Android事件分发机制完全解析，带你从源码的角度彻底理解(上)

标签： Android 事件分发 onTouch dispatchTouchEvent onTouchEvent

2013-06-20 08:30 78354人阅读 评论(142) 收藏 举报

分类：

Android疑难解析（32）

其实我一直准备写一篇关于Android事件分发机制的文章，从我的第一篇博客开始，就零零散散在好多地方使用到了Android事件分发的知识。也有好多朋友问过我各种问题，比如：onTouch和onTouchEvent有什么区别，又该如何使用？为什么给ListView引入了一个滑动菜单的功能，ListView就不能滚动了？为什么图片轮播器里的图片使用Button而不用ImageView？等等……对于这些问题，我并没有给出非常详细的回答，因为我知道如果想要彻底搞明白这些问题，掌握Android事件分发机制是必不可少的，而Android事件分发机制绝对不是三言两语就能说得清的。

在我经过较长时间的筹备之后，终于决定开始写这样一篇文章了。目前虽然网上相关的文章也不少，但我觉得没有哪篇写得特别详细的(也许我还没有找到)，多数文章只是讲了讲理论，然后配合demo运行了一下结果。而我准备带着大家从源码的角度进行分析，相信大家可以更加深刻地理解Android事件分发机制。

阅读源码讲究由浅入深，循序渐进，因此我们也从简单的开始，本篇先带大家探究View的事件分发，下篇再去探究难度更高的ViewGroup的事件分发。

那我们现在就开始吧！比如说你当前有一个非常简单的项目，只有一个Activity，并且Activity中只有一个按钮。你可能已经知道，如果想要给这个按钮注册一个点击事件，只需要调用：

```java
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d("TAG", "onClick execute");
    }
});
```

这样在onClick方法里面写实现，就可以在按钮被点击的时候执行。你可能也已经知道，如果想给这个按钮再添加一个touch事件，只需要调用：

```java
button.setOnTouchListener(new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.d("TAG", "onTouch execute, action " + event.getAction());
        return false;
    }
});
```

onTouch方法里能做的事情比onClick要多一些，比如判断手指按下、抬起、移动等事件。那么如果我两个事件都注册了，哪一个会先执行呢？我们来试一下就知道了，运行程序点击按钮，打印结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0
com.example.viewt...	TAG	onTouch execute, action 1
com.example.viewt...	TAG	onClick execute

可以看到，onTouch是优先于onClick执行的，并且onTouch执行了两次，一次是ACTION_DOWN，一次是ACTION_UP(你还可能会有多次ACTION_MOVE的执行，如果你手抖了一下)。因此事件传递的顺序是先经过onTouch，再传递到onClick。

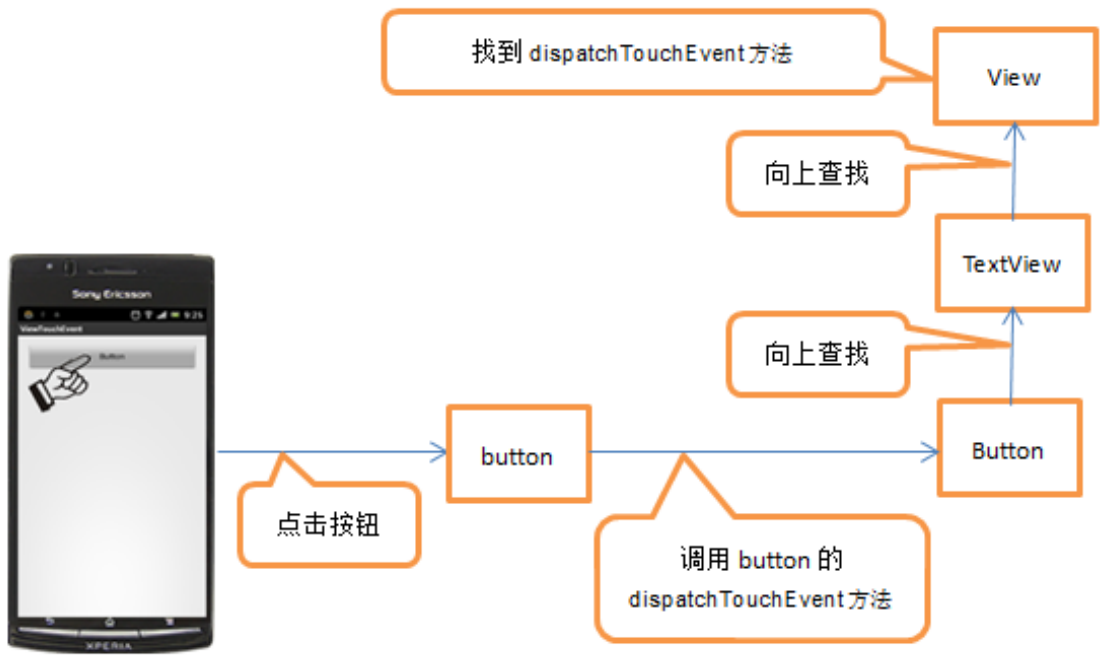
细心的朋友应该可以注意到，onTouch方法是有返回值的，这里我们返回的是false，如果我们尝试把onTouch方法里的返回值改成true，再运行一次，结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0
com.example.viewt...	TAG	onTouch execute, action 1

我们发现，onClick方法不再执行了！为什么会这样呢？你可以先理解成onTouch方法返回true就认为这个事件被onTouch消费掉了，因而不会再继续向下传递。

如果到现在为止，以上的所有知识点你都是清楚的，那么说明你对Android事件传递的基本用法应该是掌握了。不过别满足于现状，让我们从源码的角度分析一下，出现上述现象的原理是什么。

首先你需要知道一点，只要你触摸到了任何一个控件，就一定会调用该控件的dispatchTouchEvent方法。那当我们去点击按钮的时候，就会去调用Button类里的dispatchTouchEvent方法，可是你会发现Button类里并没有这个方法，那么就到它的父类TextView里去找一找，你会发现TextView里也没有这个方法，那没办法了，只好继续在TextView的父类View里找一找，这个时候你终于在View里找到了这个方法，示意图如下：



然后我们来看一下View中dispatchTouchEvent方法的源码：

```java
public boolean dispatchTouchEvent(MotionEvent event) {
    if (mOnTouchListener != null && (mViewFlags & ENABLED_MASK) == ENABLED &&
            mOnTouchListener.onTouch(this, event)) {
        return true;
    }
    return onTouchEvent(event);
}
```

这个方法非常的简洁，只有短短几行代码！我们可以看到，在这个方法内，首先是进行了一个判断，如果mOnTouchListener != null，(mViewFlags & ENABLED_MASK) == ENABLED和mOnTouchListener.onTouch(this, event)这三个条件都为真，就返回true，否则就去执行onTouchEvent(event)方法并返回。

先看一下第一个条件，mOnTouchListener这个变量是在哪里赋值的呢？我们寻找之后在View里发现了如下方法：

```java
public void setOnTouchListener(OnTouchListener l) {
    mOnTouchListener = l;
}
```

Bingo！找到了，mOnTouchListener正是在setOnTouchListener方法里赋值的，也就是说只要我们给控件注册了touch事件，mOnTouchListener就一定被赋值了。

第二个条件(mViewFlags & ENABLED_MASK) == ENABLED是判断当前点击的控件是否是enable的，按钮默认都是enable的，因此这个条件恒定为true。

第三个条件就比较关键了，mOnTouchListener.onTouch(this, event)，其实也就是去回调控件注册touch事件时的onTouch方法。也就是说如果我们在onTouch方法里返回true，就会让这三个条件全部成立，从而整个方法直接返回true。如果我们在onTouch方法里返回false，就会再去执行onTouchEvent(event)方法。

现在我们可以结合前面的例子来分析一下了，首先在dispatchTouchEvent中最先执行的就是onTouch方法，因此onTouch肯定是要优先于onClick执行的，也是印证了刚刚的打印结果。而如果在onTouch方法里返回了true，就会让dispatchTouchEvent方法直接返回true，不会再继续往下执行。而打印结果也证实了如果onTouch返回true，onClick就不会再执行了。

根据以上源码的分析，从原理上解释了我们前面例子的运行结果。而上面的分析还透漏出了一个重要的信息，那就是onClick的调用肯定是在onTouchEvent(event)方法中的！那我们马上来看下onTouchEvent的源码，如下所示：

```java
public boolean onTouchEvent(MotionEvent event) {
    final int viewFlags = mViewFlags;
```

```
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        // 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 (!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) {
                        mPrivateFlags |= PRESSED;
                        refreshDrawableState();
                        postDelayed(mUnsetPressedState,
                                ViewConfiguration.getPressedStateDuration());
                    } else if (!post(mUnsetPressedState)) {
                        // If the post failed, unpress right now
                        mUnsetPressedState.run();
                    }
                    removeTapCallback();
                }
                break;
            case MotionEvent.ACTION_DOWN:
                if (mPendingCheckForTap == null) {
                    mPendingCheckForTap = new CheckForTap();
```

```
60.                 }
61.                 mPrivateFlags |= PREPRESSED;
62.                 mHasPerformedLongPress = false;
63.                 postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
64.                 break;
65.             case MotionEvent.ACTION_CANCEL:
66.                 mPrivateFlags &= ~PRESSED;
67.                 refreshDrawableState();
68.                 removeTapCallback();
69.                 break;
70.             case MotionEvent.ACTION_MOVE:
71.                 final int x = (int) event.getX();
72.                 final int y = (int) event.getY();
73.                 // Be lenient about moving outside of buttons
74.                 int slop = mTouchSlop;
75.                 if ((x < 0 - slop) || (x >= getWidth() + slop) ||
76.                         (y < 0 - slop) || (y >= getHeight() + slop)) {
77.                     // Outside button
78.                     removeTapCallback();
79.                     if ((mPrivateFlags & PRESSED) != 0) {
80.                         // Remove any future long press/tap checks
81.                         removeLongPressCallback();
82.                         // Need to switch from pressed to not pressed
83.                         mPrivateFlags &= ~PRESSED;
84.                         refreshDrawableState();
85.                     }
86.                 }
87.                 break;
88.         }
89.         return true;
90.     }
91.     return false;
92. }
```

相较于刚才的dispatchTouchEvent方法，onTouchEvent方法复杂了很多，不过没关系，我们只挑重点看就可以了。

首先在第14行我们可以看出，如果该控件是可以点击的就会进入到第16行的switch判断中去，而如果当前的事件是抬起手指，则会进入到MotionEvent.ACTION_UP这个case当中。在经过种种判断之后，会执行到第38行的performClick()方法，那我们进入到这个方法里瞧一瞧：

```
[java]
01. public boolean performClick() {
02.     sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
03.     if (mOnClickListener != null) {
04.         playSoundEffect(SoundEffectConstants.CLICK);
05.         mOnClickListener.onClick(this);
06.         return true;
07.     }
08.     return false;
09. }
```

可以看到，只要mOnClickListener不是null，就会去调用它的onClick方法，那mOnClickListener又是在哪里赋值的呢？经过寻找后找到如下方法：

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

一切都是那么清楚了！当我们通过调用setOnClickListener方法来给控件注册一个点击事件时，就会给mOnClickListener赋值。然后每当控件被点击时，都会在performClick()方法里回调被点击控件的onClick方法。

这样View的整个事件分发的流程就让我们搞清楚了！不过别高兴的太早，现在还没结束，还有一个很重要的知识点需要说明，就是touch事件的层级传递。我们都知道如果给一个控件注册了touch事件，每次点击它的时候都会触发一系列的ACTION_DOWN，ACTION_MOVE，ACTION_UP等事件。这里需要注意，如果你在执行ACTION_DOWN的时候返回了false，后面一系列其它的action就不会再得到执行了。简单的说，就是当dispatchTouchEvent在进行事件分发的时候，只有前一个action返回true，才会触发后一个action。

说到这里，很多的朋友肯定要有巨大的疑问了。这不是在自相矛盾吗？前面的例子中，明明在onTouch事件里面返回了false，ACTION_DOWN和ACTION_UP不是都得到执行了吗？其实你只是被假象所迷惑了，让我们仔细分析一下，在前面的例子当中，我们到底返回的是什么。

参考着我们前面分析的源码，首先在onTouch事件里返回了false，就一定会进入到onTouchEvent方法中，然后我们来看一下onTouchEvent方法的细节。由于我们点击了按钮，就会进入到第14行这个if判断的内部，然后你会发现，不管当前的action是什么，最终都一定会走到第89行，返回一个true。

是不是有一种被欺骗的感觉？明明在onTouch事件里返回了false，系统还是在onTouchEvent方法中帮你返回了true。就因为这个原因，才使得前面的例子中ACTION_UP可以得到执行。

那我们可以换一个控件，将按钮替换成ImageView，然后给它也注册一个touch事件，并返回false。如下所示：

```java
imageView.setOnTouchListener(new OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        Log.d("TAG", "onTouch execute, action " + event.getAction());
        return false;
    }
});
```

运行一下程序，点击ImageView，你会发现结果如下：

Application	Tag	Text
com.example.viewt...	TAG	onTouch execute, action 0

在ACTION_DOWN执行完后，后面的一系列action都不会得到执行了。这又是为什么呢？因为ImageView和按钮不同，它是默认不可点击的，因此在onTouchEvent的第14行判断时无法进入到if的内部，直接跳到第91行返回了false，也就导致后面其它的action都无法执行了。

好了，关于View的事件分发，我想讲的东西全都在这里了。现在我们再来回顾一下开篇时提到的那三个问题，相信每个人都会有更深一层的理解。

1. onTouch和onTouchEvent有什么区别，又该如何使用？

从源码中可以看出，这两个方法都是在View的dispatchTouchEvent中调用的，onTouch优先于onTouchEvent

执行。如果在onTouch方法中通过返回true将事件消费掉，onTouchEvent将不会再执行。

另外需要注意的是，onTouch能够得到执行需要两个前提条件，第一mOnTouchListener的值不能为空，第二当前点击的控件必须是enable的。因此如果你有一个控件是非enable的，那么给它注册onTouch事件将永远得不到执行。对于这一类控件，如果我们想要监听它的touch事件，就必须通过在该控件中重写onTouchEvent方法来实现。

2．为什么给ListView引入了一个滑动菜单的功能，ListView就不能滚动了？

如果你阅读了**Android滑动框架完全解析，教你如何一分钟实现滑动菜单特效**这篇文章，你应该会知道滑动菜单的功能是通过给ListView注册了一个touch事件来实现的。如果你在onTouch方法里处理完了滑动逻辑后返回true，那么ListView本身的滚动事件就被屏蔽了，自然也就无法滑动(原理同前面例子中按钮不能点击)，因此解决办法就是在onTouch方法里返回false。

3．为什么图片轮播器里的图片使用Button而不用ImageView？

提这个问题的朋友是看过了**Android实现图片滚动控件，含页签功能，让你的应用像淘宝一样炫起来**这篇文章。当时我在图片轮播器里使用Button，主要就是因为Button是可点击的，而ImageView是不可点击的。如果想要使用ImageView，可以有两种改法。第一，在ImageView的onTouch方法里返回true，这样可以保证ACTION_DOWN之后的其它action都能得到执行，才能实现图片滚动的效果。第二，在布局文件里面给ImageView增加一个android:clickable="true"的属性，这样ImageView变成可点击的之后，即使在onTouch里返回了false，ACTION_DOWN之后的其它action也是可以得到执行的。

今天的讲解就到这里了，相信大家现在对Android事件分发机制又有了进一步的认识，在后面的文章中我会再带大家一起探究Android中ViewGroup的事件分发机制，感兴趣的朋友请继续阅读**Android事件分发机制完全解析，带你从源码的角度彻底理解(下)**。