# React Native 触摸事件处理详解

on [React-Native](http://www.race604.com/tag/react-native/), [Android](http://www.race604.com/tag/android/)

触控是移动设备的核心功能，也移动应用交互的基础，Android 和 iOS 各自都有完善的触摸事件处理机制。React Native（以下简称 RN）提供了一套统一的处理方式，能够方便的处理界面中组件的触摸事件、用户手势等。本文尝试介绍 RN 中触摸事件处理。

## 1. RN 基本触摸组件

RN 的组件除了 Text，其他组件默认是不支持点击事件，也不能响应基本触摸事件，所以 RN 中提供了几个直接处理响应事件的组件，基本上能够满大部分的点击处理需求TouchableHighlight, TouchableNativeFeedback, TouchableOpacity 和 TouchableWithoutFeedback。因为这几个组件的功能和使用方法基本类似，只是 Touch 的反馈效果不一样，所以一般我们用 Touchable\*\* 代替。Touchable\*\* 有如下几个回调方法：

* onPressIn：点击开始；
* onPressOut：点击结束或者离开；
* onPress：单击事件回调；
* onLongPress：长按事件回调。

它们的基本使用方法如下，这里以 TouchableHighlight 为例：

<TouchableHighlight

onPressIn={() => console.log("onPressIn")}

onPressOut={() => console.log("onPressOut")}

onPress={() => console.log("onPress")}

onLongPress={() => console.log("onLongPress")}

>

<Image

style={styles.button}

source={require('./img/rn\_logo.png')} />

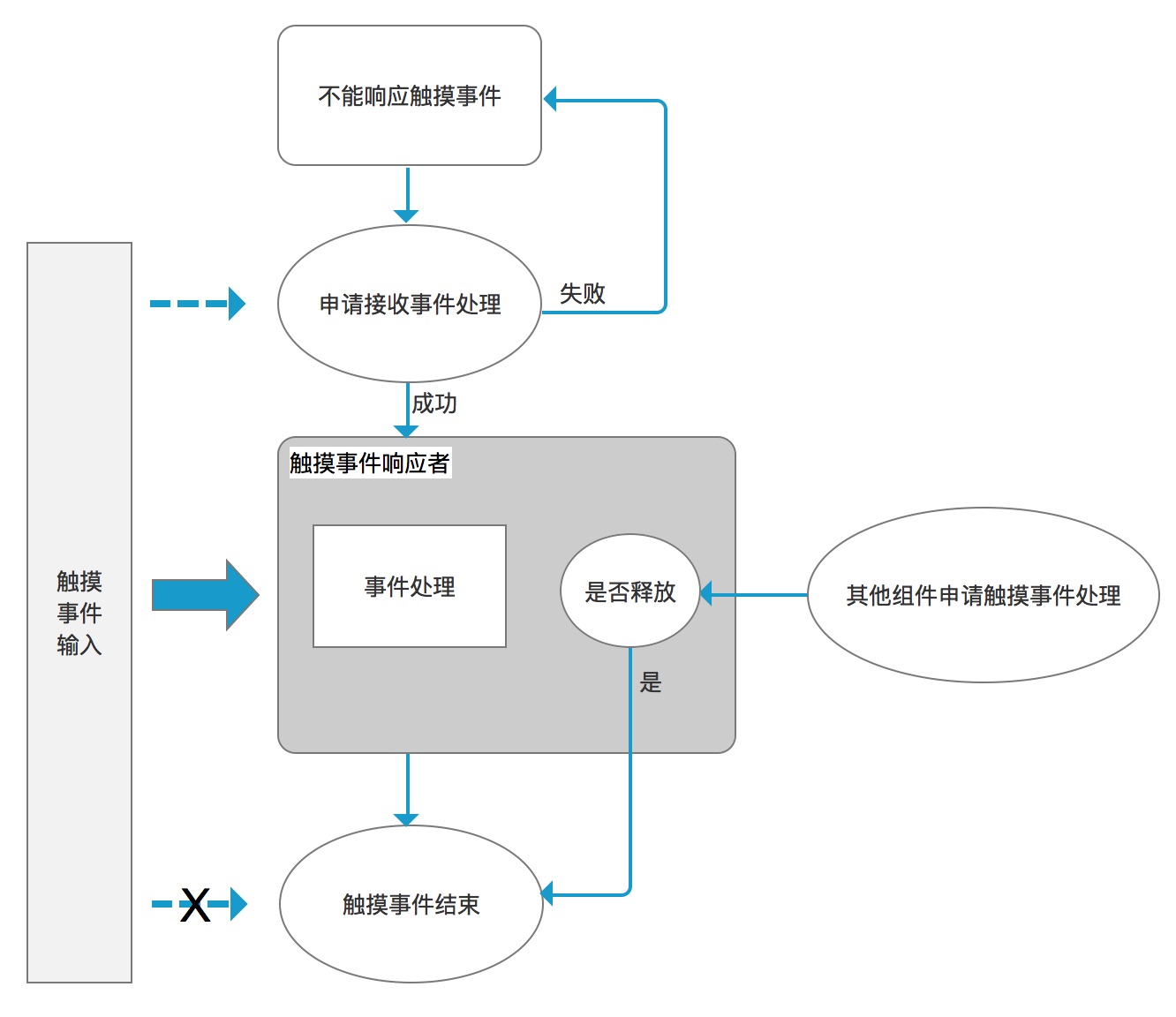
</TouchableHighlight>

RN 中提供的触摸组件使用非常简单，可以参考 [官方文档](http://facebook.github.io/react-native/docs/touchablehighlight.html)，这里也不做详细的介绍了。下面主要介绍用户触摸事件处理。

## 2. 单组件触摸事件处理

我们知道，RN 的组件默认不进行处理触摸事件。组件要处理触摸事件，首先要“申请”成为摸事件的响应者（Responder），完成事件处理以后，会释放响应者的角色。一个触摸事件处理周期，是从用户手指按下屏幕，到用户抬起手指抬起结束，这是用户的一次完整触摸操作。

单个组件的单次操作交互处理的生命周期如下：



我们来详细分析一下事件处理的生命周期，在整个事件处理的过程中，组件有可能处于两种身份中的一种，并且可以相互切换：非事件响应者和事件响应者。

#### 非事件响应者

默认情况下，触摸事件输入不会直接传递给组件，不能进行事件响应处理，也就是非事件响应者。如果组件要进行触摸事件处理，首先要申请成为事件响应者，组件有如下两个属性可以做这样的申请：

* View.props.onStartShouldSetResponder，这个属性接收一个回调函数，函数原型是 function(evt): bool，在触摸事件开始（touchDown）的时候，RN 会回调此函数，询问组件是否需要成为事件响应者，接收事件处理，如果返回 true，表示需要成为响应者；
* View.props.onMoveShouldSetResponder，它和前一个属性类似，不过这是触摸是进行过程中（touchMove），RN 询问组件是否要成为响应者，返回 true 表示是。

假如组件通过上面的方法返回了 true，表示发出了申请要成为事件响应者请求，想要接收后续的事件输入。因为同一时刻，只能有一个事件处理响应者，RN 还需要协调所有组件的事件处理请求，所以不是每个组件申请都能成功，RN 通过如下两个回调来通知告诉组件它的申请结果，：

* View.props.onResponderGrant: (evt) => {}：表示申请成功，组件成为了事件处理响应者，这时组件就开始接收后序的触摸事件输入。一般情况下，这时开始，组件进入了激活状态，并进行一些事件处理或者手势识别的初始化。
* View.props.onResponderReject: (evt) => {}：表示申请失败了，这意味者其他组件正在进行事件处理，并且它不想放弃事件处理，所以你的申请被拒绝了，后续输入事件不会传递给本组件进行处理。

#### 事件响应者

如果通过上面的步骤，组件申请成为了事件响应者，后续的事件输入都会通过回调函数通知到组件，如下：

* View.props.onResponderStart: (evt) => {}：表示手指按下时，成功申请为事件响应者的回调；
* View.props.onResponderMove: (evt) => {}：表示触摸手指移动的事件，这个回调可能非常频繁，所以这个回调函数的内容需要尽量简单；
* View.props.onResponderRelease: (evt) => {}：表示触摸完成（touchUp）的时候的回调，表示用户完成了本次的触摸交互，这里应该完成手势识别的处理，这以后，组件不再是事件响应者，组件取消激活。
* View.props.onResponderEnd: (evt) => {}：表示组件结束事件响应的回调。

从前面的图中也看到，在组件成为事件响应者期间，其他组件也可能会申请触摸事件处理。此时 RN 会通过回调询问你是否可以释放响应者角色让给其他组件。回调如下：

View.props.onResponderTerminationRequest: (evt) => bool

如果回调函数返回为 true，则表示同意释放响应者角色，同时会回调如下函数，通知组件事件响应处理被终止了：

View.props.onResponderTerminate: (evt) => {}

这个回调也会发生在系统直接终止组件的事件处理，例如用户在触摸操作过程中，突然来电话的情况。

#### 事件数据结构

从前面我们看到，触摸事件处理的回调都有一个 evt 参数，包含一个触摸事件数据 nativeEvent。nativeEvent 的详细内容如下：

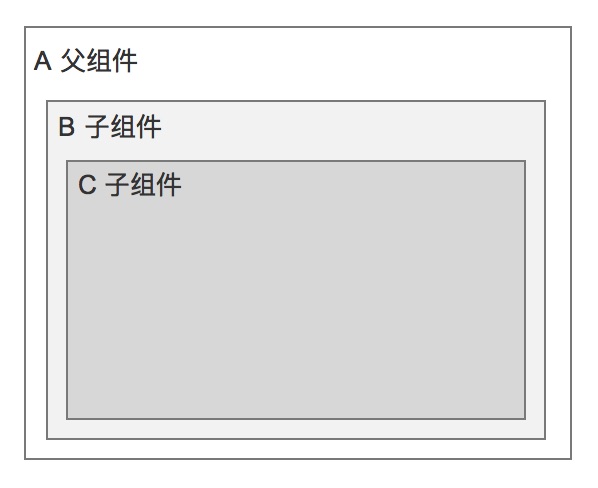
* identifier：触摸的 ID，一般对应手指，在多点触控的时候，用来区分是哪个手指的触摸事件；
* locationX 和 locationY：触摸点相对组件的位置；
* pageX 和 pageY：触摸点相对于屏幕的位置；
* timestamp：当前触摸的事件的时间戳，可以用来进行滑动计算；
* target：接收当前触摸事件的组件 ID；
* changedTouches：evt 数组，从上次回调上报的触摸事件，到这次上报之间的所有事件数组。因为用户触摸过程中，会产生大量事件，有时候可能没有及时上报，系统用这种方式批量上报；
* touches：evt 数组，多点触摸的时候，包含当前所有触摸点的事件。

这些数据中，最常用的是 locationX 和 locationY 数据，需要注意的是，因为这里是 Native 的数据，所以他们的单位是实际像素。如果要转换为 RN 中的逻辑单位，可以示使用如下方法：

var pX = evt.nativeEvent.locationX / PixelRatio.get();

## 3. 嵌套组件事件处理

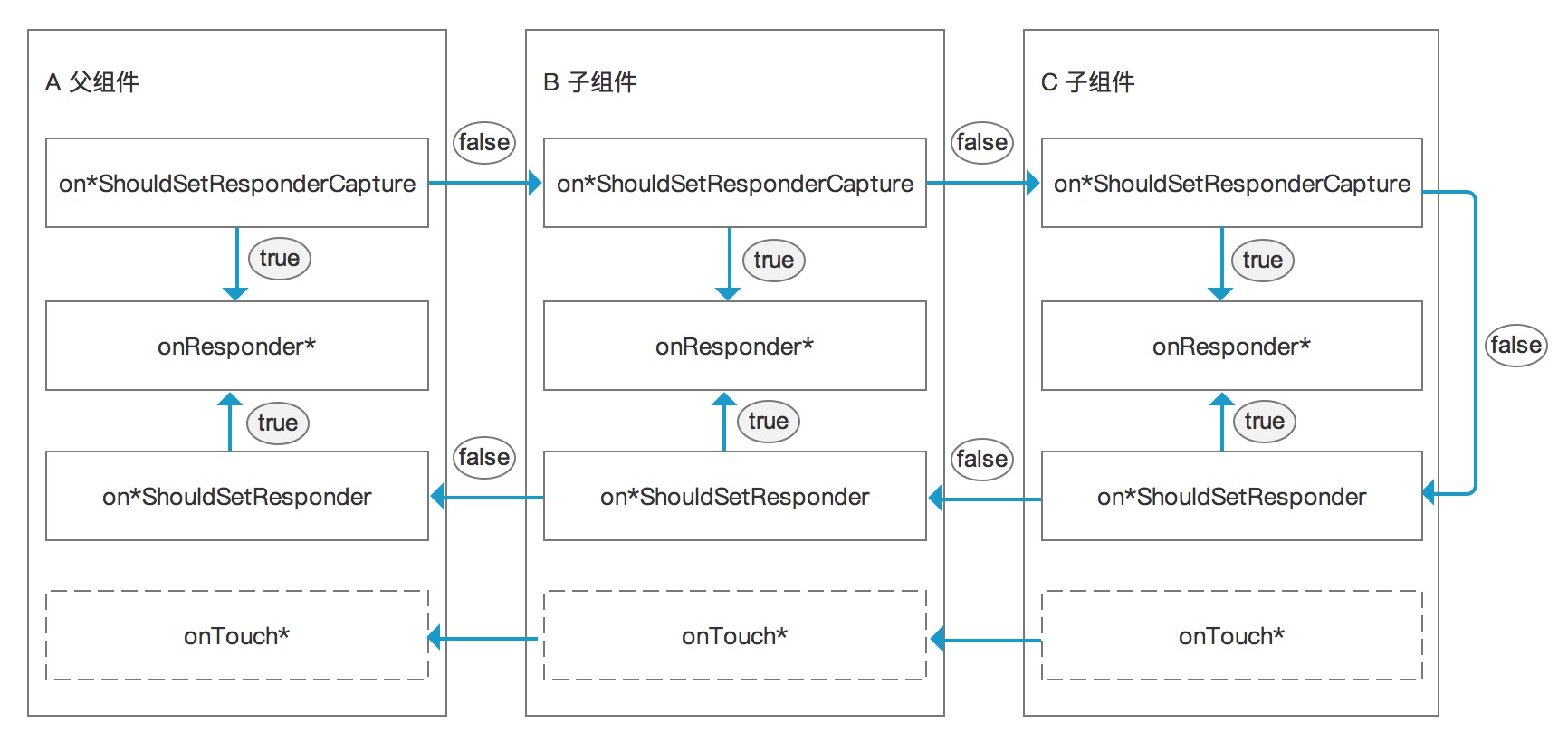
上一小节介绍的都是针对单个组件来说，事件处理的流程和机制。但是前面也提到了，当组件需要作为事件处理响应者时，需要通过 onStartShouldSetResponder 或者 onMoveShouldSetResponder 回调返回值为 true 来申请。假如当多个组件嵌套的时候，这两个回调都返回了 true 的时候，但是同一个只能有一个事件处理响应者，这种情况怎么处理呢？为了便于描述，假设我们的组件布局如下：



在 RN 中，默认情况下使用冒泡机制，响应最深的组件最先开始响应，所以前面描述的这种情况，如图中，如果 A、B、C 三个组件的 on\*ShouldSetResponder 都返回为 true， 那么只有 C 组件会得到响应成为响应者。这种机制才能保证了界面所有的组件才能得到响应。但是有些情况下，可能父组件可能需要处理事件，而禁止子组件响应。RN 提供了一个劫持机制，也就是在触摸事件往下传递的时候，先询问父组件是否需要劫持，不给子组件传递事件，也就是如下两个回调：

* View.props.onStartShouldSetResponderCapture：这个属性接收一个回调函数，函数原型是 function(evt): bool，在触摸事件开始（touchDown）的时候，RN 容器组件会回调此函数，询问组件是否要劫持事件响应者设置，自己接收事件处理，如果返回 true，表示需要劫持；
* View.props.onMoveShouldSetResponderCapture：此函数类似，不过是在触摸移动事件（touchMove）询问容器组件是否劫持。

可以把这种劫持机制看成是一种下沉机制，与上面的冒泡机制对应，我们可以总结 RN 事件处理流程如下图：



注，图中的 \* 表示可以为 Start 或者 Move，例如 on\*ShouldSetResponderCapture 表示 onStartShouldSetResponderCapture 或者 onMoveShouldSetResponderCapture，其他的类似。

触摸事件开始，首先调用 A 组件的 onStartShouldSetResponderCapture，若此回调返回 false，则按照图传递到 B 组件，然后调用 B 组件 onStartShouldSetResponderCapture，若返回 true，则事件不再传递给 C 组件，直接调用本组件的 onResponderStart，则 B 组件就成为事件响应者，后续事件直接传递给它。其他的分析类似。

注意到，图中还有 onTouchStart/onTouchStop 回调，这个回调并不受响应者的影响，在范围内的组件都会回调此函数，而且调用顺序是从最深层组件到最上层组件。

## 4. 手势识别

前面只是介绍了简单的触摸事件处理机制及其使用方法，其实连续的触摸事件，可以组成一些更高级手势，例如我们最常见的滑动屏幕内容，双指缩放（Pinch）或者旋转图片都是通过手势识别完成的。

因为有些手势是很常用的，RN 也提供了内置的手势识别库 PanResponder，它封装了上面的事件回调函数，对触摸事件数据进行加工，完成滑动手势识别，向我们提供更加高级有意义的接口，如下：

* onMoveShouldSetPanResponder: (e, gestureState) => bool
* onMoveShouldSetPanResponderCapture: (e, gestureState) => bool
* onStartShouldSetPanResponder: (e, gestureState) => bool
* onStartShouldSetPanResponderCapture: (e, gestureState) => bool
* onPanResponderReject: (e, gestureState) => {...}
* onPanResponderGrant: (e, gestureState) => {...}
* onPanResponderStart: (e, gestureState) => {...}
* onPanResponderEnd: (e, gestureState) => {...}
* onPanResponderRelease: (e, gestureState) => {...}
* onPanResponderMove: (e, gestureState) => {...}
* onPanResponderTerminate: (e, gestureState) => {...}
* onPanResponderTerminationRequest: (e, gestureState) => {...}
* onShouldBlockNativeResponder: (e, gestureState) => bool

可以看到，这些接口与前面接收的基础回调基本上是一一对应的，其功能也是类似，这里就不再赘述。这里有一个特别的回调 onShouldBlockNativeResponder 表示是否用 Native 平台的事件处理，默认是禁用的，全部使用 JS 中的事件处理，注意此函数目前只能在 Android 平台上使用。不过这里回调函数都有一个新的参数 gestureState，这是与滑动相关的数据，是对基本触摸数据的分析处理，它的内容如下：

* stateID：滑动手势的 ID，在一次完整的交互中此 ID 保持不变；
* moveX 和 moveY：自上次回调，手势移动距离；
* x0 和 y0：滑动手势识别开始的时候的在屏幕中的坐标；
* dx 和 dy：从手势开始时，到当前回调是移动距离；
* vx 和 vy：当前手势移动的速度；
* numberActiveTouches：当期触摸手指数量。

下面介绍一个简单的实例，本例实现可以使用手指拖动界面的圆形控件，使用实例如下：

import React from 'react';

import {

AppRegistry,

PanResponder,

StyleSheet,

View,

processColor,

} from 'react-native';

var CIRCLE\_SIZE = 80;

var CIRCLE\_COLOR = 'blue';

var CIRCLE\_HIGHLIGHT\_COLOR = 'green';

var PanResponderExample = React.createClass({

statics: {

title: 'PanResponder Sample',

description: 'Shows the use of PanResponder to provide basic gesture handling.',

},

\_panResponder: {},

\_previousLeft: 0,

\_previousTop: 0,

\_circleStyles: {},

circle: (null : ?{ setNativeProps(props: Object): void }),

componentWillMount: function() {

this.\_panResponder = PanResponder.create({

onStartShouldSetPanResponder: (evt, gestureState) => true,

onMoveShouldSetPanResponder: (evt, gestureState) => true,

onPanResponderGrant: this.\_handlePanResponderGrant,

onPanResponderMove: this.\_handlePanResponderMove,

onPanResponderRelease: this.\_handlePanResponderEnd,

onPanResponderTerminate: this.\_handlePanResponderEnd,

});

this.\_previousLeft = 20;

this.\_previousTop = 84;

this.\_circleStyles = {

style: {

left: this.\_previousLeft,

top: this.\_previousTop

}

};

},

componentDidMount: function() {

this.\_updatePosition();

},

render: function() {

return (

<View style={styles.container}>

<View

ref={(circle) => {

this.circle = circle;

}}

style={styles.circle}

{...this.\_panResponder.panHandlers}

/>

</View>

);

},

\_highlight: function() {

const circle = this.circle;

circle && circle.setNativeProps({

style: {

backgroundColor: processColor(CIRCLE\_HIGHLIGHT\_COLOR)

}

});

},

\_unHighlight: function() {

const circle = this.circle;

circle && circle.setNativeProps({

style: {

backgroundColor: processColor(CIRCLE\_COLOR)

}

});

},

\_updatePosition: function() {

this.circle && this.circle.setNativeProps(this.\_circleStyles);

},

\_handlePanResponderGrant: function(e: Object, gestureState: Object) {

this.\_highlight();

},

\_handlePanResponderMove: function(e: Object, gestureState: Object) {

this.\_circleStyles.style.left = this.\_previousLeft + gestureState.dx;

this.\_circleStyles.style.top = this.\_previousTop + gestureState.dy;

this.\_updatePosition();

},

\_handlePanResponderEnd: function(e: Object, gestureState: Object) {

this.\_unHighlight();

this.\_previousLeft += gestureState.dx;

this.\_previousTop += gestureState.dy;

},

});

var styles = StyleSheet.create({

circle: {

width: CIRCLE\_SIZE,

height: CIRCLE\_SIZE,

borderRadius: CIRCLE\_SIZE / 2,

backgroundColor: CIRCLE\_COLOR,

position: 'absolute',

left: 0,

top: 0,

},

container: {

flex: 1,

paddingTop: 64,

},

});

可见，在 componentWillMount 中创建一个 PanResponder 实例，并设置想好相关的属性，然后把这个对象设置给 View 的属性，如下:

<View

{...this.\_panResponder.panHandlers}

/>

其余的代码也比较简单，这里就不详述了。

## 5. 总结

通过上面的介绍，可以看到 RN 中提供了类似 Native 平台的事件处理机制，所以也可以实现各种的触摸事件处理，甚至也可以实现复杂的手势识别。

在嵌套组件的事件处理中，RN 中提供了“冒泡”和“下沉”两个方向的事件处理，这有点类似于 Android Native 上不久前才支持的 [NestedScrolling](http://www.race604.com/android-nested-scrolling/)，这就提供更加强大的事件处理机制。

另外需要注意，因为 RN 的异步通信和执行机制，前面描述的所有回调函数都是在 JS 线程中，并不是 Native 的 UI 线程，而 Native 平台的 Touch 事件都是在 UI 线程中。所以在 JS 中通过 Touch 或者手势实现动画，可能会延迟的问题。