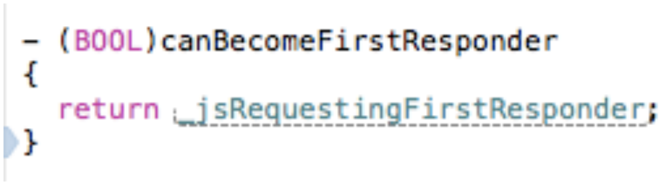
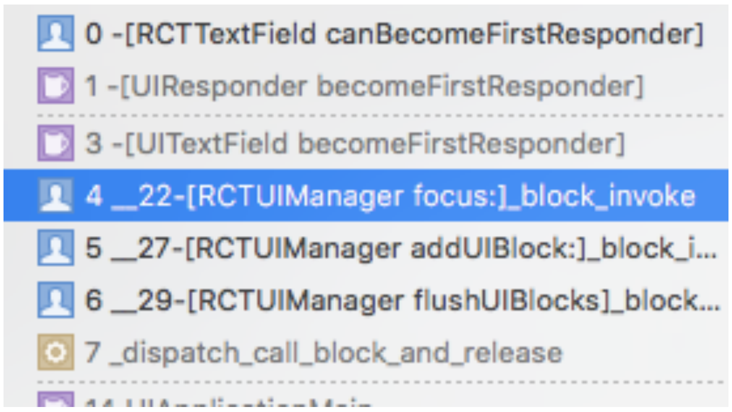
检查canBecomeFirstResponder,打断点



```
- (BOOL)canBecomeFirstResponder
{
  return _jsRequestingFirstResponder;
}
```

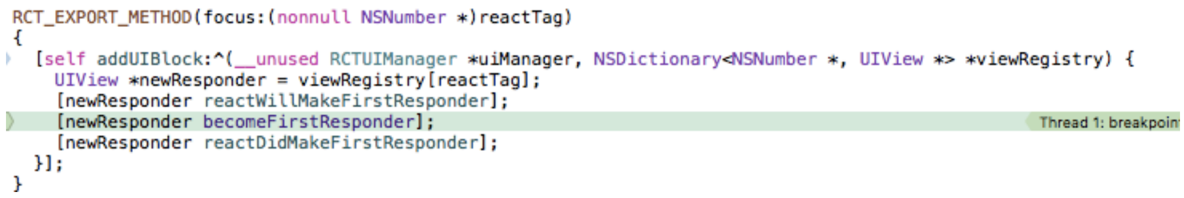
发现正常情况该方法会被调用两次（false, true），然而异常情况只会被调用一次(false)

正常情况第二次调用的调用栈



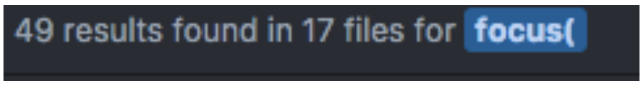
```
0 -[RCTTextField canBecomeFirstResponder]
1 -[UIResponder becomeFirstResponder]
3 -[UITextField becomeFirstResponder]
4 __22-[RCTUIManager focus:]_block_invoke
5 __27-[RCTUIManager addUIBlock:]_block_i...
6 __29-[RCTUIManager flushUIBlocks]_block...
7 _dispatch_call_block_and_release
```

该方法暴露给了js， 猜测是由js调用的



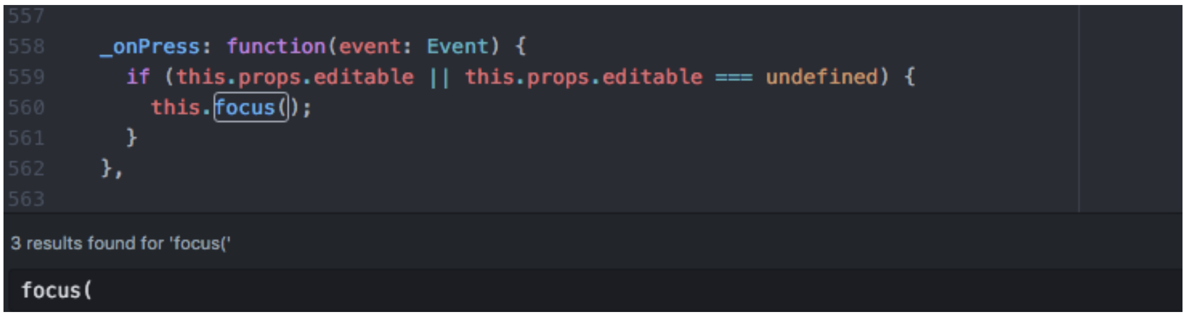
```
RCT_EXPORT_METHOD(focus:(nonnull NSNumber *)reactTag)
{
  [self addUIBlock:^(__unused RCTUIManager *uiManager, NSDictionary<NSNumber *, UIView *> *viewRegistry) {
    UIView *newResponder = viewRegistry[reactTag];
    [newResponder reactWillMakeFirstResponder];
    [newResponder becomeFirstResponder];                                        Thread 1: breakpoin
    [newResponder reactDidMakeFirstResponder];
  }];
}
```

搜索focus



```
49 results found in 17 files for focus(
```

结果太多。。。猜肯定和TextInput有关

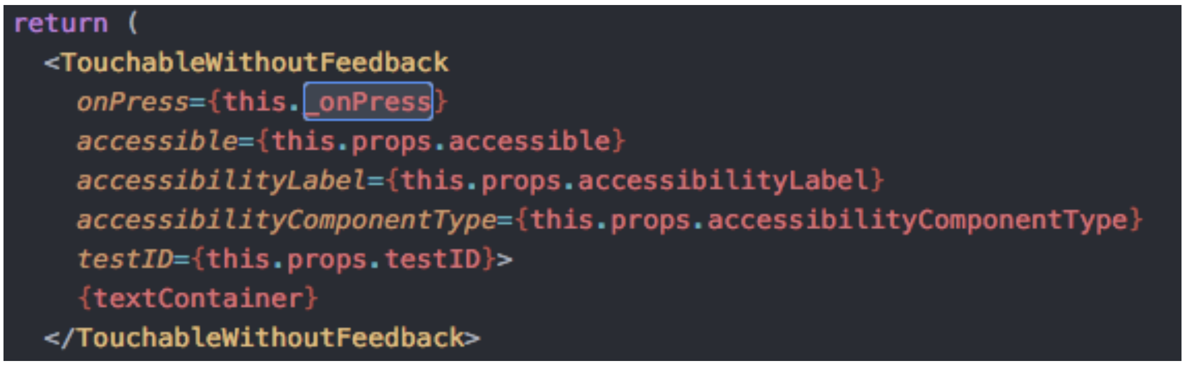


```
    _onPress: function(event: Event) {
      if (this.props.editable || this.props.editable === undefined) {
        this.focus();
      }
    },

3 results found for 'focus('
focus(
```

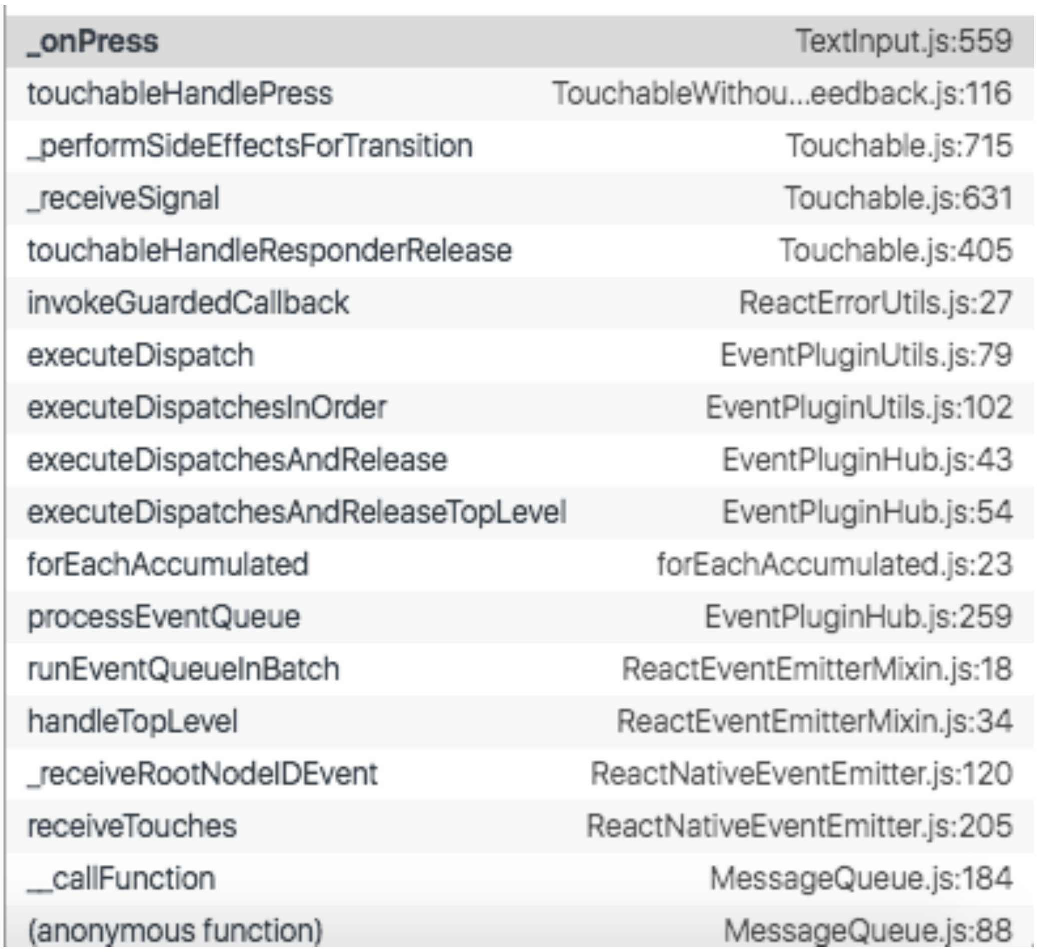
TextField在_onPress里调用了focus (继续找下去可以找到UIManager.focus())

搜索_onPress



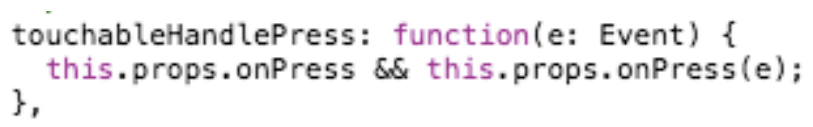
```
return (
  <TouchableWithoutFeedback
    onPress={this._onPress}
    accessible={this.props.accessible}
    accessibilityLabel={this.props.accessibilityLabel}
    accessibilityComponentType={this.props.accessibilityComponentType}
    testID={this.props.testID}>
    {textContainer}
  </TouchableWithoutFeedback>
```

在onPress回调里调用
然后在_onPress里设个断点，发现异常情况并不会调用_onPress
正常时调用_onPress的调用栈：



```
_onPress                              TextInput.js:559
touchableHandlePress                  TouchableWithou...eedback.js:116
_performSideEffectsForTransition      Touchable.js:715
_receiveSignal                        Touchable.js:631
touchableHandleResponderRelease       Touchable.js:405
invokeGuardedCallback                 ReactErrorUtils.js:27
executeDispatch                       EventPluginUtils.js:79
executeDispatchesInOrder              EventPluginUtils.js:102
executeDispatchesAndRelease           EventPluginHub.js:43
executeDispatchesAndReleaseTopLevel   EventPluginHub.js:54
forEachAccumulated                    forEachAccumulated.js:23
processEventQueue                     EventPluginHub.js:259
runEventQueueInBatch                  ReactEventEmitterMixin.js:18
handleTopLevel                        ReactEventEmitterMixin.js:34
_receiveRootNodeIDEvent               ReactNativeEventEmitter.js:120
receiveTouches                        ReactNativeEventEmitter.js:205
__callFunction                        MessageQueue.js:184
(anonymous function)                  MessageQueue.js:88
```

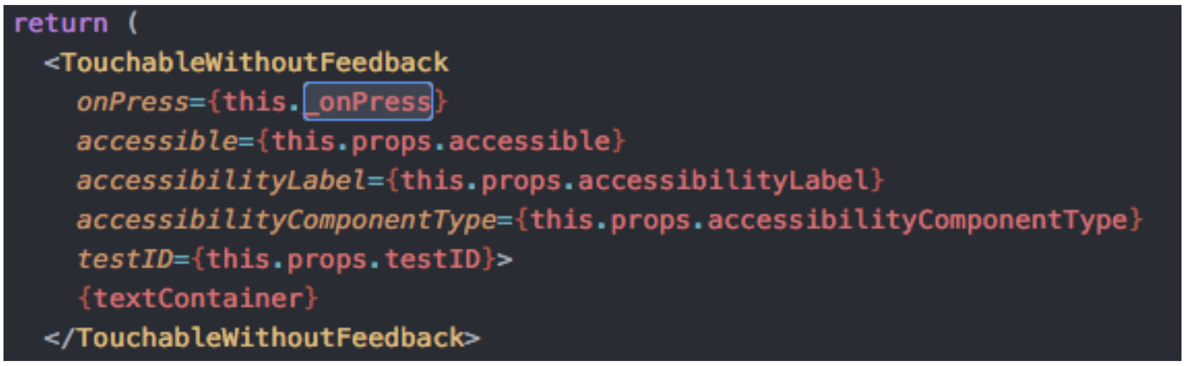
从上到下分析：
第二个只是简单确认onPress有值：



```
touchableHandlePress: function(e: Event) {
  this.props.onPress && this.props.onPress(e);
},
```

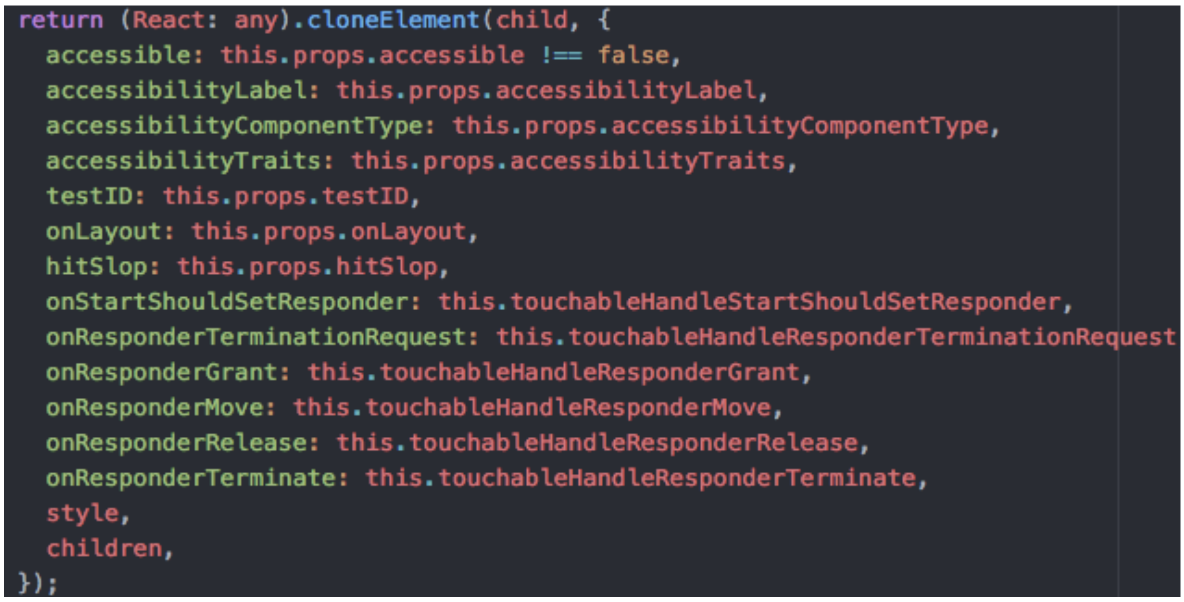
在第三个打断点,发现无论怎样都会被执行多次。。慢慢分析会比较复杂。换思路，我们先确定是事件没有发出还是传输时丢失了,我们需要先找到Js端event的源头然后推出Native发送event的位置

根据TextInput的实现可知onPress信号由TouchableWithoutFeedback接受



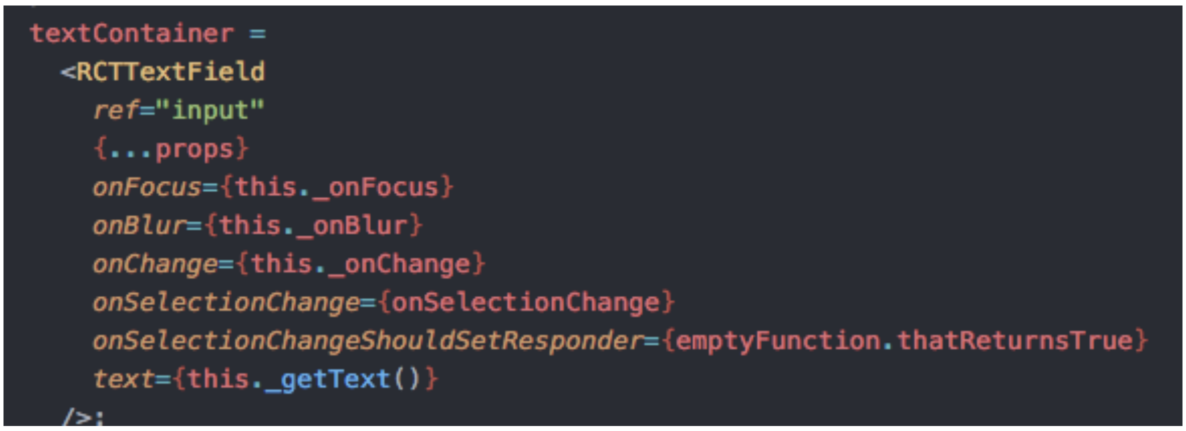
```
return (
  <TouchableWithoutFeedback
    onPress={this._onPress}
    accessible={this.props.accessible}
    accessibilityLabel={this.props.accessibilityLabel}
    accessibilityComponentType={this.props.accessibilityComponentType}
    testID={this.props.testID}>
    {textContainer}
  </TouchableWithoutFeedback>
```

由TouchableWithoutFeedback的实现可知TouchableWithoutFeedback只是将child的clone加上了一大堆方法处理的属性然后直接返回child的clone



```
return (React: any).cloneElement(child, {
  accessible: this.props.accessible !== false,
  accessibilityLabel: this.props.accessibilityLabel,
  accessibilityComponentType: this.props.accessibilityComponentType,
  accessibilityTraits: this.props.accessibilityTraits,
  testID: this.props.testID,
  onLayout: this.props.onLayout,
  hitSlop: this.props.hitSlop,
  onStartShouldSetResponder: this.touchableHandleStartShouldSetResponder,
  onResponderTerminationRequest: this.touchableHandleResponderTerminationRequest
  onResponderGrant: this.touchableHandleResponderGrant,
  onResponderMove: this.touchableHandleResponderMove,
  onResponderRelease: this.touchableHandleResponderRelease,
  onResponderTerminate: this.touchableHandleResponderTerminate,
  style,
  children,
});
```

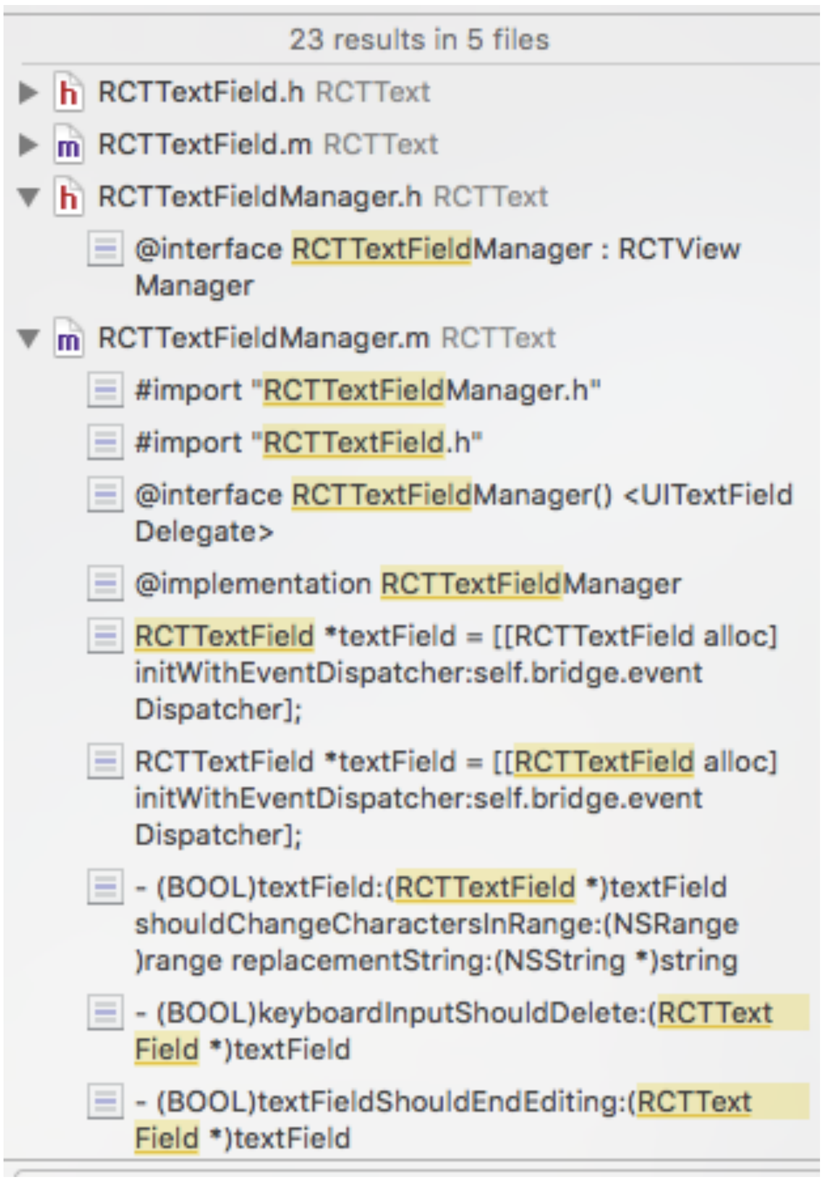
所以Js端event的源头是child 即 {textContainer} 即RCTTextField：



```
textContainer =
  <RCTTextField
    ref="input"
    {...props}
    onFocus={this._onFocus}
    onBlur={this._onBlur}
    onChange={this._onChange}
    onSelectionChange={onSelectionChange}
    onSelectionChangeShouldSetResponder={emptyFunction.thatReturnsTrue}
    text={this._getText()}
  />;
```

现在在原生找RCTTextField(一开始那个类)
原生检测touch事件无非两种方法。要么实现UIResponder的方法，要么加GestureRecognizer,
RCTTextField的实现里没有UIResponder的方法，所以确定是GestureRecognizer。
要添加GestureRecognizer必须要有RCTTextField的示例，所以必然会有

RCTTextField的引用，搜索一下。



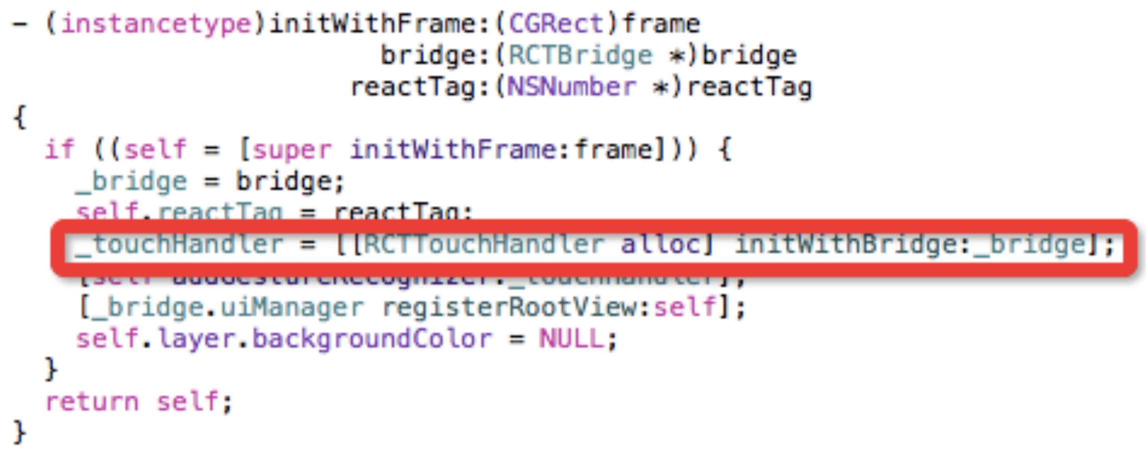
```
23 results in 5 files
▶ RCTTextField.h RCTText
▶ RCTTextField.m RCTText
▼ RCTTextFieldManager.h RCTText
    @interface RCTTextFieldManager : RCTViewManager
▼ RCTTextFieldManager.m RCTText
    #import "RCTTextFieldManager.h"
    #import "RCTTextField.h"
    @interface RCTTextFieldManager() <UITextFieldDelegate>
    @implementation RCTTextFieldManager
    RCTTextField *textField = [[RCTTextField alloc] initWithEventDispatcher:self.bridge.eventDispatcher];
    RCTTextField *textField = [[RCTTextField alloc] initWithEventDispatcher:self.bridge.eventDispatcher];
    - (BOOL)textField:(RCTTextField *)textField shouldChangeCharactersInRange:(NSRange)range replacementString:(NSString *)string
    - (BOOL)keyboardInputShouldDelete:(RCTTextField *)textField
    - (BOOL)textFieldShouldEndEditing:(RCTTextField *)textField
```

这么说来一定是在RCTTextField自身或者RCTTextFieldManager里添加的RCTTouchHandler了！

看了下。妈的没有。。。。

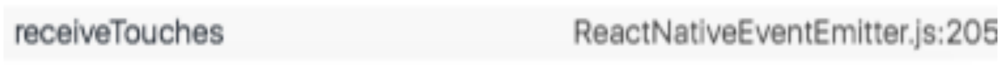
想了想，还有一种可能：响应的是parent view。那么最有可能的就是RootView了。

看了下。喜极而泣。。。。



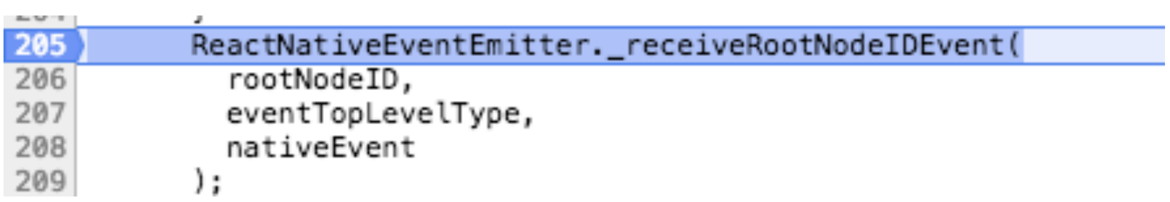
```
- (instancetype)initWithFrame:(CGRect)frame
                       bridge:(RCTBridge *)bridge
                     reactTag:(NSNumber *)reactTag
{
  if ((self = [super initWithFrame:frame])) {
    _bridge = bridge;
    self.reactTag = reactTag;
    _touchHandler = [[RCTTouchHandler alloc] initWithBridge:_bridge];
    [self addGestureRecognizer:_touchHandler];
    [_bridge.uiManager registerRootView:self];
    self.layer.backgroundColor = NULL;
  }
  return self;
}
```

在处理touch的方法handleGestureUpdate:里打个断点
看来是正常发出了。。。
看看JS调用栈有个



```
receiveTouches                    ReactNativeEventEmitter.js:205
```

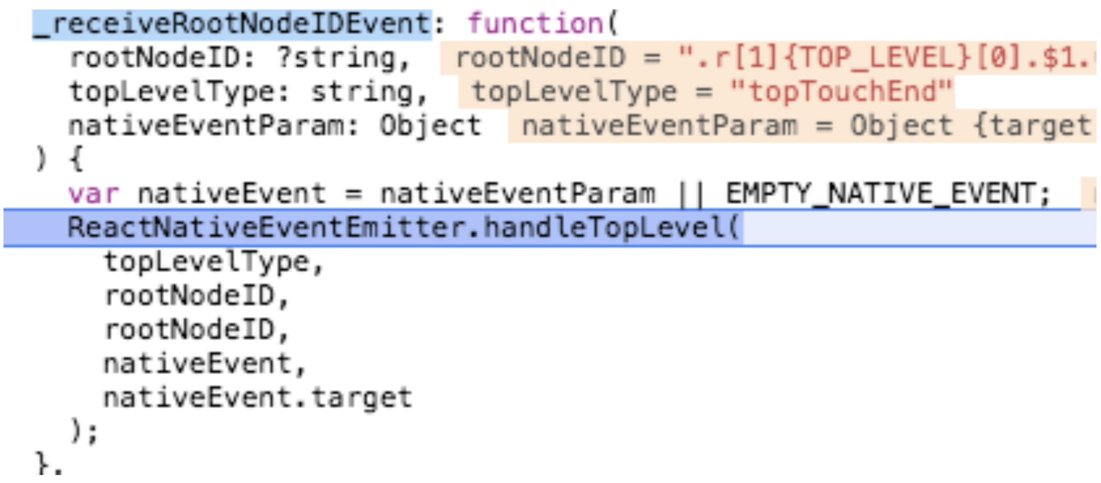
设个断点



```
205        ReactNativeEventEmitter._receiveRootNodeIDEvent(
206          rootNodeID,
207          eventTopLevelType,
208          nativeEvent
209        );
```

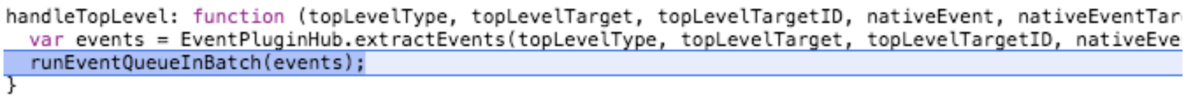
正常： topTouchStart，topTouchEnd，topFocus
异常：topTouchStart， topTouchEnd， topEndEditing，topBlur
（正常异常情况下topTouchStart和topTouchEnd的rootNodeID都相同）
然后就是开脑洞时间：
我们都知道focus是touch的结果，所以推测是topTouchStart、topTouchEnd的后续异常处理导致的bug

继续看调用栈：_receiveRootNodeIDEvent只做了简单的转发



```
_receiveRootNodeIDEvent: function(
    rootNodeID: ?string,   rootNodeID = ".r[1]{TOP_LEVEL}[0].$1.
    topLevelType: string,   topLevelType = "topTouchEnd"
    nativeEventParam: Object   nativeEventParam = Object {target
  ) {
    var nativeEvent = nativeEventParam || EMPTY_NATIVE_EVENT;
    ReactNativeEventEmitter.handleTopLevel(
      topLevelType,
      rootNodeID,
      rootNodeID,
      nativeEvent,
      nativeEvent.target
    );
  },
```

在handleTopLevel里打个断点



```
handleTopLevel: function (topLevelType, topLevelTarget, topLevelTargetID, nativeEvent, nativeEventTar
  var events = EventPluginHub.extractEvents(topLevelType, topLevelTarget, topLevelTargetID, nativeEve
  runEventQueueInBatch(events);
}
```

:

发现正常异常情况在传入topTouchEnd时传给runEventQueueInBatch的参数不同

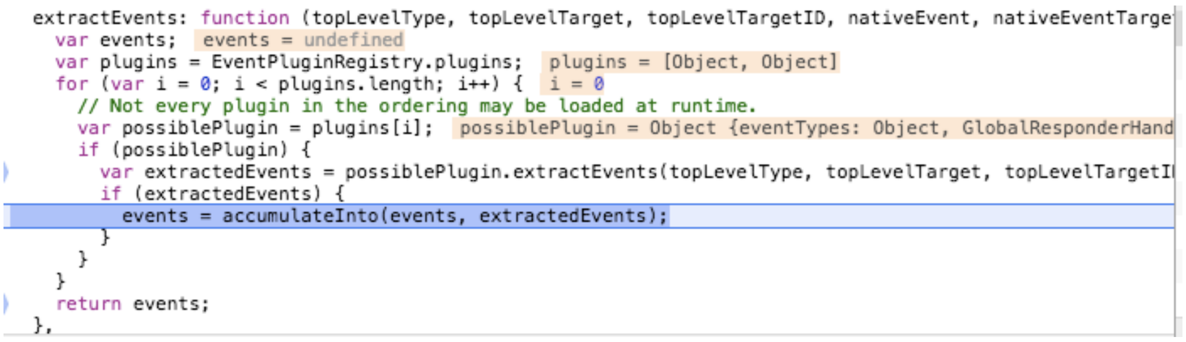
异常：event[1]. _dispatchListeners. __reactBoundMethod = function scrollResponderHandleResponderRelease(e)

正常：event[1]. _dispatchListeners. __reactBoundMethod = function touchableHandleResponderRelease(e)

我们有理由相信就是因为touchableHandleResponderRelease没被调用导致的bug

## 现在可以确定bug在EventPluginHub.extraceEvents里

来看下实现：



```
  extractEvents: function (topLevelType, topLevelTarget, topLevelTargetID, nativeEvent, nativeEventTarge
    var events;   events = undefined
    var plugins = EventPluginRegistry.plugins;   plugins = [Object, Object]
    for (var i = 0; i < plugins.length; i++) {   i = 0
      // Not every plugin in the ordering may be loaded at runtime.
      var possiblePlugin = plugins[i];   possiblePlugin = Object {eventTypes: Object, GlobalResponderHand
      if (possiblePlugin) {
        var extractedEvents = possiblePlugin.extractEvents(topLevelType, topLevelTarget, topLevelTargetI
        if (extractedEvents) {
          events = accumulateInto(events, extractedEvents);
        }
      }
    }
    return events;
  },
```

关于plugin是啥，一开始我也不知道。后来去专门看了下初始化的源码才知道。

我们知道的：

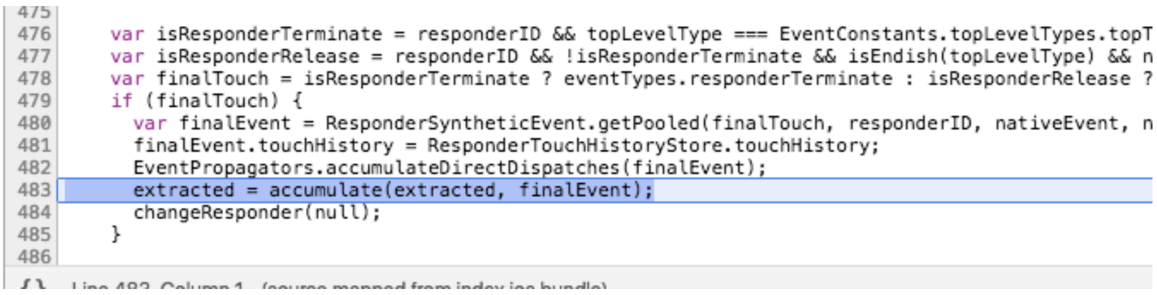
1. 正常异常情况下EventPluginRegistry.plugins返回的值都是一个长度为2的数组
2. 异常情况下接受“topTouchEnd”时第一个plugin产生的extractedEvents[1]的listener是scrollResponderHandleResponderRelease(e) 而正常情况下是touchableHandleResponderRelease(e)
3. extractedEvents在for 循环里调用

推断：

EventPluginRegistry.plugins两次返回的都是一样的数组

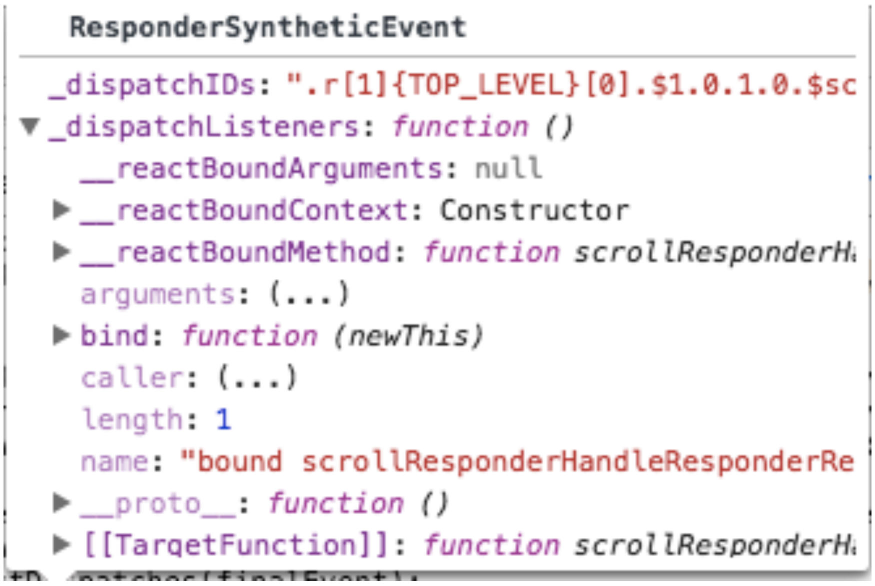
## bug在possiblePlugin.extractEvents里

进去看看。。



```
    var isResponderTerminate = responderID && topLevelType === EventConstants.topLevelTypes.topT
    var isResponderRelease = responderID && !isResponderTerminate && isEndish(topLevelType) && n
    var finalTouch = isResponderTerminate ? eventTypes.responderTerminate : isResponderRelease ?
    if (finalTouch) {
      var finalEvent = ResponderSyntheticEvent.getPooled(finalTouch, responderID, nativeEvent, n
      finalEvent.touchHistory = ResponderTouchHistoryStore.touchHistory;
      EventPropagators.accumulateDirectDispatches(finalEvent);
      extracted = accumulate(extracted, finalEvent);
      changeResponder(null);
    }
```

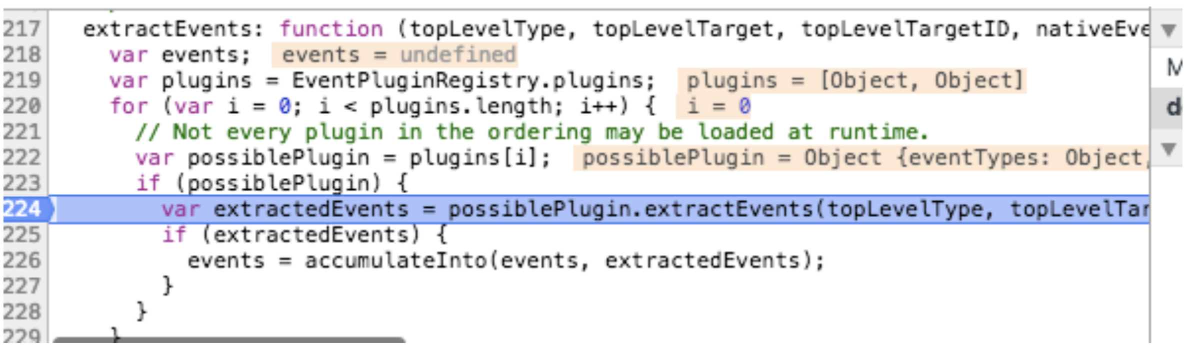
extracted的_dispatchListeners在这一行前是null
执行完这一行变为含有function scrollResponderHandleResponderRelease(e)回调
根据accumulate这个参数名大概知道是把finalEvent的内容放进extracted里了
看下finalEvent的内容验证一下



```
ResponderSyntheticEvent
  _dispatchIDs: ".r[1]{TOP_LEVEL}[0].$1.0.1.0.$sc
▼ _dispatchListeners: function ()
    __reactBoundArguments: null
  ▶ __reactBoundContext: Constructor
  ▶ __reactBoundMethod: function scrollResponderH
    arguments: (...)
  ▶ bind: function (newThis)
    caller: (...)
    length: 1
    name: "bound scrollResponderHandleResponderRe
  ▶ __proto__: function ()
  ▶ [[TargetFunction]]: function scrollResponderH
```

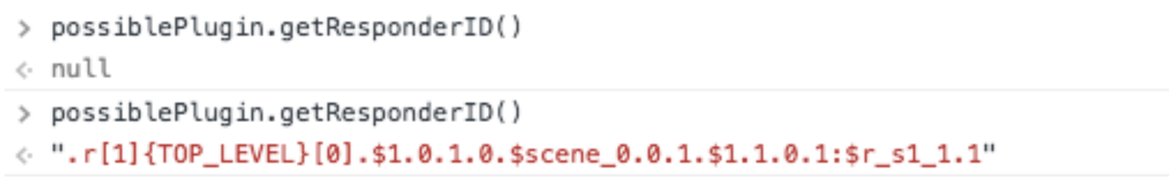
finalEvent在一开始声明
` var finalEvent = ResponderSyntheticEvent.getPooled(finalTouch, responderID, nativeEvent, nativeEventTarget);`
后大概就是这样子
然后我们看一下ResponderSyntheticEvent.getPooled的参数
正常：{
        finalTouch: "onResponderRelease"
        responderID: ".r[1]{TOP_LEVEL}[0].$1.0.1.0.$scene_0.0.1.$1.1.0.1:$r_s1_1.1"
        nativeEvent: …
        nativeEventTarget: …
}
异常：{
        finalTouch: "onResponderRelease"
        responderID: ".r[1]{TOP_LEVEL}[0].$1.0.1.0.$scene_0.0.1.$1.1"
        nativeEvent: …
        nativeEventTarget: …
}

可以看到responderID不同。。自然respond的方法也不同
于是我们watch responderID，找到它是在什么时候开始不同的:
发现responderID从ResponderSyntheticEvent.getPooled一开始就是不同的
想想既然touchEnd触发事件，那么touchStart理论上就没意义了，个人能想到唯一有可能的功能就是确定responderId
于是在touchStart时打个断点



```
extractEvents: function (topLevelType, topLevelTarget, topLevelTargetID, nativeEve
  var events;  events = undefined
  var plugins = EventPluginRegistry.plugins;  plugins = [Object, Object]
  for (var i = 0; i < plugins.length; i++) {  i = 0
    // Not every plugin in the ordering may be loaded at runtime.
    var possiblePlugin = plugins[i];  possiblePlugin = Object {eventTypes: Object,
    if (possiblePlugin) {
      var extractedEvents = possiblePlugin.extractEvents(topLevelType, topLevelTar
      if (extractedEvents) {
        events = accumulateInto(events, extractedEvents);
      }
    }
```

这行结束前后console输出一下possiblePlugin.getResponderID()

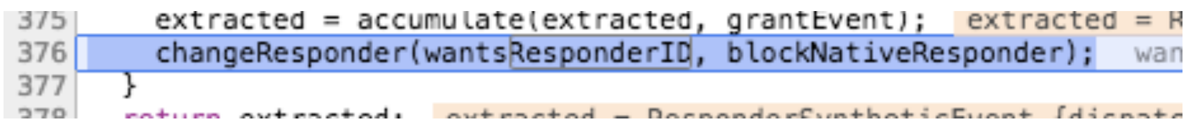


```
> possiblePlugin.getResponderID()
<· null
> possiblePlugin.getResponderID()
<· ".r[1]{TOP_LEVEL}[0].$1.0.1.0.$scene_0.0.1.$1.1.0.1:$r_s1_1.1"
```

假设正确

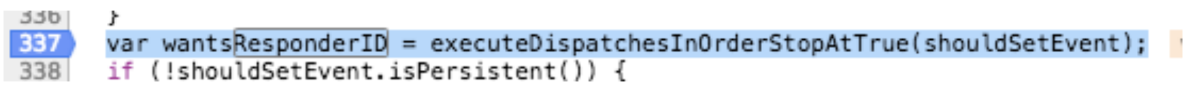
## 所以问题其实是出在接收touchStart时调用的extractEvents方法

在该方法里watch responderId:
在
    var extracted = canTriggerTransfer(topLevelType, topLevelTargetID, nativeEvent) ? setResponderAndExtractTransfer(topLevelType, topLevelTargetID, nativeEvent, nativeEventTarget) : null;
里被改变。。。。
从名字里也能看出来是setResponderAndExtractTransfer干的。。。进去看看
在



```
    extracted = accumulate(extracted, grantEvent);  extracted = R
    changeResponder(wantsResponderID, blockNativeResponder);  wan
  }
```

repsonderId被更改
更改结果为wantsResponderID
所以是wantsResponderID出错了
该执行路径下wantsResponderID只在声明时赋值了

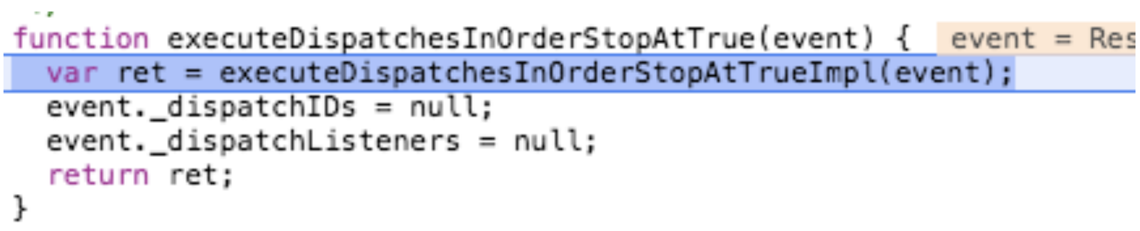


```
}
var wantsResponderID = executeDispatchesInOrderStopAtTrue(shouldSetEvent);
if (!shouldSetEvent.isPersistent()) {
```

有两种可能:

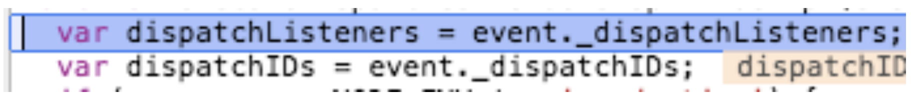
executeDispatchsInOrderStopAtTrue出错
参数shouldSetEvent不对

进executeDispatchsInOrderStopAtTrue看看：



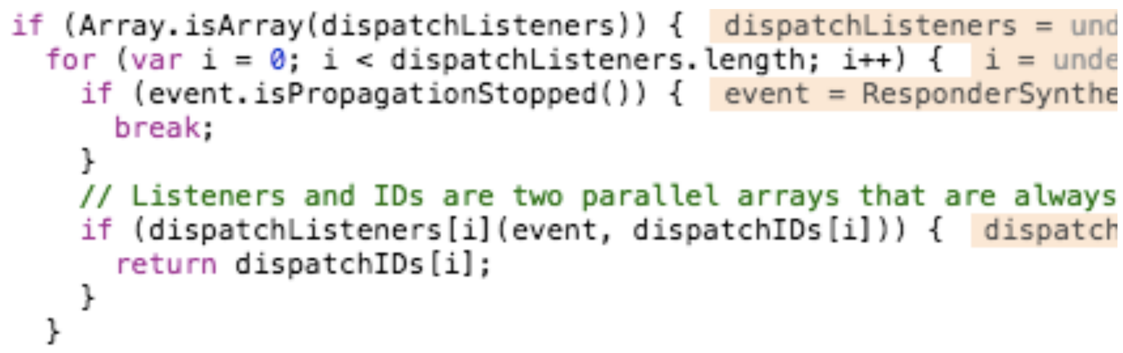
```
function executeDispatchesInOrderStopAtTrue(event) {  event = Res
  var ret = executeDispatchesInOrderStopAtTrueImpl(event);
  event._dispatchIDs = null;
  event._dispatchListeners = null;
  return ret;
}
```

再进executeDispatchesInOrderStopAtTrueImpl
核心代码如下



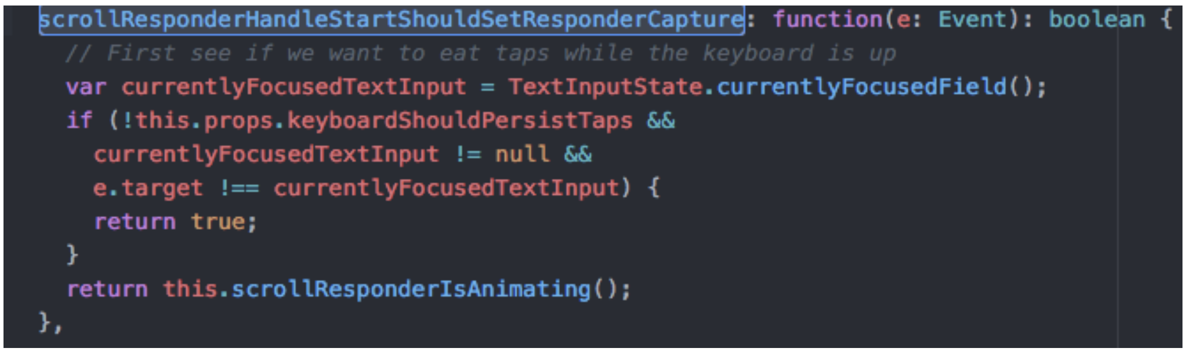
```
  var dispatchListeners = event._dispatchListeners;
  var dispatchIDs = event._dispatchIDs;  dispatchID
```

…



```
  if (Array.isArray(dispatchListeners)) {  dispatchListeners = und
    for (var i = 0; i < dispatchListeners.length; i++) {  i = unde
      if (event.isPropagationStopped()) {  event = ResponderSynthe
        break;
      }
      // Listeners and IDs are two parallel arrays that are always
      if (dispatchListeners[i](event, dispatchIDs[i])) {  dispatch
        return dispatchIDs[i];
      }
    }
```

大概就是找到第一个响应事件的listener然后返回listener所属的object的id
该循环在i=3时跳出
dispatchListeners[3] 里是scrollResponderHandleStartShouldSetResponderCapture
dispatchListeners[4] 里是touchableHandleStartShouldSetResponder
也就是说scrollResponderHandleStartShouldSetResponderCapture ”偷走”了事件
搜索下scrollResponderHandleStartShouldSetResponderCapture, 找到他的实现:



```
scrollResponderHandleStartShouldSetResponderCapture: function(e: Event): boolean {
  // First see if we want to eat taps while the keyboard is up
  var currentlyFocusedTextInput = TextInputState.currentlyFocusedField();
  if (!this.props.keyboardShouldPersistTaps &&
    currentlyFocusedTextInput != null &&
    e.target !== currentlyFocusedTextInput) {
    return true;
  }
  return this.scrollResponderIsAnimating();
},
```

注意那行注释。。。如果键盘打开他就会“eat taps”。。。。。。

**其实bug就出在这。。。下面是因为理解错误而白白多走得几步（React系统中parentView有一个方法专门“偷”子view事件，大概是因为是为了**

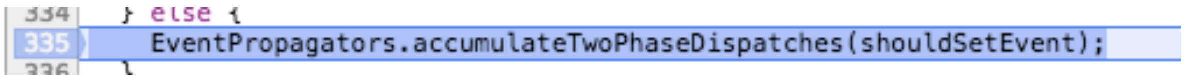
补偿没有子view处理事件过程中再把事件传给nextResponder）

所以bug出在这？其实仔细想想并不是。。。这个只是说了如果键盘打开他就会接收事件

在iOS中如果parent view 和 sub view 都能响应点击事件（UIResponder），那么subView会得到事件的优先处置权

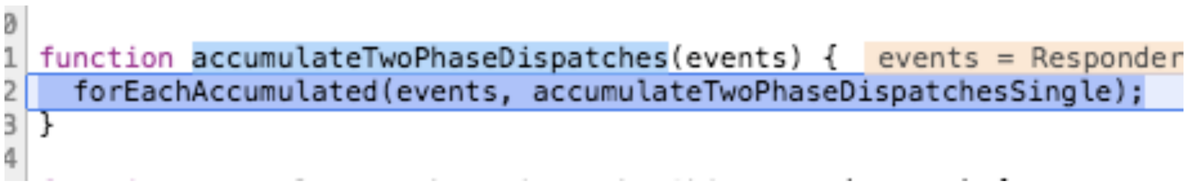
所以bug出在parentView的优先级高于subView，也就是event._dispatchListeners顺序错误

回到setResponderAndExtractTransfer方法。发现异常情况下shouldSetEvent._dispatchListeners在这一行被赋值：



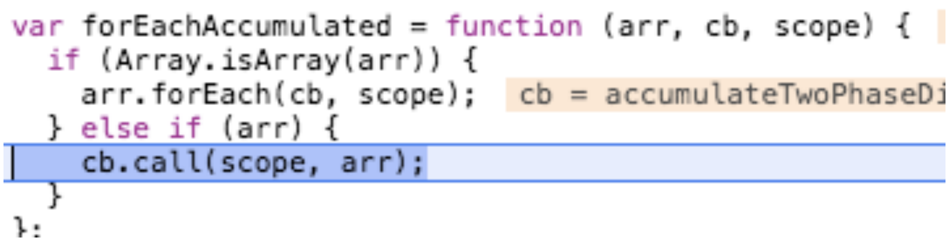
```
} else {
  EventPropagators.accumulateTwoPhaseDispatches(shouldSetEvent);
}
```

进去看看：



```
function accumulateTwoPhaseDispatches(events) {   events = Responder
  forEachAccumulated(events, accumulateTwoPhaseDispatchesSingle);
}
```

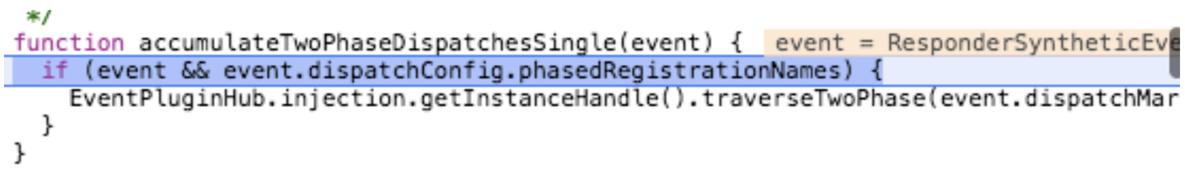
再进



```
var forEachAccumulated = function (arr, cb, scope) {
  if (Array.isArray(arr)) {
    arr.forEach(cb, scope);   cb = accumulateTwoPhaseDi
  } else if (arr) {
    cb.call(scope, arr);
  }
};
```

上三步可以简化为

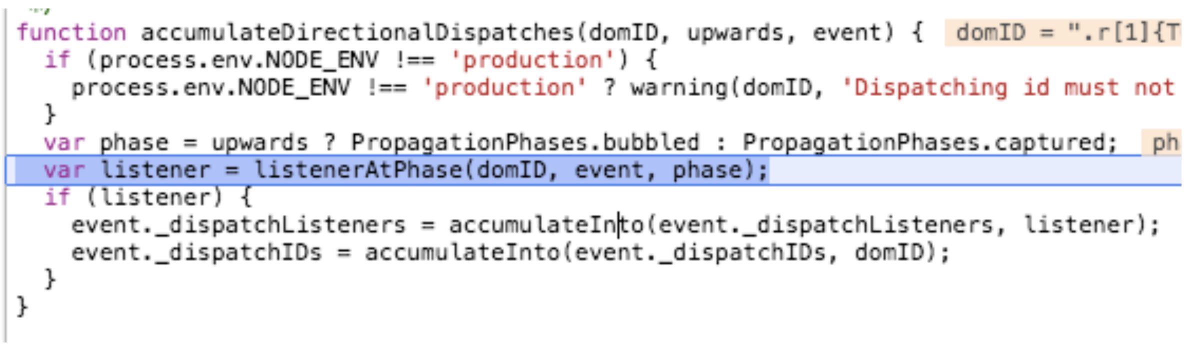
accumulateTwoPhaseDispatchesSingle(events)



```
 */
function accumulateTwoPhaseDispatchesSingle(event) {   event = ResponderSyntheticEve
  if (event && event.dispatchConfig.phasedRegistrationNames) {
    EventPluginHub.injection.getInstanceHandle().traverseTwoPhase(event.dispatchMar
  }
}
```

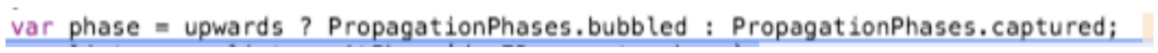
EventPluginHub.injection.getInstanceHandle().traverseTwoPhase(event.dispatchMarker,accumulateDirectionalDispatches, event);

大概做的事是从rootView遍历到标记为event.dispatchMarker，然后反向遍历（event.dispatchMarker会被访问两次，一次正向，一次反向），每个节点调用accumulateDirectionalDispatches,参数为(节点id，遍历方向,event)



```
function accumulateDirectionalDispatches(domID, upwards, event) {  domID = ".r[1]{T
  if (process.env.NODE_ENV !== 'production') {
    process.env.NODE_ENV !== 'production' ? warning(domID, 'Dispatching id must not
  }
  var phase = upwards ? PropagationPhases.bubbled : PropagationPhases.captured;  ph
  var listener = listenerAtPhase(domID, event, phase);
  if (listener) {
    event._dispatchListeners = accumulateInto(event._dispatchListeners, listener);
    event._dispatchIDs = accumulateInto(event._dispatchIDs, domID);
  }
}
```

大概就是问一下每个view：你接不接受这个event，接受的话就把你放到event的候选目标里
但是为啥会需要两个遍历阶段?



```
var phase = upwards ? PropagationPhases.bubbled : PropagationPhases.captured;
```

capturephase是啥?
查了一下文档：

**Gesture Responder System：http://facebook.github.io/react-native/releases/0.26/docs/gesture-responder-system.html#capture-shouldset-handlers**
**里面有这么一句话：**
However, sometimes a parent will want to make sure that it becomes responder. This can be handled by using the capture phase. Before the responder system bubbles up from the deepest component, it will do a capture phase, firing on*ShouldSetResponderCapture. So if a parent View wants to prevent the child from becoming responder on a touch start, it should have a onStartShouldSetResponderCapture handler which returns true.

。。。
嗯。。。其实这么说来bug原因就是scrollView不应该capture event。而是如果子view都不处理才去处理
打开scrollResponder把

scrollResponderHandleStartShouldSetResponderCapture和
scrollResponderHandleStartShouldSetResponder的实现互换就解决了。

这样还有个问题，如果打开键盘后直接点击返回键盘不消失，好办，在Touchable的响应方法里加一句这个就搞定了
`TextInputState.blurTextInput(TextInputState.currentlyFocusedField())`
当然这样textInput被点击时也会触发这个事件，但是因为iOS如果在一个时间周期内收到关、开键盘。他就会忽视前一个信号。所以一切正常啦~