Android 事件分发机制源码和实例解析

2016-02-12 安卓应用频道

(点击上方公众号，可快速关注)

> 作者：李狄青
> 链接：http://www.jianshu.com/p/7daf0feb6c2d

1. 事件分发过程的理解

1.1. 概述

事件主要有 down(MotionEvent.ACTION_DOWN)，
move（MotionEvent.ACTION_MOVE），up（MotionEvent.ACTION_UP）。

基本上的手势均由 down 事件为起点，up 事件为终点，中间可能会有一定数量的move 事件。这三种事件是大部分手势动作的基础。

事件和相关信息（比如坐标）封装成 MotionEvent。

大体的分发过程为：首先传递到 Activity，然后传给了 Activity 依附的 Window，接着由 Window 传给视图的顶层 View 也就是 DecorView，最后由 DecorView 向整个 ViewTree 分发。分发还会有回溯的过程。最后还会回到 Activity 的调用中。

Activity 的分发事件源码

```
public boolean dispatchTouchEvent(MotionEvent ev) {      if (ev.getAction() == MotionEvent.ACTION_DOWN) {
            onUserInteraction();
      }      if (getWindow().superDispatchTouchEvent(ev)) {            return true;
      }      return onTouchEvent(ev);
}
```

在 getWindow().superDispathTouchEvent 就是用来分发事件到 DecorView 中。如果整个 ViewTree 没有消费事件，会调用 Activity 的 onTouchEvent。

1.2. 主要方法

1.2.1. 概览

主要涉及到的 View 或 ViewGroup 的方法有：

dispatchTouchEvent，该方法封装了事件分发的整个过程。是事件分发的 调度者 和 指挥官 。的核心过程均在该方法中。下面的 onInterceptTouchEvent 和 onTouchEvent 的回调的调用就在该方法体中。是否传递事件到 onInterceptTouchEvent 和 onTouchEvent 由 dispatchTouchEvent 决定。

onInterceptTouchEvent，该方法决定了是否拦截事件。只有 ViewGroup 有该回调。返回 true 表示拦截，返回 false 表示不拦截。自定义 View 的时候，可以重载该方法，通过一些特定的逻辑来决定是否拦截事件。如果拦截，接下来会调用该 ViewGroup 的 onTouchEvent 来处理事件。

onTouchEvent，该方法处理了事件，并决定是否继续消费后续事件。该方法调用的前置条件：

- 该 View 拦截了事件
- 子 View 都不消费事件
- 没有子 View

该方法正式处理 MotionEvent。返回 true 表示消费，返回 false 不消费。如果消费，接下来的事件还会传递到该 View 的 dispatchTouchEvent 中；如果不消费，后面的事件不会再传过来。

onTouchListener 的 onTouch 回调，和 onTouchEvent 一样，优先级比 onTouchEvent 高，如果有设置该监听，并且 onTouch 返回 true，就不会再调用 onTouchEvent 了。如果返回 false，事件还是会传递到 onTouchEvent 中。

1.2.2. dispatchTouchEvent 方法中的一些细节处理：

大部分手势的起点为 down 事件，dispatchTouchEvent 如果收到 down 事件，会重新设置一些变量和标记

重置变量和标记的源码

```
// Handle an initial down.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();
}
```

实际的源码中，ViewGroup 继承于 View。 当子 View 不消费事件或者 ViewGroup 拦截了事件会传空值到 dispatchTransformedTouchEvent 中，内部会调用 super.dispatchTouchEvent，最终把事件传给 onTouchEvent 进行处理。

dispatchTransformedTouchEvent 关键部分

```
// Perform any necessary transformations and dispatch.if (child == null) {
      handled = super.dispatchTouchEvent(transformedEvent);
} else {      final float offsetX = mScrollX - child.mLeft;      final float offsetY = mScrollY - child.mTop;
      transformedEvent.offsetLocation(offsetX, offsetY);      if (! child.hasIdentityMatrix()) {
            transformedEvent.transform(child.getInverseMatrix());
      }

      handled = child.dispatchTouchEvent(transformedEvent);
}
```

也就是 dispatchTransformTouchEvent 完成了分发的最后过程：

a. 传入的 child 不为空，转化坐标为 child 的坐标系，调用 child.dispatchTouchEvent向 child 分发事件

b. 传入的 child 为空，调用 super.dispatchTouchEvent 分发事件到 onTouchEvent 中

1.2.3 方法的主要关系

对于一个 ViewGroup 来说，几个重要方法的关系如下

几个重要方法关系伪代码

```
public boolean dispatchTouchEvent(MotionEvent e) {      boolean consumed = false;      if (onInterceptTou
chEvent(e)) {
            consumed = onTouchEvent(e);
      } else {             for (View view: childs) {
                  consumed = view.dispatchTouchEvent(e);                    if (consumed) {
        break;
                  }
            }             if (!consumed) {
                  consumed = onTouchEvent(e);
            }
      }
      return consumed;
}
```

这是事件分发过程的简单描述，具体远比这复杂的多。

1.3. 核心行为

View 或 ViewGroup 有两个核心的行为：拦截(intercept) 和 消费(consume)。这两者是相互独立的，拦截不一定消费。是否要拦截看 onIntercepTouchEvent。是否要消费看 onTouchEvent。

注意：是否拦截还有其他因素影响。如果不是 down 事件，并且 mFirstTouchTarget 为空值，就会直接拦截事件。

在 dispatchTouchEvent 中有这样的代码

拦截的关键源码

```
// Check for interception.final boolean intercepted;if (actionMasked == MotionEvent.ACTION_DOWN
            || mFirstTouchTarget != null) {          final boolean disallowIntercept = (mGroupFlags & FL
AG_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;
}
```

从上面的源码可以看出，在不是 down 事件，并且 mFirstTouchTarget 为空的情况下，不会走 onInterceptTouchEvent 而是直接拦截。如果满足了，还会
看 FLAG_DISALLOW_INTERCEPT 标记，如果不允许拦截(disallowIntercept 为 true)，也不会走onInterceptTouchEvent，直接标记不拦截。

处理调用 onTouchEvent 的源码

```
boolean result = false;

...

ListenerInfo li = mListenerInfo;if (li != null && li.mOnTouchListener != null
                && (mViewFlags & ENABLED_MASK) == ENABLED
            && li.mOnTouchListener.onTouch(this, event)) {
      result = true;
}if (!result && onTouchEvent(event)) {
      result = true;
}
```

可以看出，在该 View 为 ENABLE 的状态并且有 mTouchListener，会先调用 onTouch。在 onTouch 返回 false 时才会继续调用 onTouchEvent。

onTouch 或者 onTouchEvent 的处理结果有：

- 返回 true，会继续消费后续事件。意味着，后面的事件将会继续传递到该 View 的 dispatchTouchEvent 方法中进行调度。父 View 会为该 View 创建一个 TouchTarget 实例加入链表中，链表的第一项为 mFirstTouchTarget。后续的 move 和 up 事件会直接交给该 View 的 dispatchTouchEvent。
- 返回 false，不再消费后续事件。意味着，后面的事件将会被父 View 拦截，而不再传递

下来。

1.4. 特殊情况

比较特殊的情况有，子 View 可以使用 requestDisallowInterceptTouchEvent 影响去父 View 的分发，可以决定父 View 是否要调用 onInterceptTouchEvent 。比如，requestDisallowInterceptTouchEvent(true)，父 View 就不用调用 onInterceptTouchEvent 来判断拦截，而就是不拦截。

该方法可以用来解决手势冲突。比如子 View 先消费了事件，但是后面父 View 也满足了手势触发的条件而拦截事件，导致子 View 手势执行一半后无法继续响应。可以使用 requestDisallowInterceptTouchEvent(true)，这样后面的事件，父 View 不会走 onInterceptTouchEvent 回调来判断是否要拦截事件，而是直接把事件继续传下来。

2. 案例分析

下面举三个简单的例子，三个类 ViewGroup1st，ViewGroup2nd 和 View0，层级关系为

```
<ViewGroup1st>
      <ViewGroup2nd>
            <View0 />
      </ViewGroup2nd></ViewGroup1st>
```

这三个类有两层 ViewGroup，最底层为 View，这几个例子主要理解 消费 行为，所以不做事件的拦截。

2.1. 案例1：均不消费 down 事件

在触摸屏幕中 View0 的区域后，输出 log 信息如下

```
12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup1st: dispatchTouchEvent before12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup1st: onInterceptTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup2nd: dispatchTouchEvent before12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup2nd: onInterceptTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/View0: dispatchTouchEvent before12-30 14:06:03.694 31323-31323/lyn.demo D/View0: onTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/View0: dispatchTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup2nd: onTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup2nd: dispatchTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup1st: onTouchEvent return:false12-30 14:06:03.694 31323-31323/lyn.demo D/ViewGroup1st: dispatchTouchEvent return:false
```

当 down 事件从 DecorView 开始了分发过程：

ViewGroup1st 收到事件，执行 onInterceptTouchEvent 返回 false，不拦截，于是调用 ViewGroup2nd 的 dispatchTouchEvent 向 ViewGroup2nd分发。

ViewGroup2nd 收到事件，dispatchTouchEvent 重复 ViewGroup1st 的分发策略。因为都不拦截，所以调用了 View0 的 dispatchTouchEvent。

View0 收到事件，而 View0 不是 ViewGroup 类型，所以把事件直接交给了 onTouchEvent。

View0 不消费事件，onTouchEvent 返回 false，dispatchTouchEvent 方法因此也返回 false。

ViewGroup2nd 因为 View0 的 dispatchTouchEvent 返回 false，确定了子类不消费事件，于是把事件传递给 onTouchEvent。但本身也不消费事件，所以 onTouchEvent 也返回 false，继续把事件上抛到 ViewGroup1st。
ViewGroup1st 重复了 ViewGroup2nd 的过程。

随后，move 事件不会再往下传了，而是直接被 Activity 拦截。

2.2. 案例2：View0 消费 down 事件

首先是 down 事件的传递，log 如下

```
12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup1st: dispatchTouchEvent before12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup1st: onInterceptTouchEvent return:false12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup2nd: dispatchTouchEvent before12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup2nd: onInterceptTouchEvent return:false12-30 14:14:09.384 7350-7350/lyn.demo D/View0: dispatchTouchEvent before12-30 14:14:09.384 7350-7350/lyn.demo D/View0: onTouchEvent return:true12-30 14:14:09.384 7350-7350/lyn.demo D/View0: dispatchTouchEvent return:true12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup2nd: dispatchTouchEvent return:true12-30 14:14:09.384 7350-7350/lyn.demo D/ViewGroup1st: dispatchTouchEvent return:true
```

ViewGroup1st 和 ViewGroup2st 的传递和案例1一样。区别在于 View0 onTouchEvent 返回 true 消费后续事件后，View0 的 dispatchTouchEvent 也返回 true，ViewGroup2nd 和 ViewGroup1st 不执行 onTouchEvent 也直接返回 true

然后稍微移动一下手指，move 事件往下传递

```
12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup1st: dispatchTouchEvent before12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup1st: onInterceptTouchEvent return:false12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup2nd: dispatchTouchEvent before12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup2nd: onInterceptTouchEvent return:false12-30 14:14:09.484 7350-7350/lyn.demo D/View0: dispatchTouchEvent before12-30 14:14:09.484 7350-7350/lyn.demo D/View0: onTouchEvent return:true12-30 14:14:09.484 7350-7350/lyn.demo D/View0: dispatchTouchEvent return:true12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup2nd: dispatchTouchEvent return:true12-30 14:14:09.484 7350-7350/lyn.demo D/ViewGroup1st: dispatchTouchEvent return:true
```

过程和 down 事件的传递一样。因为同样会经过 ViewGroup2nd 的 onInterceptTouchEvent，如果这时候 ViewGroup2nd 有拦截行为，move 事件就不会传到 View0 了。要避免这种情况发生，需要调用 View0 的 requestDisallowInterceptTouchEvent，可见 1.4 部分。

2.3. 案例3：ViewGroup2nd 消费 down 事件

首先是 down 事件的传递，log 如下

```
12-30 14:25:30.074 18848-18848/lyn.demo D/ViewGroup1st: dispatchTouchEvent before12-30 14:25:30.074 18848-18848/lyn.demo D/ViewGroup1st: onInterceptTouchEvent return:false12-30 14:25:30.084 18848-18848/lyn.demo D/ViewGroup2nd: dispatchTouchEvent before12-30 14:25:30.084 18848-18848/lyn.demo D/ViewGroup2nd: onInterceptTouchEvent return:false12-30 14:25:30.084 18848-18848/lyn.demo D/View0: dispatchTouchEvent before12-30 14:25:30.084 18848-18848/lyn.demo D/View0: onTouchEvent return:false12-30 14:25:30.084 18848-18848/lyn.demo D/View0: dispatchTouchEvent return:false12-30 14:25:30.084 18848-18848/lyn.demo D/ViewGroup2nd: onTouchEvent return:true12-30 14:25:30.084 18848-18848/lyn.demo D/ViewGroup2nd: dispatchTouchEvent return:true12-30 14:25:30.084 18848-18848/lyn.demo D/ViewGroup1st: dispatchTouchEvent return:true
```

由于 View0 不消费事件，dispatchTouchEvent 返回 false，所以执行了 ViewGroup2nd 的 onTouchEvent 方法。

ViewGroup2nd 消费事件，onTouchEvent 返回 true，之后 ViewGroup2nd 和 ViewGroup1st 的 dispatchTouchEvent 均返回 true。

动一下手指，move 事件接着传

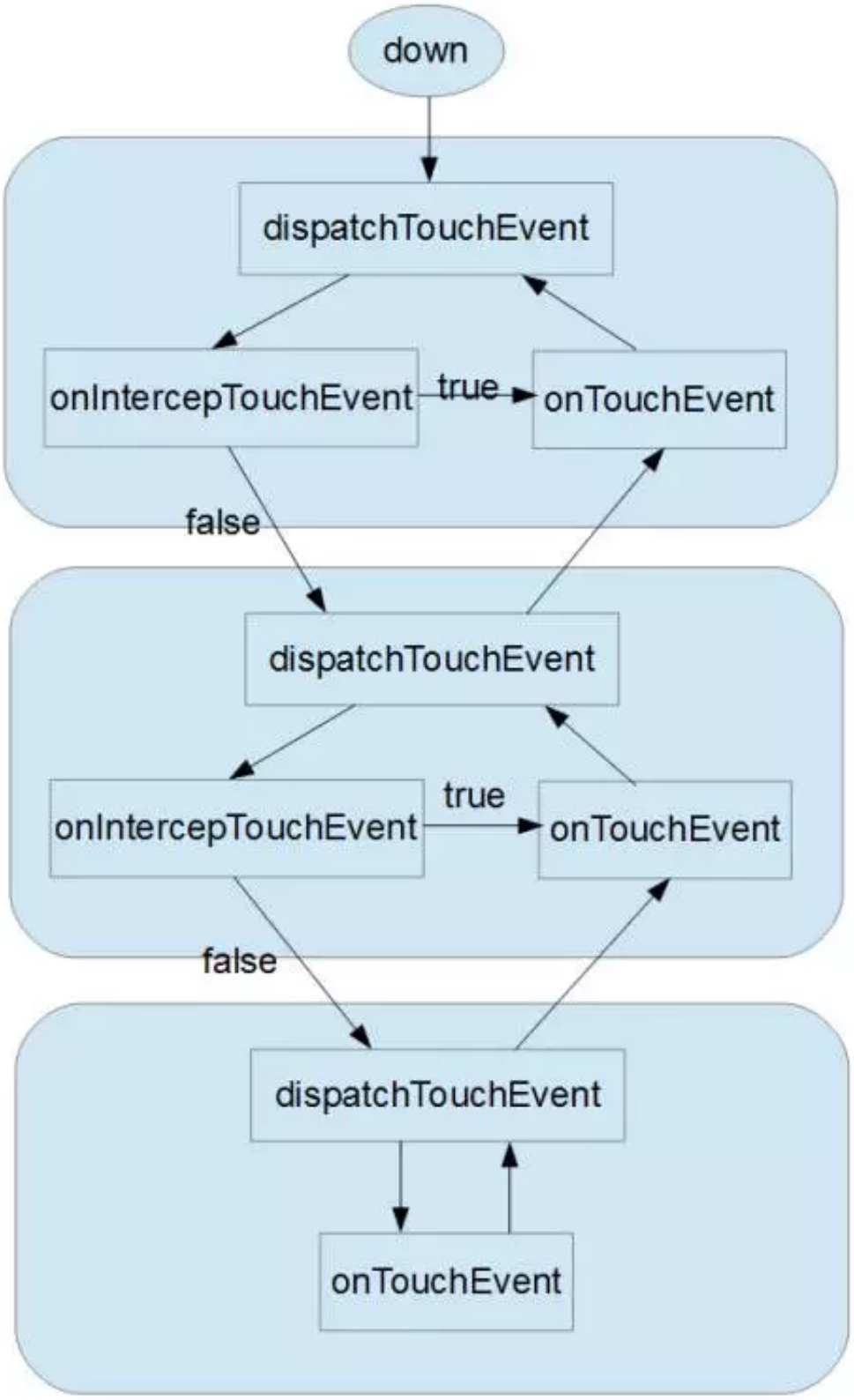
```
2-30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup1st: dispatchTouchEvent before12-30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup1st: onInterceptTouchEvent return:false12-30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup2nd: dispatchTouchEvent before12-30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup2nd: onTouchEvent return:true12-30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup2nd: dispatchTouchEvent return:true12-
```

```
30 14:25:30.174 18848-18848/lyn.demo D/ViewGroup1st: dispatchTouchEvent return:true
```

这时候，ViewGroup2nd 直接拦截了 move 事件，不再经过 onInterceptTouchEvent，也不再向 View0 分发，而是直接调用 onTouchEvent 进行处理。

3. down 事件分发图

在每个 View 都不拦截 down 事件的情况下，down 事件是这样传递的



down 事件的分发过程

阅读原文

微信扫一扫
关注该公众号