内容概述

- **MotionEvent** 概述
- Android **如何传递**触控事件?
- Android **如何处理**触控事件?
- 具代表性的**例子**说明
- **拓展 -** 注意事项、多点触控、Batching、Hover Events、Touch Delegate
- **干货**推荐

MotionEvent

Android 会把每一个触控事件包装成 MotionEvent，其中携带了许多自身事件带来的或额外的信息，包括 Touch 位置、时间、历史记录以及触控点数（屏幕上的手指数）等。

其中，事件类型包含：

- ACTION_DOWN
- ACTION_UP
- ACTION_MOVE
- ACTION_POINTER_DOWN（屏幕上已有触控点，再按下其他触控点）
- ACTION_POINTER_UP（屏幕上当前有多个触控点，松开非最后一个点）
- ACTION_CANCEL
  **
  注意：**每一个触控事件都是：ACTION_DOWN 开始，ACTION_UP 结束。

Android 如何传递触控事件?

触控事件会最先发送给前台顶端的 Activity ，最先获取触控事件的，是其中的 dispatchTouchEvent() 方法。

事件通过上层 View 或 ViewGroup，调用各自的 dispatchTouchEvent() 方法，一直往下(子View)传递。而事件回传时，View 或者 ViewGroup 则会调用 onTouchEvent()方法。

小坑：既然 Activity 是触控事件传递的第一站，那么如果在 Activity 中的 onTouchEvent() 进行操作，那么会不会马上就可以检测到触控事件呢？然而，这样是**错误的**，实际上，Activity 中的 onTouchEvent() 方法是触控事件的终点，而如果传递过程中有 View 已经消耗了该事件，那么 Activity 中的 onTouchEvent() 方法则不会调用。

处理触控事件的另外一种方式 - **onTouchListener**：与 onTouchEvent() 很相似，可以 return true 来消费触控事件；注意： onTouchListener 优先于 onTouchEvent() 进行消费

Android 如何处理触控事件?

接收到触控事件后，Activity 自身 dispatchTouchEvent() 首先会传递给 RootView（大多数情况是 ViewGroup）,而 ViewGroup 会继续传递给其中的 childView，childView 又会继续传递给 childView 的 childView，如果整个过程都没有 View 消费触控事件，那么 Activity 的 onTouchEvent() 会是整个触控事件的终点

View.dispatchTouchEvent() 方法

检查是否含有 onTouchListener ,如果有，则优先给 listener 处理；如果没有 onTouchListener 或者 listener 中没有对触控事件进行消费（没有 return true）,那么事件就会返回视图树的上层，给父控件处理。

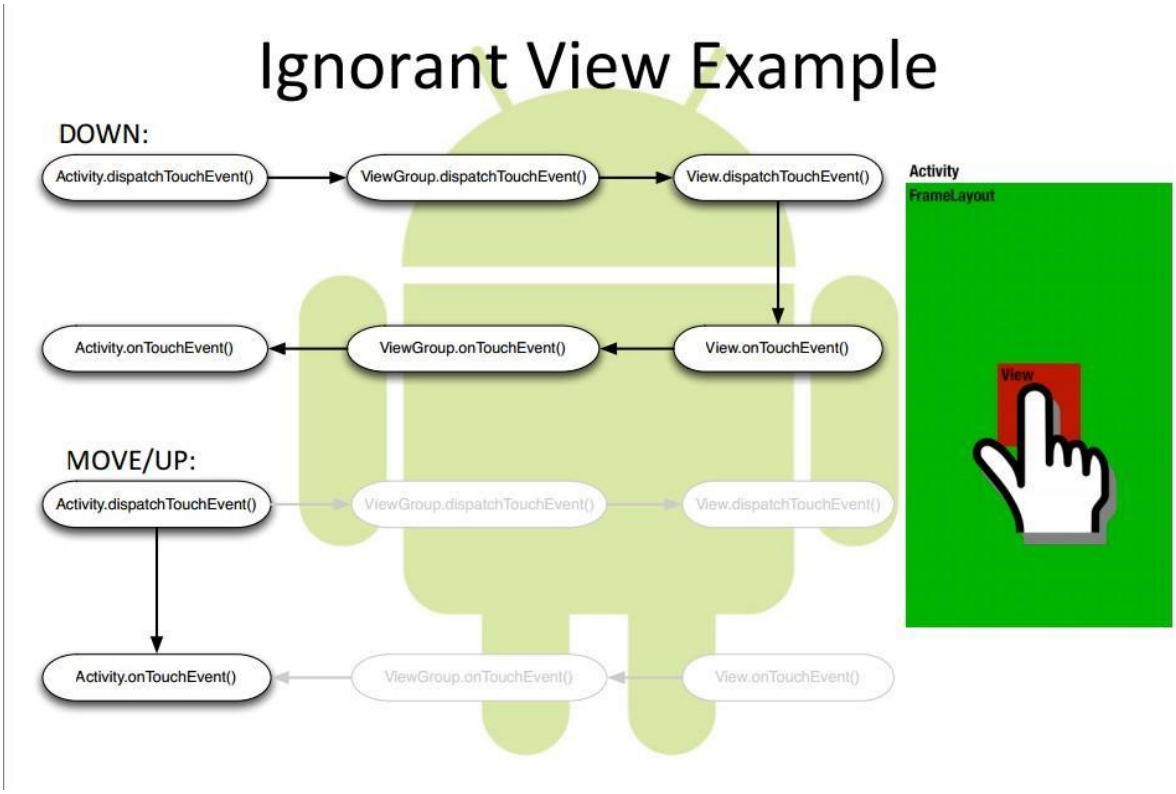
ViewGroup.dispatchTouchEvent() 方法

对比 View ， 更加复杂。

- 对 childView 进行管理，处理事件时，需要对自身 childView 进行遍历，来找出可能对触控事件感兴趣的 view**。实际上，**GroupView 会根据触控位置来判断，如果触控位置在 childView 内部，说明触控事件与其相关；如果相关的 View 不止一个（即：有重叠），那么 ViewGroup 会逆序（加入 ViewGroup 的顺序的逆序，也就是从表层 View 到底层 View ）遍历 childView，其中的处理过程与普通的事件传递相同
  。
- 另外，ViewGroup 还会对触控事件进行中断或者窃取子 View 的触控事件，可以通过以下方法来实现： **onInterceptTouchEvent(boolean flag) 方法**来实现，该方法不断监测流入此 ViewGroup 的触控事件，监测目的在于：**在某事件，结合对应手势，使 ViewGroup 停止事件分发（转而直接交给自己来处理事件）**，例子： ScrollView（内部有一个 Button），当手指放到 Button 上，然后进行上下滑动，会发现手势交给了 ScrollView，并且开始滚动（ScrollView 优先于 Button 的响应）
- 然而，ViewGroup 又有另外一个方法来打断 onInterceptTouchEvent() 方法的逻辑（childView 需要知道 parentView 是暂时屏蔽或是永久屏蔽触控事件），该方法便是 **“requestDisallowTouchIntercept()”**,它可以中断 onInterceptTouchEvent() 方法，例子：ScrollView 内部有 childView 想要处理拖拽事件而不是交给ScrollView 处理；**注意：**需要针对特定手势来设置标识，而且对当前触控事件有效（ACTION_UP 意味着当前触控事件结束，下一次 ACTION_DOWN 会重置该标识，如果仍需要拦截，需要再次调用 requestDisallowTouchIntercept()方法，也就是说，需要针对某个手势，每次都调用该方法去设置中断标识）。

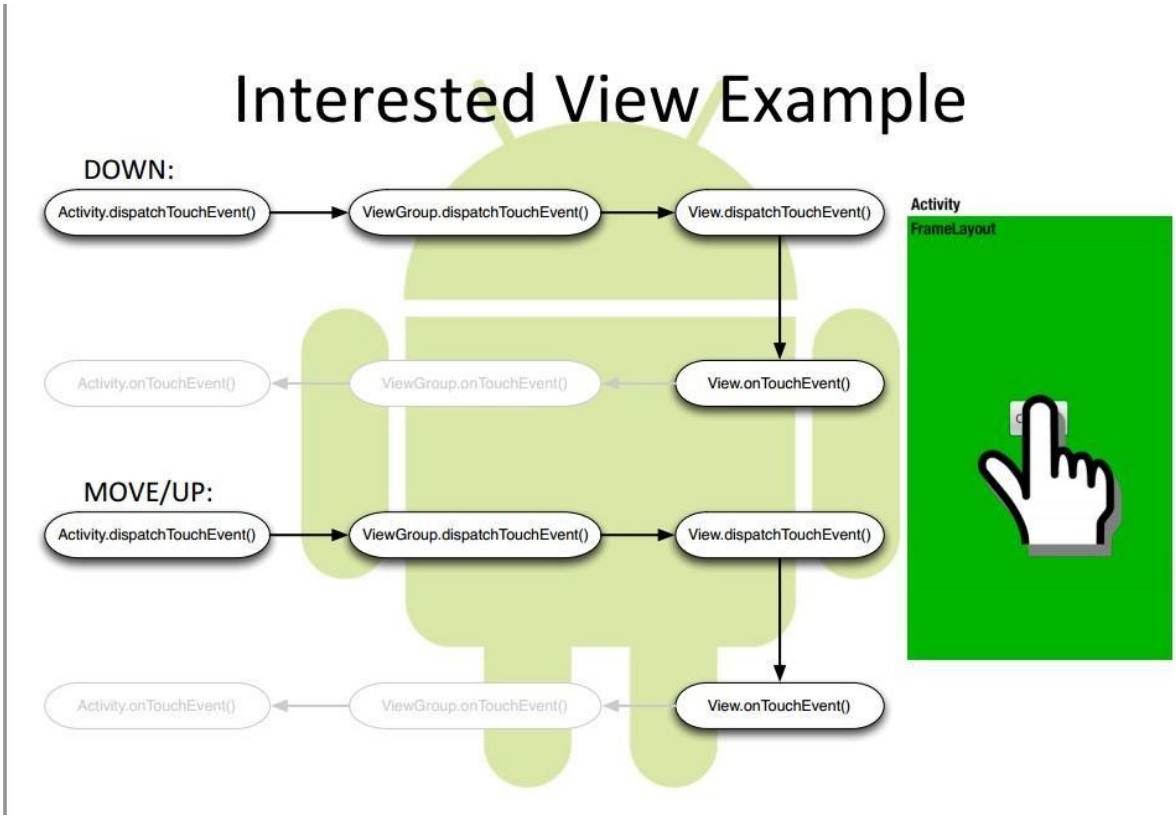
流程演示

一个 Activity 中有一个 FrameLayout(即 ViewGroup),其中又有一个 View，他们都不可点击（不会对触控事件进行任何处理，从而演示整个触控事件的传递过程）



- ACTION_DOWN：从上往下进行触控事件分发，再从下往上判断触控事件处理方式（是否拦截）
- ACTION_MOVE/ACTION_UP：框架会认为，既然之前已经知道没有 View 对 ACTION_DOWN 感兴趣了，那么 ACTION_MOVE、ACTION_UP 也就不需要再分发了（直接在 Actvity 转一圈）。

如果，将中间的 View 换成对触控事件有响应的 Button

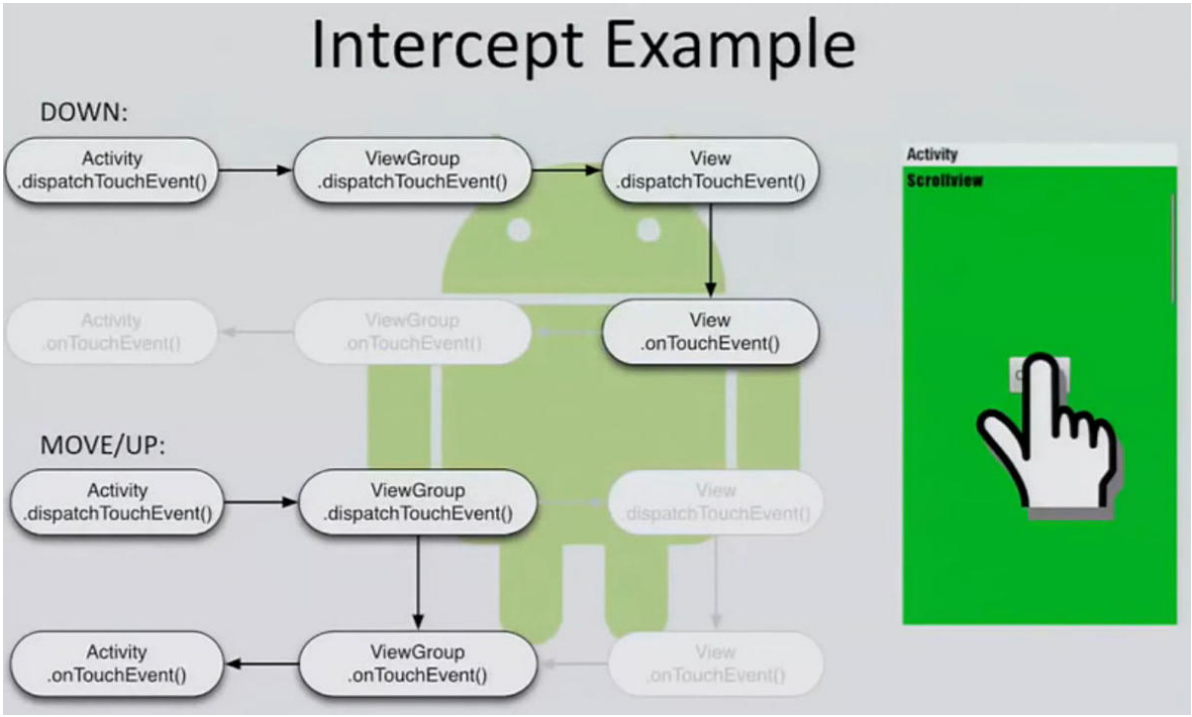


- ACTION_DOWN：依旧是同样的触控事件分发过程，但由于 Button 拦截了触控事件，即 Button 消费了本次触控事件，因此，本次触控事件在 Button 处结束
- ACTION_MOVE/ACTION_UP：所有的移动触控事件，从 MOVE 到UP，或者再放几根手指在上面（Button 上面），触控事件会走同样的流程，最终在 Button 上结束（当然，父控件的

onInterceptTouchEvent() 还是能监控事件的 ）

- 照应上面的**小坑**：Activity 中的 onTouchEvent()是分发的终点，如果中途被消费，则就不会调用了

如果，再把 FrameLayout 换成可以滚动的 ScrollView



ACTION_DOWN：触控事件同样到达 Button 处被消费
ACTION_MOVW/ACTION_UP：按住 Button，然后手指开始在屏幕上滑动，然后松开手指;ScrollView 检测到滑动事件，事件分发线路发生了改变（本身 ViewGroup 的 dispatchTouchEvent() 方法会把事件分发给内部的 childView，但 ScrollView 的 onInterceptTouchEvent() 方法返回 true，则 ViewGroup 就直接把事件发送给自身的 onTouchEvent() 而不是继续递归调用 childView 的dispatchTouchEvent() 方法 ），ScrollView （即ViewGroup）会直接截获触控事件，当然，触控位置也会随之从 Button 内部的坐标转换为自身内部的坐标。

注意：其实 Button 也会收到 ACTION_CANCEL 这一触控事件。因为当 ACTION_MOVE 触控手势被检测到时候，Button 上的触控事件会被窃取，系统需要让 Button 知道，它已经不能再接收任何触控事件了，于是给 Button 发送 ACTION_CANCEL，所以，在一个 ScrollView 中的 Button，点击它，再进行上下滑动，再松开，Button 并没有触发点击事件。 通常 ACTION_CANCEL、ACTION_UP 通常用于结束当前手势触控事件。

ViewConfiguration 类

该类携带许多供框架使用的常量，开发者需要在程序中使用这些常量，来使得你的事件处理与系统的处理一致。其中大多数都是与触控相关的，当然也会有与画图或者其他方面相关的常量存在，下面，挑几个相对重要的常量进行介绍。

getScaledTouchSlop() 方法

概述：touchSlop 是一个定义的值，此值用于跟触控事件作比较，从而决定能否从轻触转换为拖拽。

如果你将手指放置于屏幕，尽可能的不移动，但被触控到的 View 还是会收到很多 ACTION_MOVE 事件，因为手指很容易颤动，细微的振动，系统为了探知手指位置，会产生大量的 ACTION_MOVE 事件，尽管用户根本没有移动手指。此时，touchSlop 开始发挥作用了，系统会将手指移动距离与 touchSlop 常量作对比，从而判断该动作为轻触还是移动。

fling

用于判断当手指离开屏幕时候，屏幕应该滚动还是停止。通常我们都会有一种体验，快速滑动 ScrollView ，往往控件会再继续滚动一会。那么系统是如何进行判断的呢?

当手指以移动状态离开屏幕（没有停下），并且速度比系统定义的 fling 值大，系统框架就会认为此手势是 fling，可滚动控件就会根据手指移动速度进行相应的滚动，当然，系统也会有一个最大 fling 速度，当用户手指速度超过该速度时，控件的滚动速度达到最大，而不会再加快。

scaledPagingTouchSlop

该常量常用于 ViewPager，与 touchSlop(ScaleTouchSlop) 区分开，这里的 touchSlop 指的是横向翻页的 touchSlop ，不同于一般的滑动 touchSlop，当然，这并不是说明所有横向的 slop 都使用这一常量。

问题来了，为什么要设置两种 Slop： 主要是因为考虑到，当 ScrollView 嵌套与 ViewPager 中，该如何使得两者较好的工作，因此，需要 x、y轴上分别有两种不同的 Slop 检查，从而判断出用户想上下滚动还是侧滑

系统常量的好处

这些常量与屏幕分辨率无关，类似于 dp ，它们根据不同的屏幕密度来进行缩放，大大方便了开发者写出与系统行为相互适应的 App

处理触控事件的注意事项

例子：屏幕上有两个 Button A、B，现在想要做到一种效果，用户点击 A ,但在 B 做出点击响应，而用户点击 B 时候，则是 A 做出响应，即：想 A 把触控事件传递给 B，B 把触控事件传递给 A 。

一般的想法：在 A 按钮触控事件处理中（dispatch 或者 onTouchEvent 方法）调用 B 按钮的 onTouchEvent 方法来处理事件。技术角度上，确实可行，但如果这么做，整个事件响应链的剩余部分会在此处被避开了。因为在 View 的 dispatch 方法中，还做了许多其他不必要的事情。

最佳且最适当的做法：顺应系统事件分发链来做，当事件传进来，调用另一个 View 的 dispatch 方法。

onInterceptTouchEvent()

事件传递过程中，每个 View 都会先检查该方法的返回值，只要 ViewGroup 中的该方法返回 false ，事件才会走正常的传递链，而如果该方法返回 true，那么当前手势所有的后继触控事件都会直接传递给当前 ViewGroup 的 onTouchEvent() 事件，而本来会接收到触控事件目标 View 会收到 ACTION_CANCEL，这一过程是不可逆的，另外，值得注意的是，onInterceptTouchEvent() 作用是一次性的，需要每次针对特定方法调用该方法才能每次生效。

因此，ViewGroup 一旦把触控事件拦截了，那么就必须处理整个事件，而不能只处理一部分事件，然后把事件还回去。**结论：除非完全处理事件，否则不要拦截事件。**

尽可能地把不需要处理的事件传递给父类

View 中的 onTouchEvent() 方法实际上是做了许多操作来维护 View 自身的状态的（pressed、checked...），如果拦截了所有的事件，那么 View 自身的状态维护功能就会丢失，而开发者往往摸不着头脑，因此，与其直接返回 true 或者 false ，不如直接返回一个对父类的调用。

注意使用 Slop 常量进行检查

处理 ACTION_MOVE 的时候，需要使用该常量来判断用户是否真正想要移动，还是只是在轻触屏幕

考虑如何处理 ACTION_CANCEL 事件类型

大多数情况下，ACTION_CANCEL、ACTION_UP 是放在一起处理的，当时也会有情况会造成两者不同时出现的情况，例如，最常见的：用户手指弹起导致的 ACTION_UP，或者 ACTION_CANCEL 是由于 View 中事件被窃取而产生的（上文提及的：ScrollView 与 Button 滚动的情况）。当 ACTION_CANCEL 与 ACTION_UP 没有同时出现的时候，就可能需要将它们分开处理了（尽管大多数情况下，放在一起处理也是没有什么问题的，主要的操作是：重置 View 的状态）

多点触控的处理

首先需要明确的是，不论是单点还是多点触控，都是遵循系统触控框架的调用流程的，整个分发过程上，两者没有任何不同，唯一不同的仅仅在于系统报告的触控点数量超过了 1 。

MotionEvent.getPointerCount()

该方法会根据每个事件，读出当前有几个触控点正在设备跟踪记录，但设备记录的有效触控点数可能会有限，当超过设备支持的触控点数时，设备的有效追踪点数，当然就不会再增加了。

ACTION_POINTER_DOWN 与 ACTION_POINTER_UP

当用户放了 1 根手指在屏幕上，再继续放第 2、3、4 根手指时候，那么此时触发的就是 ACTION_POINTER_DOWN，而松开手指时候，触发的是 ACTION_POINTER_UP，直到只剩下 1 根手指在屏幕上，最后 1 根手指松开时候，触发 ACTION_UP

系统会为屏幕上的每个触控点设置一个 index 与 id，开发者可使用其来对整个手势每个触控点做跟踪记录。

index 可变而 id 不可变，假设：当用户放 3 根手指在屏幕上，index 与 id 分别为0，1，2；0，1，2。而用户松开了一根手指（id 为 1 的触控点消失）后，index 与 id 可能就会变成 0，1；0，2。 index 更多的是与屏幕上当前触控点相关（还剩多少个触控点），而 id 仅仅作为触控点的标记（针对某个触控点进行相关操作）

注意：通常，触控点减少时，系统默认会把**在减少的触控点上的后续触控事件**传递到前一个 index 的触控点上。

Batching 批量处理

通常，设备实际产生的触控事件往往产生的比你相像的要多很多（由硬件以及内核驱动而产生），尽管用户只是在屏幕上轻轻滑动一下。因此，View 实际上并没有检测到每一个单独的事件，而是获取到批量处理后的结果（成批事件处理中的最后一个事件）

如果因为某种原因，你需要对系统生成的每一个事件都做检测或者想得到每一个单独事件，那可能需要用这些方法来构建（MotionEvent 类中的方法）：

- 获取当前最新事件信息： getX()、getY()、getEventTime()
- ACTION_MOVE 过程中获取的触控事件：getHistoricalX()、getHistoricalY()、getHistoricalEventTime()；getHistoricalSize() -- 获取 batched events 数量

Hover Events 悬浮事件

API 14（Android 4.0）之后加入：ACTION_HOVER_ENTER、ACTION_HOVER_EXIT、ACTION_HOVER_MOVE，当然，这些都是 MotionEvent 类之中的定义的。

部分设备，例如：SAMSUNG 的手机部分就支持悬浮操作，或者是在 Android 设备上接入鼠标，鼠标悬浮在特定的 View 上面，就会触发悬浮事件，事件分发流程与普通触控事件一致。

而不同之处：回调函数与普通触控事件的**回调函数**不一样，这些事件不会在 onTouchEvent 中检测到，而是在 View.onGenericMotionEvent() 方法中，或者设置 View.onGenericMotionEventListener，虽然方法传入的参数与普通触控事件在 onTouchEvent 中传入的参数并无差异。我们可以这么认为：系统想要将悬浮事件与触控事件分开处理，所以特地分出了特殊处理的方法，而整个事件分发流程实质上是相同的。

注意：系统默认是**没有**处理**悬浮事件**的，需要靠用户来自行定制。

Touch Delegate 触摸代理

设计目的：允许父视图去定义一个特定的触摸区域，而在这个触摸区域中，所接收到的所有事件会向前传递给某个 childView。

应用例子：假如 ListView 中有一些图标太小以至于难以触摸，但又希望用户能点击时。可以使用触摸代理来定义一个区域，把此区域的所有触摸事件传递给那个很小的图标。（这种情况，其实完全可以直接把图标设置大一些，例如：Padding属性，这样更加简单快捷；触摸代理主要是针对一些无法简单扩大的 View 范围，不得不这么做的情况）