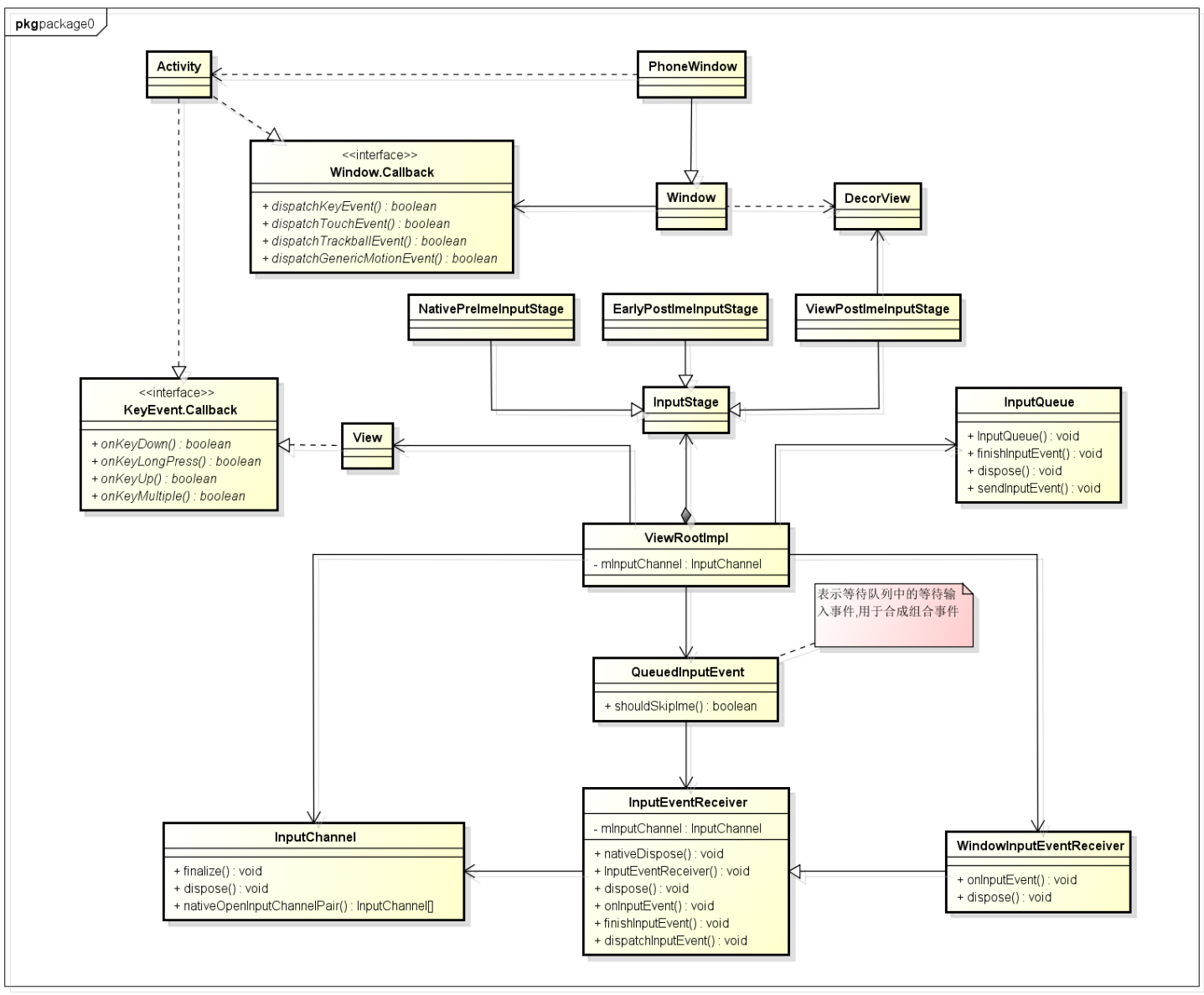
Input在APP层派发流程分析

Last edited by **caoquanli** 1 month ago

Input在APP层派发流程分析

一、概述

- Android系统的Input事件分发流程是从Native层开始的，经过InputEventReceiver接收事件，然后交给ViewRootImpl向下分发，本文介绍的是事件经ViewRootImpl向下分发的过程
- ViewRootImpl是其他View树的根，但它不是View，它的作用有以下几点：
    - 实现了View与WindowManager之间的通信
    - 完成View的绘制过程，包括measure、layout、draw过程
    - 向DecorView分发收到的用户发起的event事件，如按键，触屏等事件
- Input事件在传递到ViewRootImpl以后的类图关系如图所示：



二、ViewRootImpl事件分发流程

2.1 在ViewRootImpl的setView方法中创建了WindowInputEventReceiver，并通过WindowManagerService向InputManagerService注册InputChannel监听输入事件。WindowInputEventReceiver是ViewRootImpl的一个内部类，继承自InputEventReceiver,用于接收输入事件，在接收到事件后，调用onInputEvent方法，该方法又调用了ViewRootImpl类中的enqueueInputEvent方法，代码如下：

```
    final class WindowInputEventReceiver extends InputEventReceiver {
        public WindowInputEventReceiver(InputChannel inputChannel, Looper looper) {
            super(inputChannel, looper);
        }

        @Override
        public void onInputEvent(InputEvent event, int displayId) {
            enqueueInputEvent(event, this, 0, true);
        }
```

```
        ......
    }
```

2.2 在WindowInputEventReceiver中接收到事件之后调用的enqueueInputEvent实现如下，从代码中我们可以看到enqueueInputEvent将当前输入事件，添加到队列中，QueuedInputEvent相当于一个链表，enqueueInputEvent方法的 最后一个参数为processImmediately 是否立即处理，如果为true则会直接开始处理，直接调用doProcessInputEvents方法，否则会顺序压入主线程的MessageQueue中，对于压入MessageQueue的事件处理，最终依旧会通过ViewRootImpl中的ViewRootHandler再次调用到 doProcessInputEvents();

```
    void enqueueInputEvent(InputEvent event,
            InputEventReceiver receiver, int flags, boolean processImmediately) {
        adjustInputEventForCompatibility(event);
        QueuedInputEvent q = obtainQueuedInputEvent(event, receiver, flags);
        ......
        QueuedInputEvent last = mPendingInputEventTail;
        if (last == null) {
            mPendingInputEventHead = q;
            mPendingInputEventTail = q;
        } else {
            last.mNext = q;
            mPendingInputEventTail = q;
        }
        mPendingInputEventCount += 1;
        Trace.traceCounter(Trace.TRACE_TAG_INPUT, mPendingInputEventQueueLengthCounterName,
                mPendingInputEventCount);

        if (processImmediately) {
            doProcessInputEvents();
        } else {
            scheduleProcessInputEvents();
        }
    }
```

2.3 然后通过以下调用：doProcessInputEvents()–>deliverInputEvent(q)，在doProcessInputEvents()方法中，只要mPendingInputEventHead!=null即当前待处理事件队列还有事件需要去被处理掉,就一直循环调用deliverInputEvent()，具体处理逻辑如下:

```
void doProcessInputEvents() {
        // Deliver all pending input events in the queue.
        while (mPendingInputEventHead != null) {
            QueuedInputEvent q = mPendingInputEventHead;
            mPendingInputEventHead = q.mNext;
            if (mPendingInputEventHead == null) {
                mPendingInputEventTail = null;
            }
            q.mNext = null;

            mPendingInputEventCount -= 1;
            Trace.traceCounter(Trace.TRACE_TAG_INPUT, mPendingInputEventQueueLengthCounterN
                    mPendingInputEventCount);

            long eventTime = q.mEvent.getEventTimeNano();
            long oldestEventTime = eventTime;
            if (q.mEvent instanceof MotionEvent) {
                MotionEvent me = (MotionEvent)q.mEvent;
                if (me.getHistorySize() > 0) {
                    oldestEventTime = me.getHistoricalEventTimeNano(0);
                }
            }
            mChoreographer.mFrameInfo.updateInputEventTime(eventTime, oldestEventTime);

            deliverInputEvent(q);
        }
```

2.4 下面我们来看一下deliverInputEvent方法的实现，在该函数的方法体中实例化了InputStage对象，并通过两个判断:q.shouldSendToSynthesizer和q.shouldSkipIme选择相关的InputStage链的头,这里的InputStage是一个处理输入事件责任链，它是一个基类，提供一系列处理输入事件的方法，而具体的处理则是看它的实现类，每种实现类可以处理一定类型的输入事件。最终调用会走到：stage.deliver(q)，此处的stage即为经过判断后初始化的InputStage，为了理清该处理链的关系，在2.5节中进行进一步的分析

```
private void deliverInputEvent(QueuedInputEvent q) {
    Trace.asyncTraceBegin(Trace.TRACE_TAG_VIEW, "deliverInputEvent",
            q.mEvent.getSequenceNumber());
    if (mInputEventConsistencyVerifier != null) {
        mInputEventConsistencyVerifier.onInputEvent(q.mEvent, 0);
    }

    InputStage stage;
    if (q.shouldSendToSynthesizer()) {
        stage = mSyntheticInputStage;
    } else {
        stage = q.shouldSkipIme() ? mFirstPostImeInputStage : mFirstInputStage;
    }

    if (stage != null) {
        stage.deliver(q);
    } else {
        finishInputEvent(q);
    }
}
```

2.5 为了理清该处理链的关系,我们需要先看一下在ViewRootImpl的setView方法中构建的7个InputStage代码：

```
public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
    synchronized (this) {
        if (mView == null) {
            mView = view;
            ......
            // Set up the input pipeline.
            CharSequence counterSuffix = attrs.getTitle();
            mSyntheticInputStage = new SyntheticInputStage();
            InputStage viewPostImeStage = new ViewPostImeInputStage(mSyntheticInputStage);
            InputStage nativePostImeStage = new NativePostImeInputStage(viewPostImeStage,
                    "aq:native-post-ime:" + counterSuffix);
            InputStage earlyPostImeStage = new EarlyPostImeInputStage(nativePostImeStage);
            InputStage imeStage = new ImeInputStage(earlyPostImeStage,
                    "aq:ime:" + counterSuffix);
            InputStage viewPreImeStage = new ViewPreImeInputStage(imeStage);
            InputStage nativePreImeStage = new NativePreImeInputStage(viewPreImeStage,
                    "aq:native-pre-ime:" + counterSuffix);

            mFirstInputStage = nativePreImeStage;
            mFirstPostImeInputStage = earlyPostImeStage;
            mPendingInputEventQueueLengthCounterName = "aq:pending:" + counterSuffix;
        }
    }
}
```

可以发现他们依次以前面的一个InputStage为参数，它们依次构成一个输入事件的处理链，如果本阶段对时间没有处理，则传递到下一个对象处理，直到事件被处理。此处的stage应该被赋值为mFirstInputStage即NativePreInputStage。 NativePreImeInputStage -> ViewPreImeInputStage -> ImeInputStage ->EarlyPostImeInputStage -> NativePostImeInputStage -> ViewPostImeInputStage -> SyntheticInputStage

该7个事件处理Stage都是继承自抽象类InputStage。该抽象类中有一个mNext字段，从而可以构成一个链表结构。用来顺序处理接收到的事件,InputStage是一个抽象类其实现可以组成一个单向链表结构，实现如下:

```
abstract class InputStage {
        private final InputStage mNext;

        protected static final int FORWARD = 0;
        protected static final int FINISH_HANDLED = 1;
        protected static final int FINISH_NOT_HANDLED = 2;
        ......
        public InputStage(InputStage next) {
            mNext = next;
        }
        ......
        public final void deliver(QueuedInputEvent q) {
            if ((q.mFlags & QueuedInputEvent.FLAG_FINISHED) != 0) {
                forward(q);
            } else if (shouldDropInputEvent(q)) {
                finish(q, false);
            } else {
```

```
            apply(q, onProcess(q));
        }
    }
    ......
    protected void finish(QueuedInputEvent q, boolean handled) {
        q.mFlags |= QueuedInputEvent.FLAG_FINISHED;
        if (handled) {
            q.mFlags |= QueuedInputEvent.FLAG_FINISHED_HANDLED;
        }
        forward(q);
    }
    ......
    protected void forward(QueuedInputEvent q) {
        onDeliverToNext(q);
    }
    ......
    protected void apply(QueuedInputEvent q, int result) {
        if (result == FORWARD) {
            forward(q);
        } else if (result == FINISH_HANDLED) {
            finish(q, true);
        } else if (result == FINISH_NOT_HANDLED) {
            finish(q, false);
        } else {
            throw new IllegalArgumentException("Invalid result: " + result);
        }
    }
    ......
    protected int onProcess(QueuedInputEvent q) {
        return FORWARD;
    }
    ......
}
```

deliver()方法先判断该事件对象是否已经处理完成或者需要抛弃掉,都不满足则调用onProcess()处理该事件对象,处理完成后返回处理结果给apply()方法后续工作,根据onProcess()返回处理结果是否把事件传递给其mNext指向的下一个InputStage去处理;根据多态性，onProcess方法的具体处理逻辑要在其子类中进行实现，具体实现过程如图所示：

此图片来源于 http://newandroidbook.com/files/AndroidInput.pdf

2.6 当一个InputEvent到来时，ViewRootImpl会寻找合适它的InputStage来处理。从ViewRootImpl.setView()方法中可知,一共生成了7个InputStage的子类对象依次接龙按事件类型对应去处理,入口是NativePreImeInputStage该子类对象,代码如下：

```
@Override
protected int onProcess(QueuedInputEvent q) {
    if (mInputQueue != null && q.mEvent instanceof KeyEvent) {
        mInputQueue.sendInputEvent(q.mEvent, q, true, this);
        return DEFER;
    }
    return FORWARD;
}
```

2.7 对于一个输入事件，NativePreImeInputStage的onProcess()返回FORWARD,即交给其mNext即ViewPreImeInputStage去接龙处理，ViewPreImeInputStage.onProcess()同样返回FORWARD,交给其其mNext即ViewPreImeInputStage去接龙处理ImeInputStage去处理，依次向下传递，最终ViewPostImeInputStage可以处理它，ViewPostImeInputStage中，ViewPostImeInputStage类中的onProcess方法如下：

```
@Override
 protected int onProcess(QueuedInputEvent q) {
     if (q.mEvent instanceof KeyEvent) {
         return processKeyEvent(q);
```

```
    } else {
        final int source = q.mEvent.getSource();
        if ((source & InputDevice.SOURCE_CLASS_POINTER) != 0) {
            return processPointerEvent(q);
        } else if ((source & InputDevice.SOURCE_CLASS_TRACKBALL) != 0) {
            return processTrackballEvent(q);
        } else {
            return processGenericMotionEvent(q);
        }
    }
 }
```

2.8 当onProcess被回调时，processKeyEvent、processPointerEvent、processTrackballEvent、processGenericMotionEvent至少有一个方法就会被调用，这些方法都是属于ViewPostImeInputStage的，在这些方法中，都会调用View类的方法，如processPointerEvent方法代码如下：

```
private int processPointerEvent(QueuedInputEvent q) {
    final MotionEvent event = (MotionEvent)q.mEvent;

    ......
    boolean handled = mView.dispatchPointerEvent(event);
    ......
}
```

2.9 在2.5节ViewRootImpl.setView的代码中，我们可以发现，mView的实例化是在setView中完成的，而setView方法中传入的View参数是DecorView,由于DecroView和其父类FrameLayout,ViewGroup均没有重写此方法,故会调用View.dispatchPointerEvent方法，代码如下：

```
public final boolean dispatchPointerEvent(MotionEvent event) {
    if (event.isTouchEvent()) {
        return dispatchTouchEvent(event);
    } else {
        return dispatchGenericMotionEvent(event);
    }
}
```

会调用到DecorView中的dispatchTouchEvent(event)，这样一来，ViewPostImeInputStage将事件分发到了DecorView中，下面来看一下DecorView类中的dispatchKeyEvent方法

```
@Override
 public boolean dispatchTouchEvent(MotionEvent ev) {
     final Window.Callback cb = mWindow.getCallback();
     return cb != null && !mWindow.isDestroyed() && mFeatureId < 0
             ? cb.dispatchTouchEvent(ev) : super.dispatchTouchEvent(ev);
 }
```

从上述代码中我们可以看出，DecorView.dispatchTouchEvent()方法最终又会调用cb.dispatchTouchEvent(ev)，在Activity.attach()中已经把自己设置赋值到DecroView的外部类Window的Callback mCallback成员变量且在PhoneWindow生成DecroView对象的时候传入的mFeatureId=-1，所以这里调用了Acitivity.dispatchTouchEvent()去处理

2.10 Input事件在ViewRootImpl传递过程中各类之间的时序图如图所示：

三、Activity事件分发流程

3.1 通过上面的分析可知事件在ViewRootImpl层层传递，最终传递到了Activity，最终进入到Activity中的dispatchTouchEvent（MotionEvent ev）和dispatchKeyEvent(event)实现如下

```
public boolean dispatchTouchEvent(MotionEvent ev) {
   if (ev.getAction() == MotionEvent.ACTION_DOWN) {
```

```
        onUserInteraction();
    }
    if (getWindow().superDispatchTouchEvent(ev)) {
        return true;
    }
    return onTouchEvent(ev);
}

public boolean dispatchKeyEvent(KeyEvent event) {
    onUserInteraction();

    // Let action bars open menus in response to the menu key prioritized over
    // the window handling it
    if (event.getKeyCode() == KeyEvent.KEYCODE_MENU &&
            mActionBar != null && mActionBar.onMenuKeyEvent(event)) {
        return true;
    }

    Window win = getWindow();
    if (win.superDispatchKeyEvent(event)) {
        return true;
    }
    View decor = mDecor;
    if (decor == null) decor = win.getDecorView();
    return event.dispatch(this, decor != null
            ? decor.getKeyDispatcherState() : null, this);
}
```

- 通过以上代码，我们可以看到事件传递进来会首先进入到Activity的dispatchTouchEvent（）当所有Activity下的View不消费该次事件时会回调到Activity的 onTouchEvent(ev)
- 同样的，dispatchKeyEvent（）中在所有子View不消费该事件时会调用 **event.dispatch(this, decor != null? decor.getKeyDispatcherState() : null, this)**; 最终调用到Activity的onKeyDown（）onKeyUp（）等方法

3.2 由上述代码可知：PhoneWindow的superDispatchTouchEvent（）和superDispatchKeyEvent(event)中都直接又重新回到DecorView中调用了DecorView中的相关分发方法，实现如下：

```
@Override
public boolean superDispatchTouchEvent(MotionEvent event) {
  return mDecor.superDispatchTouchEvent(event);
}
 @Override
public boolean superDispatchKeyEvent(KeyEvent event) {
  return mDecor.superDispatchKeyEvent(event);
}
```

3.3 DecorView的superDispatchTouchEvent(event)和superDispatchKeyEvent(event)相关实现如下，其最终会调用到View的dispatchKeyEvent（event）和dispatchTouchEvent(event)：

```
public boolean superDispatchTouchEvent(MotionEvent event) {
        return super.dispatchTouchEvent(event);
    }

public boolean superDispatchKeyEvent(KeyEvent event) {
        // Give priority to closing action modes if applicable.
        if (event.getKeyCode() == KeyEvent.KEYCODE_BACK) {
            final int action = event.getAction();
            // Back cancels action modes first.
            if (mPrimaryActionMode != null) {
                if (action == KeyEvent.ACTION_UP) {
                    mPrimaryActionMode.finish();
                }
                return true;
            }
        }

        return super.dispatchKeyEvent(event);
    }
```

3.4 由于DecorView继承自FrameLayout，是一个ViewGroup，至此开始调用ViewGroup的事件分发逻辑, 也就进入了我们常见的ViewGroup的事件分发分析

四、ViewGroup事件分发流程

4.1 我们以点击事件为例简单说明一下ViewGroup的大致事件分发流程，当Activity接收到点击事件之后，会通过DectorView调用ViewGroup的dispatchTouchEvent方法，由于该方法的源码太长，所以分段说明。

```
// Check for interception.
   final boolean intercepted;
   if (actionMasked == MotionEvent.ACTION_DOWN
           || mFirstTouchTarget != null) {
       final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
       if (!disallowIntercept) {
           intercepted = onInterceptTouchEvent(ev);
           ev.setAction(action); // restore action in case it was changed
       } else {
           intercepted = false;
       }
   } else {
       // There are no touch targets and this action is not an initial down
       // so this view group continues to intercept touches.
       intercepted = true;
   }
```

从上面的代码可以看出，ViewGroup在两种情况下会判断是否拦截了当前事件：事件类型为ACTION_DOWN（即）或者mFirstTouchTarget != null,当事件类型为ACTION_DOWN时，表示当前有新的Touch事件序列到来，则需要重置触摸状态;当mFirstTouchTarget != null时，我们可以从后面的代码可以看出，当ViewGroup不拦截事件并将事件交给子元素处理时，mFirstTouchTarget会被赋值，此时mFirstTouchTarget != null，这样当ACTION_MOVE和ACTION_UP事件到来时，并且事件已被分发出去，intercepted值（用来标识事件是否由该ViewGroup所代表的View进行拦截，如果拦截则其子View不会收到后续的其他事件） 4.2 当Event的Action不为DOWN且mFirstTouchTarget==null时候），对应场景为：然后当该事件是一个序列中的非初始事件（非ActionDown），且之前的事件并没有被任何子View消费，则此时intercept 置为 true；

- 当intercept为true或者当前事件为Cancle时，跳过 4.3和4.4 直接到达4.5处执行
- 当intercept为false时且当前事件为不是Cancel时，如果当前事件的Action不为ACTION_DOWN时，也跳过 4.3和4.4 直接到达4.5处执行，否则即当前事件是ACTION_DOWN，且当前ViewGroup并不拦截、处理该事件，则会执行遍历子View把该事件依次分发到子View上，直到找到目标View
- 当intercept为false且当前事件的Action为ACTION_DOWN时则执行 顺序执行4.3和4.4

4.3 当ViewGroup不拦截事件的时候，事件会向下分发交由它的子View进行处理，首先遍历ViewGroup的所有子元素，然后判断子元素是否能够收到点击事件。是否能够收到点击事件主要由两点来衡量：子元素是否在播放动画和点击事件的坐标是否落在子元素的区域内。如果某个子元素满足这两个条件，那么事件就会传递给它处理，dispatchTransformedTouchEvent这个方法实际上就是调用子元素的dispatchTouchEvent方法，由于上面传递child不是null，因此它会直接调用子元素的dispatchTouchEvent方法，这样事件就交由子元素处理，从而完成了一轮事件分发。

4.4 在遍历子View中，如果找到目标View，调用dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign) 分发事件到子View，如果该子View没有消费则依次调用下一个，如果返回为true即该子View消费了该事件，直接break，跳出循环，此过程还涉及以下代码：

- newTouchTarget = addTouchTarget(child, idBitsToAssign);//调用addTouchTarget（）此方法内部会赋值mFirstTouchTarget，并且该TouchTarget是一个链表结构
- alreadyDispatchedToNewTouchTarget = true;//标记已经找到消费该事件的子View，并已经分发

4.5 如果遍历完毕或者没有进入遍历逻辑，然后没有子View消费该事件，mFirstTouchTarget==null，则调用dispatchTransformedTouchEvent()，此时childView为null，会直接调用Super.dispatchTouchEvent（）方法,此时逻辑进入View的dispatchTouchEvent（）

4.6 如果遍历完毕或者没有进入遍历逻辑，但是如果之前mFirstTouchTarget已经赋值不为null，则后续事件下发代码直接执行到此处，即直接找到之前消费过该事件序列的View进行分发

五、View的事件分发流程

View对点击事件的处理过程就比较简单了，因为View是一个单独的元素，它没有子元素因此无法向下传递事件，所以它只能自己处理事件，所以View的onTouchEvent方法默认返回true。对于View的事件分发逻辑，简单调用顺序如下：dispatchTouchEvent（）–>onTouchEvent()–>performClick()–>OnclickListener.onClick(this)

5.1 首先来看它的dispatchTouchEvent方法，由于源码太长，在此不再粘出，从源码中可以看出，dispatchTouchEvent方法首先会判断有没有设置OnclickListener，如果OnTouchListener中的onTouch方法返回true，那么onTouchEvent方法就不会被调用，由此可知OnTouchListener的优先级要高于onTouchEvent。

5.2 在onTouchEvent()方法中，可以看出只要CLICKABLE和LONG_CLICKABLE有一个为true，则该View就会消耗这个事件，因为返回了true，当ACTION_UP事件发生时就会调用performClick方法，如果View设置了OnclickListener,那么performClick方法就会调用其onClick方法。

```
public boolean performClick() {
    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);

    ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        return true;
    }

    return false;
}
```

5.3 View的LONG_CLICKABLE属性默认为false，而CLICKABLE属性默认为true，不过具体的View的CLICKABLE又不一定，确切来说是可点击的View其CLICKABLE属性true，比如Botton，不可点击的View的CLICKABLE为false，比如TextView。。通过setClickable和setLongClickable可以设置这两个属性。另外setOnClickListener和setOnLongClickListener会自动将View的这两个属性设为true。这一点从源码可以看出来。

```
public void setOnClickListener(OnClickListener l) {
    if (!isClickable()) {
        setClickable(true);
    }
    getListenerInfo().mOnClickListener = l;
}
```

```
public void setOnLongClickListener(OnLongClickListener l) {
    if (!isLongClickable()) {
        setLongClickable(true);
    }
    getListenerInfo().mOnLongClickListener = l;
}
```

5.4 输入事件在传入Activity以后，其分发过程可以简单总结为以下图标形式：