**[Android NestedScrolling机制完全解析 带你玩转嵌套滑动](http://blog.csdn.net/lmj623565791/article/details/52204039)**

## 一、概述

Android在support.v4包中为大家提供了两个非常神奇的类：

* NestedScrollingParent
* NestedScrollingChild

这样的效果就非常适合使用NestedScrolling机制去完成，并且CoordinatorLayout背后其实也是利用着这套机制，So，我相信你已经明白这套机制可以用来干嘛了。

但是，我相信你还有个问题

* 这个机制相比传统的自定义ViewGroup事件分发处理有什么优越的地方吗？

恩，我们简单分析下：

按照上图：

假设我们按照传统的事件分发去理解，首先我们滑动的是下面的内容区域，而移动却是外部的ViewGroup在移动，所以按照传统的方式，肯定是外部的Parent拦截了内部的Child的事件；但是，上述效果图，当Parent滑动到一定程度时，Child又开始滑动了，中间整个过程是没有间断的。从正常的事件分发（不手动调用分发事件，不手动去发出事件）角度去做是不可能的，因为当Parent拦截之后，是没有办法再把事件交给Child的，事件分发，对于拦截，相当于一锤子买卖，只要拦截了，当前手势接下来的事件都会交给Parent(拦截者)来处理。

但是NestedScrolling机制来处理这个事情就很好办，所以对这个机制进行深入学习，一来有助于我们编写嵌套滑动时一些特殊的效果；二来是我为了对CoordinatorLayout做分析的铺垫~~~

ps：具体在哪个v4版本中添加的，就不去深究了，如果你的v4中没有上述两个类，升级下你的v4版本。NestedScrolling机制这个词，个人称呼，不清楚官方有没有这么叫，勿深究。

## 二、预期效果

当然讲解这两个类，肯定要有案例的支撑，不然太过于空洞了。好在，我这里有个非常好的案例可以来描述：

很久以前，我写过这样一篇文章：

* [Android 自定义控件 轻松实现360软件详情页](http://blog.csdn.net/lmj623565791/article/details/43649913)

完全按照传统的方式去编写的，而且为了连续滑动，做了一些非常特殊处理，比如手动去分发DOWN事件类的，有兴趣可以阅读下。

今天我们就利用这个效果，作为NestedSroll机制的案例，最后我们还会简单分析一下源码，其实源码还是比较简单的~~

ps:CoordinatorLayout可以很方便实现该效果，后续的文章也会对CoordinateLayout做一些分析。

## 三、实现

上述效果图，分为3部分：顶部布局；中间的ViewPager指示器；以及底部的RecyclerView；

RecyclerView其实就是NestedSrollingChild的实现类，所以本例主要的角色是去实现NestedScrollingParent.

### （1）布局文件

首先预览下布局文件，脑子里面有个大致的布局：

<com.zhy.view.StickyNavLayout xmlns:tools="http://schemas.android.com/tools"

xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical" >

<RelativeLayout

android:id="@id/id\_stickynavlayout\_topview"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="#4400ff00" >

<TextView

android:layout\_width="match\_parent"

android:layout\_height="256dp"

android:gravity="center"

android:text="软件介绍"

android:textSize="30sp"

android:textStyle="bold" />

</RelativeLayout>

<com.zhy.view.SimpleViewPagerIndicator

android:id="@id/id\_stickynavlayout\_indicator"

android:layout\_width="match\_parent"

android:layout\_height="50dp"

android:background="#ffffffff" >

</com.zhy.view.SimpleViewPagerIndicator>

<android.support.v4.view.ViewPager

android:id="@id/id\_stickynavlayout\_viewpager"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

>

</android.support.v4.view.ViewPager>

</com.zhy.view.StickyNavLayout>

StickyNavLayout是直接继承自LinearLayout的，并且设置的是orientation="vertical"，所以直观的就是控件按顺序纵向排列，至于测量需要做一些特殊的处理，因为不是本文的重点，可以自己查看源码，或者上面提到的文章。

### (2) 实现NestedScrollingParent

NestedScrollingParent是一个接口，实现它需要实现如下方法：

public boolean onStartNestedScroll(View child, View target, int nestedScrollAxes);

public void onNestedScrollAccepted(View child, View target, int nestedScrollAxes);

public void onNestedScroll(View target, int dxConsumed, int dyConsumed,

int dxUnconsumed, int dyUnconsumed);

public void onNestedPreScroll(View target, int dx, int dy, int[] consumed);

public boolean onNestedFling(View target, float velocityX, float velocityY, boolean consumed);

public boolean onNestedPreFling(View target, float velocityX, float velocityY);

public int getNestedScrollAxes();

在写具体的实现前，先对需要用到的上述方法做一下简单的介绍：

* onStartNestedScroll该方法，一定要按照自己的需求返回true，该方法决定了当前控件是否能接收到其内部View(非并非是直接子View)滑动时的参数；假设你只涉及到纵向滑动，这里可以根据nestedScrollAxes这个参数，进行纵向判断。
* onNestedPreScroll该方法的会传入内部View移动的dx,dy，如果你需要消耗一定的dx,dy，就通过最后一个参数consumed进行指定，例如我要消耗一半的dy，就可以写consumed[1]=dy/2
* onNestedFling你可以捕获对内部View的fling事件，如果return true则表示拦截掉内部View的事件。

主要关注的就是这三个方法~

这里内部View表示不一定非要是直接子View，只要是内部View即可。

下面看一下我们具体的实现：

public class StickyNavLayout extends LinearLayout implements NestedScrollingParent

{

@Override

public boolean onStartNestedScroll(View child, View target, int nestedScrollAxes)

{

return (nestedScrollAxes & ViewCompat.SCROLL\_AXIS\_VERTICAL) != 0;

}

@Override

public void onNestedPreScroll(View target, int dx, int dy, int[] consumed)

{

boolean hiddenTop = dy > 0 && getScrollY() < mTopViewHeight;

boolean showTop = dy < 0 && getScrollY() > 0 && !ViewCompat.canScrollVertically(target, -1);

if (hiddenTop || showTop)

{

scrollBy(0, dy);

consumed[1] = dy;

}

}

@Override

public boolean onNestedPreFling(View target, float velocityX, float velocityY)

{

if (getScrollY() >= mTopViewHeight) return false;

fling((int) velocityY);

return true;

}

}

* onStartNestedScroll中，我们判断了如果是纵向返回true，这个一般是需要内部的View去传入的，你要是不确定，或者担心内部View编写的不规范，你可以直接return true；
* onNestedPreScroll中，我们判断，如果是上滑且顶部控件未完全隐藏，则消耗掉dy，即consumed[1]=dy;如果是下滑且内部View已经无法继续下拉，则消耗掉dy，即consumed[1]=dy，消耗掉的意思，就是自己去执行scrollBy，实际上就是我们的StickNavLayout滑动。
* 此外，这里还处理了fling，通过onNestedPreFling方法，这个可以根据自己需求定了，当顶部控件显示时，fling可以让顶部控件隐藏或者显示。

对于fling方法，我们利用了OverScroll的fling的方法，对于边界检测，是重写了scrollTo方法：

public void fling(int velocityY)

{

mScroller.fling(0, getScrollY(), 0, velocityY, 0, 0, 0, mTopViewHeight);

invalidate();

}

@Overridepublic void scrollTo(int x, int y)

{

if (y < 0)

{

y = 0;

}

if (y > mTopViewHeight)

{

y = mTopViewHeight;

}

if (y != getScrollY())

{

super.scrollTo(x, y);

}

}

详细的解释可以看上面提到的博客，这里就不重复了。

到这里呢，可以看到NestedScrolling机制说白了非常简单：

就是NestedScrollingParent内部的View，在滑动到时候，会首先将dx、dy传入给NestedScrollingParent，NestedScrollingParent可以决定是否对其进行消耗，一般会根据需求消耗部分或者全部(不过这里并没有实际的约束，你可以随便写消耗多少，可能会对内部View造成一定的影响）。

用白话和原本的事件分发机制作对比就是这样的（针对正常流程下一次手势）：

* 事件分发是这样的：子View首先得到事件处理权，处理过程中，父View可以对其拦截，但是拦截了以后就无法再还给子View（本次手势内）。
* NestedScrolling机制是这样的：内部View在滚动的时候，首先将dx,dy交给NestedScrollingParent，NestedScrollingParent可对其进行部分消耗，剩余的部分还给内部View。

具体的源码会比本博文复杂，因为涉及到触摸非内部View区域的一些交互，非本博文重点，可以参考源码。

## 四、原理

原理其实就是看内部View什么时候回调NestedScrollingParent各种方法的，直接定位到内部View的onTouchEvent：

@Overridepublic boolean onTouchEvent(MotionEvent e) {

switch (action) {

case MotionEvent.ACTION\_DOWN: {

int nestedScrollAxis = ViewCompat.SCROLL\_AXIS\_NONE;

if (canScrollHorizontally) {

nestedScrollAxis |= ViewCompat.SCROLL\_AXIS\_HORIZONTAL;

}

if (canScrollVertically) {

nestedScrollAxis |= ViewCompat.SCROLL\_AXIS\_VERTICAL;

}

startNestedScroll(nestedScrollAxis);

} break;

case MotionEvent.ACTION\_MOVE: {

if (dispatchNestedPreScroll(dx, dy, mScrollConsumed, mScrollOffset)) {

dx -= mScrollConsumed[0];

dy -= mScrollConsumed[1];

vtev.offsetLocation(mScrollOffset[0], mScrollOffset[1]);

}

} break;

case MotionEvent.ACTION\_UP: {

fling((int) xvel, (int) yvel);

resetTouch();

} break;

case MotionEvent.ACTION\_CANCEL: {

cancelTouch();

} break;

}

return true;

}

可以看到:

ACTION\_DOWN调用了startNestedScroll；ACTION\_MOVE中调用了dispatchNestedPreScroll；ACTION\_UP可能会触发fling以调用resetTouch。

startNestedScroll内部实际上：

#NestedScrollingChildHelperpublic boolean startNestedScroll(int axes) {

if (hasNestedScrollingParent()) {

// Already in progress

return true;

}

if (isNestedScrollingEnabled()) {

ViewParent p = mView.getParent();

View child = mView;

while (p != null) {

if (ViewParentCompat.onStartNestedScroll(p, child, mView, axes)) {

mNestedScrollingParent = p;

ViewParentCompat.onNestedScrollAccepted(p, child, mView, axes);

return true;

}

if (p instanceof View) {

child = (View) p;

}

p = p.getParent();

}

}

return false;

}

去寻找NestedScrollingParent，然后回调onStartNestedScroll和onNestedScrollAccepted。

dispatchNestedPreScroll中会回调onNestedPreScroll方法，内部的scrollByInternal中还会回调onNestedScroll方法。

fling中会回调onNestedPreFling和onNestedFling方法。

resetTouch中则会回调onStopNestedScroll。

代码其实没什么贴的，大家直接找到onTouchEvent一眼就能看到，调用的方法名都是dispatchNestedXXX方法，实际内部都是通过NestedScrollingChildHelper实现的。

所以如果你需要实现和NestedScrollingParent协作的内部View，记得实现NestedScrollingChild，然后内部借助NestedScrollingChildHelper这个辅助类，核心的方法都封装好了，你只需要在恰当的实际去传入参数调用方法即可。

ok，这样的一个机制一定要去试试，很多滑动相关的效果都可以借此实现；此外，后面可能会对CoordinatorLayout写一些博文~