩下的看代码：

```
package com.zhy.base.adapter.recyclerview;

import android.content.Context;
import android.support.v7.widget.RecyclerView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

import com.zhy.base.adapter.ViewHolder;

import java.util.List;

/**
 * Created by zhy on 16/4/9.
 */
public abstract class CommonAdapter<T> extends RecyclerView.Adapter<ViewHolder>
{
    protected Context mContext;
    protected int mLayoutId;
    protected List<T> mDatas;
    protected LayoutInflater mInflater;


    public CommonAdapter(Context context, int layoutId, List<T> datas)
    {
        mContext = context;
        mInflater = LayoutInflater.from(context);
        mLayoutId = layoutId;
        mDatas = datas;
    }

    @Override
    public ViewHolder onCreateViewHolder(final ViewGroup parent, int viewType)
    {
        ViewHolder viewHolder = ViewHolder.get(mContext, parent, mLayoutId);
        return viewHolder;
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position)
    {
        holder.updatePosition(position);
        convert(holder, mDatas.get(position));
    }

    public abstract void convert(ViewHolder holder, T t);
```

```
    @Override
    public int getItemCount()
    {
        return mDatas.size();
    }
}
12345678910111213141516171819202122232425262728293031323334353637383940414243444
5464748495051525354
```

继承自`RecyclerView.Adapter`，需要复写的方法还是比较少的。首先我们使用过程中传输我们的数据集mDatas，和我们item的布局文件layoutId。

`onCreateViewHolder`时，通过layoutId即可利用我们的通用的ViewHolder生成实例。

`onBindViewHolder`这里主要用于数据、事件绑定，我们这里直接抽象出去，让用户去操作。可以看到我们修改了下参数，用户可以拿到当前Item所需要的对象和viewHolder去操作。

那么现在用户的使用是这样的：

```
mRecyclerView.setAdapter(new CommonAdapter<String>(this, R.layout.item_list,
mDatas)
{
    @Override
    public void convert(ViewHolder holder, String s)
    {
        TextView tv = holder.getView(R.id.id_item_list_title);
        tv.setText(s);
    }
});123456789
```

看到这里，爽了很多，目前我们仅仅写了很少的代码，但是我们的通用的Adapter感觉已经初步完成了。

可以看到我们这里通过viewholder根据控件的id拿到控件，然后再进行数据绑定和事件操作，我们还能做些什么简化呢？

恩，我们可以通过一些辅助方法简化我们的代码，所以继续往下看。

五、进一步封装ViewHolder

我们的Item实际上使用的控件较多时候可能都是`TextView`,`ImageView`等，我们一般在convert方法都是去设置文本，图片什么的，那么我们可以在ViewHolder里面，写上如下的一些辅助方法：

```
class ViewHolder extends RecyclerView.Adapter<ViewHolder>
{
    //...
    public ViewHolder setText(int viewId, String text)
    {
        TextView tv = getView(viewId);
        tv.setText(text);
        return this;
    }

    public ViewHolder setImageResource(int viewId, int resId)
    {
        ImageView view = getView(viewId);
```

```
        view.setImageResource(resId);
        return this;
    }

    public ViewHolder setOnClickListener(int viewId,
                                         View.OnClickListener listener)
    {
        View view = getView(viewId);
        view.setOnClickListener(listener);
        return this;
    }
}
12345678910111213141516171819202122232425261234567891011121314151617181920212223242526
```

当然上面只给出了几个方法，你可以把常用控件的方法都写进去，并且在使用过程中不断完善即可。

有了一堆辅助方法后，我们的操作更加简化了一步。

```
mRecyclerView.setAdapter(new CommonAdapter<String>(this, R.layout.item_list,
mDatas)
{
    @Override
    public void convert(ViewHolder holder, String s)
    {
        //TextView tv = holder.getView(R.id.id_item_list_title);
        //tv.setText(s);
        holder.setText(R.id.id_item_list_title,s);
    }
});123456789101234567891012345678910
```

ok，到这里，我们的针对单种ViewItemType的通用Adapter就完成了，代码很简单也很少，但是简化效果非常明显。

ok，接下来我们考虑多种ItemViewType的情况。

六、多种ItemViewType

多种ItemViewType，一般我们的写法是：

- 复写`getItemViewType`，根据我们的bean去返回不同的类型
- `onCreateViewHolder`中根据itemView去生成不同的ViewHolder

如果大家还记得，我们的ViewHolder是通用的，唯一依赖的就是个layoutId。那么上述第二条就变成，根据不同的itemView告诉我用哪个layoutId即可，生成viewholder这种事我们通用adapter来做。

于是，引入一个接口：

```
public interface MultiItemTypeSupport<T>
{
    int getLayoutId(int itemType);

    int getItemViewType(int position, T t);
}123456123456
```

可以很清楚的看到，这个接口实际就是完成我们上述的两条工作。用户在使用过程中，通过实现上面两个方法，指明不同的Bean返回什么itemViewType,不同的itemView所对应的layoutId.

ok，有了上面这个接口，我们的参数就够了，下面开始我们的`MultiItemCommonAdapter`的编写。

```
public abstract class MultiItemCommonAdapter<T> extends CommonAdapter<T>
{
    protected MultiItemTypeSupport<T> mMultiItemTypeSupport;

    public MultiItemCommonAdapter(Context context, List<T> datas,
                                  MultiItemTypeSupport<T> multiItemTypeSupport)
    {
        super(context, -1, datas);
        mMultiItemTypeSupport = multiItemTypeSupport;
    }

    @Override
    public int getItemViewType(int position)
    {
        return mMultiItemTypeSupport.getItemViewType(position,
mDatas.get(position));
    }

    @Override
    public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
    {
        int layoutId = mMultiItemTypeSupport.getLayoutId(viewType);
        ViewHolder holder = ViewHolder.get(mContext, parent, layoutId;
        return holder;
    }

}12345678910111213141516171819202122232425 26
```

几乎没有几行代码，感觉简直不需要消耗脑细胞。`getItemViewType`用户的传入的`MultiItemTypeSupport.getItemViewType`完成，`onCreateViewHolder`中根据`MultiItemTypeSupport.getLayoutId`返回的layoutId，去生成ViewHolder即可。

ok，这样的话，我们的多种ItemViewType的支持也就完成了，一路下来感觉还是蛮轻松的~~~

七、添加分类Header

话说添加分类header，其实就是我们多种ItemViewType的一种，那么我们需要知道哪些参数呢？

简单思考下，我们需要：

1. header所对应的布局文件
2. 显示header的title对应的TextView
3. 显示的title是什么（一般肯定根据Bean生成）

ok，这样的话，我们依然引入一个接口，用于提供上述3各参数

```
public interface SectionSupport<T>
{
    public int sectionHeaderLayoutId();

    public int sectionTitleTextViewId();

    public String getTitle(T t);
}12345678
```

方法名应该很明确了，这里引入泛型，对应我们使用时的数据类型Bean。

刚才也说了我们的分类header是多种ItemViewType的一种，那么直接继承`MultiItemCommonAdapter`实现。

```
public abstract class SectionAdapter<T> extends MultiItemCommonAdapter<T>
{
    private SectionSupport mSectionSupport;
    private static final int TYPE_SECTION = 0;
    private LinkedHashMap<String, Integer> mSections;

    private MultiItemTypeSupport<T> headerItemTypeSupport = new
MultiItemTypeSupport<T>()
    {
        @Override
        public int getLayoutId(int itemType)
        {
            if (itemType == TYPE_SECTION)
                return mSectionSupport.sectionHeaderLayoutId();
            else
                return mLayoutId;
        }
        @Override
        public int getItemViewType(int position, T o)
        {
             return mSections.values().contains(position) ?
                    TYPE_SECTION :
                    1;
        }
    };

    @Override
    public int getItemViewType(int position)
    {
        return mMultiItemTypeSupport.getItemViewType(position, null);
    }

    final RecyclerView.AdapterDataObserver observer = new
RecyclerView.AdapterDataObserver()
    {
        @Override
        public void onChanged()
        {
            super.onChanged();
            findSections();
        }
    };


    public SectionAdapter(Context context, int layoutId, List<T> datas,
SectionSupport sectionSupport)
    {
        super(context, datas, null);
        mLayoutId = layoutId;
        mMultiItemTypeSupport = headerItemTypeSupport;
        mSectionSupport = sectionSupport;
        mSections = new LinkedHashMap<>();
```

```
        findSections();
        registerAdapterDataObserver(observer);
    }

    @Override
    protected boolean isEnabled(int viewType)
    {
        if (viewType == TYPE_SECTION)
            return false;
        return super.isEnabled(viewType);
    }

    @Override
    public void onDetachedFromRecyclerView(RecyclerView recyclerView)
    {
        super.onDetachedFromRecyclerView(recyclerView);
        unregisterAdapterDataObserver(observer);
    }

    public void findSections()
    {
        int n = mDatas.size();
        int nSections = 0;
        mSections.clear();

        for (int i = 0; i < n; i++)
        {
            String sectionName = mSectionSupport.getTitle(mDatas.get(i));

            if (!mSections.containsKey(sectionName))
            {
                mSections.put(sectionName, i + nSections);
                nSections++;
            }
        }

    }


    @Override
    public int getItemCount()
    {
        return super.getItemCount() + mSections.size();
    }

    public int getIndexForPosition(int position)
    {
        int nSections = 0;

        Set<Map.Entry<String, Integer>> entrySet = mSections.entrySet();
        for (Map.Entry<String, Integer> entry : entrySet)
        {
            if (entry.getValue() < position)
            {
                nSections++;
            }
        }
        return position - nSections;
```

```
    }

    @Override
    public void onBindViewHolder(ViewHolder holder, int position)
    {
        position = getIndexForPosition(position);
        if (holder.getItemViewType() == TYPE_SECTION)
        {
            holder.setText(mSectionSupport.sectionTitleTextViewId(),
mSectionSupport.getTitle(mDatas.get(position)));
            return;
        }
        super.onBindViewHolder(holder, position);
    }
}12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849505152535455565758596061626364656667686970717273747576777879808182838485868788899091929394959697989910010110210310410510610710810911011111211311411516117118119120121
```

根据我们之前的代码，使用`MultiItemCommonAdapter`，需要提供一个`MultiItemTypeSupport`，我们这里当然也不例外。可以看到上述代码，我们初始化了成员变量`headerItemTypeSupport`，分别对`getLayoutId`和`getItemViewType`进行了实现。

- `getLayoutId`如果type是header类型，则返回`mSectionSupport.sectionHeaderLayoutId()`；否则则返回mLayout.
- `getItemViewType`根据位置判断，如果当前是header所在位置，返回header类型常量；否则返回1.

ok，可以看到我们构造方法中调用了`findSections()`，主要为了存储我们的title和对应的position，通过一个Map`mSections`来存储。

那么对应的`getItemCount()`方法，我们多了几个title肯定总数会增加，所以需要复写。

在`onBindViewHolder`中我们有一行重置position的代码，因为我们的position变大了，所以在实际上绑定我们数据时，这个position需要还原，代码逻辑见`getIndexForPosition(position)`。

最后一点就是，每当我们的数据发生变化，我们的title集合，即`mSections`就可能会发生变化，所以需要重新生成，本来准备复写`notifyDataSetChanged`方法，在里面重新生成，没想到这个方法是final的，于是利用了`registerAdapterDataObserver(observer);`，在数据发生变化回调中重新生成，记得在`onDetachedFromRecyclerView`里面对注册的observer进行解注册。

ok，到此我们的增加Header就结束了~~

恩，上面是针对普通的Item增加header的代码，如果是针对多种ItemViewType呢？其实也很简单，这种方式需要传入`MultiItemTypeSupport`。那么对于headerItemTypeSupport中的`getItemViewType`等方法，不是header类型时，交给传入的`MultiItemTypeSupport`即可，大致的代码如下：

```
headerItemTypeSupport = new MultiItemTypeSupport<T>()
{
    @Override
    public int getLayoutId(int itemType)
    {
        if (itemType == TYPE_SECTION)
            return mSectionSupport.sectionHeaderLayoutId();
        else
            return multiItemTypeSupport.getLayoutId(itemType);
    }
```

```
    @Override
    public int getItemViewType(int position, T o)
    {
        int positionVal = getIndexForPosition(position);
        return mSections.values().contains(position) ?
                TYPE_SECTION :
                multiItemTypeSupport.getItemViewType(positionVal, o);
    }
};12345678910111213141516171819201
```

那么这样的话，今天的博客就结束了，有几点需要说明下：

本来是想接着以前的万能Adapter后面写，但是为了本文的独立和完整性，还是尽可能没有去依赖上篇博客的内容了。

此外，文章最后给出的开源代码与上述代码存在些许的差异，因为开源部分源码整合了ListView,RecyclerView等，而本文上述代码完全针对RecyclerView进行编写。

对于ItemClick,ItemLongClick的代码就不赘述了，其实都是通过itemView.setXXXListener完成，详细的参考代码即可。