View的事件分发

1. **事件 MotionEvent**

ACTION\_DOWN 手指刚接触屏幕

ACTION\_MOVE 手指在屏幕上移动

ACTION\_UP 手指从屏幕上松开的一瞬间

TouchSlop:系统所能识别的被认为是滑动的最小距离：ViewConfiguration.get(getContext).getScaledTouchSlop()

VelocityTracker（速度追踪）GestureDetector（手势检测，单击、滑动、长按、双击等）和 Scroller（弹性滑动对象）

**二、View的事件分发**

一：点击事件的传递规则（就是对MotionEvent事件的分发过程）

dispatchTouchEvent (MotionEvent ev) 它就是来执行分发的，进行逻辑上的拦截和处理控制

boolean onInterceptTouchEvent (MotionEvent event) 判断是否拦截事件

onTouchEvent (MotionEvent evnt) 处理事件的

他们之间的 关系用伪代码表述如下:

public boolean dispatchTouchEvent（Motion Event）{

boolean consum = false;

if (onInterceptTouchEvent (ev)) { consume=onTocuhEvent(ev) }

else consum = child.dispatchTouchEvent(ev);

return consum;

}

如果一个View的onTouchEvent 返回false，那么它的父容器的 onTouchEvent 将会被调用，依次类推，直到回退到Activity的onTouchEvent，不论它返回什么，事件都会终止

一些结论

* 同一个事件只能被一个View处理（注意不是事件序列）
* 同一个事件序列指的从手指接触屏幕的那一刻起，到手指离开的那一刻结束，在这个过程中的一系列事件
* 一个事件流中的事件可以由不同的View处理
* 父View在任何时候都可以拦截事件（拦截不一定发生在onInterceptTouchEvent中，也可能是 dispatchTouchEvent(改变了分发逻辑) ）
* 子View不可以控制父View对事件的拦截和处理（不过在一个事件序列中，好像有个函数 disallow…）
* 某个View一旦决定拦截，那么这一个事件序列都只能由他来处理（如果事件序列能传递给它的话），意思其实是它不会继续向下（子View）传递了，不用再调用它自身的 onInterceptTouchEvent了，这点别被模糊了，。父ViewGroup依然可以拦截它并接管事件序列
  + 注意，拦截了的话，只能说有一点是确定的，就是它自身的 onIntercept 不会再被调用，就是说我已经拦截了，下次就不用判断了。当然，这个事件序列是不是一定只能由他处理呢，这要看拦截之后的处理（onTouchEvent）是不是消费了这个事件，并且，父ViewGroup依然有权抢夺它的事件消费权（即是它接管了这个事件序列）
* *某个View一旦开始处理事件（onTouchEvent调用，处理不一定消费了哈），如果它不消耗ACTION\_DOWN事件（onTouchEvent返回了false），那么同一个事件序列中的其它事件都不会再交给他来处理，并且将事件重新交给他的父元素去处理（父元素的onTouchEvent 被调用）*
* 如果View不消耗除ACTION\_DOWN以外的其它事件，那么这个点击事件会消失，此时后续事件并不会回溯给父ViewGroup的onTouchEvent，并且当前View可以继续接到后续事件
* ViewGroup默认不拦截任何事件（Intercept返回false），
* *View的onTouchEvent默认都会消耗事件（返回true，*不管你对event进行各种复杂操作了没，你返回true，这个事件就是被消费了），除非它是不可点击的（clickable 和 longClickable 同时为false）。我的一些想法：当View的onTouchEvent 返回false，父View（或者自己，具体逻辑是放在哪一层要看看源码）是不是会记录上一个事件是什么事件（ACTION\_DOWN和其它两个处理毕竟不一样）

所以，View并不能直接说它默认就一定会消耗事件，而是要看它的属性是不是可点击的，可点击，那默认就消耗，否则默认不消耗

* onClick 发生的前提是当前View是可点击的，并且它收到了down和up的事件
* 事件传递过程是由外向内的，即事件总是先传递给父元素，然后再由父元素分发给子View通过 requestDisallowInterceptTouchEvent 方法可以在子元素中干预父元素的事件分发过程，但是 ACTION\_DOWN 事件除外

**三、从源码的角度来分析完整的分发架构**

1. VIEW的事件分发

public class View{

**public boolean dispatchTouchEvent(MotionEvent ev){** *//注意，View没有拦截逻辑*

boolean result = false;

// 先判断是否设置了onTouchListener,如果设置了,就调用

if (mOnTouchListener != null && mOnTouchListener.onTouch(ev)) {

result = true;

}

*// 如果没有设置onTouchListener,或者调用了onTouch后返回了false,那么就调用onTouchEvent*

*//说明 onTouchListenerf的优先级要高于 onTouchEvent，也可以理解，因为用户优先嘛，不必每次去改写View*

if (!result && onTouchEvent(ev)) {

result = true;

}

return result;

**}**

**public boolean onTouchEvent(MotionEvent ev){**

// 如果View可点击

if (isClickable()) {

switch (ev.getAction()) {

...

case MotionEvent.ACTION\_UP:

...

// 判断是否设置了onClickListener,设置了就调用,并消耗事件

if (mOnClickListener != null) {

mOnClickListener.onClick(this);

}

...

break;

...

}

// 可点击的View,默认会消耗掉事件

return true;

}

// 不可点击的View,默认不消耗事件

return false;

**}**

}

1. ViewGroup的事件分发

public class ViewGroup extends View{

// 一旦有子元素消耗了事件,这个子元素就被赋值给mFirstTouchTarget

public View mFirstTouchTarget = null;

public boolean dispatchTouchEvent(MotionEvent ev){

int action = ev.getAction();

boolean handled = false;

boolean intercepted;

**// DOWN事件,或者有子元素消耗事件,则进入if语句内,判断是否拦截**

**if** (action == MotionEvent.ACTION\_DOWN || mFirstTouchTarget != null) {

// 取出 是否允许拦截 的标记,默认是false。子元素可以通过requestDisallowInterceptTouchEvent(true);将

这个标记设置为true

boolean disallow = getFlagDisallowIntercept();

if (!disallow) {

// 允许拦截,调用onInterceptTouchEvent判断是否拦截

intercepted = onInterceptTouchEvent(ev);

}else {

// 不允许拦截,直接不拦截

intercepted = false;

}

**} else** {

// 非DOWN事件并且没有子元素消耗了事件（子元素onTouchEvent 返回false）,直接拦截

intercepted = true;

**}**

***// 如果有子元素消耗事件,alreadyDispatched就被赋值为true***

***boolean alreadyDispatched = false;***

if (!intercepted) {

// 不拦截事件

// 遍历子元素

for (int i = 0 ; i < mChildList().size() ; i++) {

View child = mChildList.get(i);

float x = ev.getX();

float y = ev.getY();

if (x >= child.left && x <= child.right && y >= child.top && y <= child.bottom) {

// 事件位于子元素所在区域内,将事件分发给子元素

if (child.dispatchTouchEvent(ev)) {

// 子元素消耗了事件

mFirstTouchTarget = child;

*alreadyDispatched = true;*

// 跳出循环,不再分发

break;

}

}

}

}

if (mFirstTouchTarget == null) {

// 没有子元素消耗事件

// ViewGroup自行处理事件

super.dispatchTouchEvent(ev);

}else {

***// 有子元素消耗了事件,则后续事件全部交给该子元素处理,不管他是否消耗事件 ? ? ?***

*if (alreadyDispatched)* {

mFirstTouchTarget.dispatchTouchEvent(ev);

handled = true;

}

}

return handled;

}

public boolean onTouchEvent(MotionEvent ev){

super.onTouchEvent(ev);

}

public boolean onInterceptTouchEvent(MotionEvent ev){

return false;

}

}