Android事件分发机制总结抽象--钓钩模型

用户体验设计师巧妙地利用有限的手机屏幕空间，完美设计出极简实用的交互功能，如果多问一句软件工程师 “怎么做到的” ？  
答案必须是从事件分发机制的高超运用说起。

在6年的Android应用开发职业生涯中，我接触到最多的也正是如何运用事件分发机制和自定义控件，堆砌出一幅幅可交互的精致业务功能画面。下图是我分别在手机百度和美团App上研发的列表拖动排序和卡片抽屉效果代表作。



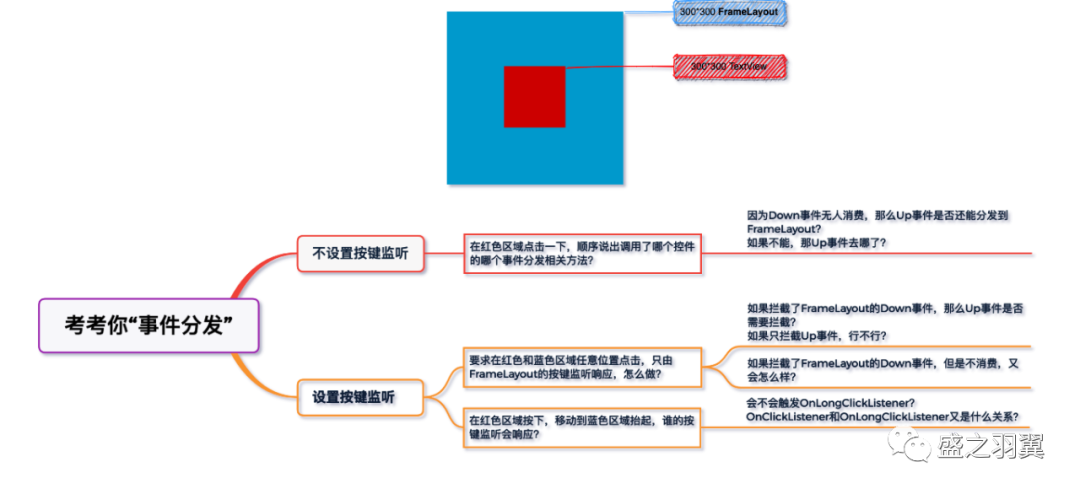
2018年在我编码技战术水平的小巅峰期，首次将MECE（Mutually Exclusive Collectively Exhaustive，相互独立，完全穷尽）分析法用于专业技术原理剖析，“**正面硬刚**” 事件分发机制写下 [Android事件分发-来龙去脉](http://mp.weixin.qq.com/s?__biz=MzU5ODUzODQ2Ng==&mid=2247483666&idx=1&sn=59b5470f4c47ab5bfef9fdf55e457d15&chksm=fe43e09fc93469890b306ed1aa6a5736ecb789c54117a6fc81f0111f41a8402cf79117a300f8&scene=21" \l "wechat_redirect" \t "/Users/shengshuqiang/Documents\\x/_blank)，此后一度自诩事件分发 “不敢说精通”，自我膨胀了一段时间（程序猿的快乐就是这么简单）。

须知道，一山更比一山高。今年力行 “书上学” 苦练基本功，认真学习了玉刚哥的《Android开发艺术探索》，书中的几个问题 “**侧面迂回**” 暴击了我掌握的事件分发机制不过尔尔。  
猛然意识到 “问题驱动理解” 这种学习方式简单有效，我也来试试。

考考你

▼

提问，谁不会呢？好像张嘴就能来，但我们需要的是能检验出水平且有区分度的那种。这就不禁让我想到了工作中令人难忘的事--写线上故障 CaseStudy ，相信亲身经历过的小伙伴一定忘不了直击灵魂深处的 “**5 Whys**”（针对问题的原因层层递进问5个为什么，差不多也就从事物的表象深入到了本质）。  
当然，我们没必要老是跟自己过不去。差不多问3个就行了。问题呢，也不能太啰嗦，大道至简，最好能从最 “简单” 的问题来接近事物的本质。我尝试构造一个简易场景来推演**三个大问题**和**几个小问题**，帮助自己理解精进事件分发机制页面中有一个300\*300的蓝色背景FrameLayout，正中有一个100\*100的红色背景TextView。  
如下图所示：▼



接下来的问题只需要围绕FrameLayout和TextView两个控件的顺序说出事件分发相关方法调用即可。因为场景固定，不存在如果，即答案对应的是唯一路径，不存在如果...就...为了便于理解，在回答上述问题前，我先介绍一下事件分发机制的核心方法以及对应的功能：**dispatchTouchEvent：**控件事件分发主体逻辑，View中的该方法用于调用OnTouchListener.onTouch和onTouchEvent，ViewGroup中该方法用于判断是否拦截，不拦截则遍历子控件分发。

**onInterceptTouchEvent：**是否拦截事件。若拦截事件，则事件不会分发给子控件，而是直接给自己消费。

**onTouchEvent：**消费事件主体逻辑，用于处理按键状态、OnClickListener.onClick和OnLongClickListener.onLongClick。

不设按键监听点击分发

▼

1. 不设置按键监听，在红色区域点击一下，顺序说出调用了哪个控件的哪个事件分发相关方法？

这个问题比较简单，无需赘述，答案如下：

① 调用FrameLayout的dispatchTouchEvent，即对应ViewGroup中的dispatchTouchEvent方法。  
    ② 调用FrameLayout的onInterceptTouchEvent。因为没有重写事件拦截，所以返回默认false。  
       ③ 调用TextView的dispatchTouchEvent，即对应View中的dispatchTouchEvent方法。  
             ④ 调用TextView的onTouchEvent。因为onInterceptTouchEvent只有ViewGroup有，TextView不是ViewGroup，也就不存在事件拦截方法。因为未设置相关按键监听消费事件，所以返回默认false。  
        ⑤ 调用FrameLayout的super.dispatchTouchEvent，即对应View中的dispatchTouchEvent方法。因为子控件TextView没有消费事件，转由FrameLayout尝试消费事件。  
            ⑥ 调用FrameLayout的onTouchEvent。因为未设置相关按键监听消费事件，所以返回默认false。

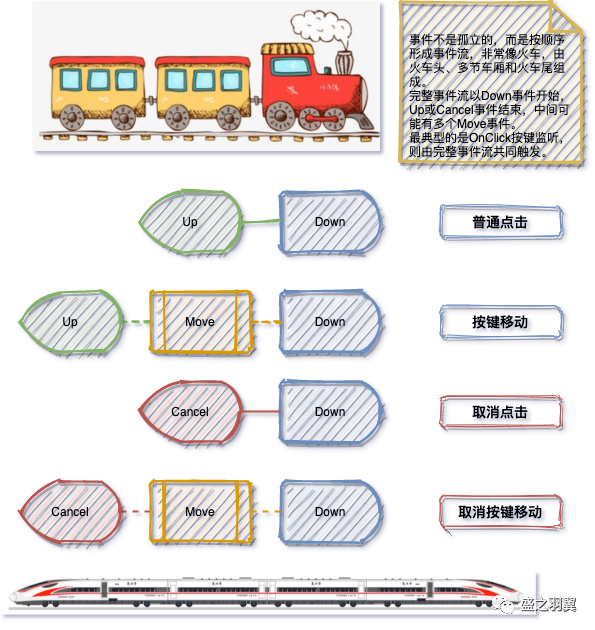
相信这个问题难不倒大部分同学。但是，问题结束了吗？  
众所周知，普通点击事件包含DOWN事件和UP事件，上面说的只是DOWN事件，UP事件呢？

1.1 因为DWON事件无人消费，那么UP事件是否还能分发到FrameLayout？  
如果不能，那UP事件去哪了？

这个问题其实我刚开始自问自答时，也没有回答上来。

在回答这个问题前，有必要科普一下Android开发者文档中描述的**事件流一致性保证**(Consistency Guarantees)：

按下开始，中间可能伴随着移动，直到松开或者取消结束。  
DOWN -> MOVE(\*) -> UP/CANCEL。

简单来说，一条事件流就像一辆火车，车头和车尾是必须要有的，中间的车厢可有可无，有的话可以是任意节。DOWN事件相当于火车头，UP或CANCEL相当于火车尾，MOVE事件相当于火车厢。我们所熟悉的OnClick按键监听就是由完整事件流共同决定是否触发响应。  
事件流火车模型如下图所示：▼

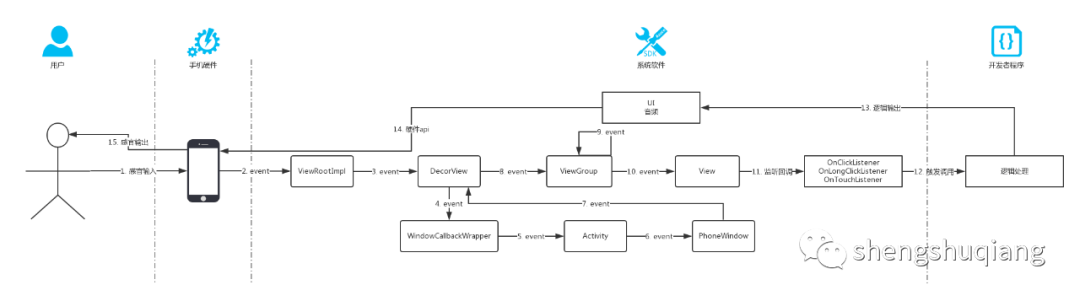
如果控件及其子孙控件都没有消费DOWN事件，则该控件不会收到接下来的事件流。  
《Android开发艺术探索》中的比喻十分生动形象：领导给你安排一件事，如果你中间掉链子，那就没有然后了，因为机会只有一次。

按此逻辑，DOWN事件没有消费，那应该是不会收到UP事件了。如果是这样，那么问题来了，UP事件去哪了？毕竟没有控件消费UP事件。

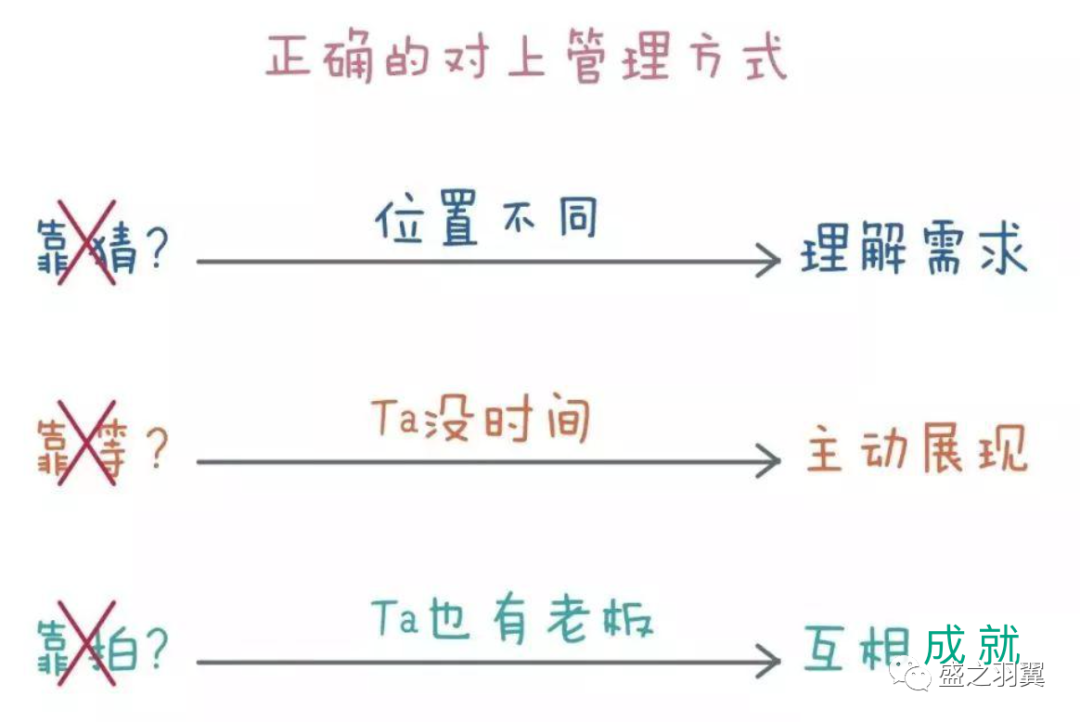
凭直觉，可能是给Activity消费了，通过自定义重写Activity的dispatchTouchEvent和onTouchEvent，FrameLayout的dispatchTouchEvent、onInterceptTouchEvent和onTouchEvent，FrameLayout的dispatchTouchEvent和onTouchEvent，加上日志，点击一下。

答案一目了然：UP事件会继续调用Activity的dispatchTouchEvent和onTouchEvent，但不会再调用FrameLayout和TextView。

阅读过源码的同学大概知道，Activity并没有事件分发逻辑，兜兜转转最终调用的还是DecorView的事件分发，而DecorView是继承自ViewGroup，也就是事件分发主体逻辑还是由ViewGroup和View完成的。  
按键响应调用如下图所示：▼



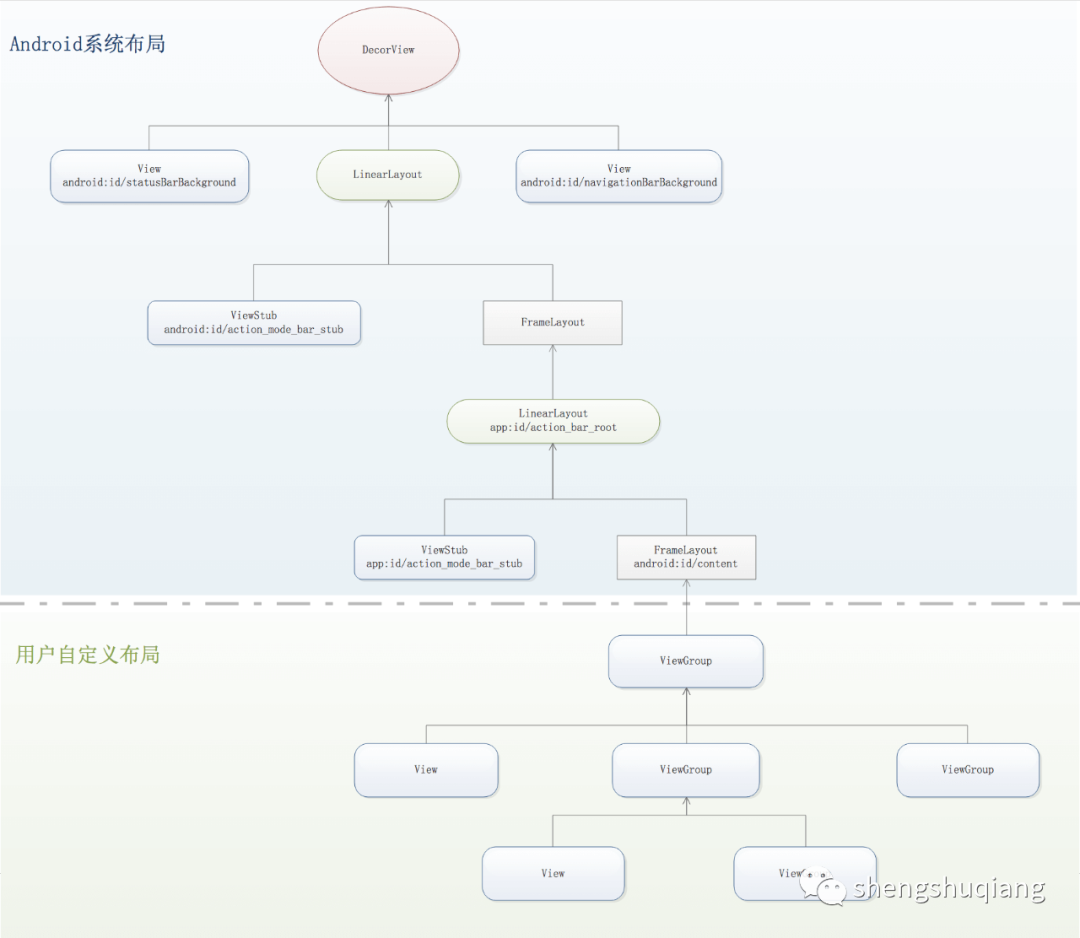
所以，事件大概率被DecorView消费了。如果继续靠猜，那效率就有点低了。最直接最有效的方式就是Debug源码。



在build.gradle中将compileSdkVersion和targetSdkVersion指定成和Android模拟器一样的版本，并且在Debug调试时下载对应源码。

接下来，只是时间问题。

多说一句，千万别在ViewGroup或View中直接断点，这么做会很容易让你内心崩溃...  
因为所有控件都会继承View（包括ViewGroup），而你在Activity中的setContentView并不是View树的全部，像状态栏、导航栏等都属于页面内容的一部分，而这些，系统帮你做了。  
页面布局如下图所示：▼

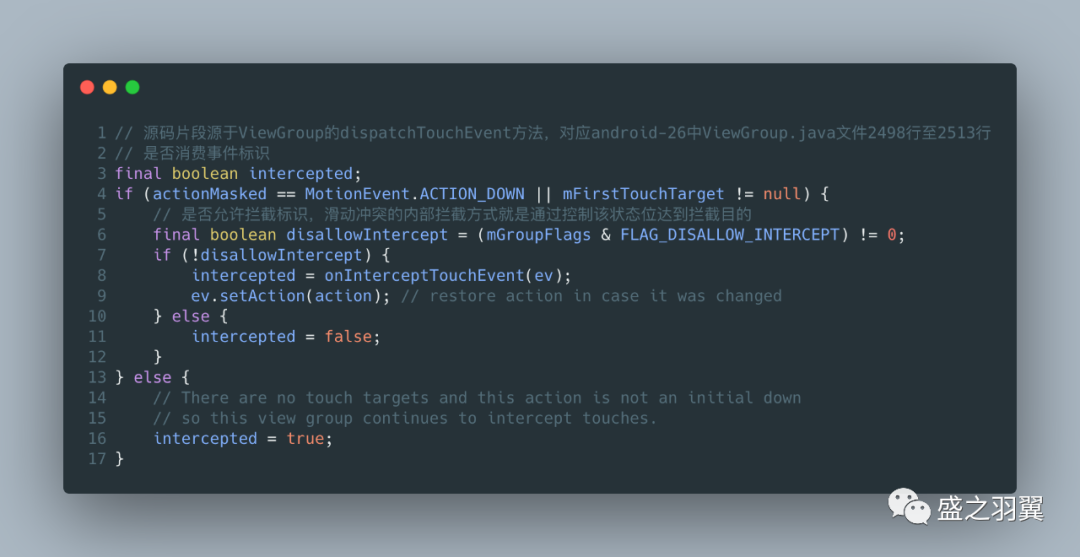


科学的操作是先通过日志摸清情况，找到规律，然后控制局面，有的放矢，通过自定义控件重写相关方法，在自定义控件中打断点，断住后单点跟进，精准查看逻辑。

细节建议读者实操一遍，我直接说结果了：

①  **DOWN事件**：TextView和FrameLayout未消费DOWN事件，会继续向上回传到DecorView，调用DecorView的onTouchEvent。  
但DecorView也不消费，继续传给Activity，调用Activity的onTouchEvent，Activity返回false。  
简而言之，DOWN事件会陆续调用到DecorView和Activity，始终没有被消费。  
②  **UP事件**：Activity的dispatchTouchEvent先调用到，接着调用DecorView的dispatchTouchEvent。  
因为mFirstTouchTarget为null，不会调用onInterceptTouchEvent，但会设置intercepted状态位为true。逻辑见下述ViewGroup中dispatchTouchEvent源码片段，执行逻辑为第4行和16行。  
接着调用DecorView的onTouchEvent，显然，DecorView也不消费，继续传给Activity，调用Activity的onTouchEvent，Activity返回false。  
简而言之，UP事件也不会被消费，而且只会调用DecorView和Activity的事件分发相关方法，其他控件将无法收到事件分发调用。

ViewGroup中dispatchTouchEvent逻辑源码片段如下图所示：▼



这个问题看似简单，但实际能回答上来的才是真的高手。



但可能有同学会问，不设置按键监听情况下，没啥实际意义，大部分人不会关心这种情况，换一题。



设按键监听&拦截点击分发

▼

2. FrameLayout和TextView均设置按键监听，要求在红色和蓝色区域任意位置点击，只由FrameLayout的按键监听响应，怎么做？

这个简单，我来！

重写FrameLayout的onInterceptTouchEvent方法返回true。

答案没毛病。但小问题接踵而至，DOWN事件和UP事件可能都会触发调用onInterceptTouchEvent，上面的答案不区分DOWN还是UP，简单粗暴的返回了true。DOWN事件一定要返回true吗？返回false行不行？UP事件呢，需不需要返回true？

2.1 只能拦截DOWN事件吗？拦截DWON事件后，UP事件需不需要返回true？

这里有必要先科普一下按键监听OnClickListener的小知识点：

OnClick是由UP事件的onTouchEvent触发调用的，但是触发的前提条件是已标记PFLAG\_PRESSED按下状态位，而标记操作恰恰是在DOWN事件中做的。这也就解释了事件流的连续性。MOVE事件呢？这是第三题，这里先按下不表。

基于上述理论，DOWN事件是一定要拦截的。但是UP事件，想必有部分同学开始模棱两可了，返回true肯定对，返回false好像也对...



从常识判断，如果一个返回布尔值的纯函数，调用后返回ture和返回false效果一样，那这个调用肯定是冗余的。onInterceptTouchEvent基本可以看成是这种纯函数。  
基于对Android Framework工程师的基本尊重，犯这种低级错误没有道理。  
那么结论只能是：onInterceptTouchEvent在DOWN事件返回true，那么后续UP事件根本不会再调用onInterceptTouchEvent。

换个角度看，如果onInterceptTouchEvent在DOWN事件返回true，意味着本控件将拦截处理后续的事件流，后续事件调用自然也就用不着再问要不要拦截。

事实也是如此。  
①  onInterceptTouchEvent只能拦截DOWN事件，否则FrameLayout的按键监听不会响应。  
②  无需处理UP事件，因为DOWN事件拦截后，后续事件流根本不会再调用onInterceptTouchEvent。因为FrameLayout直接在DOWN事件就拦截了，TextView没有机会消费事件，不会有什么问题。但如果我们继续向前走一步，进一步窥探事件分发机制。

2.2 不拦截DWON事件只拦截UP事件，UP事件到底由谁消费？

按理说，DOWN事件由TextView消费，UP事件被FrameLayout拦截，当然是FrameLayout消费。但如果是这样，TextView怎么办，考虑过被拦截的子控件的感受了吗？好比领导给了机会，我也兢兢业业的投入工作，然后就戛然而止...让不让干好歹给个痛快话，我还在干杵着呢...显然，拦截的控件满意了，但被拦截的控件也不能不管，成熟的事件分发机制必须能妥善解决这些 “民事纠纷”。这就涉及到了一个高级知识点了--CANCEL事件。  
这年头，不知道CANCEL事件的都不好意思说自己精通事件分发（反正我不敢说精通）。



当ViewGroup的子类重写onInterceptTouchEvent返回true拦截事件后，如果存在被拦截的子控件（该事件流的头部事件已被子控件消费），子控件将会收到一个CANCEL事件被告知事件流到此为止。以上是事件拦截的大致逻辑，但是细心的同学会发现，上面只回答了CANCEL事件到哪去，那它是从哪来的呢？被拦截的那个事件，又是谁消费的？



相信这个问题难不倒深入阅读分析事件分发源码的同学，答案如下：

①  被拦截的事件会被转换为CANCEL事件，即event.setAction(MotionEvent.ACTION\_CANCEL)，会传递给被拦截的子控件告知事件流取消，View中的onTouchEvent会消费CANCEL事件返回true。  
②  此后的事件流，将调用拦截控件的dispatchTouchEvent和onTouchEvent。

其实这里面还有一个问题...

2.3 如果拦截了FrameLayout的DOWN事件，但是不消费，又会怎么样？

再这么推演下去，没完没了了，换一题。

设按键监听&按键移动分发

▼

3. 都设置按键监听，在红色区域按下，移动到蓝色区域抬起，谁的按键监听会响应？

这个问题，好像还真没想过...



基于上述按键逻辑，DOWN事件由TextView消费没有争议，关键问题就是第一个不在红色区域但在蓝色区域的Move事件怎么处理，以及最终的UP事件到底是谁消费？

太伤头发了...

分享个生活小妙招放松一下：当我们在按下按钮那一刻，后悔了怎么办？  
我的做法是，手按着不放，慢慢移动到按钮以外区域，然后再小心抬起，如愿以偿的没有触发点击操作（终于在付款的最后一刻冷静了下来，机智）。

基于这个常识，上面问题的答案是FrameLayout和TextView的监听事件均不会调到。  
突然想到我爸问过我一个问题：公山羊和母山羊谁有胡子？  
我当然没有观察过山羊的胡子，不过问题既然这么问，答案必须是反常识的。  
母山羊有胡子，我得意地大声回答。  
这时，我爸哈哈大笑，都有胡子...

言归正传，为什么监听事件都不会调用到？

答案都在源码里，我直接公布了：

① DOWN和红色区域内的Move事件都由TextView消费。  
第一个在蓝色区域的Move事件以及之后的Move事件和UP事件依旧还是TextView消费（没想到吧）。  
② 如果整个事件流都是TextView消费，那么为什么没有响应OnClick？问题的关键在于Move事件会根据当前坐标是否在控件内来判断是否取消PFLAG\_PRESSED按下状态位。第一个蓝色区域的Move事件会将按下状态位标记为未按下（不用机灵地以为移出去再移回来可以响应，没有机会了，Move只能取消按下状态，只有DOWN才能标记按下状态）。UP事件时会检查按下状态位，只有按下情况才会触发OnClick。  
③ 过程中不会有CANCEL事件，这是一部分同学对CANCEL事件的误解。  
④ CANCEL事件产生**两个前提**条件：子控件已经消费了DOWN事件，但父控件拦截了之后的事件。

可能好奇的同学内心还犯嘀咕，会不会有OnLongClickListener？

3.1 会不会触发OnLongClickListener？OnClickListener和OnLongClickListener又是什么关系?

这个问题问得好！答案我也直接说了：

① 和OnClickListener在UP事件触发不同的是，OnLongClickListener在DOWN事件触发，不过不是立即执行，而是延时执行，默认500ms。  
② OnClickListener和OnLongClickListener最多只有一个会执行。  
Move事件除了会根据当前坐标是否在控件内来判断是否取消按下状态位，也会来判断是否移除延迟执行OnLongClick。  
UP事件在触发OnClick前，会检查是否已经执行过OnLongClick逻辑（注意，是实际执行，不是触发延迟），  
如果执行过OnLongClick监听，则不会触发OnClick，  
如果没有执行过OnLongClick监听，会先移除延迟执行OnLongClick再触发OnClick。  
拦截产生的CANCEL也会移除延迟执行OnLongClick。总结

▼

受《Android开发艺术探索》的启发，尝试使用简明扼要的伪代码来总结回顾一下事件分发机制。



事件分发的难点在于一连串的事件流，把单点的独立问题变成了多点的连续问题，而且所有控件都走这套逻辑，目不暇接难免稀里哗啦稀里糊涂。

用个段子总结一下吧：

来了个项目，领导优先“分发”下去，问你接不接？  
当然，没有人能强迫你，你可以不接（这对应事件分发不消费场景）。那后果就是，没有然后了，你不干有的是人干，机会只有一次。  
所以你信心满满地对领导说，我好好干（这对应事件分发消费场景）。  
然后这个项目的人力物力财力都会源源不断（一个项目对应一个完整事件流）给到你，大家都开心。  
过了一段时间，领导发现项目不及预期，找你来了场触及灵魂的沟通。  
最后领导和你说，现在我来负责这个项目（这对应事件拦截），你好好休息一段时间（这是你收到的CANCEL事件）。  
后续的资源不断调拨给领导（对应拦截后的事件流改道），领导也没得选，只能自己加班加点干（拦截事件流后要对事件流负责到底，不论你干不干，这就是“项目闭环”）。  
公司管这叫“补位”。“分发”和“补位”是领导的基本素质。