下拉刷新组合控件的制作小结

在涉及联网操作的很多应用中会涉及到，下拉刷新的功能，国外一个 Johan Nilsson 的高人写了一个 listview 下拉刷新代码，因为项目中的需要，我将其进行扩展了一下，形成了一个 NPullToFreshContainer 类，该类和 ScrollView 一样限制了只能包含一个一级顶层控件或控件容器，至于该顶层控件就没有强制为具体的类型，只要派生自 View 就可以了，假如该顶层控件为 ViewGroup 时，里面可以放置 ScrollView、AdapterView 等的派生类，自然也就将下拉刷新从 listView 扩展为多种 View 了。对于实现新浪微博的下拉刷新功能该控件绰绰有余哦。

下拉刷新主要有上下两部分组成：上面一部分是下拉才出现的刷新视图，这里称其为好 headView；下面一部分是需要更新的内容视图，这里称其为 contentView。下图中 HelloWorld 上面的就是 headView，包括 HelloWorld 这个 textView 到关闭按钮这部分都是 contentView。

具体可以参看 Demo 源码中的 main.xml 文件。

下面来步骤解析该控件的实现。

首先，准备好 **headview** 资源

这里照搬 Johan Nilsson 里面用到的下拉刷新头资源，即 pull_to_refresh_header.xml 文件，由于 Demo 中有源码在这里就不罗列了。

第二，新建一个 NPullToFreshContainer，派生自 FrameLayout

之所以使用 FrameLayout，主要个人觉得 FrameLayout 层叠效果相对自由些，假如你想使用其它 ViewGroup 派生类，也可以做些尝试。由于派生自 FrameLayout，为此需要重载构造函数，在这里我们三个构造函数全部重载下，每一个里面都调用安装 headView 的 init 操作。

第三，编写 init 函数，添加 headView

这里就直接贴代码如下

```
        private void init(Context context) {
            mRefreshView = LayoutInflater.from(getContext()).inflate(R.layout.pull_to_refresh_header, null);

            mRefreshView.setBackgroundColor(0xf7f7f8);
            mRefreshViewImage = (ImageView)mRefreshView.findViewById(R.id.pull_to_refresh_image);
            mRefreshViewImage.setScaleType(ImageView.ScaleType.FIT_CENTER);
            float density = context.getResources().getDisplayMetrics().density;
            mRefreshViewImage.setMinimumHeight((int)(50 * density));

            mText = (TextView)mRefreshView.findViewById(R.id.pull_to_refresh_text);
            mDateTv = (TextView)mRefreshView.findViewById(R.id.pull_to_refresh_updated_at);
            mDateTv.setVisibility(View.INVISIBLE);
            mProgressBar = (ProgressBar)mRefreshView.findViewById(R.id.pull_to_refresh_progress);

            mAnimationUp = new RotateAnimation(0, -180, RotateAnimation.RELATIVE_TO_SELF, 0.5f,RotateAnimation.RELATIVE_TO_SELF, 0.5f);
            mAnimationUp.setInterpolator(new LinearInterpolator());
            mAnimationUp.setDuration(100);
            mAnimationUp.setFillAfter(true);

            mAnimationDown = new RotateAnimation(-180, 0,
                    RotateAnimation.RELATIVE_TO_SELF, 0.5f,
                    RotateAnimation.RELATIVE_TO_SELF, 0.5f);
            mAnimationDown.setInterpolator(new LinearInterpolator());
            mAnimationDown.setDuration(100);
            mAnimationDown.setFillAfter(true);
```

```
            //add pullToRefreshView
            if (mFirstLayout) {
                measureView(mRefreshView);
                HEAD_VIEW_HEIGHT = mRefreshView.getMeasuredHeight();
                mFirstLayout = false;
            }

            addView(mRefreshView, new
ViewGroup.LayoutParams(ViewGroup.LayoutParams.FILL_PARENT,
ViewGroup.LayoutParams.WRAP_CONTENT));
            mFling = new Flinger();
            //获取当前控件的最小滑动误判距离
            final ViewConfiguration configuration = ViewConfiguration.get(context);
            mTouchSlop = configuration.getScaledTouchSlop();
        }
```

HEAD_VIEW_HEIGHT 是 headView 的高度，原本这个值是放到 OnLayout 函数中去获取的，结果发现在某些特殊布局中，使用 WRAP_CONTENT 这个高度布局 addView 时，得到的 HEAD_VIEW_HEIGHT 高度不准确，为此在这里就通过调用 measureView 函数自测 View 的方式，提前将 headView 的高度量测出来固定好，而不是等 addView 后，在 onMeasure 或 onLayout 中获取高度了。measureView 函数的具体代码如下：

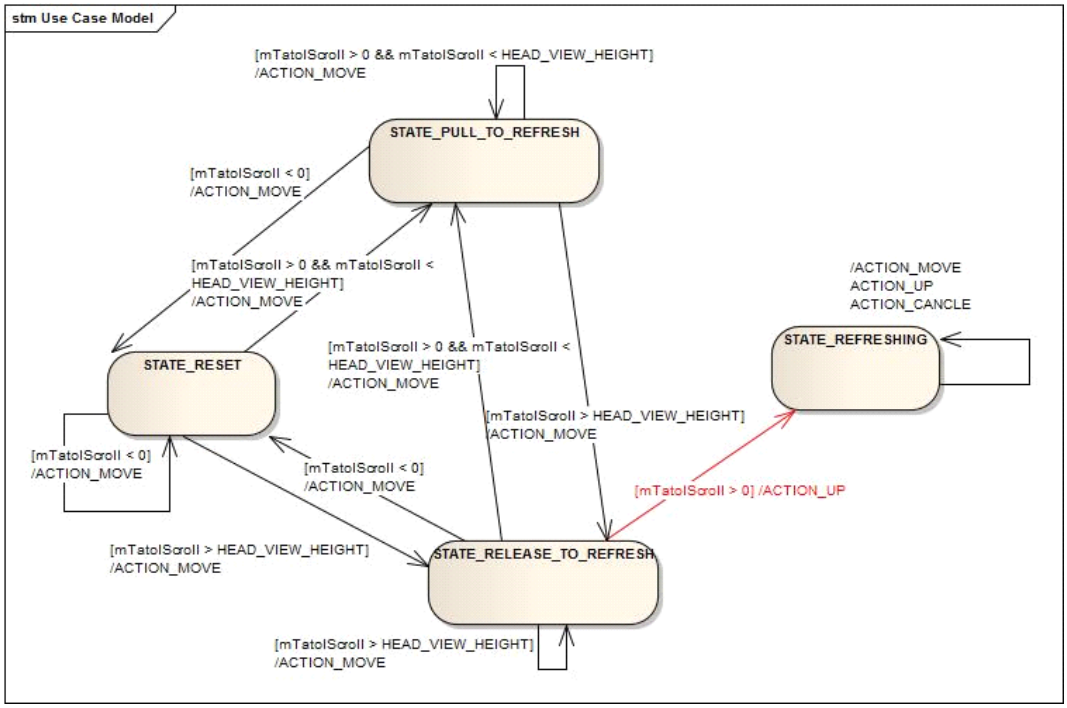
```
        private void measureView(View child) {
            ViewGroup.LayoutParams p = child.getLayoutParams();
            if (p == null) {
                p = new
ViewGroup.LayoutParams(ViewGroup.LayoutParams.FILL_PARENT,
ViewGroup.LayoutParams.WRAP_CONTENT);
            }

            int childWidthSpec = ViewGroup.getChildMeasureSpec(0, 0 + 0, p.width);
            int lpHeight = p.height;
            int childHeightSpec;
            if (lpHeight > 0) {
                childHeightSpec = MeasureSpec.makeMeasureSpec(lpHeight,
MeasureSpec.EXACTLY);
            } else {
                childHeightSpec = MeasureSpec.makeMeasureSpec(0,
MeasureSpec.UNSPECIFIED);
            }
            child.measure(childWidthSpec, childHeightSpec);
        }
```

至于代码内容，这里就不做过多展开了。

第四，由 Touch 事件来计算拖动位移

下拉效果的实现，靠的就是触摸时获得视图的位移重绘来实现的。根据 Touch 事件传递的原则，分别重载 onInterceptTouchEvent 和 onTouchEvent 函数，前者用于判断下拉刷新的第一次响应，即是否让当前 NPullToFreshContainer 对象成为事件的消费者并处理逻辑，后者用于成为事件消费者后的具体逻辑操作。

程序中为下拉刷新简单设置了起始/重置状态（STATE_RESET）、下拉可以刷新状态（STATE_PULL_TO_REFRESH）、释放进入刷新状态（STATE_RELEASE_TO_REFRESH）和刷新状态（STATE_REFRESHING）四个状态，头视图的高度为 HEAD_VIEW_HEIGHT，这四个状态在 ACTION_MOVE 和 ACTION_UP 事件中与内容视图的 mTop 位置的位移 mTatolScroll 之间就存在如下的切换关系



由于每个状态都标识出来的话，整个图会显得很乱，所以图上只绘制了 ACTION_MOVE 的动作时的状态转换、ACTION_UP 动作时的两个状态转换和 ACTION_CANCLE 的一个状态转换，其余条件下 ACTION_UP 和 ACTION_CANCLE 动作都将状态转换为 STATE_RESET，至于这点状态图上就没有标识出来了。当然也可以参阅 Demo 中两个函数的源码。在这里需要说明的是，为了避免与内部的 ScrollView 和 AdapterView 等派生类滚动效果混淆，为此，只有当内部可滚动的视图位于顶层位置时下拉效果才有效，这一步我采用了一个可以迭代询问的 isTouchView 函数来实现，具体参看下面源码。

```
        private boolean isTouchView(View view){
            boolean FirstLayout = true;
            if(view instanceof ViewGroup){
                int count = ((ViewGroup) view).getChildCount();
                for(int i = 0; i < count; i++){
                    View aView = ((ViewGroup) view).getChildAt(i);
                    int viewScrollY =    aView.getScrollY();
                    if(viewScrollY != 0){
                        return false;
```

```
                    }

                    if(aView instanceof AdapterView)
                    {
                        AdapterView adapterView = (AdapterView)aView;
                        final int position = adapterView.getFirstVisiblePosition();
                        if(position != 0){
                            return false;
                        }

                        if (adapterView.getChildCount() > 0) {
                            int vTop = adapterView.getChildAt(0).getTop();
                            if(vTop != 0){
                                return false;
                            }
                        }
                    }

                    if(isTouchView(aView)){
                        continue;
                    }else{
                        return false;
                    }
                }
            }else{
                int viewScrollY =    view.getScrollY();
                if(viewScrollY != 0){
                    return false;
                }else{
                    return true;
                }
            }
            return FirstLayout;
        }
```

第五，有上述获得的位移来重绘视图

根据 View 视图绘制的原理，在这里没有重载 onDraw 函数，而是重载了 onLayout 函数，并自写了一个 onInvalidate 函数来具体实现拖动产生位移后的 headView 和 contentView 两部分的绘制。这部分相对简单，就不做详细介绍了。

第六，增加一个 **Runnable** 对象，用于复原回滚操作

在下拉释放和更新完毕等过程中，已经下拉的 headView 需要弹性回复到刷新和隐藏状态，这里就参考常见 FlingRunnable 的写法，复写了一个 Flinger 类，内部最主要的对象就是 Scroller，他其实并不是视图上看到的滚动滑块之类的对象，他就是在内存中负责处理滚动操作的一个常用类。具体参看源码，这里同样不过多展开。

第七，增加设置刷新时调用异步操作的回调函数

下拉刷新时往往是调用联网获取内容或其他长任务的异步操作，NPullToFreshContainer 本身是不调用异步操作的，他只是负责实现 UI 上的效果。为此我们需要将异步操作的调用，通过设置回调函数的方式在下拉释放进行刷新时由外部的回调函数来调用异步操作。其实也是参考 Johan 大神的，具体代码如下：

```
public interface OnContainerRefreshListener {
/**
 * * Called when the list should be refreshed. *
 * <p>
 * * A call to {@link PullToRefreshListView #onRefreshComplete()} is *
 * expected to indicate that the refresh has completed.
 */
public void onContainerRefresh();
}
/**
 * Register a callback to be invoked when this list should be refreshed.
 *
 * @param onRefreshListener The callback to run.
 */
public void setOnRefreshListener(OnContainerRefreshListener onRefreshListener) {
    mOnRefreshListener = onRefreshListener;
}

public void onRefresh() {
if (mOnRefreshListener != null) {
    mOnRefreshListener.onContainerRefresh();
}
}
```

第八，开放外部主动调用刷新和结束刷新的操作函数

类似新浪微博，并非一定要通过下拉拖动的方式来调用刷新，也可以通过外部按钮来实现下拉刷新的操作，为此就必须提供外部主动调用刷新的操作函数，这里是 doRefresh 函数，同时外部调用异步操作函数结束是也需要来结束 UI 的刷新动画，这里就通过 void

onComplete(final String date)函数，其中的字符串在 Demo 中是时间信息。由于 Demo 中没有调用异步操作，所以必须通过点击完成按钮来模拟异步操作的结束。

在这里需要注意的时，原本想在 doRefresh 操作中来主动实现 contentView 内部位移控件的置顶操作，比如刷新开始时，将 ScrollView 的偏移复位，但是后来发现非顶层 View 的复位操作差异化很大，控制起来没有像 isTouchView 那样好控制，为此这个复位操作就交给外部对象自己去控制了。

好了，大致上整个下拉刷新组合控件就完成，详情可以参考源码。简单起见，当刷新过程中，上推 headView 只是做了简单的隐藏 headView 的操作，如果有需求需要上推时停止异步操作，就要另外增加操作函数，这里就不再展开讨论了。

Demo 代码下载地址 http://files.cnblogs.com/franksunny/ScrollerDemo.rar