转载请注明链接：https://blog.csdn.net/feather_wch/article/details/81503716

思维导图还没完成，之后补全。。

View的事件分发-思维导图版

1-View基础

1.1-什么是View

1、什么是View

> 1. View是所有控件的基类
> 2. View有一个特殊子类ViewGroup，ViewGroup能包含一组View，但ViewGroup的本身也是View。

3. 由于View和ViewGourp的存在，意味着View可以是单个控件也可以是一组控件。这种结构形成了View树。

1.2-View的位置参数

2、View的位置参数：top,left,right,bottom

1. top-左上角的y轴坐标(全部是相对坐标，相对于父容器)
2. left-左上角的x轴坐标
3. right-右下角的x轴坐标
4. bottom-右下角的y轴坐标
5. 在View中获取这些成员变量的方法，是getLeft(),getRight(),getTop(),getBottom()即可

3、View从3.0开始新增的参数：x,y,translationX,translationY

1. x,y是View当前左上角的坐标
2. translationX,translationY是在滑动/动画后，View当前位置和View最原始位置的距离。
3. 因此得出等式：x(View左上角当前位置) = left(View左上角初始位置) + translationX(View左上角偏移的距离)
4. View平移时top、left等参数不变，改变的是x,y,tranlsationX和tranlsationY

1.3-MotionEvent

4、MotionEvent包含的手指触摸事件

1. ACTION_DOWN\MOVE\UP对应三个触摸事件。
2. getX/getY能获得触摸点的坐标，相当于当前View左上角的(x,y)
3. getRawX/getRawY，获得触摸点相当于手机左上角的(x,y)坐标

2-事件分发机制

1、事件分发

1. 点击事件的对象就是MotionEvent，因此事件的分发，就是MotionEvent的分发过程，
2. 点击事件有三个重要方法来完成：dispatchTouchEvent、onInterceptTouchEvent和onTouchEvent

2.1-三个重要方法

32、事件分发伪代码:

```
public boolean dispatchTouchEvent(MotionEvent ev){
  boolean consume = false;
  if(onInterceptTouchEvent(ev)){
    consume = onTouchEvent(ev);
  }else{
    consume = child.dispatchTouchEvent(ev);
  }
  return consume;
}
```

2.1.1-dispatchTouchEvent

2、dispatchTouchEvent的作用

> 1. 用于进行事件的分发
> 2. 只要事件传给当前View，该方法一定会被调用
> 3. 返回结果受到当前View的onTouchEvent和下级View的dispatchTouchEvent影响
> 4. 表示是否消耗当前事件

2.1.2-onInterceptTouchEvent

3、onInterceptTouchEvent的作用

> 1. 在dispatchTouchEvent的内部调用，用于判断是否拦截某个事件

2.1.3-onTouchEvent

4、onTouchEvent的作用

> 1. 在dispatchTouchEvent的中调用，用于处理点击事件
> 2. 返回结果表示是否消耗当前事件

2.2-事件传递规则与要点

2.2.0-事件传递规则

33、事件的传递规则：

> 1. 点击事件产生后，会先传递给根ViewGroup，并调用dispatchTouchEvent
> 2. 之后会通过onInterceptTouchEvent判断是否拦截该事件，如果true，则表示拦截并交给该ViewGroup的onTouchEvent方法进行处理
> 3. 如果不拦截，则当前事件会传递给子元素，调用子元素的dispatchTouchEvent，如此反复直到事件被处理

34、View处理事件的优先级

1. 在View需要处理事件时，会先调用OnTouchListener的onTouch方法，并判断onTouch的返回值
2. 返回true，表示处理完成，不会调用onTouchEvent方法
3. 返回false，表示未完成，调用onTouchEvent方法进行处理
4. 可见，onTouchEvent的优先级没有OnTouchListener高
5. 平时常用的OnClickListener优先级最低，属于事件传递尾端

35、点击事件传递过程遵循如下顺序：

1. Activity->Window->View->分发
2. 如果View的onTouchEvent返回false，则父容器的onTouchEvent会被调用，最终可以传递到Activity的onTouchEvent

36、事件传递规则要点

1. View一旦拦截事件，则整个事件序列都由它处理(ACTION_DOWN\UP等)，onInterceptTouchEvent不会再调用(因为默认都拦截了)
2. 但是一个事件序列也可以通过特殊方法交给其他View处理(onTouchEvent)
3. 如果View开始处理事件(已经拦截)，如果不消耗ACTIO_DOWN事件(onTouchEvent返回false)，则同一事件序列的剩余内容都直接交给父onTouchEvent处理
4. View消耗了ACTION_DOWN，但不处理其他的事件，整个事件序列会消失(父onTouchEvent)不会调用。这些消失的点击事件最终会传给Activity处理。
5. ViewGroup默认不拦截任何事件(onInterceptTouchEvent默认返回false)
6. View没有onInterceptTouchEvent方法，一旦有事件传递给View，onTouchEvent就会被调用
7. View的onTouchEvent默认都会消耗事件return true, 除非该View不可点击(clickable和longClickable同时为false)
8. View的enable属性不影响onTouchEvent的默认返回值。即使是disable状态。
9. onClick的发生前提是当前View可点击，并且收到了down和up事件
10. 事件传递过程是由父到子，层层分发，可以通过requestDisallowInterceptTouchEvent让子元素干预父元素的事件分发(ACTION_DOWN除外)

2.2.1-Activity的事件分发

37、Activity事件分发的过程

1. 事件分发过程：Activity->Window->Decor View(当前界面的底层容器，setContentView的View的父容器)->View
2. Activity的dispatchTouchEvent，会交给Window处理（`getWindow().superDispatchTouchEvent()`），
3. 返回true：事件全部结束
4. 返回false：所有View都没有处理(onTouchEvent返回false)，则调用Activity的onTouchEvent

2.2.2-Window的事件分发

38、Window事件分发

> 1. Window和superDispatchTouchEvent分别是抽象类和抽象方法
> 2. Window的实现类是PhoneWindow
> 3. PhoneWindow的 `superDispatchTouchEvent()` 直接调用 `mDecor.superDispatchTouchEvent()` ,也就是直接传给了DecorView

2.2.3-DecorView的事件分发

39、DecorView的事件分发

> 1. DecorView继承自FrameLayout
> 2. DecorView的 `superDispatchTouchEvent()` 会调用 `super.dispatchTouchEvent()` ——也就是 `ViewGroup` 的 `dispatchTouchEvent` 方法，之后就会层层分发下去。

2.2.4-根View的事件分发

40、根View的事件分发

> 1. 顶层View调用dispatchTouchEvent
> 2. 调用onInterceptTouchEvent方法
> 3. 返回true，事件由当前View处理。如果有onTouchiListener，会执行onTouch，并且屏蔽掉onTouchEvent。没有则执行onTouchEvent。如果设置了onClickListener，会在onTouchEvent后执行onClickListener
> 4. 返回false，不拦截，交给子View重复如上步骤。

2.2.5-ViewGroup的事件分发

41、ViewGroup的dispatchTouchEvent事件分发解析

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    boolean handled = false;
    //1. 过滤掉不符合安全策略的事件
    if (onFilterTouchEventForSecurity(ev)) {
        final boolean intercepted;
        /**==========================================
         * 2. 一旦一系列事件中的某个事件被拦截，后续的事件都会直接拦截，不会再判断
         *  情景1：为MotionEvent.ACTION_DOWN，等式为true，进入判断是否拦截
         *  情景2：不为ACTION_DOWN，(mFirstTouchTarget!=null)系列事件都没有被拦截，等式为true，进
         *  情景3：不为ACTION_DOWN，(mFirstTouchTarget=null)前面事件被拦截，等式为false
         *==========================================*/
        if (actionMasked == MotionEvent.ACTION_DOWN || mFirstTouchTarget != null) {
            /**==========================================
             *3. 由子View请求ViewGroup不要拦截该事件
             * 1-子View会通过`requestDisallowInterceptTouchEvent`设置FLAG_DISALLOW_INTERCEPT标志
             * 2-ACTION_DOWN会重置FLAG_DISALLOW_INTERCEPT标志位，因此无法被子View影响
             *==========================================*/
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
            if (!disallowIntercept) {
                //4. 判断ViewGroup是否拦截该事件
                intercepted = onInterceptTouchEvent(ev);
                ev.setAction(action); // restore action in case it was changed
            } else {
                //5. 由子View控制不拦截该事件(前提是DOWN没有被拦截)
                intercepted = false;
            }
        } else {
            //6. ACTION_UP\MOVE等系列事件被拦截过，因此后续的全部拦截，不会重新判断
            intercepted = true;
        }
        ......
        //7. 没有被拦截，交给子View处理
        if (!canceled && !intercepted) {
            //8. 遍历所有子元素，并判断是否能接受点击事件，以及点击事件坐标是否在子元素内。
            for (int i = childrenCount - 1; i >= 0; i--) {
                //判断是否能接受点击事件，不能就直接continue
                if (childWithAccessibilityFocus != null) {
                    if (childWithAccessibilityFocus != child) {
                        continue;
                    }
                }
                //判断点击事件坐标是否在子元素内，不在就直接continue
                if (!canViewReceivePointerEvents(child) || !isTransformedTouchPointInView(x, y,
                    continue;
                }
                //分发给子View处理，内部就是调用子元素的`dispatchTouchEvent`方法
                if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                    //子View消耗并且处理该事件
                    alreadyDispatchedToNewTouchTarget = true;
                    break;
                }
            }
        }
        //9. 事件被拦截或者子View未消耗该事件：自己处理该事件
```

```
        if (mFirstTouchTarget == null) {
            handled = dispatchTransformedTouchEvent(ev, canceled, null, TouchTarget.ALL_POINTER
        }
        .......
    }
    return handled;
}
```

2.2.6-View的事件分发和事件处理

方法	优先级
OnTouchListener的onTouch	1
View的onTouchEvent	2
TouchDelegate的onTouchEvent	3
OnClickListener的onClick	4

> 1. 优先级从`1~4`， 顺序执行
> 2. 只要有 某一层消耗了该事件`(return true)`，后续的方法都不会执行

42、View对点击事件的处理过程(不包括ViewGroup)

```
/**=========================
 * 1. 事件分发(OnTouchListener或者onTouchEvent直接处理)
 *=========================*/
public boolean dispatchTouchEvent(MotionEvent event) {
    boolean result = false;
    ......
    //1. 采用安全策略过滤事件
    if (onFilterTouchEventForSecurity(event)) {
        ListenerInfo li = mListenerInfo;
        //2. 判断是否有OnTouchListener，返回true，则处理完成
        if (li != null && li.mOnTouchListener != null
                && (mViewFlags & ENABLED_MASK) == ENABLED
                && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        //3. 如果OnTouch返回true，不会调用onTouchEvent
        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
    ......
    return result;
}
/**=========================
 * 2. 事件处理onTouchEvent
 *=========================*/
public boolean onTouchEvent(MotionEvent event) {
    //0. 获取点击状态
    final boolean clickable = ((viewFlags & CLICKABLE) == CLICKABLE
            || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)
            || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE;
    //1. View不可用状态下(可点击状态-会消耗该事件,不可点击不消耗)
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (action == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        return clickable;//根据是否可点击决定是否消耗
    }
    //2. 如果有代理，会执行代理的onTouchEvent方法（会消耗该事件）
    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }
    //3. View可点击(消耗该事件)
    if (clickable || (viewFlags & TOOLTIP) == TOOLTIP) {
        switch (action) {
            case MotionEvent.ACTION_UP:
                  ......
                //5. 如果设置了`OnClickListener`，performClick内部会调用onClick方法
                  performClick();
                  ......
                break;
        }
```

```
            return true;
        }
        //4. 可用状态&&没有代理&&不可点击：不消耗该事件
        return false;
    }
```

3-滑动冲突

滑动冲突的三种场景

43、滑动冲突的三种场景

> 1. 内层和外层滑动方向不一致：一个垂直，一个水平
> 2. 内存和外层滑动方向一致：均垂直or水平
> 3. 前两者层层嵌套

滑动冲突处理原则和解决办法

44、 滑动冲突处理原则

> 1. 对于内外层滑动方向不同，只需要根据滑动方向来给相应控件拦截
> 2. 对于内外层滑动方向相同，需要根据业务来进行事件拦截
> 3. 前两者嵌套的情况，根据前两种原则层层处理即可。

45、 滑动冲突解决办法

> 1. 外部拦截：在父容器进行拦截处理，需要重写父容器的onInterceptTouchEvent方法
> 2. 内部拦截：父容器不拦截任何事件，事件都传递给子元素。子元素需要就处理，否则给父容器处理。需要配合 `requestDisallowInterceprtTouchEvent` 方法。

外部拦截

46、外部拦截法要点

> 1. 父容器的 `onInterceptTouchEvent` 方法中处理
> 2. ACTION_DOWN不拦截，一旦拦截会导致后续事件都直接交给父容器处理。
> 3. ACTION_MOVE中根据情况进行拦截，拦截：return true，不拦截：return false（外部拦截核心）
> 4. ACTION_UP不拦截，如果父控件拦截UP，会导致子元素接收不到UP进一步会让onClick方法无法触发。此外UP拦截也没什么用。

49、外部拦截，自定义ScrollView

```kotlin
//Kotlin
class CustomScrollView(context: Context,
                       attrs: AttributeSet?,
                       defStyleAttr: Int,
                       defStyleRes: Int): ScrollView(context, attrs, defStyleAttr, defStyleRes)

    constructor(context: Context) : this(context, null, 0, 0)
    constructor(context: Context, attrs: AttributeSet?) : this(context, attrs, 0, 0)
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int) : this(context, attr

    var lastX: Int = 0
    var lastY: Int = 0

    override fun dispatchTouchEvent(ev: MotionEvent): Boolean {

        val curX = ev.x.toInt()
        val curY = ev.y.toInt()

        when(ev.action){
            ACTION_DOWN -> {
                parent.requestDisallowInterceptTouchEvent(true)
            }
            ACTION_MOVE -> {
                //如果是水平滑动则交给父容器处理
                if(Math.abs(curX - lastX) > Math.abs(curY - lastY)){
                    parent.requestDisallowInterceptTouchEvent(false)
                }
            }
            ACTION_UP -> null
            else -> null
        }
        lastX = curX
        lastY = curY
        return super.dispatchTouchEvent(ev)
    }
}
```

内部拦截

47、内部拦截法要点

> 1. 子View的 `dispatchTouchEvent` 方法处理
> 2. ACTION_DOWN，让父容器不拦截(也不能拦截，否则会导致后续事件都无法传递到子View)- `parent.requestDisallowInterceptTouchEvent(true)`
> 3. ACTION_MOVE,如父容器需要该事件，则父容器拦截 requestDisallowInterceptTouchEvent(false)
> 4. ACTION_UP，无操作，正常执行

48、内部拦截Kotlin代码

```kotlin
//Kotlin
class CustomHorizontalScrollView(context: Context,
                                 attrs: AttributeSet?,
                                 defStyleAttr: Int,
                                 defStyleRes: Int): HorizontalScrollView(context, attrs, defSty
  //构造器
    constructor(context: Context): this(context, null, 0, 0)
    constructor(context: Context, attrs: AttributeSet?): this(context, attrs, 0, 0)
    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int): this(context, attrs

    var downX: Int = 0
    var downY: Int = 0
  //拦截处理
    override fun onInterceptTouchEvent(ev: MotionEvent): Boolean {
        var intercepted = super.onInterceptTouchEvent(ev)
        when(ev.action){
        //不拦截
            ACTION_DOWN -> {
                downX = ev.x.toInt()
                downY = ev.y.toInt()
                intercepted = false
            }
        //判断是否拦截
            ACTION_MOVE -> {
                val curX = ev.x.toInt()
                val curY = ev.y.toInt()
                //水平滑动进行拦截
                if(Math.abs(curX - downX) > Math.abs(curY - downY)){
                    intercepted = true
                }
            }
        //不拦截
            ACTION_UP -> intercepted = false
            else -> null
        }
        return intercepted
    }
}
```