安卓 View onTouch onClick 事件分发处理机制

deepwaterooo

June 6, 2022

Contents

1 Button 的 onTouch，onClick，onLongClick 事件发生先后顺序和关联：

1.1 一，onTouch 返回 false

- 首先是 onTouch 事件的 down 事件发生，此时，如果长按，触发 onLongClick 事件；
- 然后是 onTouch 事件的 up 事件发生，up 完毕，最后触发 onClick 事件。

1.2 二，onTouch 返回 true

- 首先是 onTouch 事件的 down 事件发生，然后是 onTouch 事件的 up 事件发生；期间不触发 onClick 和 onLongClick 事件

1.3 三，onTouch：down 返回 true，up 返回 false：结果同二。

- 机制分析：
  - onTouch 事件中：down 事件返回值标记此次事件是否为点击事件（返回 false，是点击事件；返回 true，不记为点击事件），而 up 事件标记此次事件结束时间，也就是判断是否为长按。
- 只要当 down 返回 true 时候，系统将不把本次事件记录为点击事件，也就不会触发 onClick 或者 onLongClick 事件了。因此尽管当 up 的时候返回 false，系统也不会继续触发 onClick 事件了。

1.4 四，onTouch：down 返回 false，up 返回 true：

- 首先是 onTouch 事件的 down 事件发生，此时：
- 长按，触发 onLongClick 事件，然后是 onTouch 事件的 up 事件发生，完毕。
- 短按，先触发 onTouch 的 up 事件，到一定时间后，自动触发 onLongClick 事件。
- 机制分析：
  - onTouch 事件中：down 事件返回值标记此次事件是否为点击事件（返回 false，是点击事件；返回 true，不记为点击事件），而 up 事件标记此次事件结束时间，也就是判断是否为长按。
  - 当 down 返回 false，标记此次事件为点击事件，而 up 返回了 true，则表示此次事件一直没有结束，也就是一直长按下去了，达到长按临界时间后，自然触发长按事件，而 onClick 事件没有触发到

1.5 some code theories

```
/**
 * Implement this method to handle touch screen motion events.
 *
 * @param event The motion event.
 * @return True if the event was handled, false otherwise.
 */
public boolean onTouchEvent(MotionEvent event) {
    final int viewFlags = mViewFlags;
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (event.getAction() == MotionEvent.ACTION_UP && (mPrivateFlags & PRESSED) != 0) {
            mPrivateFlags &= ~PRESSED;
            refreshDrawableState();
        }
        // A disabled view that is clickable still consumes the touch
        // events, it just doesn't respond to them.
        return (((viewFlags & CLICKABLE) == CLICKABLE ||
                 (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));
    }
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }
    if (((viewFlags & CLICKABLE) == CLICKABLE ||
         (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
        switch (event.getAction()) {
        case MotionEvent.ACTION_UP:
            boolean prepressed = (mPrivateFlags & PREPRESSED) != 0;
            if ((mPrivateFlags & PRESSED) != 0 || prepressed) {
                // take focus if we don't have it already and we should in
                // touch mode.
                boolean focusTaken = false;
                if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
```

```
            focusTaken = requestFocus();
        }
        if (prepressed) {
            // The button is being released before we actually
            // showed it as pressed.  Make it show the pressed
            // state now (before scheduling the click) to ensure
            // the user sees it.
            mPrivateFlags |= PRESSED;
            refreshDrawableState();
        }
        if (!mHasPerformedLongPress) {
            // This is a tap, so remove the longpress check
            removeLongPressCallback();
            // Only perform take click actions if we were in the pressed state
            if (!focusTaken) {                                  focustaken 为 false 时, click才触发。
                    // Use a Runnable and post this rather than calling
                    // performClick directly. This lets other visual state
                    // of the view update before click actions start.
                    if (mPerformClick == null) {
                        mPerformClick = new PerformClick();
                    }
                if (!post(mPerformClick)) {
                    performClick();
                }
            }
        }
        if (mUnsetPressedState == null) {
            mUnsetPressedState = new UnsetPressedState();
        }
        if (prepressed) {
            postDelayed(mUnsetPressedState,
                    ViewConfiguration.getPressedStateDuration());
        } else if (!post(mUnsetPressedState)) {
            // If the post failed, unpress right now
            mUnsetPressedState.run();
        }
        removeTapCallback();
    }
    break;
case MotionEvent.ACTION_DOWN:
    mHasPerformedLongPress = false;
    if (performButtonActionOnTouchDown(event)) {
        break;
    }
    // Walk up the hierarchy to determine if we're inside a scrolling container.
    boolean isInScrollingContainer = isInScrollingContainer();
    // For views inside a scrolling container, delay the pressed feedback for
    // a short period in case this is a scroll.
    if (isInScrollingContainer) {
        mPrivateFlags |= PREPRESSED;
        if (mPendingCheckForTap == null) {
            mPendingCheckForTap = new CheckForTap();
        }
        postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
    } else {
        // Not inside a scrolling container, so show the feedback right away
        mPrivateFlags |= PRESSED;
        refreshDrawableState();
        checkForLongClick(0);
    }
    break;
case MotionEvent.ACTION_CANCEL:
    mPrivateFlags &= ~PRESSED;
    refreshDrawableState();
    removeTapCallback();
    break;
case MotionEvent.ACTION_MOVE:
    final int x = (int) event.getX();
    final int y = (int) event.getY();
    // Be lenient about moving outside of buttons
    if (!pointInView(x, y, mTouchSlop)) {
        // Outside button
        removeTapCallback();
        if ((mPrivateFlags & PRESSED) != 0) {
            // Remove any future long press/tap checks
```

```
                    removeLongPressCallback();
                    // Need to switch from pressed to not pressed
                    mPrivateFlags &= ~PRESSED;
                    refreshDrawableState();
                }
            }
            break;
        }
        return true;
    }
```

2 SimpleOnGestureListener 静态内部类

2.1 构造方法

- GestureDetector 一共有 5 种构造函数，但有 2 种被废弃了，1 种是重复的，所以只需要关注其中的 2 种构造函数即可，如下：

```
public GestureDetector(Context context, OnGestureListener listener)
public GestureDetector(Context context, OnGestureListener listener, Handler handler)
```

- 第 1 种构造函数里面需要传递两个参数，Context(上下文) 和 OnGestureListener(手势监听器)，这个很容易理解，也是最经常使用的一种。
- 第 2 种构造函数则需要多传递一个 Handler 作为参数，这个有什么作用呢？其实作用也非常简单，这个 Handler 主要是为了给 GestureDetector 提供一个 Looper。
- 在通常情况下是不需这个 Handler 的，因为它会在内部自动创建一个 Handler 用于处理数据，如果在主线程中创建 GestureDetector，那么它内部创建的 Handler 会自动获得主线程的 Looper，然而如果在一个没有创建 Looper 的子线程中创建 GestureDetector 则需要传递一个带有 Looper 的 Handler 给它，否则就会因为无法获取到 Looper 导致创建失败。重点是传递的 Handler 一定要有 Looper，重点是 Looper，而非 Handler。

2.2 创建方法：在没有 Looper 的地方使用 GestureDetector 时可以通过以下方法：

```
// 方法一：在主线程创建 Handler，使用第 2 种构造方法进行创建
final Handler handler = new Handler();
new Thread(new Runnable() {
        @Override public void run() {
            final GestureDetector detector = new GestureDetector(getContext(), new GestureDetector.SimpleOnGestureListener(
            // ...
        }
    }).start();

// 方法二：在子线程创建 Handler，并且指定 Looper，使用第 2 种构造方法进行创建
new Thread(new Runnable() {
        @Override public void run() {
            final Handler handler = new Handler(Looper.getMainLooper());
            final GestureDetector detector = new GestureDetector(getContext(), new GestureDetector.SimpleOnGestureListener(
            // ...
        }
    }).start();

// 方法三：子线程准备了 Looper，那么可以直接使用第 1 种构造方法进行创建
new Thread(new Runnable() {
        @Override public void run() {
            Looper.prepare(); // 初始化 Looper(重点)
            final GestureDetector detector = new GestureDetector(getContext(), new GestureDetector.SimpleOnGestureListener(
            // ...
        }
    }).start();
```

2.3 接口方法的定义

- GestureDetector 类中已经为我们定义了一个静态内部类 SimpleOnGestureListener，它实现了 OnGestureListener，OnDoubleTapListener，OnContextClickListener 接口，定义为

```
 public static class SimpleOnGestureListener
     implements OnGestureListener, OnDoubleTapListener, OnContextClickListener {
 }
// 下面是一个例子
private class simpleGestureListener extends GestureDetector.SimpleOnGestureListener {
    /*****OnGestureListener 的函数 *****/
    @Override public boolean onDown(MotionEvent e) {
        return false;
    }
    @Override public void onShowPress(MotionEvent e) {
    }
    @Override public boolean onSingleTapUp(MotionEvent e) {
        return true;
    }
    @Override public boolean onScroll(MotionEvent e1, MotionEvent e2,
                            float distanceX, float distanceY) {
        return true;
    }
    @Override public void onLongPress(MotionEvent e) {
    }
    @Override public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX,
                           float velocityY) {
        return true;
    }

    /*****OnDoubleTapListener 的函数 *****/
    @Override public boolean onSingleTapConfirmed(MotionEvent e) {
        return true;
    }
    @Override public boolean onDoubleTap(MotionEvent e) {
        return true;
    }
    @Override public boolean onDoubleTapEvent(MotionEvent e) {
        return true;
    }
}
```

- SimpleOnGestureListener 类内重写接口中的所有方法，但是都是空实现，返回的布尔值都是 false。主要作用是方便我们继承这个类有选择的复写回调方法，而不是实现接口去重写所有的方法。
- onTouchEvent() 方法用来分析传入的事件，如果匹配的话就去触发 OnGestureListener 中相应的回调方法。
- 如果要监听双击事件必须调用 GestureDector.setOnDoubleTapListener()
- 上面所有的回调方法的返回值都是 boolean 类型，和 View 的事件传递机制一样，返回 true 表示消耗了事件，flase 表示没有消耗。
- 要实现捕捉屏幕手势，除了在 Activity 中创建 gestureDetector 外，还有一种思路：构建一个 Overlay, 这个 Overlay 实现 OnGestureListener 接口，使其维护自己的 GestureDetector。在主视图上添加这个 Overlay, 并传入相应的 listener，即可实现捕捉手势的功能。

3 onInterceptTouchEvent()

- 这个方法其实以前不怎么关注，后来自定义布局用的多了，发现很多时候都必须要重载这个函数，
- 一般重载这个函数地方就是你自定义了一个布局，extends LinearLayout 等等布局

- onInterceptTouchEvent() 是 ViewGroup 类中的方法, 而不是来自 Activity.
- 您可以通过将逻辑从 onInterceptTouchEvent() 移动到 dispatchTouchEvent(MotionEvent ev) 来实现所需的行为. 请记住调用 dispatchTouchEvent(MotionEvent ev) 的超类实现来处理应该正常处理的事件.
- 另请注意, 只有在 delta 大于 system constant for touch slop 时才应考虑移动. 我建议用户通过测试 yDelta / 2> 确保用户正在按照您想要的方向滑动. xDelta 而不是 yDelta> xDelta.

```
public class Game extends Activity {
    private int mSlop;
    private float mDownX;
    private float mDownY;
    private boolean mSwiping;
    @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.game_activity);
        ViewConfiguration vc = ViewConfiguration.get(this);
        mSlop = vc.getScaledTouchSlop();
//other code....
    }
    @Override
        public boolean dispatchTouchEvent(MotionEvent ev) {
        switch (ev.getAction()) {
        case MotionEvent.ACTION_DOWN:
            mDownX = ev.getX();
            mDownY = ev.getY();
            mSwiping = false;
            break;
        case MotionEvent.ACTION_CANCEL:
        case MotionEvent.ACTION_UP:
            if(mSwiping) {
                swipeScreen(); //if action recognized as swipe then swipe
            }
            break;
        case MotionEvent.ACTION_MOVE:
            float x = ev.getX();
            float y = ev.getY();
            float xDelta = Math.abs(x - mDownX);
            float yDelta = Math.abs(y - mDownY);
            if (yDelta > mSlop && yDelta / 2 > xDelta) {
                mSwiping = true;
                return true;
            }
            break;
        }
        return super.dispatchTouchEvent(ev);
    }
}
```

- 一般除了自己的业务处理外，返回值只有两种，

3.1 第一种：让父类去处理返回值，跟踪 android 源码你会知道返回的是 false

- 作用：让自定义布局上面的所有子 view 例如 button imageview 等可以被点击

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    // TODO Auto-generated method stub
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {
        downX = (int) ev.getX();
        downY = (int) ev.getY();
        windowX = (int) ev.getX();
        windowY = (int) ev.getY();
        setOnItemClickListener(ev);
    }
    return super.onInterceptTouchEvent(ev);
}
```

3.2 第二种：直接返回 true

- 作用：让自定义布局上面的所有子 view 不可以被点击

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    return true;
}
```

3.3 ViewGroup 中有一个 onInterceptTouchEvent 方法，我们来看一下这个方法的源码

```
/**
 * Implement this method to intercept all touch screen motion events.  This
 * allows you to watch events as they are dispatched to your children, and
 * take ownership of the current gesture at any point.
 *
 * <p>Using this function takes some care, as it has a fairly complicated
 * interaction with {@link View#onTouchEvent(MotionEvent)
 * View.onTouchEvent(MotionEvent)}, and using it requires implementing
 * that method as well as this one in the correct way.  Events will be
 * received in the following order:
 *
 * <ol>
 * <li> You will receive the down event here.
 * <li> The down event will be handled either by a child of this view
 * group, or given to your own onTouchEvent() method to handle; this means
 * you should implement onTouchEvent() to return true, so you will
 * continue to see the rest of the gesture (instead of looking for
 * a parent view to handle it).  Also, by returning true from
 * onTouchEvent(), you will not receive any following
 * events in onInterceptTouchEvent() and all touch processing must
 * happen in onTouchEvent() like normal.
 * <li> For as long as you return false from this function, each following
 * event (up to and including the final up) will be delivered first here
 * and then to the target's onTouchEvent().
 * <li> If you return true from here, you will not receive any
 * following events: the target view will receive the same event but
 * with the action {@link MotionEvent#ACTION_CANCEL}, and all further
 * events will be delivered to your onTouchEvent() method and no longer
 * appear here.
 * </ol>
 *
 * @param ev The motion event being dispatched down the hierarchy.
 * @return Return true to steal motion events from the children and have
 * them dispatched to this ViewGroup through onTouchEvent().
 * The current target will receive an ACTION_CANCEL event, and no further
 * messages will be delivered here.
 */
public boolean onInterceptTouchEvent(MotionEvent ev) {
//  返回 false：表示当前的 layoutgroup 不处理触摸事件，交由其子视图们去处理
    return false;
//  返回 true：表示当前的 layoutgroup 拦截处理所有的触摸事件，其所有的子视图将无法接收到任何触摸事件
}
```

4 android 事件分发原理

4.1 事件分发 4 大方法

```
dispatchTouchEvent    // 事件分发
onInterceptTouchEvent // 事件拦截
onTouchEvent          // 事件处理
requestDisallowInterceptTouchEvent(true)
```

- ViewGroup 4 个方法都有，View 没有事件拦截方法。这 4 个方法就是 Android 事件分发和处理的具体方法了，我们在实际处理时都是重写这 4 个方法。
- 这 3 个方法都有一个 boolean 的返回值，onInterceptTouchEvent 方法表示我要拦截这个事件，onTouchEvent 表示我处理了事件

4.1.1 dispatchTouchEvent

- dispatchTouchEvent 方法作为 view 的事件处理的 API 入口，内部会调用 onInterceptTouchEvent 和 onTouchEvent 来计算返回的 boolean 值，一般我们都不会动 dispatchTouchEvent 方法，因为没有意思。但是我们直接返回 true 时，表示我消费这个事件了

4.1.2 onInterceptTouchEvent

- onInterceptTouchEvent 方法默认是返回 false 的，表示不会拦截这个事件，但是若是返回 true 则表示这个事件我拦截住了，就不再往我下一级 view 传递了，然后会把事件交给自己的 onTouchEvent 方法

4.1.3 onTouchEvent

- onTouchEvent 方法可以获取具体的触控参数，进行手势方向，当前手势类型判断，可以记录触摸点的 x，y 坐标。onTouchEvent 方法若是返回 true ，则表示这个事件我已经处理过了，那么整个事件传递过程就结速了，否则的话这个事件会原路返回，去一级一级的跑上一层 view 的 onTouchEvent 方法，直到有人返回 true 或是没有了。

4.1.4 requestDisallowInterceptTouchEvent

- view.getParent().requestDisallowInterceptTouchEvent(true) , 可以让内层的 view 获取到事件，而忽略上层视图的拦截。true 表示接受事件，fasle 表示不接受事件。原理分析看这里：
- ScrollView 嵌套 ScrollView 的滑动冲突: https://www.jianshu.com/p/eedc98eecf02

4.2 事件分发的顺序

- 总的来说，事件是由其父视图向子视图传递，按前面的图来说是这样。

```
Activity - > PhoneWindow - > DecorView - > ViewGroup - > ... - > View
```

- window 根据视图树顺序，从外到内遍历所有的 view ，询问每一个遍历出来的 view 你要不要消费触摸事件
- 每个 view 会跑自己的 dispatchTouchEvent 方法，判断自己是否需要消费这个事件
- dispatchTouchEvent 方法首先会询问 onInterceptTouchEvent 方法，是否要拦截这个事件。
- 若是拦截则会跑自己的 onTouchEvent 方法，若不拦截则会传递给下一级 view，调用下一级 view 的 dispatchTouchEvent 方法
- onTouchEvent 若是返回 true ，表示事件实际处理过了，那么整个事件传递在这个节点就结速了，不再传递了。若是返回 fasle，这个事件会原路返回，去一级一级的跑上一层 view 的 onTouchEvent 方法，直到有人返回 true 或是没有了。
- 再总结一遍的话：

- **事件机制：**
  ①分发： 将TouchEvent对象从Activity对象开始，由外向内分发给对应的布局和子View对象（由外向内分发）。

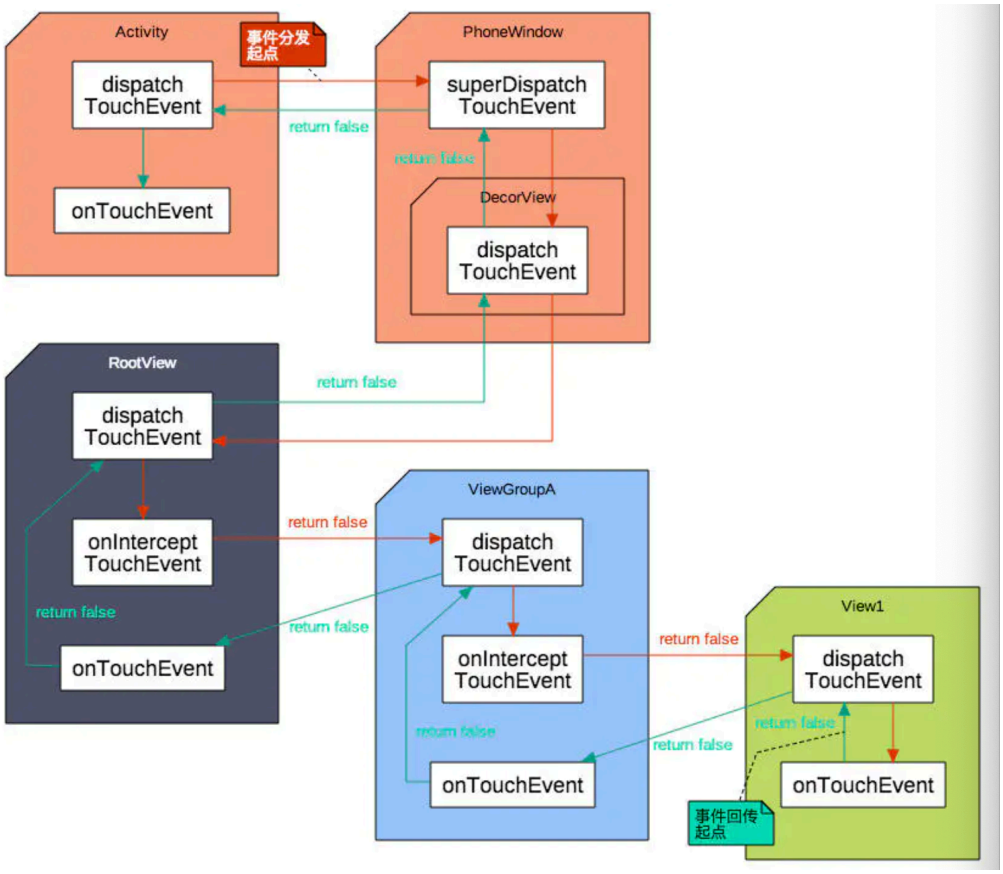
  ②处理： 回调OnTouchListener的boolean onTouch()
  回调View的boolean onTouchEvent()

  ③消费： 回调方法返回true

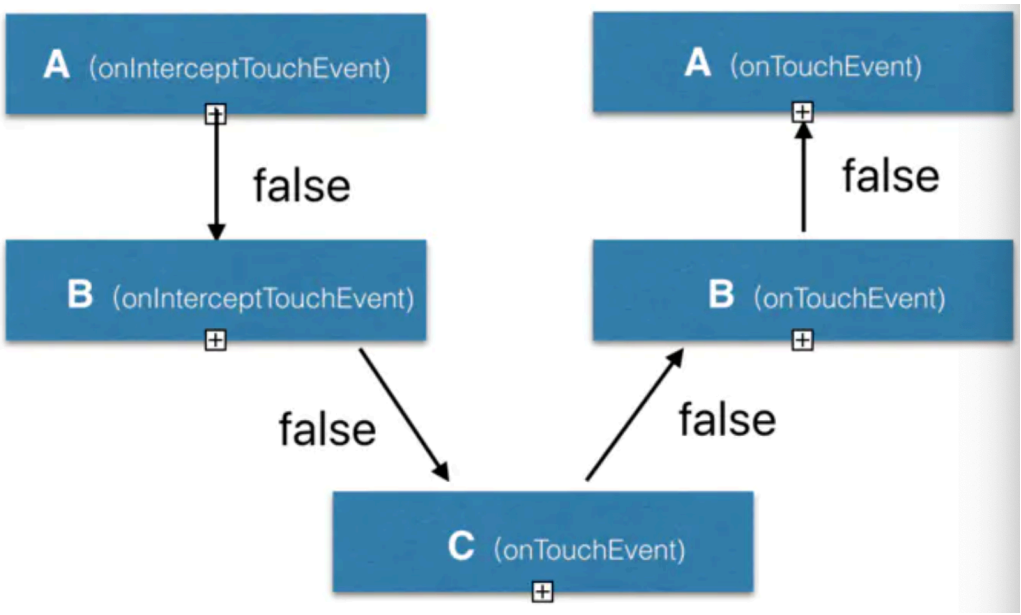
  ④拦截： onInterceptTouchEvent()执行返回true
  如果返回true，TouchEvent就不会再传入子View对象

  ⑤反拦截： view.getParent().requestDisallowInterceptTouchEvent(true)
  使父View不能再拦截，事件就会分发到当前View对象
  拦截与反拦截，都是在分发的时候就要决定的。

- 用一个图来形象描述的话：



- another simpler one:



4.3 我复制一下核心要点:

- 事件分发原理: 责任链模式，事件层层传递，直到被消费。
- View 的 dispatchTouchEvent 主要用于调度自身的监听器和 onTouchEvent。
- View 的事件的调度顺序是 onTouchListener > onTouchEvent > onLongClickListener > onClickListener 。
- 不论 View 自身是否注册点击事件，只要 View 是可点击的就会消费事件。
- 事件是否被消费由返回值决定，true 表示消费，false 表示不消费，与是否使用了事件无关。
- ViewGroup 中可能有多个 ChildView 时，将事件分配给包含点击位置的 ChildView。
- ViewGroup 和 ChildView 同时注册了事件监听器 (onClick 等)，由 ChildView 消费。
- 一次触摸流程中产生事件应被同一 View 消费，全部接收或者全部拒绝。
- 只要接受 ACTION$_{DOWN}$ 就意味着接受所有的事件，拒绝 ACTION$_{DOWN}$ 则不会收到后内容。
- 如果当前正在处理的事件被上层 View 拦截，会收到一个 ACTION$_{CANCEL}$

5 ScrollView 嵌套 ScrollView 的滑动冲突

- https://www.jianshu.com/p/eedc98eecf02

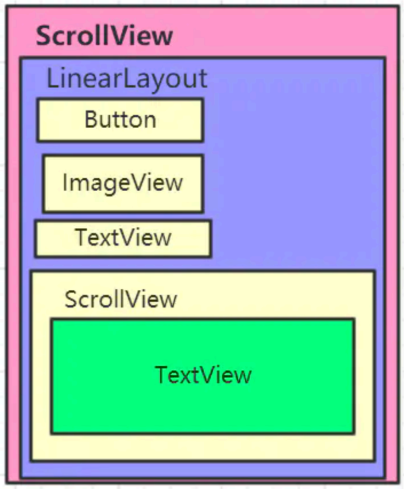
5.1 结论：在自定义控件中如下重写 onInterceptTouchEvent 就告诉所有父 View：不要拦截事件，让我消费!!

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    getParent().requestDisallowInterceptTouchEvent(true);
    return super.onInterceptTouchEvent(ev);
}
```

- 这是一个从源码角度分析滑动冲突的原因
- 以及在源码中理解为何能解决滑动冲突

5.2 滑动冲突 case 模拟

- 这是 MainActivity 主界面的布局内容:



- xml:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- 这里外面还有个大的 scrollview -->
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
            xmlns:app="http://schemas.android.com/apk/res-auto"
            xmlns:tools="http://schemas.android.com/tools"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            tools:context="com.solory.learnview.MainActivity">
  <!-- viewgroup 的子类 -->
  <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
    <!-- 几个不同的视图 -->
    <Button
        android:id="@+id/btn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="8dp" />
    <ImageView
        android:id="@+id/imageView"
        android:layout_width="105dp"
        android:layout_height="86dp"
        android:layout_margin="8dp"
        app:srcCompat="@mipmap/ic_launcher_round" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/article_1"
        android:textSize="36sp" />
    <!-- 关注的重点:  -->
    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="80dp"
        android:background="@color/colorPrimary">
      <TextView
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:text="@string/article_2" />
```

```
    </ScrollView>
  </LinearLayout>
</ScrollView>
```

- MainActivity 不用动。跑起来：外面的大的可以上下滚动，但是下面的那个小的动不了!!! 外面的 ScrollView 正常滑动，但是里面的那个 ScrollView 动不了。

5.3 直接给出解决方案再看如何解决：

- 新建一个类继承 ScrollView

```
public class MyScrollView extends ScrollView {
    public MyScrollView(Context context) {
        this(context,null);
    }
    public MyScrollView(Context context, AttributeSet attrs) {
        this(context, attrs,0);
    }
    public MyScrollView(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }
// 覆写了这个方法
    @Override public boolean onInterceptTouchEvent(MotionEvent ev) {
        // 关键点在这
        getParent().requestDisallowInterceptTouchEvent(true);
        return super.onInterceptTouchEvent(ev);
    }
}
```

- buildProject, 然后在 xml 中将里面的 ScrollView 修改成这个 MyScrollView。

```
<com.solory.learnview.MyScrollView
    android:layout_width="match_parent"
    android:layout_height="80dp"
    android:background="@color/colorPrimary">
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/article_2" />
</com.solory.learnview.MyScrollView>
```

- 跑起来：

- 问题解决。

5.4 现在研究为什么：为什么在重写的 onInterceptTouchEvent(MotionEvent ev) 中神奇的一句代码，就把问题解决了？

```
getParent().requestDisallowInterceptTouchEvent(true);
```

- 先看 ScollView 源码中的 onInterceptTouchEvent:

```
 @Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    /*
     * 这个方法决定了我们是否要拦截这个事件。
     * 如果我们返回 true, onMotionEvent 方法将被调用,
     * 我们将在那执行实际的滚动操作。
     */
    /*
    * 最常见的情况: 用户在拖拽中。
    * 他在动他的手指。我们想要截取这个
    * 事件.
    */
    final int action = ev.getAction();
    if ((action == MotionEvent.ACTION_MOVE) && (mIsBeingDragged)) {
        return true;
```

```
    }
    if (super.onInterceptTouchEvent(ev)) {
        return true;
    }
    /*
     * 如果我们不能滚动，不要试图截取触摸。
     */
    if (getScrollY() == 0 && !canScrollVertically(1)) {
        return false;
    }
```

- 清晰明了，干净简单，后面还有一大段代码，就不放了。这里一进来就是一个判断，如果进来的是 ACTION$_{MOVE}$，那么直接返回 true，直接拦截，那么后面就没他的子 View 什么事了（不懂的话去看一下 ViewGroup 的 dispatchTouchEvent 方法），event 被传入他自己的 onTouchEvent 中去进行滚动操作了。
  - 那么我们一开始内部的 ScrollView 滑动没有响应的原因就是，那时候手指是在滑动的，一直不断传入 ACTION$_{MOVE}$，所以 event 一直被外部的 ScrollView 在如上的操作中拦截了。
  - 意思就是只要你手指在 ScrollView 上滑动，ScrollView 内部的子 View 就永远接收不到任何事件，就是永远无响应。
- 冲突的原因明白了，现在看如何解决的
- 回头看 MyScrollView 是如何解决的：

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    getParent().requestDisallowInterceptTouchEvent(true);
    return super.onInterceptTouchEvent(ev);
}
```

- 意思就是取得父类，然后请求父类不拦截 TouchEvent 的意思。
- 首先 getParent 就是返回父类，在这里是返回的那个 LinearLayout，然后点 requestDisallowInterceptTouchEvent 进去看，发现是一个叫做 ViewParent 的接口中的抽象方法，

```
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {  }
```

- 注释的英文：

```
当一个子 View 不想要他的父 View 和它的祖先 View 们拦截触摸事件的时候。调用该方法
他的父 View 应该将该方法接着向上传递给每一个祖先 View 们。
```

- 抽象方法的话，看一下是谁实现了，因为继承的 ScrollView，所以先看对应的 ScrollView 中的实现

```
//-----ScrollView 中
@Override
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
    if (disallowIntercept) {
    // 我也不知道这个方法干嘛的，反正不影响整体思路，先跳过。
        recycleVelocityTracker();
    }
    // 无论如何，都会执行父类的该方法。
    super.requestDisallowInterceptTouchEvent(disallowIntercept);
}
```

- 那么我们查看父类中的实现，ViewGroup 中：

```
//-----ViewGroup 中
@Override
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
    if (disallowIntercept == ((mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0)) {
        // We're already in this state, assume our ancestors are too
```

```
        return;
    }
    if (disallowIntercept) {
        mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
    } else {
        mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
    }
    // Pass it up to our parent
    if (mParent != null) {
        mParent.requestDisallowInterceptTouchEvent(disallowIntercept);
    }
}
```

- 意思就是 **将自己的 FLAG 更改，变成 disallowIntercept，并且递归，只要有父 View，就把父 View 的 FLAG 同样设置**。
- 大意为，设置了这个方法，MyScrollView 就通过递归，告诉了他的父 View 和向上的所有祖先 View：统统不要拦截事件！交给我来！
- 那这个 FLAG 是在哪里发挥作用?当然是在 ViewGroup 的 dispatchTouchEvent(MotionEvent event) 内部，并且用一个 if 条件先于 onInterceptMotionEvent(MotionEvent event) 来判断
- 图片为证：

- 但是!

```
//-----MyScrollView 中
@Override public boolean onInterceptTouchEvent(MotionEvent ev) {
        getParent().requestDisallowInterceptTouchEvent(true);
        return super.onInterceptTouchEvent(ev);
    }
```

- 这段代码的 getParent().requestDisallowInterceptTouchEvent(true); 能执行到的前提是 MyScrollView 能执行 onInterceptTouchEvent，也就是能执行 dispatchTouchEvent，可是事件早都被外层的 ScrollView 拦截了，你还怎么获取父类然后请求不要拦截 TouchEvent?
- 当时我在这里思考了蛮久的，那么我们返回到 ScrollView 的 onInterceptTouchEvent 里去看吧

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    final int action = ev.getAction();
    if ((action == MotionEvent.ACTION_MOVE) && (mIsBeingDragged)) {
        return true;
    }
    if (super.onInterceptTouchEvent(ev)) {
        return true;
```

```
    }
    if (getScrollY() == 0 && !canScrollVertically(1)) {
        return false;
    }
```

- 他只拦截 ACTION$MOVE \quad ACTION\,DOWN \quad ACTION\,DOWN \qquad View \qquad MyScrollView \qquad event \quad FL$

5.5 总结:

- 在自定义控件中重写 onInterceptTouchEvent 就告诉所有父 View：不要拦截事件，让我消费!!

```
@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    getParent().requestDisallowInterceptTouchEvent(true);
    return super.onInterceptTouchEvent(ev);
}
```

6 Android 触摸屏事件派发机制详解与源码分析一 (View 篇)

- https://blog.csdn.net/yanbober/article/details/45887547

6.1 基础实例现象

6.1.1 例子

- 从一个例子分析说起吧。如下是一个很简单不过的 Android 实例：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/mylayout">
    <Button
        android:id="@+id/my_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="click test"/>
</LinearLayout>

public class ListenerActivity extends Activity implements View.OnTouchListener, View.OnClickListener {
    private LinearLayout mLayout;
    private Button mButton;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLayout = (LinearLayout) this.findViewById(R.id.mylayout);
        mButton = (Button) this.findViewById(R.id.my_btn);
        mLayout.setOnTouchListener(this);
        mButton.setOnTouchListener(this);
        mLayout.setOnClickListener(this);
        mButton.setOnClickListener(this);
    }
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.i(null, "OnTouchListener--onTouch-- action="+event.getAction()+" --"+v);
        return false;
    }
    @Override
    public void onClick(View v) {
        Log.i(null, "OnClickListener--onClick--"+v);
    }
}
```

6.1.2 现象

- 如上代码很简单，但凡学过几天 Android 的人都能看懂吧。Activity 中有一个 LinearLayout（ViewGroup 的子类，ViewGroup 是 View 的子类）布局，布局中包含一个按钮（View 的子类）；然后分别对这两个控件设置了 Touch 与 Click 的监听事件，具体运行结果如下：
- 我们看下 onTouch 和 onClick，从参数都能看出来 onTouch 比 onClick 强大灵活，毕竟多了一个 event 参数。这样 onTouch 里就可以处理 ACTION$_{DOWN\ ACTION UP\ ACTION MOVE}$ 1 B 各一次是因为按下抬起两个触摸动作被触发）然后才触发 onClick 事件；在 2 中也同理类似 1；在 3 中会发现 onTouch 被多次调运后才调运 onClick，是因为手指晃动了，所以触发了 ACTION$_{DOWN}$->ACTION$_{MOVE_}$->ACTION$_{UP}$
- 如果你眼睛比较尖你会看见 onTouch 会有一个返回值，而且在上面返回了 false。你可能会疑惑这个返回值有啥效果？那就验证一下吧，我们将上面的 onTouch 返回值改为 ture。如下：

```
@Override
public boolean onTouch(View v, MotionEvent event) {
    Log.i(null, "OnTouchListener--onTouch-- action="+event.getAction()+" --"+v);
    return true;
}
```

6.1.3 总结结论

- 好了，经过这个简单的实例验证你可以总结发现：
  - Android 控件的 Listener 事件触发顺序是先触发 onTouch，其次 onClick。
  - 如果控件的 onTouch 返回 true 将会阻止事件继续传递，返回 false 事件会继续传递。
- 对于伸手党码农来说其实到这足矣应付常规的 App 事件监听处理使用开发了，但是对于复杂的事件监听处理或者想自定义控件的码农来说这才是刚刚开始，只是个热身。既然这样那就继续喽。。。

6.2 Android 5.1.1(API 22) View 触摸屏事件传递源码分析

6.2.1 写在前面的话

- 其实 Android 源码无论哪个版本对于触摸屏事件的传递机制都类似，这里只是选用了目前最新版本的源码来分析而已。分析 Android View 事件传递机制之前有必要先看下源码的一些关系，如下是几个继承关系图：
- 怎么样？看了官方这个继承图是不是明白了上面例子中说的 LinearLayout 是 ViewGroup 的子类，ViewGroup 是 View 的子类，Button 是 View 的子类关系呢？其实，在 Android 中所有的控件无非都是 ViewGroup 或者 View 的子类，说高尚点就是所有控件都是 View 的子类。
- 这里通过继承关系是说明一切控件都是 View，同时 View 与 ViewGroup 又存在一些区别，所以该模块才只单单先分析 View 触摸屏事件传递机制。

6.2.2 从 View 的 dispatchTouchEvent 方法说起

- 在 Android 中你只要触摸控件首先都会触发控件的 dispatchTouchEvent 方法（其实这个方法一般都没在具体的控件类中，而在他的父类 View 中），所以我们先来看下 View 的 dispatchTouchEvent 方法，如下：

```
public boolean dispatchTouchEvent(MotionEvent event) {
    // If the event should be handled by accessibility focus first.
    if (event.isTargetAccessibilityFocus()) {
        // We don't have focus or no virtual descendant has it, do not handle the event.
```

```
        if (!isAccessibilityFocusedViewOrHost()) {
            return false;
        }
        // We have focus and got the event, then use normal event dispatch.
        event.setTargetAccessibilityFocus(false);
    }
    boolean result = false;
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(event, 0);
    }
    final int actionMasked = event.getActionMasked();
    if (actionMasked == MotionEvent.ACTION_DOWN) {
        // Defensive cleanup for new gesture
        stopNestedScroll();
    }
    if (onFilterTouchEventForSecurity(event)) { // 判断当前 View 是否没被遮住等
        //noinspection SimplifiableIfStatement
// ListenerInfo 是 View 的静态内部类，用来定义一堆关于 View 的 XXXListener 等方法
        ListenerInfo li = mListenerInfo;
// 首先 li 对象自然不会为 null，li.mOnTouchListener 呢 ?
        if (li != null && li.mOnTouchListener != null
// 通过位与运算确定控件（View）是不是 ENABLED 的，默认控件都是 ENABLED 的；
            && (mViewFlags & ENABLED_MASK) == ENABLED
// 判断 onTouch 的返回值是不是 true
            && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
    if (!result && mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }
    // Clean up after nested scrolls if this is the end of a gesture;
    // also cancel it if we tried an ACTION_DOWN but we didn't want the rest
    // of the gesture.
    if (actionMasked == MotionEvent.ACTION_UP ||
        actionMasked == MotionEvent.ACTION_CANCEL ||
        (actionMasked == MotionEvent.ACTION_DOWN && !result)) {
        stopNestedScroll();
    }
    return result;
}
```

- dispatchTouchEvent 的代码有点长，咱们看重点就可以。前面都是设置一些标记和处理 input 与手势等传递，到 24 行的 if (onFilterTouchEventForSecurity(event)) 语句判断当前 View 是否没被遮住等，接着 26 行定义 ListenerInfo 局部变量，ListenerInfo 是 View 的静态内部类，用来定义一堆关于 View 的 XXXListener 等方法；接着 if (li != null && li.mOnTouchListener != null && (mViewFlags & ENABLED$_{MASK}$) == ENABLED && li.mOnTouchListener.onTouch(this, event)) 语句就是重点，首先 li 对象自然不会为 null，li.mOnTouchListener 呢？你会发现 ListenerInfo 的 mOnTouchListener 成员是在哪儿赋值的呢？怎么确认他是不是 null 呢？通过在 View 类里搜索可以看到：

```
/**
 * Register a callback to be invoked when a touch event is sent to this view.
 * @param l the touch listener to attach to this view
 */
public void setOnTouchListener(OnTouchListener l) {
    getListenerInfo().mOnTouchListener = l;
}
```

- li.mOnTouchListener 是不是 null 取决于控件（View）是否设置 setOnTouchListener 监听，在上面的实例中我们是设置过 Button 的 setOnTouchListener 方法的，所以也不为 null；接着通过位与运算确定控件（View）是不是 ENABLED 的，默认控件都是 ENABLED 的；接着判断 onTouch 的返回值是不是 true。通过如上判断之后如果都为 true 则设置默认为 false 的 result 为 true，那么接下来的 if (!result && onTouchEvent(event)) 就不会执行，最终 dispatchTouchEvent 也会返回 true。而如果 if (li != null && li.mOnTouchListener != null &&

(mViewFlags & ENABLED$_{MASK}$) == ENABLED && li.mOnTouchListener.onTouch(this, event)) 语句有一个为 false 则 if (!result && onTouchEvent(event)) 就会执行，如果 onTouchEvent(event) 返回 false 则 dispatchTouchEvent 返回 false，否则返回 true。

- 这下再看前面的实例部分明白了吧？控件触摸就会调运 dispatchTouchEvent 方法，而在 dispatchTouchEvent 中先执行的是 onTouch 方法，所以验证了实例结论总结中的 onTouch 优先于 onClick 执行道理。如果控件是 ENABLE 且在 onTouch 方法里返回了 true 则 dispatchTouchEvent 方法也返回 true，不会再继续往下执行；反之，onTouch 返回 false 则会继续向下执行 onTouchEvent 方法，且 dispatchTouchEvent 的返回值与 onTouchEvent 返回值相同。
- 所以依据这个结论和上面实例打印结果你指定已经大胆猜测认为 onClick 一定与 onTouchEvent 有关系？是不是呢？先告诉你，是的。下面我们会分析。

1. 总结结论
   - 在 View 的触摸屏传递机制中通过分析 dispatchTouchEvent 方法源码我们会得出如下基本结论：
     - 触摸控件（View）首先执行 dispatchTouchEvent 方法。
     - 在 dispatchTouchEvent 方法中先执行 onTouch 方法，后执行 onClick 方法（onClick 方法在 onTouchEvent 中执行，下面会分析）。
     - 如果控件（View）的 onTouch 返回 false 或者 mOnTouchListener 为 null（控件没有设置 setOnTouchListener 方法）或者控件不是 enable 的情况下会调运 onTouchEvent，dispatchTouchEvent 返回值与 onTouchEvent 返回一样。
     - 如果控件不是 enable 的设置了 onTouch 方法也不会执行，只能通过重写控件的 onTouchEvent 方法处理（上面已经处理分析了），dispatchTouchEvent 返回值与 onTouchEvent 返回一样。
     - 如果控件（View）是 enable 且 onTouch 返回 true 情况下，dispatchTouchEvent 直接返回 true，不会调用 onTouchEvent 方法。
   - 上面说了 onClick 一定与 onTouchEvent 有关系，那么接下来就分析分析 dispatchTouchEvent 方法中的 onTouchEvent 方法。

6.2.3 继续说说 View 的 dispatchTouchEvent 方法中调运的 onTouchEvent 方法

- 上面说了 dispatchTouchEvent 方法中如果 onTouch 返回 false 或者 mOnTouchListener 为 null（控件没有设置 setOnTouchListener 方法）或者控件不是 enable 的情况下会调运 onTouchEvent，所以接着看就知道了，如下：

```
public boolean onTouchEvent(MotionEvent event) {
    final float x = event.getX();
    final float y = event.getY();
    final int viewFlags = mViewFlags;
// 如果控件（View）是 disenable 状态，同时是可以 clickable 的则 onTouchEvent 直接消费事件返回 true，
// 反之如果控件（View）是 disenable 状态，同时是 disclickable 的则 onTouchEvent 直接 false
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (event.getAction() == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        // A disabled view that is clickable still consumes the touch
        // events, it just doesn't respond to them.
        return (((viewFlags & CLICKABLE) == CLICKABLE ||
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));
    }
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }
```

```
// 如果一个控件是 enable 且 disclickable 则 onTouchEvent 直接返回 false 了;
// 反之，如果一个控件是 enable 且 clickable 则继续进入过于一个 event 的 switch 判断中，
    if (((viewFlags & CLICKABLE) == CLICKABLE ||
         (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
        switch (event.getAction()) {
        case MotionEvent.ACTION_UP:
            // 首先判断了是否按下过
            boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
            if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                // take focus if we don't have it already and we should in
                // touch mode.
                // 是不是可以得到焦点
                boolean focusTaken = false;
                if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
                    // 然后尝试获取焦点
                    focusTaken = requestFocus();
                }
                if (prepressed) {
                    // The button is being released before we actually
                    // showed it as pressed.  Make it show the pressed
                    // state now (before scheduling the click) to ensure
                    // the user sees it.
                    setPressed(true, x, y);
                }
// 然后判断如果不是 longPressed 则通过 post 在 UI Thread 中执行一个 PerformClick 的 Runnable，也就是 performClick 方法
                if (!mHasPerformedLongPress) {
                    // This is a tap, so remove the longpress check
                    removeLongPressCallback();
                    // Only perform take click actions if we were in the pressed state
                    if (!focusTaken) {
                        // Use a Runnable and post this rather than calling
                        // performClick directly. This lets other visual state
                        // of the view update before click actions start.
                        if (mPerformClick == null) {
                            mPerformClick = new PerformClick();
                        }
                        if (!post(mPerformClick)) {
                            performClick();
                        }
                    }
                }
                if (mUnsetPressedState == null) {
                    mUnsetPressedState = new UnsetPressedState();
                }
                if (prepressed) {
                    postDelayed(mUnsetPressedState,
                                ViewConfiguration.getPressedStateDuration());
                } else if (!post(mUnsetPressedState)) {
                    // If the post failed, unpress right now
                    mUnsetPressedState.run();
                }
                removeTapCallback();
            }
            break;
        case MotionEvent.ACTION_DOWN:
            mHasPerformedLongPress = false;
            if (performButtonActionOnTouchDown(event)) {
                break;
            }
            // Walk up the hierarchy to determine if we're inside a scrolling container.
            boolean isInScrollingContainer = isInScrollingContainer();
            // For views inside a scrolling container, delay the pressed feedback for
            // a short period in case this is a scroll.
            if (isInScrollingContainer) {
                mPrivateFlags |= PFLAG_PREPRESSED;
                if (mPendingCheckForTap == null) {
                    mPendingCheckForTap = new CheckForTap();
                }
                mPendingCheckForTap.x = event.getX();
                mPendingCheckForTap.y = event.getY();
                postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
            } else {
                // Not inside a scrolling container, so show the feedback right away
                setPressed(true, x, y);
                checkForLongClick(0);
```

```
            }
            break;
        case MotionEvent.ACTION_CANCEL:
            setPressed(false);
            removeTapCallback();
            removeLongPressCallback();
            break;
        case MotionEvent.ACTION_MOVE:
            drawableHotspotChanged(x, y);
            // Be lenient about moving outside of buttons
            if (!pointInView(x, y, mTouchSlop)) {
                // Outside button
                removeTapCallback();
                if ((mPrivateFlags & PFLAG_PRESSED) != 0) {
                    // Remove any future long press/tap checks
                    removeLongPressCallback();
                    setPressed(false);
                }
            }
            break;
        }
// 最终 onTouchEvent 都返回了 true
        return true;
    }
// 如果一个控件是 enable 且 disclickable 则 onTouchEvent 直接返回 false 了;
    return false;
}
```

我勒个去！一个方法比一个方法代码多。好吧，那咱们继续只挑重点来说明呗。首先地 6 到 14 行可以看出，如果控件（View）是 disenable 状态，同时是可以 clickable 的则 onTouchEvent 直接消费事件返回 true，反之如果控件（View）是 disenable 状态，同时是 disclickable 的则 onTouchEvent 直接 false。多说一句，关于控件的 enable 或者 clickable 属性可以通过 java 或者 xml 直接设置，每个 view 都有这些属性。接着 22 行可以看见，如果一个控件是 enable 且 disclickable 则 onTouchEvent 直接返回 false 了；反之，如果一个控件是 enable 且 clickable 则继续进入过于一个 event 的 switch 判断中，然后最终 onTouchEvent 都返回了 true。switch 的 ACTION$_{DOWN\ ACTION MOVE\ \ \ \ \ ACTION UP\ \ \ \ \ longPressed\ \ post UI}$ Thread 中执行一个 PerformClick 的 Runnable，也就是 performClick 方法。具体如下：

```
public boolean performClick() {
    final boolean result;
    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
    return result;
}
```

- 这个方法也是先定义一个 ListenerInfo 的变量然后赋值，接着判断 li.mOnClickListener 是不是为 null，决定执行不执行 onClick。你指定现在已经很机智了，和 onTouch 一样，搜一下 mOnClickListener 在哪赋值的呗，结果发现：

```
public void setOnClickListener(OnClickListener l) {
    if (!isClickable()) {
        setClickable(true);
    }
    getListenerInfo().mOnClickListener = l;
}
```

- 看见了吧！控件只要监听了 onClick 方法则 mOnClickListener 就不为 null，而且有意思的是如果调运 setOnClickListener 方法设置监听且控件是 disclickable 的情况下默认会帮设置为 clickable。

- 我勒个去！！！ 惊讶吧！！！ 猜的没错 onClick 就在 onTouchEvent 中执行的，而且是在 onTouchEvent 的 ACTION$_{UP}$

1. 总结结论

   - onTouchEvent 方法中会在 ACTION$_{UP}$ $onClick$
   - 当 dispatchTouchEvent 在进行事件分发的时候，只有前一个 action 返回 true，才会触发下一个 action。
   - 到此上面例子中关于 Button 点击的各种打印的真实原因都找到了可靠的证据，也就是说 View 的触摸屏事件传递机制其实也就这么回事。

6.3 透过源码继续进阶实例验证

- 其实上面分析完 View 的触摸传递机制之后已经足够用了。如下的实例验证可以说是加深阅读源码的理解，还有一个主要作用就是为将来自定义控件打下坚实基础。因为自定义控件中时常会与这几个方法打交道。

6.3.1 例子

- 我们自定义一个 Button（Button 实质继承自 View），如下：

```
public class TestButton extends Button {
    public TestButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
        return super.dispatchTouchEvent(event);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "onTouchEvent-- action="+event.getAction());
        return super.onTouchEvent(event);
    }
}
```

- 其他代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/mylayout">
    <com.zzci.light.TestButton
        android:id="@+id/my_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="click test"/>
</LinearLayout>

public class ListenerActivity extends Activity implements View.OnTouchListener, View.OnClickListener {
    private LinearLayout mLayout;
    private TestButton mButton;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLayout = (LinearLayout) this.findViewById(R.id.mylayout);
        mButton = (TestButton) this.findViewById(R.id.my_btn);
        mLayout.setOnTouchListener(this);
        mButton.setOnTouchListener(this);
        mLayout.setOnClickListener(this);
```

```
        mButton.setOnClickListener(this);
    }
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.i(null, "OnTouchListener--onTouch-- action="+event.getAction()+" --"+v);
        return false;
    }
    @Override
    public void onClick(View v) {
        Log.i(null, "OnClickListener--onClick--"+v);
    }
}
```

- 其实这段代码只是对上面例子中的 Button 换为了自定义 Button 而已。

6.3.2 现象分析

1. 点击 Button（手抽筋了一下）

    - dispatchTouchEvent 方法先派发 down 事件，完事调运 onTouch，完事调运 onTouchEvent 返回 true，同时 dispatchTouchEvent 返回 true，然后 dispatchTouchEvent 继续派发 move 或者 up 事件，循环，直到 onTouchEvent 处理 up 事件时调运 onClick 事件，完事返回 true，同时 dispatchTouchEvent 返回 true；一次完整的 View 事件派发流程结束。

2. 简单修改 onTouchEvent 返回值为 true

    - 将 TestButton 类的 onTouchEvent 方法修改如下，其他和基础代码保持不变：

    ```
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "onTouchEvent-- action="+event.getAction());
        return true;
    }
    ```

    - 可以发现，当自定义了控件（View）的 onTouchEvent 直接返回 true 而不调运 super 方法时，事件派发机制如同 4.2.1 类似，只是最后 up 事件没有触发 onClick 而已（因为没有调用 super）。
    - 所以可想而知，如果 TestButton 类的 onTouchEvent 修改为如下：

    ```
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "onTouchEvent-- action="+event.getAction());
        super.onTouchEvent(event);
        return true;
    }
    ```

    - 整个派发机制和 4.2.1 完全类似。

3. 简单修改 onTouchEvent 返回值为 false

    - 将 TestButton 类的 onTouchEvent 方法修改如下，其他和基础代码保持不变：

    ```
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "onTouchEvent-- action="+event.getAction());
        return false;
    }
    ```

    - 你会发现如果 onTouchEvent 返回 false（也即 dispatchTouchEvent 一旦返回 false 将不再继续派发其他 action，立即停止派发），这里只派发了 down 事件。至于后面触发了 LinearLayout 的 touch 与 click 事件我们这里不做关注，下一篇博客会详细解释为啥（其实你可以想下的，LinearLayout 是 ViewGroup 的子类，你懂的），这里你只用知道 View 的 onTouchEvent 返回 false 会阻止继续派发事件。

- 同理修改如下：

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    Log.i(null, "onTouchEvent-- action="+event.getAction());
    super.onTouchEvent(event);
    return false;
}
```

4. 简单修改 dispatchTouchEvent 返回值为 true

   - 将 TestButton 类的 dispatchTouchEvent 方法修改如下，其他和基础代码保持不变：

   ```
   @Override
   public boolean dispatchTouchEvent(MotionEvent event) {
       Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
       return true;
   }
   ```

   - 你会发现如果 dispatchTouchEvent 直接返回 true 且不调运 super 任何事件都得不到触发。
   - 继续修改如下呢?
   - 将 TestButton 类的 dispatchTouchEvent 方法修改如下，其他和基础代码保持不变：

   ```
   @Override
   public boolean dispatchTouchEvent(MotionEvent event) {
       Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
       super.dispatchTouchEvent(event);
       return true;
   }
   ```

   - 可以发现所有事件都可以得到正常派发，和 4.2.1 类似。

5. 简单修改 dispatchTouchEvent 返回值为 false

   - 将 TestButton 类的 dispatchTouchEvent 方法修改如下，其他和基础代码保持不变：

   ```
   @Override
   public boolean dispatchTouchEvent(MotionEvent event) {
       Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
       return false;
   }
   ```

   - 你会发现事件不进行任何继续触发，关于点击 Button 触发了 LinearLayout 的事件暂时不用关注，下篇详解。
   - 继续修改如下呢?
   - 将 TestButton 类的 dispatchTouchEvent 方法修改如下，其他和基础代码保持不变：

   ```
   @Override
   public boolean dispatchTouchEvent(MotionEvent event) {
       Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
       super.dispatchTouchEvent(event);
       return false;
   }
   ```

   - 你会发现结果和 4.2.3 的第二部分结果一样，也就是说如果 dispatchTouchEvent 返回 false 事件将不再继续派发下一次。

6. 简单修改 dispatchTouchEvent 与 onTouchEvent 返回值

   - 修改 dispatchTouchEvent 返回值为 true，onTouchEvent 为 false：

- 将 TestButton 类的 dispatchTouchEvent 方法和 onTouchEvent 方法修改如下，其他和基础代码保持不变：

```
@Override
public boolean dispatchTouchEvent(MotionEvent event) {
    Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
    super.dispatchTouchEvent(event);
    return true;
}
@Override
public boolean onTouchEvent(MotionEvent event) {
    Log.i(null, "onTouchEvent-- action=" + event.getAction());
    super.onTouchEvent(event);
    return false;
}
```

- 修改 dispatchTouchEvent 返回值为 false，onTouchEvent 为 true：
- 将 TestButton 类的 dispatchTouchEvent 方法和 onTouchEvent 方法修改如下，其他和基础代码保持不变：

```
@Override
public boolean dispatchTouchEvent(MotionEvent event) {
    Log.i(null, "dispatchTouchEvent-- action=" + event.getAction());
    super.dispatchTouchEvent(event);
    return false;
}
@Override
public boolean onTouchEvent(MotionEvent event) {
    Log.i(null, "onTouchEvent-- action=" + event.getAction());
    super.onTouchEvent(event);
    return true;
}
```

- 由此对比得出结论，dispatchTouchEvent 事件派发是传递的，如果返回值为 false 将停止下次事件派发，如果返回 true 将继续下次派发。譬如，当前派发 down 事件，如果返回 true 则继续派发 up，如果返回 false 派发完 down 就停止了。

6.3.3 总结

这个例子组合了很多种情况的值去验证上面源码的分析，同时也为自定义控件打下了基础。仔细理解这个例子对于 View 的事件传递就差不多了。

6.4 总结 View 触摸屏事件传递机制

- 上面例子也测试了，源码也分析了，总得有个最终结论方便平时写代码作为参考依据呀，不能每次都再去分析一遍源码，那得多蛋疼呢!
- 综合得出 Android View 的触摸屏事件传递机制有如下特征：
  - 触摸控件（View）首先执行 dispatchTouchEvent 方法。
  - 在 dispatchTouchEvent 方法中先执行 onTouch 方法，后执行 onClick 方法（onClick 方法在 onTouchEvent 中执行，下面会分析）。
  - 如果控件（View）的 onTouch 返回 false 或者 mOnTouchListener 为 null（控件没有设置 setOnTouchListener 方法）或者控件不是 enable 的情况下会调运 onTouchEvent，dispatchTouchEvent 返回值与 onTouchEvent 返回一样。
  - 如果控件不是 enable 的设置了 onTouch 方法也不会执行，只能通过重写控件的 onTouchEvent 方法处理（上面已经处理分析了），dispatchTouchEvent 返回值与 onTouchEvent 返回一样。

- 如果控件（View）是 enable 且 onTouch 返回 true 情况下，dispatchTouchEvent 直接返回 true，不会调用 onTouchEvent 方法。
- 当 dispatchTouchEvent 在进行事件分发的时候，只有前一个 action 返回 true，才会触发下一个 action（也就是说 dispatchTouchEvent 返回 true 才会进行下一次 action 派发）。

7 Android 触摸屏事件派发机制详解与源码分析二（ViewGroup 篇）

- http://static.kancloud.cn/digest/androidframeworks/127775

7.1 基础实例现象

7.1.1 例子

- 这个例子布局等还和上一篇的例子相似，只是重写了 Button 和 LinearLayout 而已，所以效果图不在提供，具体参见上一篇。
- 首先我们简单的自定义一个 Button（View 的子类），再自定义一个 LinearLayout（ViewGroup 的子类），其实没有自定义任何属性，只是重写部分方法（添加了打印，方便查看）而已，如下：

```
public class TestButton extends Button {
    public TestButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        Log.i(null, "TestButton dispatchTouchEvent-- action=" + event.getAction());
        return super.dispatchTouchEvent(event);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "TestButton onTouchEvent-- action=" + event.getAction());
        return super.onTouchEvent(event);
    }
}
public class TestLinearLayout extends LinearLayout {
    public TestLinearLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        Log.i(null, "TestLinearLayout onInterceptTouchEvent-- action=" + ev.getAction());
        return super.onInterceptTouchEvent(ev);
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        Log.i(null, "TestLinearLayout dispatchTouchEvent-- action=" + event.getAction());
        return super.dispatchTouchEvent(event);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "TestLinearLayout onTouchEvent-- action=" + event.getAction());
        return super.onTouchEvent(event);
    }
}
```

- 如上两个控件很简单吧，不解释，继续看其他代码：

```
<?xml version="1.0" encoding="utf-8"?>
<com.zzci.light.TestLinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    android:id="@+id/mylayout">
    <com.zzci.light.TestButton
        android:id="@+id/my_btn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="click test"/>
</com.zzci.light.TestLinearLayout>
```

```
public class ListenerActivity extends Activity implements View.OnTouchListener, View.OnClickListener {
    private TestLinearLayout mLayout;
    private TestButton mButton;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mLayout = (TestLinearLayout) this.findViewById(R.id.mylayout);
        mButton = (TestButton) this.findViewById(R.id.my_btn);
        mLayout.setOnTouchListener(this);
        mButton.setOnTouchListener(this);
        mLayout.setOnClickListener(this);
        mButton.setOnClickListener(this);
    }
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.i(null, "OnTouchListener--onTouch-- action="+event.getAction()+" --"+v);
        return false;
    }
    @Override
    public void onClick(View v) {
        Log.i(null, "OnClickListener--onClick--"+v);
    }
}
```

- 到此基础示例的代码编写完成。没有啥难度，很简单易懂，不多解释了。

7.1.2 运行现象

- 当直接点击 Button 时打印现象如下：

```
TestLinearLayout dispatchTouchEvent-- action=0
TestLinearLayout onInterceptTouchEvent-- action=0
TestButton dispatchTouchEvent-- action=0
OnTouchListener--onTouch-- action=0 --com.zzci.light.TestButton
TestButton onTouchEvent-- action=0
TestLinearLayout dispatchTouchEvent-- action=1
TestLinearLayout onInterceptTouchEvent-- action=1
TestButton dispatchTouchEvent-- action=1
OnTouchListener--onTouch-- action=1 --com.zzci.light.TestButton
TestButton onTouchEvent-- action=1
OnClickListener--onClick--com.zzci.light.TestButton
```

- 分析：你会发现这个结果好惊讶吧，点击了 Button 却先执行了 TestLinearLayout（ViewGroup）的 dispatchTouchEvent，接着执行 TestLinearLayout（ViewGroup）的 onInterceptTouchEvent，接着执行 TestButton（TestLinearLayout 包含的成员 View）的 dispatchTouchEvent，接着就是 View 触摸事件的分发流程，上一篇已经讲过了。也就是说当点击 View 时事件派发每一个 down，up 的 action 顺序是先触发最父级控件
- （这里为 LinearLayout）的 dispatchTouchEvent->onInterceptTouchEvent-> 然后向前一级传递（这里就是传递到 Button View）。
- 那么继续看，当直接点击除 Button 以外的其他部分时打印如下：

```
TestLinearLayout dispatchTouchEvent-- action=0
TestLinearLayout onInterceptTouchEvent-- action=0
OnTouchListener--onTouch-- action=0 --com.zzci.light.TestLinearLayout
TestLinearLayout onTouchEvent-- action=0
TestLinearLayout dispatchTouchEvent-- action=1
OnTouchListener--onTouch-- action=1 --com.zzci.light.TestLinearLayout
TestLinearLayout onTouchEvent-- action=1
OnClickListener--onClick--com.zzci.light.TestLinearLayout
```

- 分析：你会发现一个奇怪的现象，派发 ACTION$_{DOWN\ action}$=0）事件时顺序为 dispatchTouchEvent->onInterceptTouchEvent->onTouch->onTouchEvent，而接着派发 ACTION$_{UP\ action}$=1）事件时与上面顺序不同的时竟然没触发 onInterceptTouchEvent 方法。这是为啥呢？
- 我也纳闷，那就留着下面分析源码再找答案吧，先记住这个问题。
- 有了上面这个例子你是不是发现包含 ViewGroup 与 View 的事件触发有些相似又有很大差异吧（PS：在 Android 中继承 View 实现的控件已经是最小单位了，也即在 XML 布局等操作中不能再包含子项了，而继承 ViewGroup 实现的控件通常不是最小单位，可以包含不确定数目的子项）。具体差异是啥呢？咱们类似上篇一样，带着这个实例疑惑去看源码找答案吧。

7.2 Android 5.1.1(API 22) ViewGroup 触摸屏事件传递源码分析

- 通过上面例子的打印我们可以确定分析源码的顺序，那就开始分析呗。

7.2.1 从 ViewGroup 的 dispatchTouchEvent 方法说起

- 前一篇的 3-2 小节说在 Android 中你只要触摸控件首先都会触发控件的 dispatchTouchEvent 方法（其实这个方法一般都没在具体的控件类中，而在他的父类 View 中）。这其实是思维单单局限在 View 的角度去看待的，这里通过上面的例子你是否发现触摸控件会先从他的父级 dispatchTouchEvent 方法开始派发呢？是的，所以咱们先从 ViewGroup 的 dispatchTouchEvent 方法说起，如下：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onTouchEvent(ev, 1);
    }
    // If the event targets the accessibility focused view and this is it, start
    // normal event dispatch. Maybe a descendant is what will handle the click.
    if (ev.isTargetAccessibilityFocus() && isAccessibilityFocusedViewOrHost()) {
        ev.setTargetAccessibilityFocus(false);
    }
    boolean handled = false;
    if (onFilterTouchEventForSecurity(ev)) {
        final int action = ev.getAction();
        final int actionMasked = action & MotionEvent.ACTION_MASK;
        // Handle an initial down.
// 因为 ACTION_DOWN 是一系列事件的开端，当是 ACTION_DOWN 时进行一些初始化操作:
        if (actionMasked == MotionEvent.ACTION_DOWN) {
            // Throw away all previous state when starting a new touch gesture.
            // The framework may have dropped the up or cancel event for the previous gesture
            // due to an app switch, ANR, or some other state change.
        // 清除以往的 Touch 状态然后开始新的手势
            // 在这里你会发现 cancelAndClearTouchTargets(ev) 方法中有一个非常重要的操作就是将 mFirstTouchTarget 设置为了 null
            // 这个是分析 ViewGroup 的 dispatchTouchEvent 方法第一步中重点要记住的一个地方
            cancelAndClearTouchTargets(ev);
        // 接着在 resetTouchState() 方法中重置 Touch 状态标识
            resetTouchState();
        }
        // Check for interception.
// 检查是否要拦截
    // 使用变量 intercepted 来标记 ViewGroup 是否拦截 Touch 事件的传递
        final boolean intercepted;
        // 当事件为 ACTION_DOWN 或者 mFirstTouchTarget 不为 null(即已经找到能够接收 touch 事件的目标组件) 时 if 成立
        if (actionMasked == MotionEvent.ACTION_DOWN || mFirstTouchTarget != null) {
            // 当事件为 ACTION_DOWN 或者 mFirstTouchTarget 不为 null 时判断 disallowIntercept(禁止拦截) 标志位
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                intercepted = onInterceptTouchEvent(ev);
                ev.setAction(action); // restore action in case it was changed
            } else {
                intercepted = false;
            }
        } else {
            // There are no touch targets and this action is not an initial down
            // so this view group continues to intercept touches.
```

```
            intercepted = true;
        }
        // If intercepted, start normal event dispatch. Also if there is already
        // a view that is handling the gesture, do normal event dispatch.
        if (intercepted || mFirstTouchTarget != null) {
            ev.setTargetAccessibilityFocus(false);
        }
        // Check for cancelation.
        final boolean canceled = resetCancelNextUpFlag(this)
            || actionMasked == MotionEvent.ACTION_CANCEL;
        // Update list of touch targets for pointer down, if needed.
        final boolean split = (mGroupFlags & FLAG_SPLIT_MOTION_EVENTS) != 0;
        TouchTarget newTouchTarget = null;
        boolean alreadyDispatchedToNewTouchTarget = false;
// 如果没有取消没有拦截，就进入执行下面的逻辑：
        if (!canceled && !intercepted) {
            // If the event is targeting accessiiblity focus we give it to the
            // view that has accessibility focus and if it does not handle it
            // we clear the flag and dispatch the event to all children as usual.
            // We are looking up the accessibility focused host to avoid keeping
            // state since these events are very rare.
            View childWithAccessibilityFocus = ev.isTargetAccessibilityFocus()
                ? findChildWithAccessibilityFocus() : null;
            if (actionMasked == MotionEvent.ACTION_DOWN
                || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
                final int actionIndex = ev.getActionIndex(); // always 0 for down
                final int idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex)
                    : TouchTarget.ALL_POINTER_IDS;
                // Clean up earlier touch targets for this pointer id in case they
                // have become out of sync.
                removePointersFromTouchTargets(idBitsToAssign);
                final int childrenCount = mChildrenCount;
                // 判断了 childrenCount 个数是否不为 0
                if (newTouchTarget == null && childrenCount != 0) {
                    final float x = ev.getX(actionIndex);
                    final float y = ev.getY(actionIndex);
                    // Find a child that can receive the event.
                    // Scan children from front to back.
                    final ArrayList<View> preorderedList = buildOrderedChildList();
                    final boolean customOrder = preorderedList == null
                        && isChildrenDrawingOrderEnabled();
                    final View[] children = mChildren;
// 倒序遍历所有的子 view：倒序是因为：
// 这是因为 preorderedList 中的顺序是按照 addView 或者 XML 布局文件中的顺序来的，后 addView 添加的子 View，会因为 Android 的 UI
// 假如点击的地方有两个子 View 都包含的点击的坐标，那么后被添加到布局中的那个子 view 会先响应事件；
// 这样其实也是符合人的思维方式的，因为后被添加的子 view 会浮在上层，所以我们去点击的时候一般都会希望点击最上层的那个组件先去响应事件
                    for (int i = childrenCount - 1; i >= 0; i--) { // 倒序遍历所有的子 view
                        final int childIndex = customOrder
                            ? getChildDrawingOrder(childrenCount, i) : i;
                        final View child = (preorderedList == null)
                            ? children[childIndex] : preorderedList.get(childIndex);
                        // If there is a view that has accessibility focus we want it
                        // to get the event first and if not handled we will perform a
                        // normal dispatch. We may do a double iteration but this is
                        // safer given the timeframe.
                        if (childWithAccessibilityFocus != null) {
                            if (childWithAccessibilityFocus != child) {
                                continue;
                            }
                            childWithAccessibilityFocus = null;
                            i = childrenCount - 1;
                        }
                        if (!canViewReceivePointerEvents(child)
                            || !isTransformedTouchPointInView(x, y, child, null)) {
                            ev.setTargetAccessibilityFocus(false);
                            continue;
                        }
// 通过 getTouchTarget 去查找当前子 View 是否在 mFirstTouchTarget.next 这条 target 链中的某一个 targe 中，如果在则返回这个 targe
                        newTouchTarget = getTouchTarget(child);
                        if (newTouchTarget != null) {
                            // 说明找到了接收 Touch 事件的子 View，即 newTouchTarget，那么，既然已经找到了，所以执行 break 跳出
                            // Child is already receiving touch within its bounds.
                            // Give it the new pointer in addition to the ones it is handling.
                            newTouchTarget.pointerIdBits |= idBitsToAssign;
```

```
                            break;
                        }
                        resetCancelNextUpFlag(child);
// 调用方法 dispatchTransformedTouchEvent() 将 Touch 事件传递给特定的子 View。
// 该方法十分重要，在该方法中为一个递归调用，会递归调用 dispatchTouchEvent() 方法。
// 在 dispatchTouchEvent() 中如果子 View 为 ViewGroup 并且 Touch 没有被拦截，那么递归调用 dispatchTouchEvent()，
// 如果子 View 为 View 那么就会调用其 onTouchEvent()。dispatchTransformedTouchEvent 方法如果返回 true 则表示子 View 消费掉该事件
                        if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                            // Child wants to receive touch within its bounds.
                            mLastTouchDownTime = ev.getDownTime();
                            if (preorderedList != null) {
                                // childIndex points into presorted list, find original index
                                for (int j = 0; j < childrenCount; j++) {
                                    if (children[childIndex] == mChildren[j]) {
                                        mLastTouchDownIndex = j;
                                        break;
                                    }
                                }
                            } else {
                                mLastTouchDownIndex = childIndex;
                            }
                            mLastTouchDownX = ev.getX();
                            mLastTouchDownY = ev.getY();
                            // 给 newTouchTarget 赋值
                            newTouchTarget = addTouchTarget(child, idBitsToAssign);
                            // 给 alreadyDispatchedToNewTouchTarget 赋值为 true
                            alreadyDispatchedToNewTouchTarget = true;
                            // 执行 break，因为该 for 循环遍历子 View 判断哪个子 View 接受 Touch 事件，既然已经找到了就跳出该外层
                            break;
                        }
                        // The accessibility focus didn't handle the event, so clear
                        // the flag and do a normal dispatch to all children.
                        ev.setTargetAccessibilityFocus(false);
                    }
                    if (preorderedList != null) preorderedList.clear();
                } // (newTouchTarget == null && childrenCount != 0)  结束
// 该 if 表示经过前面的 for 循环没有找到子 View 接收 Touch 事件并且之前的 mFirstTouchTarget 不为空则为真
                if (newTouchTarget == null && mFirstTouchTarget != null) {
                    // Did not find a child to receive the event.
                    // Assign the pointer to the least recently added target.
                    newTouchTarget = mFirstTouchTarget; // newTouchTarget 指向了最初的 TouchTarget
                    while (newTouchTarget.next != null) {
                        newTouchTarget = newTouchTarget.next;
                    }
                    newTouchTarget.pointerIdBits |= idBitsToAssign;
                }
            }
        }
        // Dispatch to touch targets.
// if 判断的 mFirstTouchTarget 为 null 时，也就是说 Touch 事件未被消费，即没有找到能够消费 touch 事件的子组件或 Touch 事件被拦截了
// 则调用 ViewGroup 的 dispatchTransformedTouchEvent() 方法处理 Touch 事件（和普通 View 一样），即子 View 没有消费 Touch 事件，
// 那么子 View 的上层 ViewGroup 才会调用其 onTouchEvent() 处理 Touch 事件。
        if (mFirstTouchTarget == null) {
            // No touch targets so treat this as an ordinary view.
            handled = dispatchTransformedTouchEvent(ev, canceled, null,
                                                    TouchTarget.ALL_POINTER_IDS);
        } else {
            // 找到了可以消费 Touch 事件的子 View 且后续 Touch 事件可以传递到该子 View
            // 对于非 ACTION_DOWN 事件继续传递给目标子组件进行处理，依然是递归调用 dispatchTransformedTouchEvent() 方法来实现的处
            // Dispatch to touch targets, excluding the new touch target if we already
            // dispatched to it.  Cancel touch targets if necessary.
            TouchTarget predecessor = null;
            TouchTarget target = mFirstTouchTarget;
            while (target != null) {
                final TouchTarget next = target.next;
                if (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {
                    handled = true;
                } else {
                    final boolean cancelChild = resetCancelNextUpFlag(target.child)
                            || intercepted;
                    if (dispatchTransformedTouchEvent(ev, cancelChild,
                                                      target.child, target.pointerIdBits)) {
                        handled = true;
                    }
                    if (cancelChild) {
```

```
                    if (predecessor == null) {
                        mFirstTouchTarget = next;
                    } else {
                        predecessor.next = next;
                    }
                    target.recycle();
                    target = next;
                    continue;
                }
            }
            predecessor = target;
            target = next;
        }
    }
    // Update list of touch targets for pointer up or cancel, if needed.
    if (canceled
        || actionMasked == MotionEvent.ACTION_UP
        || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
        resetTouchState();
    } else if (split && actionMasked == MotionEvent.ACTION_POINTER_UP) {
        final int actionIndex = ev.getActionIndex();
        final int idBitsToRemove = 1 << ev.getPointerId(actionIndex);
        removePointersFromTouchTargets(idBitsToRemove);
    }
}
if (!handled && mInputEventConsistencyVerifier != null) {
    mInputEventConsistencyVerifier.onUnhandledEvent(ev, 1);
}
return handled;
}
```

- 我勒个去!!! 这比 View 的 dispatchTouchEvent 方法长很多啊，那就只关注重点分析吧。
- 第一步，17-24 行，对 ACTION$_{DOWN}$
- 因为 ACTION$_{DOWN}$ $ACTION DOWN$ $Touch$ $cancelAndClearTouchTargets$(ev) 方法中有一个非常重要的操作就是将 mFirstTouchTarget 设置为了 null（刚开始分析大眼瞄一眼没留意，结果越往下看越迷糊，所以这个是分析 ViewGroup 的 dispatchTouchEvent 方法第一步中重点要记住的一个地方），接着在 resetTouchState() 方法中重置 Touch 状态标识。
- 第二步，26-47 行，检查是否要拦截。
- 在 dispatchTouchEvent(MotionEvent ev) 这段代码中使用变量 intercepted 来标记 ViewGroup 是否拦截 Touch 事件的传递，该变量类似第一步的 mFirstTouchTarget 变量，在后续代码中起着很重要的作用。if (actionMasked = MotionEvent.ACTION_DOWN || mFirstTouchTarget ! null) 这一条判断语句说明当事件为 ACTION$_{DOWN}$ $mFirstTouchTarget$ $null$(即已经找到能够接收 touch 事件的目标组件) 时 if 成立，否则 if 不成立，然后将 intercepted 设置为 true，也即拦截事件。当当事件为 ACTION$_{DOWN}$ $mFirstTouchTarget$ $null$ $disallowIntercept$(禁止拦截) 标志位，而这个标记在 ViewGroup 中提供了 public 的设置方法，如下：

```
public void requestDisallowInterceptTouchEvent(boolean disallowIntercept) {
        if (disallowIntercept == ((mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0)) {
            // We're already in this state, assume our ancestors are too
            return;
        }
        if (disallowIntercept) {
            mGroupFlags |= FLAG_DISALLOW_INTERCEPT;
        } else {
            mGroupFlags &= ~FLAG_DISALLOW_INTERCEPT;
        }
        // Pass it up to our parent
        if (mParent != null) {
            mParent.requestDisallowInterceptTouchEvent(disallowIntercept);
        }
    }
```

- 所以你可以在其他地方调用 requestDisallowInterceptTouchEvent(boolean disallowIntercept) 方法，从而禁止执行是否需要拦截的判断。

- 当 disallowIntercept 为 true（禁止拦截判断）时则 intercepted 直接设置为 false，否则调用 onInterceptTouchEvent(ev) 方法，然后将结果赋值给 intercepted。那就来看下 ViewGroup 与众不同与 View 特有的 onInterceptTouchEvent 方法，如下：

```
public boolean onInterceptTouchEvent(MotionEvent ev) {
        return false;
    }
```

- 看见了吧，默认的 onInterceptTouchEvent 方法只是返回了一个 false，也即 intercepted=false。所以可以说明上面例子的部分打印（dispatchTouchEvent->onInterceptTouchEvent->onTouchEvent 这里很明显表明在 ViewGroup 的 dispatchTouchEvent() 中默认（不在其他地方调运 requestDisallowInterceptTouchEvent 方法设置禁止拦截标记）首先调用了 onInterceptTouchEvent() 方法。

- 第三步，49-51 行，检查 cancel。

- 通过标记和 action 检查 cancel，然后将结果赋值给局部 boolean 变量 canceled。

- 第四步，53-函数结束，事件分发。

- 54 行首先可以看见获取一个 boolean 变量标记 split 来标记，默认是 true，作用是是否把事件分发给多个子 View，这个同样在 ViewGroup 中提供了 public 的方法设置，如下：

```
public void setMotionEventSplittingEnabled(boolean split) {
        // TODO Applications really shouldn't change this setting mid-touch event,
        // but perhaps this should handle that case and send ACTION_CANCELs to any child views
        // with gestures in progress when this is changed.
        if (split) {
            mGroupFlags |= FLAG_SPLIT_MOTION_EVENTS;
        } else {
            mGroupFlags &= ~FLAG_SPLIT_MOTION_EVENTS;
        }
    }
```

- 接着 57 行 if (!canceled && !intercepted) 判断表明，事件不是 ACTION$_{CANCEL}$ $ViewGroup$ $intercepte$ 拦截）则会进入其中。事件分发步骤中关于 ACTION$_{DOWN}$ 67 if if (actionMasked = MotionEvent.ACTION_DOWN || (split && actionMasked = MotionEvent.ACTION$_{POINTERDOW}$ || actionMasked == MotionEvent.ACTION$_{HOVERMOVE}$) 处理 ACTION$_{DOWN}$

- 在 79 行判断了 childrenCount 个数是否不为 0，然后接着在 84 行拿到了子 View 的 list 集合 preorderedList；接着在 88 行通过一个 for 循环 i 从 childrenCount - 1 开始遍历到 0，倒序遍历所有的子 view，这是因为 preorderedList 中的顺序是按照 addView 或者 XML 布局文件中的顺序来的，后 addView 添加的子 View，会因为 Android 的 UI 后刷新机制显示在上层；假如点击的地方有两个子 View 都包含的点击的坐标，那么后被添加到布局中的那个子 view 会先响应事件；这样其实也是符合人的思维方式的，因为后被添加的子 view 会浮在上层，所以我们去点击的时候一般都会希望点击最上层的那个组件先去响应事件。

- 接着在 106 到 112 行通过 getTouchTarget 去查找当前子 View 是否在 mFirstTouchTarget.next 这条 target 链中的某一个 targe 中，如果在则返回这个 target，否则返回 null。在这段代码的 if 判断通过说明找到了接收 Touch 事件的子 View，即 newTouchTarget，那么，既然已经找到了，所以执行 break 跳出 for 循环。如果没有 break 则继续向下执行走到 115 行开始到 134 行，这里你可以看见一段 if 判断的代码 if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign))，这个被 if 的大括弧括起来的一段代码很重要，具体解释如下：

- 调用方法 dispatchTransformedTouchEvent() 将 Touch 事件传递给特定的子 View。该方法十分重要，在该方法中为一个递归调用，会递归调用 dispatchTouchEvent() 方法。在 dispatchTouchEvent() 中如果子 View 为 ViewGroup 并且 Touch 没有被拦截那么递归调用 dispatchTouchEvent()，如果子 View 为 View 那么就会调用其 onTouchEvent()。dispatchTransformedTouchEvent 方法如果返回 true 则表示子 View 消费掉该事件，同时进入

- 该 if 判断。满足 if 语句后重要的操作有：
  - 给 newTouchTarget 赋值；
  - 给 alreadyDispatchedToNewTouchTarget 赋值为 true；
  - 执行 break，因为该 for 循环遍历子 View 判断哪个子 View 接受 Touch 事件，既然已经找到了就跳出该外层 for 循环；
- 如果 115 行 if 判断中的 dispatchTransformedTouchEvent() 方法返回 false，即子 View 的 onTouchEvent 返回 false(即 Touch 事件未被消费)，那么就不满足该 if 条件，也就无法执行 addTouchTarget()，从而导致 mFirstTouchTarget 为 null(没法对 mFirstTouchTarget 赋值，因为上面分析了 mFirstTouchTarget 一进来是 ACTION$_{DOWN}$ $null$ $View$ $ACTIONMOVE$ $ACTIONUP$ 了，所以之后一系列判断无法通过)。
- 如果 115 行 if 判断中的 dispatchTransformedTouchEvent() 方法返回 true，即子 View 的 onTouchEvent 返回 true(即 Touch 事件被消费)，那么就满足该 if 条件，从而 mFirstTouchTarget 不为 null。
- 继续看 143 行的判断 if (newTouchTarget = null && mFirstTouchTarget ! null)。该 if 表示经过前面的 for 循环没有找到子 View 接收 Touch 事件并且之前的 mFirstTouchTarget 不为空则为真，然后 newTouchTarget 指向了最初的 TouchTarget。
- 通过上面 67 到 157 行关于事件分发步骤中 ACTION$_{DOWN}$ $ACTIONDOWN$ $dispatchTransforme$ 方法，该方法返回值具备如下特征：

return	description	set
true	事件被消费	mFirstTouchTarget!=null
false	事件未被消费	mFirstTouchTarget==null

- 因为在 dispatchTransformedTouchEvent() 会调用递归调用 dispatchTouchEvent() 和 onTouchEvent()，所以 dispatchTransformedTouchEvent() 的返回值实际上是由 onTouchEvent() 决定的。简单地说 onTouchEvent() 是否消费了 Touch 事件的返回值决定了 dispatchTransformedTouchEvent() 的返回值，从而决定 mFirstTouchTarget 是否为 null，进一步决定了 ViewGroup 是否处理 Touch 事件，这一点在 160 行开始的代码中有体现。如下分析事件分发步骤中关于 ACTION$_{DOWN}$ 160
- 事件分发步骤中关于 ACTION$_{DOWN}$
- 可以看到，如果派发的事件不是 ACTION$_{DOWN}$ $ACTIONDOWN$ $mFirstTouchTarget$ $null$ (mFirstTouchTarget == null) 与 else 判断了 mFirstTouchTarget 值是否为 null 的情况，完全符合如上分析。那我们分情况继续分析一下：
- 当 161 行 if 判断的 mFirstTouchTarget 为 null 时，也就是说 Touch 事件未被消费，即没有找到能够消费 touch 事件的子组件或 Touch 事件被拦截了，则调用 ViewGroup 的 dispatchTransformedTouchEvent() 方法处理 Touch 事件（和普通 View 一样），即子 View 没有消费 Touch 事件，那么子 View 的上层 ViewGroup 才会调用其 onTouchEvent() 处理 Touch 事件。具体就是在调用 dispatchTransformedTouchEvent() 时第三个参数为 null，关于 dispatchTransformedTouchEvent 方法下面会分析，暂时先记住就行。
- 这下再回想上面例子，点击 Button 时为啥触发了 Button 的一系列 touch 方法而没有触发父级 LinearLayout 的 touch 方法的疑惑？明白了吧？
- 子 view 对于 Touch 事件处理返回 true 那么其上层的 ViewGroup 就无法处理 Touch 事件了，子 view 对于 Touch 事件处理返回 false 那么其上层的 ViewGroup 才可以处理 Touch 事件。

- 当 161 行 if 判断的 mFirstTouchTarget 不为 null 时，也就是说找到了可以消费 Touch 事件的子 View 且后续 Touch 事件可以传递到该子 View。可以看见在源码的 else 中对于非 $ACTION_{DOWN}$ $dispatchTransformedTouchEvent()$ 方法来实现的处理。
- 到此 ViewGroup 的 dispatchTouchEvent 方法分析完毕。
- 上面说了 ViewGroup 的 dispatchTouchEvent 方法详细情况，也知道在其中可能会执行 onInterceptTouchEvent 方法，所以接下来咱们先简单分析一下这个方法。
- 如下系统源码：

```
public boolean onInterceptTouchEvent(MotionEvent ev) {
        return false;
    }
```

- 看到了吧，这个方法算是 ViewGroup 不同于 View 特有的一个事件派发调运方法。在源码中可以看到这个方法实现很简单，但是有一堆注释。其实上面
- 分析了，如果 ViewGroup 的 onInterceptTouchEvent 返回 false 就不阻止事件继续传递派发，否则阻止传递派发。
- 对了，还记得在 dispatchTouchEvent 方法中除过可能执行的 onInterceptTouchEvent 以外在后面派发事件时执行的 dispatchTransformedTouchEvent 方法吗？上面分析 dispatchTouchEvent 时说了下面会仔细分析，那么现在就来继续看看这个方法吧。

7.2.2 继续说说 ViewGroup 的 dispatchTouchEvent 中执行的 dispatchTransformedTouchEvent 方法

- ViewGroup 的 dispatchTransformedTouchEvent 方法系统源码如下：

```
private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
                                              View child, int desiredPointerIdBits) {
    final boolean handled;
    // Canceling motions is a special case.  We don't need to perform any transformations
    // or filtering.  The important part is the action, not the contents.
    final int oldAction = event.getAction();
    if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
        event.setAction(MotionEvent.ACTION_CANCEL);
        if (child == null) {
            handled = super.dispatchTouchEvent(event);
        } else {
            handled = child.dispatchTouchEvent(event);
        }
        event.setAction(oldAction);
        return handled;
    }
    // Calculate the number of pointers to deliver.
    final int oldPointerIdBits = event.getPointerIdBits();
    final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;
    // If for some reason we ended up in an inconsistent state where it looks like we
    // might produce a motion event with no pointers in it, then drop the event.
    if (newPointerIdBits == 0) {
        return false;
    }
    // If the number of pointers is the same and we don't need to perform any fancy
    // irreversible transformations, then we can reuse the motion event for this
    // dispatch as long as we are careful to revert any changes we make.
    // Otherwise we need to make a copy.
    final MotionEvent transformedEvent;
    if (newPointerIdBits == oldPointerIdBits) {
        if (child == null || child.hasIdentityMatrix()) {
            if (child == null) {
                handled = super.dispatchTouchEvent(event);
            } else {
                final float offsetX = mScrollX - child.mLeft;
                final float offsetY = mScrollY - child.mTop;
```

```
                event.offsetLocation(offsetX, offsetY);
                handled = child.dispatchTouchEvent(event);
                event.offsetLocation(-offsetX, -offsetY);
            }
            return handled;
        }
        transformedEvent = MotionEvent.obtain(event);
    } else {
        transformedEvent = event.split(newPointerIdBits);
    }
    // Perform any necessary transformations and dispatch.
    if (child == null) {
        handled = super.dispatchTouchEvent(transformedEvent);
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        transformedEvent.offsetLocation(offsetX, offsetY);
        if (! child.hasIdentityMatrix()) {
            transformedEvent.transform(child.getInverseMatrix());
        }
        handled = child.dispatchTouchEvent(transformedEvent);
    }
    // Done.
    transformedEvent.recycle();
    return handled;
}
```

- 看到了吧，这个方法也算是 ViewGroup 不同于 View 特有的一个事件派发调运方法，而且奇葩的就是这个方法也很长。那也继续分析吧。。。
- 上面分析了，在 dispatchTouchEvent() 中调用 dispatchTransformedTouchEvent() 将事件分发给子 View 处理。在此我们需要重点分析该方法的第三个参数（View child）。在 dispatchTouchEvent() 中多次调用了 dispatchTransformedTouchEvent() 方法，而且有时候第三个参数为 null，有时又不是，他们到底有啥区别呢？这段源码中很明显展示了结果。在 dispatchTransformedTouchEvent() 源码中可以发现多次对于 child 是否为 null 的判断，并且均做出如下类似的操作。其中，当 child = null 时会将 Touch 事件传递给该 ViewGroup 自身的 dispatchTouchEvent() 处理，即 super.dispatchTouchEvent(event)（也就是 View 的这个方法，因为 ViewGroup 的父类是 View）；当 child ! null 时会调用该子 view(当然该 view 可能是一个 View 也可能是一个 ViewGroup) 的 dispatchTouchEvent(event) 处理，即 child.dispatchTouchEvent(event)。别的代码几乎没啥需要具体注意分析的。
- 所以，到此你也会发现 ViewGroup 没有重写 View 的 onTouchEvent(MotionEvent event) 方法，也就是说接下来的调运关系就是上一篇分析的流程了，这里不在多说。
- 好了，到此你是不是即明白了上面实例演示的代码结果，也明白了上一篇最后升级实例验证模块留下的点击 Button 触发了 LinearLayout 的一些疑惑呢？答案自然是必须的！

7.3 Android 5.1.1(API 22) ViewGroup 触摸屏事件传递总结

- 如上就是所有 ViewGroup 关于触摸屏事件的传递机制源码分析与实例演示。具体总结如下：
- Android 事件派发是先传递到最顶级的 ViewGroup，再由 ViewGroup 递归传递到 View 的。
- 在 ViewGroup 中可以通过 onInterceptTouchEvent 方法对事件传递进行拦截，onInterceptTouchEvent 方法返回 true 代表不允许事件继续向子 View 传递，返回 false 代表不对事件进行拦截，默认返回 false。
- 子 View 中如果将传递的事件消费掉，ViewGroup 中将无法接收到任何事件。

8 Android 触摸屏事件派发机制详解与源码分析三 (Activity 篇)

- https://blog.csdn.net/yanbober/article/details/45932123

8.1 实例验证

8.1.1 代码

- 如下实例与前面实例相同，一个 Button 在 LinearLayout 里，只不过我们这次重写了 Activity 的一些方法而已。具体如下：

- 自定义的 Button 与 LinearLayout：

```
public class TestButton extends Button {
    public TestButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        Log.i(null, "TestButton--dispatchTouchEvent--action="+event.getAction());
        return super.dispatchTouchEvent(event);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "TestButton--onTouchEvent--action="+event.getAction());
        return super.onTouchEvent(event);
    }
}
public class TestLinearLayout extends LinearLayout {
    public TestLinearLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean onInterceptTouchEvent(MotionEvent ev) {
        Log.i(null, "TestLinearLayout--onInterceptTouchEvent--action="+ev.getAction());
        return super.onInterceptTouchEvent(ev);
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        Log.i(null, "TestLinearLayout--dispatchTouchEvent--action=" + event.getAction());
        return super.dispatchTouchEvent(event);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "TestLinearLayout--onTouchEvent--action="+event.getAction());
        return super.onTouchEvent(event);
    }
}
```

- 整个界面的布局文件：

```
<com.example.yanbo.myapplication.TestLinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/layout">
    <com.example.yanbo.myapplication.TestButton
        android:text="click test"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/button"/>
</com.example.yanbo.myapplication.TestLinearLayout>
```

- 整个界面 Activity，重写了 Activity 的一些关于触摸派发的方法（三个）：

```
public class MainActivity extends Activity implements View.OnClickListener, View.OnTouchListener {
    private TestButton mButton;
    private TestLinearLayout mLayout;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        mButton = (TestButton) this.findViewById(R.id.button);
        mLayout = (TestLinearLayout) this.findViewById(R.id.layout);
        mButton.setOnClickListener(this);
        mLayout.setOnClickListener(this);
```

```
        mButton.setOnTouchListener(this);
        mLayout.setOnTouchListener(this);
    }
    @Override
    public void onClick(View v) {
        Log.i(null, "onClick----v=" + v);
    }
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.i(null, "onTouch--action="+event.getAction()+"--v="+v);
        return false;
    }
    @Override
    public boolean dispatchTouchEvent(MotionEvent ev) {
        Log.i(null, "MainActivity--dispatchTouchEvent--action=" + ev.getAction());
        return super.dispatchTouchEvent(ev);
    }
    @Override
    public void onUserInteraction() {
        Log.i(null, "MainActivity--onUserInteraction");
        super.onUserInteraction();
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.i(null, "MainActivity--onTouchEvent--action="+event.getAction());
        return super.onTouchEvent(event);
    }
}
```

- 如上就是实例测试代码，非常简单，没必要分析，直接看结果吧。

8.1.2 结果分析

- 直接点击 Button 按钮打印如下：

```
MainActivity--dispatchTouchEvent--action=0
MainActivity--onUserInteraction
TestLinearLayout--dispatchTouchEvent--action=0
TestLinearLayout--onInterceptTouchEvent--action=0
TestButton--dispatchTouchEvent--action=0
onTouch--action=0--v=com.example.yanbo.myapplication.TestButton
TestButton--onTouchEvent--action=0

MainActivity--dispatchTouchEvent--action=1
TestLinearLayout--dispatchTouchEvent--action=1
TestLinearLayout--onInterceptTouchEvent--action=1
TestButton--dispatchTouchEvent--action=1
onTouch--action=1--v=com.example.yanbo.myapplication.TestButton
TestButton--onTouchEvent--action=1
onClick----v=com.example.yanbo.myapplication.TestButton
```

- 分析可以发现，当点击 Button 时除过派发 Activity 的几个新方法之外其他完全符合前面两篇分析的 View 与 ViewGroup 的触摸事件派发机制。对于 Activity 来说，ACTION$_{DOWN}$ $dispatchTouchEv$

- 直接点击 Button 以外的其他区域：

```
MainActivity--dispatchTouchEvent--action=0
MainActivity--onUserInteraction
TestLinearLayout--dispatchTouchEvent--action=0
TestLinearLayout--onInterceptTouchEvent--action=0
onTouch--action=0--v=com.example.yanbo.myapplication.TestLinearLayout
TestLinearLayout--onTouchEvent--action=0

MainActivity--dispatchTouchEvent--action=1
TestLinearLayout--dispatchTouchEvent--action=1
onTouch--action=1--v=com.example.yanbo.myapplication.TestLinearLayout
TestLinearLayout--onTouchEvent--action=1
onClick----v=com.example.yanbo.myapplication.TestLinearLayout
```

- 怎么样？完全符合上面点击 Button 结果分析的猜想。
- 那接下来还是要看看 Activity 里关于这几个方法的源码了。

8.2 Android 5.1.1(API 22) Activity 触摸屏事件传递源码分析

- 通过上面例子的打印我们可以确定分析源码的顺序，那就开始分析呗。

8.2.1 从 Activity 的 dispatchTouchEvent 方法说起

1. 开始分析

    - 先上源码，如下：

```
/**
 * Called to process touch screen events.  You can override this to
 * intercept all touch screen events before they are dispatched to the
 * window.  Be sure to call this implementation for touch screen events
 * that should be handled normally.
 *
 * @param ev The touch screen event.
 *
 * @return boolean Return true if this event was consumed.
 */
public boolean dispatchTouchEvent(MotionEvent ev) {
    // 只有 ACTION_DOWN 事件派发时调运了 onUserInteraction 方法
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {
        onUserInteraction();
    }
// 本质执行的是一个 ViewGroup 的 dispatchTouchEvent 方法
// (这个 ViewGroup 是 Activity 特有的 root view, 也就是 id 为 content 的 FrameLayout 布局)
    if (getWindow().superDispatchTouchEvent(ev)) {
        return true;
    }
    return onTouchEvent(ev);
}
```

    - 哎呦！这次看着代码好少的样子，不过别高兴，浓缩才是精华，这里代码虽少，涉及的问题点还是很多的，那么咱们就来一点一点分析吧。
    - 12 到 14 行看见了吧?上面例子咱们看见只有 ACTION$_{DOWN}$ $onUserInteraction$
    - 好了，自己分析 15 到 17 行，看着简单吧，我勒个去，我怎么有点懵，这是哪的方法?咱们分析分析吧。
    - 首先分析 Activity 的 attach 方法可以发现 getWindow() 返回的就是 PhoneWindow 对象(PhoneWindow 为抽象 Window 的实现子类)，那就简单了，也就相当于 PhoneWindow 类的方法，而 PhoneWindow 类实现于 Window 抽象类，所以先看下 Window 类中抽象方法的定义，如下：

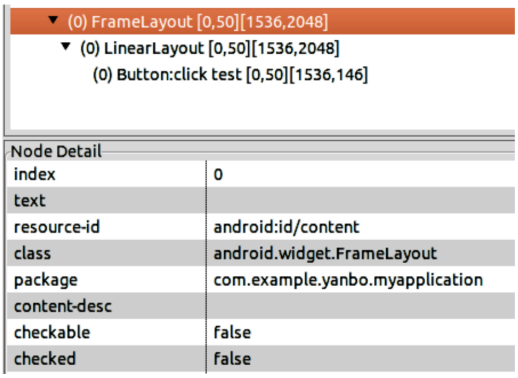
```
/**
     * Used by custom windows, such as Dialog, to pass the touch screen event
     * further down the view hierarchy. Application developers should
     * not need to implement or call this.
     *
     */
// 用户不需要重写实现的方法
    public abstract boolean superDispatchTouchEvent(MotionEvent event);
```

    - 看见注释没有？用户不需要重写实现的方法，实质也不能，在 Activity 中没有提供重写的机会，因为 Window 是以组合模式与 Activity 建立关系的。好了，看完了抽象的 Window 方法，那就去 PhoneWindow 里看下 Window 抽象方法的实现吧，如下：

```
@Override
public boolean superDispatchTouchEvent(MotionEvent event) {
    return mDecor.superDispatchTouchEvent(event);
}
```

    - 又是看着好简单的样子哦，实际又是一堆问题，继续分析。你会发现在 PhoneWindow 的 superDispatchTouchEvent 方法里又直接返回了另一个 mDecor 对象的 superDispatch-TouchEvent 方法，mDecor 是啥？继续分析吧。

- 在 PhoneWindow 类里发现，mDecor 是 DecorView 类的实例，同时 DecorView 是 PhoneWindow 的内部类。最惊人的发现是 DecorView extends FrameLayout implements RootViewSurfaceTaker，看见没有？它是一个真正 Activity 的 root view，它继承了 FrameLayout。怎么验证他一定是 root view 呢？很简单，不知道大家是不是熟悉 Android App 开发技巧中关于 UI 布局优化使用的 SDK 工具 Hierarchy Viewer。咱们通过他来看下上面刚刚展示的那个例子的 Hierarchy Viewer 你就明白了，如下我在 Ubuntu 上截图的 Hierarchy Viewer 分析结果：



- 看见没有，我们上面例子中 Activity 中 setContentView 时放入的 xml layout 是一个 LinearLayout，其中包含一个 Button，上图展示了我们放置的 LinearLayout 被放置在一个 id 为 content 的 FrameLayout 的布局中，这也就是为啥 Activity 的 setContentView 方法叫 set content view 了，就是把我们的 xml 放入了这个 id 为 content 的 FrameLayout 中。
- 赶快回过头，你是不是发现上面 PhoneWindow 的 superDispatchTouchEvent 直接返回了 DecorView 的 superDispatchTouchEvent，而 DecorView 又是 FrameLayout 的子类，FrameLayout 又是 ViewGroup 的子类。机智的你想到了啥木有？
- 没想到就继续看下 DecorView 类的 superDispatchTouchEvent 方法吧，如下：

```
public boolean superDispatchTouchEvent(MotionEvent event) {
    return super.dispatchTouchEvent(event);
}
```

- 这回你一定恍然大悟了吧，不然就得脑补前面两篇博客的内容了。。。
- 搞半天 Activity 的 dispatchTouchEvent 方法的 15 行 if (getWindow().superDispatchTouchEver 本质执行的是一个 ViewGroup 的 dispatchTouchEvent 方法（这个 ViewGroup 是 Activity 特有的 root view，也就是 id 为 content 的 FrameLayout 布局），接下来就不用多说了吧，完全是前面两篇分析的执行过程。
- 接下来依据派发事件返回值决定是否触发 Activity 的 onTouchEvent 方法。

2. 小总结一下

- 在 Activity 的触摸屏事件派发中：
  - 首先会触发 Activity 的 dispatchTouchEvent 方法。
  - dispatchTouchEvent 方法中如果是 ACTION$_{DOWN}$ $onUserInteraction$
  - 接着在 dispatchTouchEvent 方法中会通过 Activity 的 root View（id 为 content 的 FrameLayout），实质是 ViewGroup，通过 super.dispatchTouchEvent 把 touchevent 派发给各个 activity 的子 view，也就是我们再 Activity.onCreat 方法中 setContentView 时设置的 view。
  - 若 Activity 下面的子 view 拦截了 touchevent 事件 (返回 true) 则 Activity.onTouchEvent 方法就不会执行。

8.2.2 继续 Activity 的 dispatchTouchEvent 方法中调运的 onUserInteraction 方法

- 如下源码：

```
/**
 * Called whenever a key, touch, or trackball event is dispatched to the
 * activity.  Implement this method if you wish to know that the user has
 * interacted with the device in some way while your activity is running.
 * This callback and {@link #onUserLeaveHint} are intended to help
 * activities manage status bar notifications intelligently; specifically,
 * for helping activities determine the proper time to cancel a notfication.
 *
 * <p>All calls to your activity's {@link #onUserLeaveHint} callback will
 * be accompanied by calls to {@link #onUserInteraction}.  This
 * ensures that your activity will be told of relevant user activity such
 * as pulling down the notification pane and touching an item there.
 *
 * <p>Note that this callback will be invoked for the touch down action
 * that begins a touch gesture, but may not be invoked for the touch-moved
 * and touch-up actions that follow.
 *
 * @see #onUserLeaveHint()
 */
public void onUserInteraction() {
}
```

- 搞了半天就像上面说的，这是一个空方法，那它的作用是啥呢?
- 此方法是 activity 的方法，当此 activity 在栈顶时，触屏点击按 home，back，menu 键等都会触发此方法。下拉 statubar、旋转屏幕、锁屏不会触发此方法。所以它会用在屏保应用上，因为当你触屏机器就会立马触发一个事件，而这个事件又不太明确是什么，正好屏保满足此需求；或者对于一个 Activity，控制多长时间没有用户点响应的时候，自己消失等。
- 这个方法也分析完了，那就剩下 onTouchEvent 方法了，如下继续分析。

8.2.3 继续 Activity 的 dispatchTouchEvent 方法中调运的 onTouchEvent 方法

- 如下源码：

```
/**
 * Called when a touch screen event was not handled by any of the views
 * under it.  This is most useful to process touch events that happen
 * outside of your window bounds, where there is no view to receive it.
 *
 * @param event The touch screen event being processed.
 *
 * @return Return true if you have consumed the event, false if you haven't.
 * The default implementation always returns false.
 */
public boolean onTouchEvent(MotionEvent event) {
    if (mWindow.shouldCloseOnTouch(this, event)) {
        finish();
        return true;
    }
    return false;
}
```

- 看见没有，这个方法看起来好简单的样子。
- 如果一个屏幕触摸事件没有被这个 Activity 下的任何 View 所处理，Activity 的 onTouchEvent 将会调用。这对于处理 window 边界之外的 Touch 事件非常有用，因为通常是没有 View 会接收到它们的。返回值为 true 表明你已经消费了这个事件，false 则表示没有消费，默认实现中返回 false。
- 继续分析吧，重点就一句，mWindow.shouldCloseOnTouch(this, event) 中的 mWindow 实际就是上面分析 dispatchTouchEvent 方法里的 getWindow() 对象，所以直接到 Window 抽象类和 PhoneWindow 子类查看吧，发现 PhoneWindow 没有重写 Window 的 shouldCloseOnTouch 方法，所以看下 Window 类的 shouldCloseOnTouch 实现吧，如下：

```
/** @hide */
public boolean shouldCloseOnTouch(Context context, MotionEvent event) {
    if (mCloseOnTouchOutside && event.getAction() == MotionEvent.ACTION_DOWN
            && isOutOfBounds(context, event) && peekDecorView() != null) {
        return true;
    }
    return false;
}
```

- 这其实就是一个判断,判断 mCloseOnTouchOutside 标记及是否为 ACTION$_{DOWN}$ $event$ x y $Bound$
- 所以，到此 Activity 的 onTouchEvent 分析完毕。

8.3 Android 触摸事件综合总结

- 到此整个 Android 的 Activity->ViewGroup->View 的触摸屏事件分发机制完全分析完毕。这时候你可以回过头看这三篇文章的例子，你会完全明白那些打印的含义与原理。
- 当然，了解这些源码机制不仅对你写普通代码时有帮助，最重要的是对你想自定义装逼控件时有不可磨灭的基础性指导作用与技巧提示作用。