可能是讲解 Android 事件分发最好的文章

2016-07-03 安卓应用频道

(点击上方公众号，可快速关注)

> 来源： milter
> 链接：http://www.jianshu.com/p/2be492c1df96

> 我几乎看过国内讲解Android事件分发的所有文章，但遗憾的是都没有这篇讲的好，原因有二：它阐明了具体的事件分发机制的设计意图，让人既知其然，又知其所以然；它没有贴源码，吓唬本宝宝。所以我决定将它翻译出来，造福广大Android开发者。原文请看这里：http://balpha.de/2013/07/android-development-what-i-wish-i-had-known-earlier/

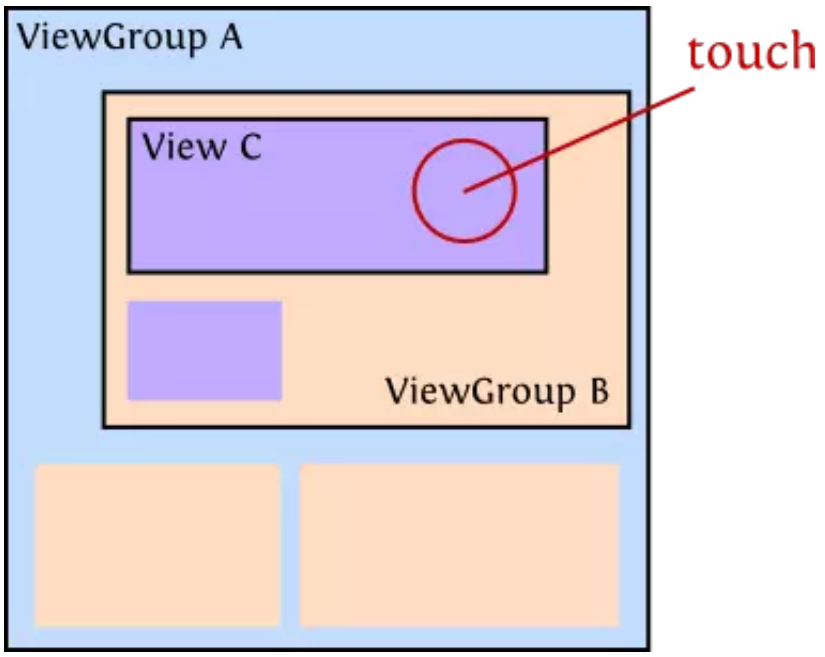
有时，你必须要自己处理触摸事件（touch events）而不能依赖于有可用的onSomethingListener。我就遇到过这样的时候，当时我很想有一篇文章能简单地解释触摸事件是怎样在视图层次（view hierarchy）中传播的，从而可以将之作为进一步深入学习的起点。这篇博客是我的一次尝试，它看起来有点长，但这是因为我是按照触摸事件的传播过程一步一步来写的。

一些假设

我们只考虑最重要的四个触摸事件，即：**DOWN,MOVE,UP和CANCEL。一个手势**（gesture）是一个事件列，以一个DOWN事件开始（当用户触摸屏幕时产生），后跟0个或多个MOVE事件（当用户四处移动手指时产生），最后跟一个单独的UP或CANCEL事件（当用户手指离开屏幕或者系统告诉你手势（gesture）由于其他原因结束时产生）。当我们说到“手势剩余部分”时指的是手势后续的MOVE事件和最后的UP或CANCEL事件。

在这里我也不考虑多点触摸手势（我们只假设用一个手指）并且忽略多个MOVE事件可以被归为一组这一实际情况。最后，我们假设文中的view都没有注册onTouchListener。

我们将要讨论的视图层次是这样的：最外层是一个ViewGroup A，包含一个或多个子view（children），其中一个子view是ViewGroup B，ViewGroupB中又包含一个或多个子view，其中一个子view是 View C,C不是一个ViewGroup。这里我们忽略同层级view之间可能的交叉叠加。



android-touch.png

假设用户首先触摸到的屏幕上的点是C上的某个点，该点被标记为触摸点（touch point），DOWN事件就在该点产生。然后用户移动手指并最后离开屏幕，此过程中手指是否离开C的区域无关紧要，关键是手势（gesture）是从哪里开始的。

默认情况

假设上面的A,B,C都没有覆写默认的事件传播行为，那么下面就是事件传播的过程：

- DOWN事件被传到C的onTouchEvent方法中，该方法返回false，表示“我不关心这个手势（gesture）”。
- 因此，DOWN事件被传到B的onTouchEvent方法中，该方法同样返回false，表示B也不关心这个手势。
- 同样，因为B不关心这个手势，DOWN事件被传到A的onTouchEvent方法中，该方法也返回false。

由于没有view关心这个手势（gesture），它们将不再会从“手势剩余部分”中接收任何事件。

处理事件

现在，让我们假设C实际上是关心这个手势（gesture）的，原因可能是C被设置成可点击的（clickable）或者你覆写了C的onTouchEvent方法。

- DOWN事件被传递给C的onTouchEvent方法，该方法可以做任何它想做的事情，最后返回true。
- 因为C说它正在处理这个手势（gesture），则DOWN事件将不再被传递给B和A的onTouchEvent方法。
- 因为C说它正在处理这个手势（gesture），所以“手势剩余部分”的事件也将传递给C的onTouchEvent方法，此时该方法返回true或false都无关紧要了，但是为保持一致最好还是返回true。

个人理解：从这里可以看出，各个View的onTouchEvent方法对DOWN事件的处理，代表了该View对以此DOWN开始的整个手势（gesture）的处理意愿，返回true代表愿意处理该gesture，返回false代表不愿意处理该gesture。

onInterceptTouchEvent

现在我们将讨论一个新的方法：onInterceptTouchEvent，它只存在于ViewGroup中，普通的View中没有这个方法。在任何一个view的onTouchEvent被调用之前，它的父辈们（ancestors）将先获得拦截这个事件的一次机会，换句话说，它们可以窃取该事件。在刚才的“处理事件”部分中，我们遗漏了这一过程，现在，让我们把它加上：

- DOWN事件被传给A的onInterceptTouchEvent，该方法返回false，表示它不想拦截。
- DOWN又被传递给B的onInterceptTouchEvent，它也不想拦截，因此该方法也返回false。
- 现在，DOWN事件被传递到C的onTouchEvent方法，该方法返回true，因为它想处理以该事件为首的手势（gesture）。
- 现在，该手势的下一个事件MOVE到来了。这个MOVE事件再一次被传递给A的onInterceptTouchEvent方法，该方法再一次返回false，B也同样如此。
- 然后，MOVE事件被传递给C的onTouchEvent，就像在前一部分中一样。
- “手势剩余部分”中其他事件的处理过程和上面一样，假如A和B的onInterceptTouchEvent方法继续返回false的话。

这里有两点需要注意：

- 虽然ViewGroup A和B的onInterceptTouchEvent方法对DOWN事件返回了false，后续的事件依然会传递给它们的onInterceptTouchEvent方法，这一点与onTouchEvent的行为是不一样的。
- 假如DOWN事件传给C的onTouchEvent方法时，它返回了false，DOWN事件会继续

向上传递给B和A的onTouchEvent，即使它们在onInterceptTouchEvent方法中说它们不想拦截这个DOWN事件，但没办法，没有子View愿意处理该事件。

个人理解：感谢@编程世界的孩子 的提醒，由此可见，DOWN事件的处理实际上经历了一下一上两个过程，下是指A->B的onInterceptTouchEvent，上是指C->B->A的onTouchEvent，当然，任意一步的方法中返回true,都能阻止它继续传播。

拦截事件

现在，让我们更进一步，假设B没有拦截DOWN事件，但它拦截了接下来的MOVE事件。原因可能是B是一个scrolling view。当用户仅仅在它的区域内点击（tap）时，被点击到的元素应当能处理该点击事件。但是当用户手指移动了一定的距离后，就不能再视该手势（gesture）为点击了——很明显，用户是想scroll。这就是为什么B要接管该手势（gesture）。
下面是事件被处理的顺序：

- DOWN事件被依次传到A和B的onInterceptTouchEvent方法中，它们都返回的false，因为它们目前还不想拦截。
- DOWN事件传递到C的onTouchEvent方法，返回了true。
- 在后续到来MOVE事件时，A的onInterceptTouchEvent方法仍然返回false。
- B的onInterceptTouchEvent方法收到了该MOVE事件，此时B注意到用户手指移动距离已经超过了一定的threshold（或者称为slop）。因此，B的onInterceptTouchEvent方法决定返回true，从而接管该手势（gesture）后续的处理。
- 然后，这个MOVE事件将会被系统变成一个CANCEL事件，这个CANCEL事件将会传递给C的onTouchEvent方法。
- 现在，又来了一个MOVE事件，它被传递给A的onInterceptTouchEvent方法，A还是不关心该事件，因此onInterceptTouchEvent方法继续返回false。
- 此时，该MOVE事件将不会再传递给B的onInterceptTouchEvent方法，该方法一旦返回一次true，就再也不会被调用了。事实上，该MOVE以及“手势剩余部分”都将传递给B的onTouchEvent方法（除非A决定拦截“手势剩余部分”）。
- C再也不会收到该手势（gesture）产生的任何事件了。

下面的一些小事情可能会令你感到吃惊：

- 如果一个ViewGroup拦截了最初的DOWN事件，该事件仍然会传递到该ViewGroup的onTouchEvent方法中。
- 另一方面，如果ViewGroup拦截了一个半路的事件（比如，MOVE），这个事件将会

被系统变成一个CANCEL事件，并传递给之前处理该手势（gesture）的子View，而且不会再传递（无论是被拦截的MOVE还是系统生成的CANCEL）给ViewGroup的onTouchEvent方法。只有再到来的事件才会传递到ViewGroup的onTouchEvent方法中。

从此开始，你可以更进一步。比如对mouthful-method （实在不知道该怎么翻译啦！）requestDisallowInterceptTouchEvent,C可以用该方法阻止B窃取事件。如果你想更加疯狂一点，你可以在你自己的ViewGroup中直接覆写dispatchTouchEvent方法，并对传递进来的事件做任何你想做的处理。但这样的话你可能会破坏一些约定，所以应当小心。

好了，文章翻译完了，不知道你是否和我一样，读完此文消解了许多的困惑，如果是的话，点zan吧！
