一.概述

事件分发有多种类型, 本文主要介绍Touch相关的事件分发.

- 整个事件分发流程中，会有大量MotionEvent对象，该对象用于记录所有与移动相关的事件，比如手指触摸屏幕事件。
- 一次完整的MotionEvent事件，是从用户触摸屏幕到离开屏幕。整个过程的动作序列：ACTION*DOWN(1次) -> ACTION*MOVE(N次) -> ACTION_UP(1次),
- 多点触摸，每一个触摸点Pointer会有一个id和index。对于多指操作，通过pointerindex来获取指定Pointer的触屏位置。比如，对于单点操作时获取x坐标通过getX()，而多点操作获取x坐标通过getX(pointerindex)

对于View,ViewGroup,Activity都能处理Touch事件, 它们之间处理的先后顺序和方法有所不同.

1.1 View

- View是所有视图对象的父类，实现了动画相关的接口Drawable.Callback， 按键相关的接口KeyEvent.Callback， 交互相关的接口AccessibilityEventSource。比如Button继承自View。
- TouchEvent事件处理相关的方法： > dispatchTouchEvent(MotionEvent event)

> onTouchEvent(MotionEvent event)

1.2 ViewGroup

- ViewGroup，是一个abstract类，一组View的集合，可以包含View和ViewGroup,是所有布局的父类或间接父类。继承了View，实现了ViewParent（用于与父视图交互的接口）， ViewManager（用于添加、删除、更新子视图到Activity的接口）。比如常用的LinearLayout，RelativeLayout都是继承自ViewGroup。
- TouchEvent事件处理相关的方法： > dispatchTouchEvent(MotionEvent event)

> onInterceptTouchEvent(MotionEvent ev)
>
> onTouchEvent(MotionEvent event)

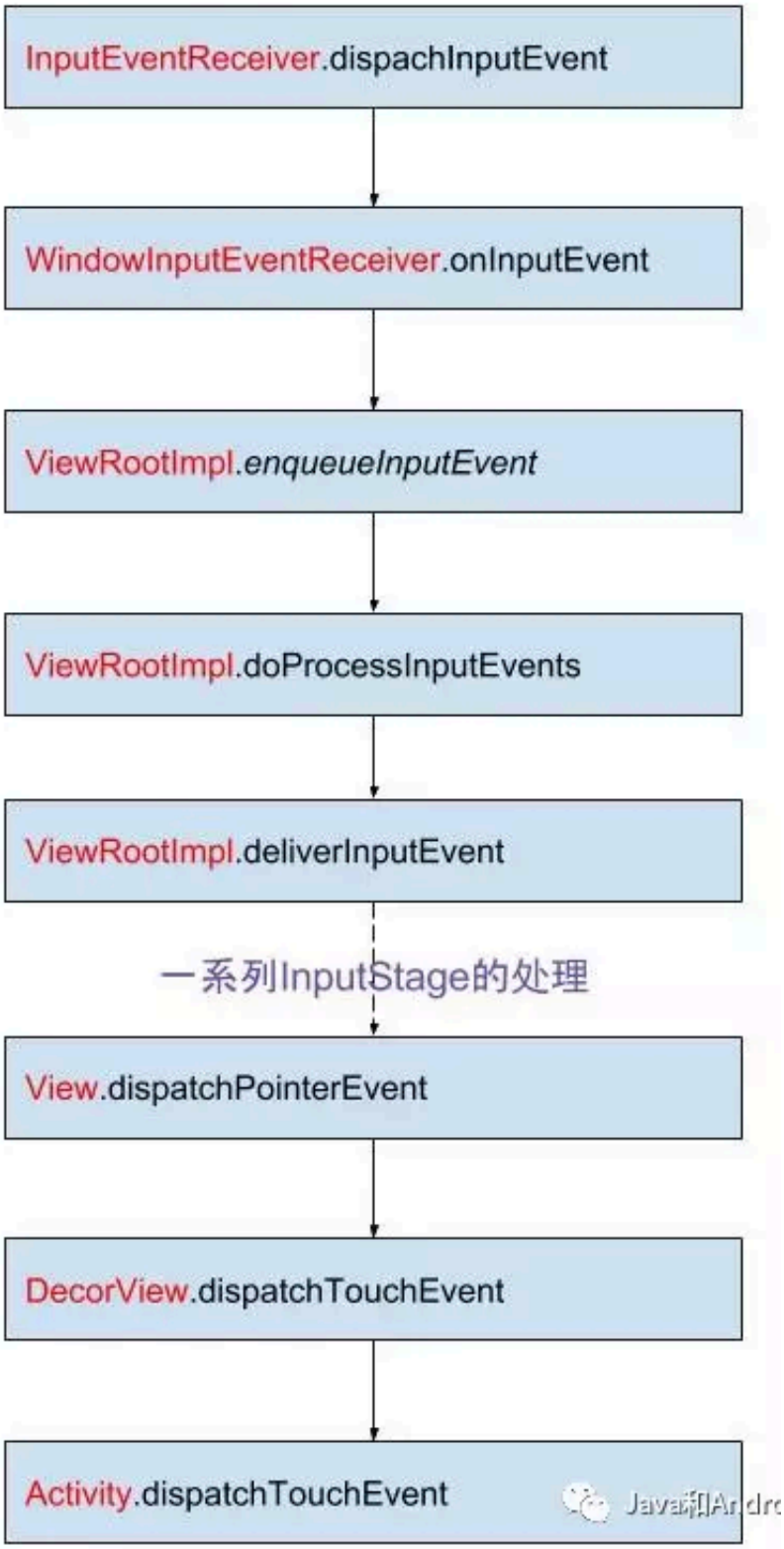
1.3 Activity

- Activity是Android四大基本组件之一，当手指触摸到屏幕时，屏幕硬件一行行不断地扫描每个像素点，获取到触摸事件后，从底层产生中断上报。再通过native层调用Java层InputEventReceiver中的dispatchInputEvent方法。经过层层调用，交由Activity的dispatchTouchEvent方法来处理。
- TouchEvent事件处理相关的方法： > dispatchTouchEvent(MotionEvent event)

> onTouchEvent(MotionEvent event)

分发原理

Input系统—进程交互文章的小节[3.3]已介绍事件分发过程的前期工作. 当UI主线程收到底层上报的input事件, 便会调用InputEventReceiver.dispachInputEvent方法.



2.1 DecorView.dispatchTouchEvent

[-> PhoneWindow.java ::DecorView]

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    final Callback cb = getCallback();
    // [见小节2.2]
    return cb != null && !isDestroyed() && mFeatureId < 0 ? cb.dispatchTouchEvent(ev)
            : super.dispatchTouchEvent(ev);
}
```


此处cb是指Window的内部接口Callback. 对于Activity实现了Window.Callback接口. 故接下来调用Activity类.

2.2 Activity.dispatchTouchEvent

[-> Activity.java]

```
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {
        //第一次按下操作时，用户希望能与设备进行交互，可通过实现该方法
        onUserInteraction();
    }

    //获取当前Activity的顶层窗口是PhoneWindow [见小节2.3]
    if (getWindow().superDispatchTouchEvent(ev)) {
        return true;
    }
    //当没有任何view处理时，交由activity的onTouchEvent处理
    return onTouchEvent(ev); // [见小节2.2.1]
}
```


如果重写Activity的该方法，则会在分发事件之前拦截所有的触摸事件. 另外此处getWindow()返回的是Activity的mWindow成员变量, 该变量赋值过程是在Activity.attach()方法, 可知其类型为PhoneWindow.

2.2.1 Activity.onTouchEvent

[-> Activity.java]

```
public boolean onTouchEvent(MotionEvent event) {
    //当窗口需要关闭时，消费掉当前event
    if (mWindow.shouldCloseOnTouch(this, event)) {
        finish();
        return true;
    }

    return false;
}
```


2.3 superDispatchTouchEvent

[-> PhoneWindow.java]

```
public boolean superDispatchTouchEvent(KeyEvent event) {
    return mDecor.superDispatcTouchEvent(event); // [见小节2.4]
}
```


PhoneWindow的最顶View是DecorView，再交由DecorView处理。而DecorView的父类的父类是ViewGroup，接着调用 ViewGroup.dispatchTouchEvent()方法。为了精简篇幅，有些中间函数调用不涉及关键逻辑，可能会直接跳过。

2.4 ViewGroup.dispatchTouchEvent

```java
public boolean dispatchTouchEvent(MotionEvent ev) {
    ...
    boolean handled = false;
    //根据隐私策略而来决定是否过滤本次触摸事件,
    if (onFilterTouchEventForSecurity(ev)) {  // [见小节2.4.1]
        final int action = ev.getAction();
        final int actionMasked = action & MotionEvent.ACTION_MASK;

        if (actionMasked == MotionEvent.ACTION_DOWN) {
            // 发生ACTION_DOWN事件, 则取消并清除之前所有的触摸targets
            cancelAndClearTouchTargets(ev);
            resetTouchState(); // 重置触摸状态
        }

        // 发生ACTION_DOWN事件或者已经发生过ACTION_DOWN;才进入此区域, 主要功能是拦截器
        //只有发生过ACTION_DOWN事件, 则mFirstTouchTarget != null;
        final boolean intercepted;
        if (actionMasked == MotionEvent.ACTION_DOWN
                || mFirstTouchTarget != null) {
            //可通过调用requestDisallowInterceptTouchEvent, 不让父View拦截事件
            final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) !
            //判断是否允许调用拦截器
            if (!disallowIntercept) {
                //调用拦截方法
                intercepted = onInterceptTouchEvent(ev);  // [见小节2.4.2]
                ev.setAction(action);
            } else {
                intercepted = false;
            }
        } else {
            // 当没有触摸targets, 且不是down事件时, 开始持续拦截触摸。
            intercepted = true;
        }
        ...
```

```
//不取消事件，同时不拦截事件，并且是Down事件才进入该区域
if (!canceled && !intercepted) {
    //把事件分发给所有的子视图，寻找可以获取焦点的视图。
    View childWithAccessibilityFocus = ev.isTargetAccessibilityFocus()
            ? findChildWithAccessibilityFocus() : null;

    if (actionMasked == MotionEvent.ACTION_DOWN
            || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
            || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
        final int actionIndex = ev.getActionIndex(); // down事件等于0
        final int idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex)
                : TouchTarget.ALL_POINTER_IDS;

        removePointersFromTouchTargets(idBitsToAssign); //清空早先的触摸对象

        final int childrenCount = mChildrenCount;
        //第一次down事件，同时子视图不会空时
        if (newTouchTarget == null && childrenCount != 0) {
            final float x = ev.getX(actionIndex);
            final float y = ev.getY(actionIndex);
            // 从视图最上层到下层，获取所有能接收该事件的子视图
            final ArrayList<View> preorderedList = buildOrderedChildList(); //
            final boolean customOrder = preorderedList == null
                    && isChildrenDrawingOrderEnabled();
            final View[] children = mChildren;

            /* 从最底层的父视图开始遍历，
            ** 找寻newTouchTarget，并赋予view与 pointerIdBits；
            ** 如果已经存在找寻newTouchTarget，说明正在接收触摸事件，则跳出循环。
            */
            for (int i = childrenCount - 1; i >= 0; i--) {
                final int childIndex = customOrder
                        ? getChildDrawingOrder(childrenCount, i) : i;
                final View child = (preorderedList ==
                        ? children[childIndex] : preorderedList.get(childIndex)
```

```
// 如果当前视图无法获取用户焦点，则跳过本次循环
if (childWithAccessibilityFocus != null) {
    if (childWithAccessibilityFocus != child) {
        continue;
    }
    childWithAccessibilityFocus = null;
    i = childrenCount - 1;
}
//如果view不可见，或者触摸的坐标点不在view的范围内，则跳过本次循环
if (!canViewReceivePointerEvents(child)
        || !isTransformedTouchPointInView(x, y, child, null)) {
    ev.setTargetAccessibilityFocus(false);
    continue;
}

newTouchTarget = getTouchTarget(child);
// 已经开始接收触摸事件,并退出整个循环。
if (newTouchTarget != null) {
    newTouchTarget.pointerIdBits |= idBitsToAssign;
    break;
}

//重置取消或抬起标志位
//如果触摸位置在child的区域内，则把事件分发给子View或ViewGroup
if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAss
    // 获取TouchDown的时间点
    mLastTouchDownTime = ev.getDownTime();
    // 获取TouchDown的Index
    if (preorderedList != null) {
        for (int j = 0; j < childrenCount; j++) {
            if (children[childIndex] == mChildren[j]) {
                mLastTouchDownIndex = j;
                break;
            }
```

```
                    } else {
                        mLastTouchDownIndex = childIndex;
                    }

                    //获取TouchDown的x,y坐标
                    mLastTouchDownX = ev.getX();
                    mLastTouchDownY = ev.getY();
                    //添加TouchTarget,则mFirstTouchTarget != null。 [见小节2.4.5]
                    newTouchTarget = addTouchTarget(child, idBitsToAssign);
                    //表示以及分发给NewTouchTarget
                    alreadyDispatchedToNewTouchTarget = true;
                    break;
                }
                ev.setTargetAccessibilityFocus(false);
            }
            // 清除视图列表
            if (preorderedList != null) preorderedList.clear();
        }

        if (newTouchTarget == null && mFirstTouchTarget != null) {
            //将mFirstTouchTarget的链表最后的touchTarget赋给newTouchTarget
            newTouchTarget = mFirstTouchTarget;
            while (newTouchTarget.next != null) {
                newTouchTarget = newTouchTarget.next;
            }
            newTouchTarget.pointerIdBits |= idBitsToAssign;
        }
    }
}
```

```java
// mFirstTouchTarget赋值是在通过addTouchTarget方法获取的;
// 只有处理ACTION_DOWN事件，才会进入addTouchTarget方法。
// 这也正是当View没有消费ACTION_DOWN事件，则不会接收其他MOVE,UP等事件的原因
if (mFirstTouchTarget == null) {
    //没有触摸target,则由当前ViewGroup来处理
    handled = dispatchTransformedTouchEvent(ev, canceled, null,
            TouchTarget.ALL_POINTER_IDS);
} else {
    //如果View消费ACTION_DOWN事件，那么MOVE,UP等事件相继开始执行
    TouchTarget predecessor = null;
    TouchTarget target = mFirstTouchTarget;
    while (target != null) {
        final TouchTarget next = target.next;
        if (alreadyDispatchedToNewTouchTarget && target == newTouchTarget) {
            handled = true;
        } else {
            final boolean cancelChild = resetCancelNextUpFlag(target.child)
                    || intercepted;
            if (dispatchTransformedTouchEvent(ev, cancelChild,
                    target.child, target.pointerIdBits)) {
                handled = true;
            }
            if (cancelChild) {
                if (predecessor == null) {
                    mFirstTouchTarget = next;
                } else {
                    predecessor.next = next;
                }
                target.recycle();
                target = next;
                continue;
            }
        }
        predecessor = target;
        target = next;
```

```
            }
        }

        //当发生抬起或取消事件，更新触摸targets
        if (canceled
                || actionMasked == MotionEvent.ACTION_UP
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
            resetTouchState();
        } else if (split && actionMasked == MotionEvent.ACTION_POINTER_UP) {
            final int actionIndex = ev.getActionIndex();
            final int idBitsToRemove = 1 << ev.getPointerId(actionIndex);
            removePointersFromTouchTargets(idBitsToRemove);
        }
    } //此处大括号，是if (onFilterTouchEventForSecurity(ev))的结尾

    //通知verifier由于当前时间未处理，那么该事件其余的都将被忽略
    if (!handled && mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(ev, 1);
    }
    return handled;
}
```


2.4.1 onFilterTouchEventForSecurity

```
public boolean onFilterTouchEventForSecurity(MotionEvent event) {
    if ((mViewFlags & FILTER_TOUCHES_WHEN_OBSCURED) != 0
            && (event.getFlags() & MotionEvent.FLAG_WINDOW_IS_OBSCURED) != 0) {
        //隐私包含,则丢弃该事件
        return false;
    }
    return true;
}
```


2.4.2 onInterceptTouchEvent

```
public boolean onInterceptTouchEvent(MotionEvent ev) {
    return false;
}
```


2.4.3 buildOrderedChildList

```java
ArrayList<View> buildOrderedChildList() {
    final int count = mChildrenCount;
    if (count <= 1 || !hasChildWithZ()) return null;

    if (mPreSortedChildren == null) {
        mPreSortedChildren = new ArrayList<View>(count);
    } else {
        mPreSortedChildren.ensureCapacity(count);
    }

    final boolean useCustomOrder = isChildrenDrawingOrderEnabled();
    for (int i = 0; i < mChildrenCount; i++) {
        // 添加下一个子视图到列表
        int childIndex = useCustomOrder ? getChildDrawingOrder(mChildrenCount, i) : i;
        View nextChild = mChildren[childIndex];
        float currentZ = nextChild.getZ(); //获取Z轴

        int insertIndex = i;
        //按Z轴，从小到大排序所有的子视图
        while (insertIndex > 0 && mPreSortedChildren.get(insertIndex - 1).getZ() > curr
            insertIndex--;
        }
        mPreSortedChildren.add(insertIndex, nextChild);
    }
    return mPreSortedChildren;
}
```


获取一个视图组的先序列表，通过虚拟的Z轴来排序。

> public float getZ() {
> return getElevation() + getTranslationZ(); } getZ()用于获取Z轴坐标。屏幕只有x,y坐标，而Z是虚拟的，可通过setElevation(),setTranslationZ()或者setZ()方法来修改Z轴的坐标值。

2.4.4 dispatchTransformedTouchEvent

```java
private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
        View child, int desiredPointerIdBits) {
    final boolean handled;

    // 发生取消操作时，不再执行后续的任何操作
    final int oldAction = event.getAction();
    if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
        event.setAction(MotionEvent.ACTION_CANCEL);
        if (child == null) {
            handled = super.dispatchTouchEvent(event);
        } else {
            handled = child.dispatchTouchEvent(event);
        }
        event.setAction(oldAction);
        return handled;
    }

    final int oldPointerIdBits = event.getPointerIdBits();
    final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;

    //由于某些原因，发生不一致的操作，那么将抛弃该事件
    if (newPointerIdBits == 0) {
        return false;
    }

    //分发的主要区域
    final MotionEvent transformedEvent;
    //判断预期的pointer id与事件的pointer id是否相等
    if (newPointerIdBits == oldPointerIdBits) {
        if (child == null || child.hasIdentityMatrix()) {
            if (child == null) {
                //不存在子视图时，ViewGroup调用View.dispatchTouchEvent分发事件，再调用ViewGro
                handled = super.dispatchTouchEvent(event);
            } else {
```

```
                final float offsetX = mScrollX - child.mLeft;
                final float offsetY = mScrollY - child.mTop;
                event.offsetLocation(offsetX, offsetY);
                //将触摸事件分发给子ViewGroup或View;
                //如果是ViewGroup，则调用代码(2.1);
                //如果是View，则调用代码(3.1);
                handled = child.dispatchTouchEvent(event);

                event.offsetLocation(-offsetX, -offsetY); //调整该事件的位置
            }
            return handled;
        }
        transformedEvent = MotionEvent.obtain(event); //拷贝该事件，来创建一个新的MotionEve
    } else {
        //分离事件，获取包含newPointerIdBits的MotionEvent
        transformedEvent = event.split(newPointerIdBits);
    }

    if (child == null) {
        //不存在子视图时，ViewGroup调用View.dispatchTouchEvent分发事件，再调用ViewGroup.onTou
        handled = super.dispatchTouchEvent(transformedEvent);  // [见小节2.4]
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        transformedEvent.offsetLocation(offsetX, offsetY);
        if (! child.hasIdentityMatrix()) {
            //将该视图的矩阵进行转换
            transformedEvent.transform(child.getInverseMatrix());
        }
        //将触摸事件分发给子ViewGroup或View;
        /如果是ViewGroup，则 [见小节2.4]; 如果是View，则[见小节2.5];
        handled = child.dispatchTouchEvent(transformedEvent);
    }
```



```
    //回收transformedEvent
    transformedEvent.recycle();
    return handled;
}
```


该方法是ViewGroup真正处理事件的地方，分发子View来处理事件，过滤掉不相干的pointer ids。当子视图为null时，MotionEvent将会发送给该ViewGroup。最终调用View.dispatchTouchEvent方法来分发事件。

2.4.5 addTouchTarget

> private TouchTarget addTouchTarget(View child, int pointerIdBits) { TouchTarget target = TouchTarget.obtain(child, pointerIdBits); target.next = mFirstTouchTarget; mFirstTouchTarget = target; return target; } 调用该方法，获取了TouchTarget，同时mFirstTouchTarget不再为null。

2.5 View.dispatchTouchEvent

[-> View.java]

```java
public boolean dispatchTouchEvent(MotionEvent event) {
    ...

    final int actionMasked = event.getActionMasked();
    if (actionMasked == MotionEvent.ACTION_DOWN) {
        //在Down事件之前，如果存在滚动操作则停止。不存在则不进行操作
        stopNestedScroll();
    }

    // mOnTouchListener.onTouch优先于onTouchEvent。
    if (onFilterTouchEventForSecurity(event)) {
        //当存在OnTouchListener，且视图状态为ENABLED时，调用onTouch()方法
        ListenerInfo li = mListenerInfo;
        if (li != null && li.mOnTouchListener != null
                && (mViewFlags & ENABLED_MASK) == ENABLED
                && li.mOnTouchListener.onTouch(this, event)) {
            result = true; //如果已经消费事件，则返回True
        }
        //如果OnTouch ()没有消费Touch事件则调用OnTouchEvent()
        if (!result && onTouchEvent(event)) { // [见小节2.5.1]
            result = true; //如果已经消费事件，则返回True
        }
    }

    if (!result && mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onUnhandledEvent(event, 0);
    }

    // 处理取消或抬起操作
```



```java
    if (actionMasked == MotionEvent.ACTION_UP ||
            actionMasked == MotionEvent.ACTION_CANCEL ||
            (actionMasked == MotionEvent.ACTION_DOWN && !result)) {
        stopNestedScroll();
    }

    return result;
}
```


1.先由OnTouchListener的OnTouch()来处理事件，当返回True，则消费该事件，否则进入2。
2.onTouchEvent处理事件，的那个返回True时，消费该事件。否则不会处理

2.5.1 View.onTouchEvent

```java
public boolean onTouchEvent(MotionEvent event) {
    final float x = event.getX();
    final float y = event.getY();
    final int viewFlags = mViewFlags;

    // 当View状态为DISABLED，如果可点击或可长按，则返回True，即消费事件
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (event.getAction() == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSE
            setPressed(false);
        }
        return (((viewFlags & CLICKABLE) == CLICKABLE ||
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE));
    }

    if (mTouchDelegate != null) {
        if (mTouchDelegate.onTouchEvent(event)) {
            return true;
        }
    }

    //当View状态为ENABLED，如果可点击或可长按，则返回True，即消费事件;
    //与前面的的结合，可得出结论:只要view是可点击或可长按，则消费该事件.
    if (((viewFlags & CLICKABLE) == CLICKABLE ||
            (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_UP:
                boolean prepressed = (mPrivateFlags & PFLAG_PREPRESSED) != 0;
                if ((mPrivateFlags & PFLAG_PRESSED) != 0 || prepressed) {
                    boolean focusTaken = false;
                    if (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
                        focusTaken = requestFocus();
                    }
```

```
            if (prepressed) {
                setPressed(true, x, y);
            }

            if (!mHasPerformedLongPress) {
                //这是Tap操作，移除长按回调方法
                removeLongPressCallback();

                if (!focusTaken) {
                    if (mPerformClick == null) {
                        mPerformClick = new PerformClick();
                    }
                    //调用View.OnClickListener
                    if (!post(mPerformClick)) {
                        performClick();
                    }
                }
            }

            if (mUnsetPressedState == null) {
                mUnsetPressedState = new UnsetPressedState();
            }

            if (prepressed) {
                postDelayed(mUnsetPressedState,
                        ViewConfiguration.getPressedStateDuration());
            } else if (!post(mUnsetPressedState)) {
                mUnsetPressedState.run();
            }

            removeTapCallback();
        }
        break;
```

```
        case MotionEvent.ACTION_DOWN:
            mHasPerformedLongPress = false;

            if (performButtonActionOnTouchDown(event)) {
                break;
            }

            //获取是否处于可滚动的视图内
            boolean isInScrollingContainer = isInScrollingContainer();

            if (isInScrollingContainer) {
                mPrivateFlags |= PFLAG_PREPRESSED;
                if (mPendingCheckForTap == null) {
                    mPendingCheckForTap = new CheckForTap();
                }
                mPendingCheckForTap.x = event.getX();
                mPendingCheckForTap.y = event.getY();
                //当处于可滚动视图内，则延迟TAP_TIMEOUT，再反馈按压状态，用来判断用户是否想要
                postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout()
            } else {
                //当不再滚动视图内，则立刻反馈按压状态
                setPressed(true, x, y);
                checkForLongClick(0); //检测是否是长按
            }
            break;

        case MotionEvent.ACTION_CANCEL:
            setPressed(false);
            removeTapCallback();
            removeLongPressCallback();
            break;

        case MotionEvent.ACTION_MOVE:
            drawableHotspotChanged(x, y);
```

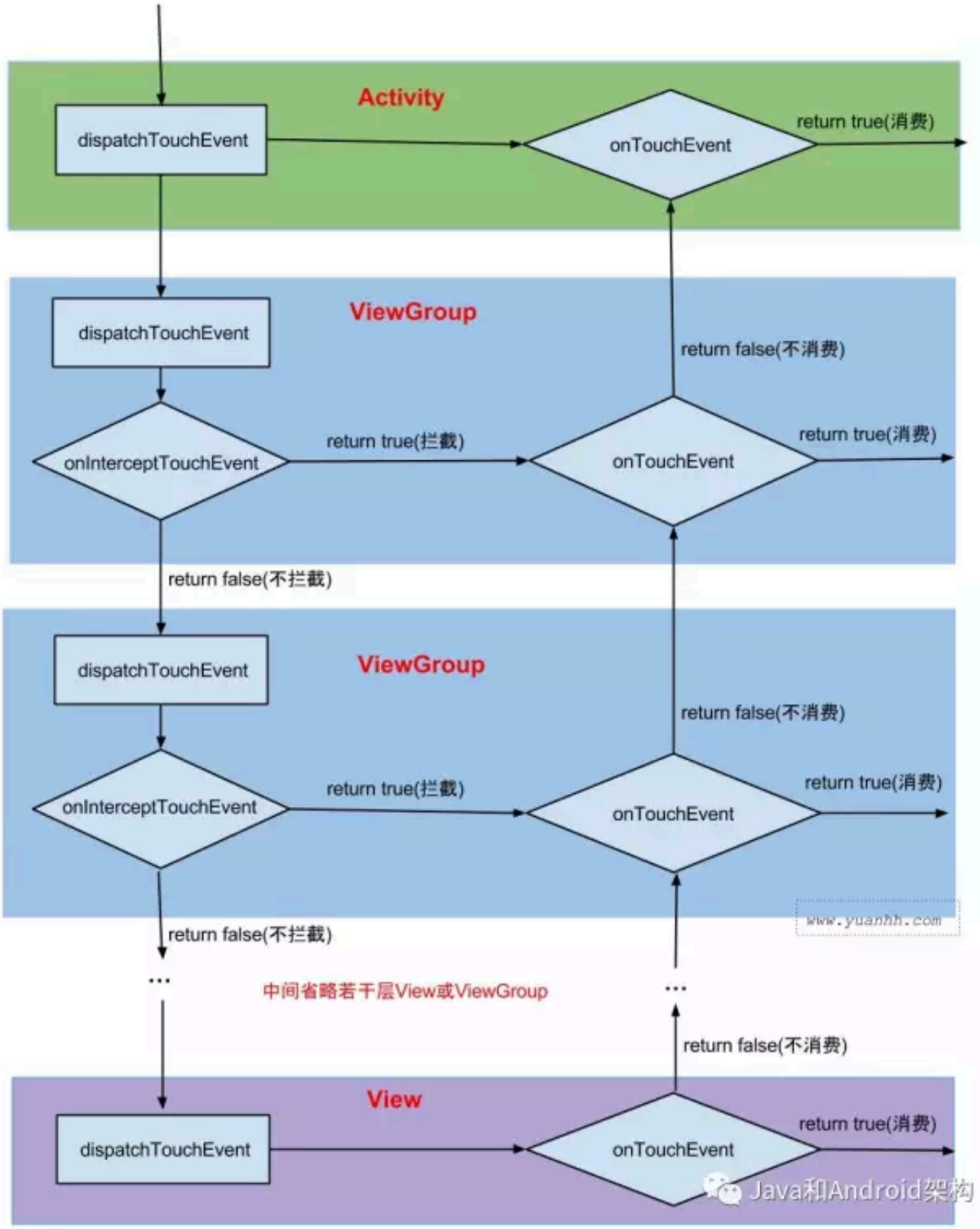


```
            if (!pointInView(x, y, mTouchSlop)) {
                removeTapCallback();
                if ((mPrivateFlags & PFLAG_PRESSED) != 0) {
                    removeLongPressCallback();
                    setPressed(false);
                }
            }
            break;
        }

        return true;
    }
    return false;
}
```


三. 总结

事件分发流程图:



- onInterceptTouchEvent返回值true表示事件拦截， onTouch/onTouchEvent 返回值true表示事件消费。
- 触摸事件先交由Activity.dispatchTouchEvent。再一层层往下分发，当中间的ViewGroup都不拦截时，进入最底层的View后，开始由最底层的OnTouchEvent来处理，如果一直不消费，则最后返回到

Activity.OnTouchEvent。

- ViewGroup才有onInterceptTouchEvent拦截方法。在分发过程中，中间任何一层ViewGroup都可以直接拦截，则不再往下分发，而是交由发生拦截操作的ViewGroup的OnTouchEvent来处理。
- 子View可调用requestDisallowInterceptTouchEvent方法，来设置disallowIntercept=true，从而阻止父ViewGroup的onInterceptTouchEvent拦截操作。
- OnTouchEvent由下往上冒泡时，当中间任何一层的OnTouchEvent消费该事件，则不再往上传递，表示事件已处理。
- 如果View没有消费ACTION*DOWN事件，则之后的ACTION*MOVE等事件都不会再接收。
- 只要View.onTouchEvent是可点击或可长按，则消费该事件.
- onTouch优先于onTouchEvent执行，上面流程图中省略，onTouch的位置在onTouchEvent前面。当onTouch返回true,则不执行onTouchEvent,否则会执行onTouchEvent。onTouch只有View设置了OnTouchListener，且是enable的才执行该方法。