View事件分发及源码分析

linwq8 于 2024-12-22 16:41:30 发布

概念

事件的分发主要是对MotionEvent的分发，当产生一个EotionEvent事件，需要把这个事件分发到某个具体的View进行处理。
备注：以下主要是针对触摸的事件源进行分析，源码是基于7.1版本

事件源类型

1. PointerEvent：是一种指针事件，它主要用于处理多点触控以及新型输入设备（如触控笔、触摸屏等）的输入。
2. TrackballEvent:是一种轨迹球事件，它主要用于处理轨迹球（一种小型球体，通过滚动来输入方向）的输入。
3. GenericMotionEvent:是一种通用运动事件，它主要用于处理非标准触控输入设备（如游戏手柄、鼠标、滚轮、触控板等）的输入。

MotionEvent事件类型

事件类型	说明
ACTION_DOWN	触摸屏幕时触发一次
ACTION_MOVE	在屏幕上移动时，会多次触发
ACTION_UP	离开屏幕时触发一次
ACTION_CANCE	取消事件

事件涉及的主要方法

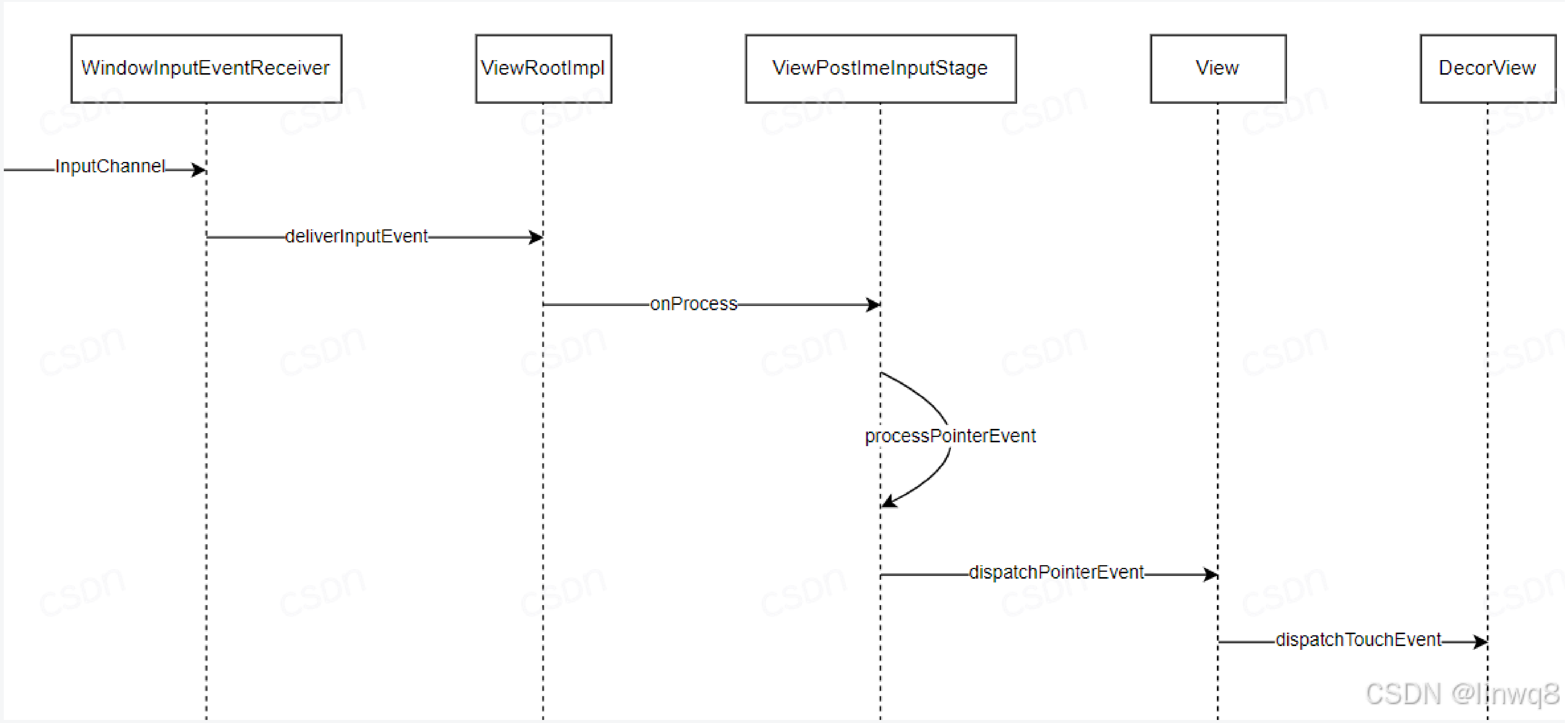
- public boolean dispatchTouchEvent(MotionEvent e)
  用来进行事件的分发。如果事件能传到ViewGroup或者View,该方法会 被调用。返回值受当前ViewGroup或则View的onTouchEvent()影响、或者下级View的dispatchTouchEvent()影响。
- public boolean onInterceptTouchEvent(MotionEvent e)
  用来拦截事件。主要作用于ViewGroup，如果当前ViewGroup拦截了事件，则后续同个系列的事件该方法不会再被调用。返回值表示是否拦截。

- -public boolean onTouchEvent(MotionEvent e)
  处理触摸事件。返回结果表示是否消耗事件，如果不消耗，后续同个系列事件不会再触发执行。

事件分发流程及源码分析

流程：事件流程顺序是：Activity—>Window—>DecorView—>具体View及ViewGroup

1. 事件分发到Activity



- 点击屏幕后经过一系列传递到WindowInputEventReceiver.java中，该类是ViewRootImpl的内部类。

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

        @Override
        public void onInputEvent(InputEvent event) {
            //将事件入队
            enqueueInputEvent(event, this, 0, true);
        }
```
AI助手

- 调用ViewRootImpl中的enqueueInputEvent()入队，doProcessInputEvents()进行事件处理，deliverInputEvent(q)分发事件。

```
void enqueueInputEvent(InputEvent event,
            InputEventReceiver receiver, int flags, boolean processImmediately) {
        adjustInputEventForCompatibility(event);
        //事件入队
        QueuedInputEvent q = obtainQueuedInputEvent(event, receiver, flags);

        //.....代码省略
        if (processImmediately) {
            //处理和分发事件
            doProcessInputEvents();
        } else {
            scheduleProcessInputEvents();
        }
    }
```
AI助手

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

AI助手

```
    private void deliverInputEvent(QueuedInputEvent q) {

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

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

AI助手

- 调用InputStage中的onProcess(),由其子类ViewPostImeInputStag重写执行，在onProcess中根据事件类型进行处理。

```
final class ViewPostImeInputStage extends InputStage {
        public ViewPostImeInputStage(InputStage next) {
            super(next);
        }

        @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);
                }
            }
```

AI助手

- 屏幕滑动事件处理processPointerEvent(q),调用DecorView进行分发

```
private int processPointerEvent(QueuedInputEvent q) {
            final MotionEvent event = (MotionEvent)q.mEvent;
            ...
            //触摸点击View是DecorView
            final View eventTarget =
                    (event.isFromSource(InputDevice.SOURCE_MOUSE) && mCapturingView != null) ?
                            mCapturingView : mView;
            mAttachInfo.mHandlingPointerEvent = true;
            boolean handled = eventTarget.dispatchPointerEvent(event);
                        ...
            return handled ? FINISH_HANDLED : FORWARD;
        }
```

AI助手

- 事件会交由DecorView的dispatchPointerEvent()向下级View 进行分发处理，再调用DecorView的dispatchTouchEvent()进行分发事件，最后通过Window.Callback回调将事件传入Activity。

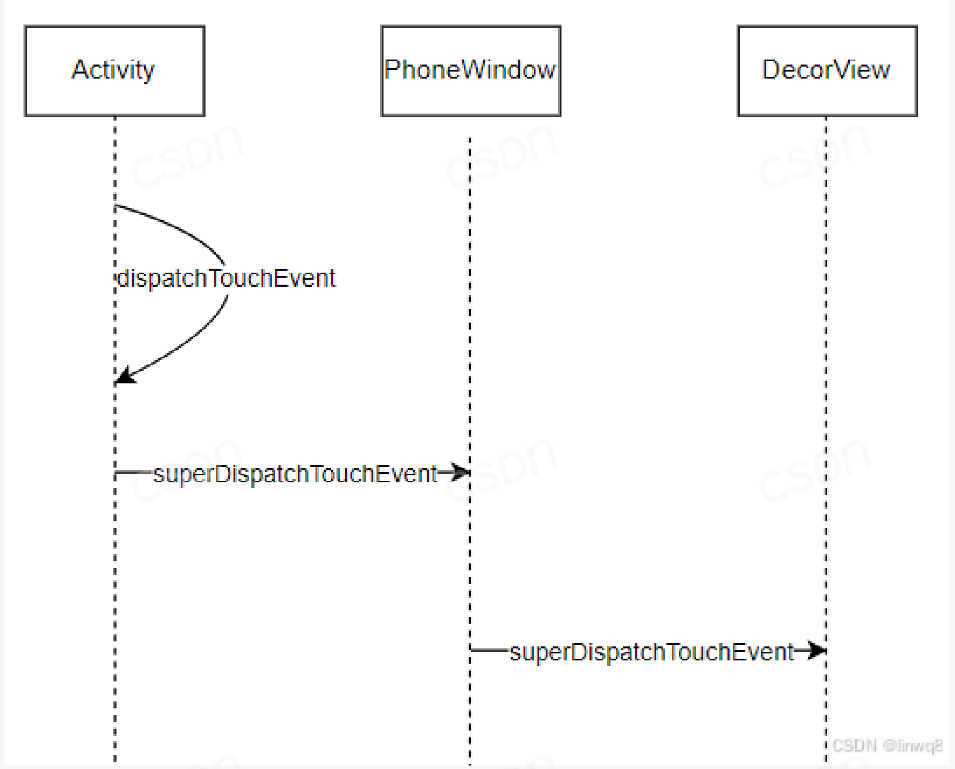
```java

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

```java
    @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);
    }
```
AI助手

2. Activity事件分发过程

- 由Activity内部的Window进行执行分发，具体执行由其实现类PhoneWindow实现，最后会由DecorView进行分发到顶层View。



```java
//Activity.java
public boolean dispatchTouchEvent(MotionEvent ev) {
    if (ev.getAction() == MotionEvent.ACTION_DOWN) {
        onUserInteraction();
    }
    //调用window进行分发
    if (getWindow().superDispatchTouchEvent(ev)) {
        return true;
    }
    //如果上面分发成功则Activity的onTouchEvent不会执行
    return onTouchEvent(ev);
}
```

AI助手

```java
//PhoneWindow.java
@Override
public boolean superDispatchTouchEvent(MotionEvent event) {
    //调用DecorView进行分发
    return mDecor.superDispatchTouchEvent(event);
}
```

AI助手

3. Activity事件分发过程

- 当前ViewGroup先判断是否需要拦截事件。

```
    //ViewGroup中的dispatchTouchEvent(MotionEvent ev) 方法
if (onFilterTouchEventForSecurity(ev)) {
    ...
    final boolean intercepted;//是否拦截事件
    //mFirstTouchTarget!=null,表示由子View进行事件处理
    if (actionMasked == MotionEvent.ACTION_DOWN
|| mFirstTouchTarget != null) {
        final boolean disallowIntercept = (mGroupFlags & FLAG_DISALLOW_INTERCEPT) != 0;
        //是否允许拦截事件，可通过requestDisallowInterceptTouchEvent()进行设置
        /**设置后无法拦截除了DOWN事件的事件，因为ViewGroup最开始在分发事件的时候
           会重置标记。
        */
        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;
    }
}
```

AI助手

```
//DOWN事件会清除标记
if (actionMasked == MotionEvent.ACTION_DOWN) {
    // Throw away all previous state when starting a new touch gesture.
    // The framework may have dropped the up or cancel event for the previous gesture
    // due to an app switch, ANR, or some other state change.
     cancelAndClearTouchTargets(ev);
     resetTouchState();
}
```

AI助手

- 如果拦截事件的化，ViewGroup自己处理事件，判断是否有设置OnTouchListener，如果没有则会执行onTouchEvent。
- 如果不拦截则会将事件传递到下级View，循环执行直到事件被消耗。

```
    //不拦截情况
    if (actionMasked == MotionEvent.ACTION_DOWN
            || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
            || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {
    ...
        final int childrenCount = mChildrenCount;//子对象个数
        if (newTouchTarget == null && childrenCount != 0) {//子对象不为0
            final float x = ev.getX(actionIndex);
            final float y = ev.getY(actionIndex);
            ...
            //循环查找能处理事件的View
            final View[] children = mChildren;
            for (int i = childrenCount - 1; i >= 0; i--) {
                final int childIndex = getAndVerifyPreorderedIndex(
childrenCount, i, customOrder);
                final View child = getAndVerifyPreorderedView(
preorderedList, children, childIndex);
            ...
                //判断能否接收事件和计算(x, y)是否在此View的范围内
                if (!canViewReceivePointerEvents(child)
                        || !isTransformedTouchPointInView(x, y, child, null)) {

```

```
                ev.setTargetAccessibilityFocus(false);
                //不符合跳过
                continue;
            }
            //找到合适的child
            newTouchTarget = getTouchTarget(child);
            if (newTouchTarget != null) {
                // Child is already receiving touch within its bounds.
                // Give it the new pointer in addition to the ones it is handling.
                 newTouchTarget.pointerIdBits |= idBitsToAssign;
                 break;
            }

            resetCancelNextUpFlag(child);
            //事件继续传递，如果是ViewGroup则执行dispatchTouchEvent
            //如果是View则执行dispatchTouchEvent
            if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
            ...
                break;//事件执行完成

            }
        }
    }
}
```

AI助手

4. View事件分发过程

- 判断是否有设置了OnTouchListener，如果设置了且OnTouchListener.onTouch()返回true,则onTouchEvent()不会执行

```
public boolean dispatchTouchEvent(MotionEvent event) {
    ...
    if (onFilterTouchEventForSecurity(event)) {
        if ((mViewFlags & ENABLED_MASK) == ENABLED && handleScrollBarDragging(event)) {
                result = true;
        }
        //noinspection SimplifiableIfStatement
        ListenerInfo li = mListenerInfo;
        //判断是否有设置了OnTouchListener
        if (li != null && li.mOnTouchListener != null
                    && (mViewFlags & ENABLED_MASK) == ENABLED
                    && li.mOnTouchListener.onTouch(this, event)) {
            result = true;
        }
        //处理事件

        if (!result && onTouchEvent(event)) {
            result = true;
        }
    }
}
```

AI助手

- onTouchEvent()分析

```
public boolean onTouchEvent(MotionEvent event) {
    ...
    //View处于不可用状态也会消耗事件
    if ((viewFlags & ENABLED_MASK) == DISABLED) {
        if (action == MotionEvent.ACTION_UP && (mPrivateFlags & PFLAG_PRESSED) != 0) {
            setPressed(false);
        }
        // A disabled view that is clickable still consumes the touch
        // events, it just doesn't respond to them.
        return (((viewFlags & CLICKABLE) == CLICKABLE
                    || (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE)
                    || (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE);
```

```java
        }
        //代理设置
        if (mTouchDelegate != null) {
            if (mTouchDelegate.onTouchEvent(event)) {
                return true;
            }
        }
        if (((viewFlags & CLICKABLE) == CLICKABLE ||
                (viewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) ||
                (viewFlags & CONTEXT_CLICKABLE) == CONTEXT_CLICKABLE) {
            switch (action) {
                case MotionEvent.ACTION_UP:
                    ...
                    if (!mHasPerformedLongPress && !mIgnoreNextUpEvent) {
                        // This is a tap, so remove the longpress check
                        removeLongPressCallback();

                        // Only perform take click actions if we were in the pressed state
                        if (!focusTaken) {
                            // Use a Runnable and post this rather than calling
                            // performClick directly. This lets other visual state
                            // of the view update before click actions start.
                            if (mPerformClick == null) {
                                mPerformClick = new PerformClick();
                            }
                            if (!post(mPerformClick)) {
                               //有设置onClick会执行
                                performClick();
                            }
                            }
                            ...
                        }
                    break;
            }
       }
```

AI助手