

17 | Gesture（下）：如何解决多视图多手势的冲突问题？

2022-05-06 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 23:41 大小 21.70M



你好，我是蒋宏伟。

前一节课，我们讲解了手势进阶的一些内容，也分析了如何解决单视图多手势冲突的问题，但这个 Demo 其实挺基础的。今天我们要再深入一点，看一个稍微复杂点的案例，就是 Android 的回弹下拉刷新。

在 Gesture 的第一篇中我提到过，实现 Android 回弹下拉刷新的难点在于 Android 的 ScrollView 组件就没有滚动回弹属性 `bounces`。而 iOS 的 ScrollView 组件是有滚动回弹属性 `bounces` 的，而且是默认开启的。

在 Android 回弹下拉刷新案例中，会用到 Gesture 上中下三篇中的所有知识点，包括如何将手势库 Gesture 和动画库 Reanimated 搭配一起使用，如何解决单视图多手势的冲突问题，如何解决多视图多手势的冲突问题。

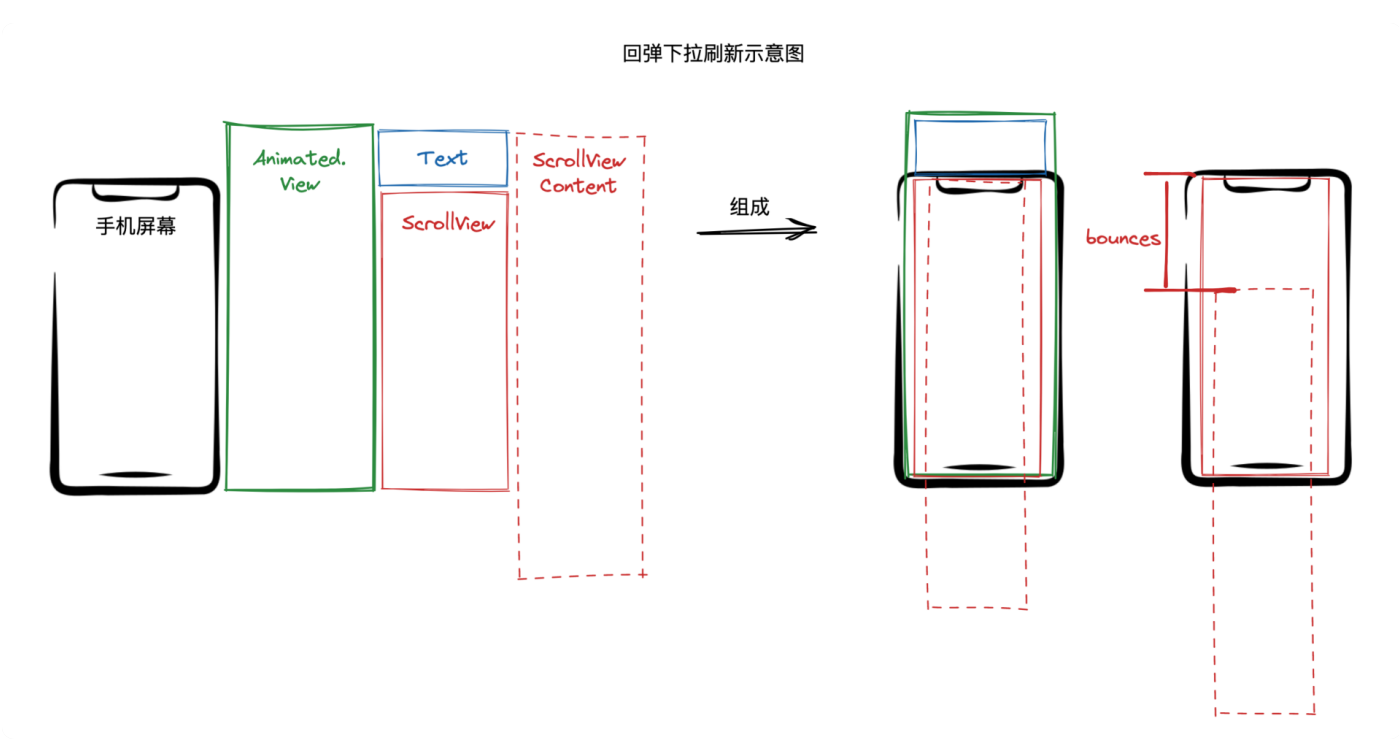
今天这一讲，一方面我会重点和你介绍如何解决多视图多手势的冲突问题，另一方面我会把 Gesture 上中下三篇的内容给你串起来，帮你实现 Android 回弹下拉刷新的效果。

Android 回弹下拉刷新

在真实的业务开发中，实现双端下拉刷新的正确逻辑是：**iOS 基于 bounces 实现，Android 基于手势实现**。不过，为了方便，我在写 Demo 的时候，直接把 iOS 的 bounces 效果关了，双端统一使用手势实现，省去了 if else 的代码，这样你看代码会容易一些。

那我们又应该从哪里开始进行实现回弹下拉刷新？

要实现回弹下拉刷新，首先要理解 Android 回弹滚动的原理，我们先来看一下 **Android 回弹滚动的结构示意图**：



你可以看到，Android 回弹滚动涉及的结构一共有 5 种，最外层是手机屏幕，手机屏幕内有一个比屏幕高一些的 Animated.View。Animated.View 比屏幕高出来的部分，正好等于 Loading 视图的高度，Loading 视图这里我直接用 Text 元素代替了。Animated.View 和屏幕同高的部分是 ScrollView 视图。在 ScrollView 视图的内部，ScrollView 的内容 Content 部分是比 ScrollView 容器更高的，这样内容才能滚动。

如果从 ScrollView 内容的最底部开始，一直往上滚，滚到内容的最顶部。在没有 bounces 回弹效果的容器中，内容的顶部和手机屏幕的顶部是平齐的。如果支持 bounces 回

弹或手势回弹，那么内容还可以继续往下拉，内容的顶部可以低于手机屏幕顶部。

这里有个小细节。你可能注意到了，我并没有使用 **absolute** 绝对定位将 **Text** 定位到手机屏幕上方的位置，而是增加了其父容器 **Animated.View** 的高度。这是因为，我以前遇到过 **Android** 手机子视图超出父容器后不显示的问题。为了避免超出不显示，在 **Animated.View** 的子视图这里，我采用的是从上到下的默认布局方式，把子视图都包裹在 **Animated.View** 视图内部，而不是让子视图 **Loading** 浮在 **Animated.View** 视图的外面。

那上述回弹下拉刷新的 **JSX** 实现是什么样的呢？

JSX 部分的核心代码如下：

 复制代码

```
1  const LOADING_HEIGHT = 30
2  const {height: windowHeight} = useWindowDimensions()
3  const wrapperHeight = windowHeight + LOADING_HEIGHT
4  // ...
5  return (
6    <Animated.View style={[{ height: wrapperHeight }, animatedStyle]}>
7      <Text style={{height: LOADING_HEIGHT}}>loading...</Text>
8      <GestureDetector gesture={Gesture.Simultaneous(scrollGesture, panGesture)}>
9        <Animated.ScrollView bounces={false} onScroll={scrollHandler}/>
10     </GestureDetector>
11   </Animated.View>
12 )
```

在上述结构代码中，我给 **ScrollView** 添加了两个可以同时执行的手势：一个是 **ScrollView** 自身的滚动手势 **scrollGesture**；另一个是拖拽手势 **panGesture**。同时我还给 **Animated.View** 添加了一个动画样式 **animatedStyle**。

这样做的目的是，我们可以通过滚动手势、拖拽手势和动画的配合，实现 **Android** 回弹下拉效果。

了解完 **JSX** 结构后，我们再来看驱动拖拽动画的共享值。**回弹下拉一共涉及了两个共享值，也就是 **scrollY** 和 **refreshY**。**

- **scrollY**： **ScrollView** 滚动偏移量，相关代码如下：

```

1  const scrollY = useSharedValue(0);
2
3  const scrollHandler = useAnimatedScrollHandler({
4    onScroll: e => {
5      // 记录偏移量，只读不写
6      scrollY.value = e.contentOffset.y;
7    },
8  });
9
10 // ...
11 <Animated.ScrollView onScroll={scrollHandler} scrollEventThrottle={1}/>

```

由于 `ScrollView` 的滚动偏移量是由原生平台控制的，`Animated` 动画库和 `Gesture` 手势库都控制不了，因此 `scrollY` 只可读、不可写。读取 `scrollY` 靠的是 `ScrollView` 的 `onScroll` 回调和 `Reanimated` 的 `useAnimatedScrollHandler` 的配合，整个过程在 UI 线程中进行。另外，我们也通过 `scrollEventThrottle` 属性，将两次 `onScroll` 回调的执行间隔设置为 `1ms`，以此来保证获取 `scrollY` 的时效性。

- `refreshY`: `Animated.View` 拖拽偏移量，示例代码如下：

```

1  const LOADING_HEIGHT = 30
2  const refreshY = useSharedValue(-LOADING_HEIGHT);
3  const {height: windowHeight} = useWindowDimensions()
4  const wrapperHeight = windowHeight + LOADING_HEIGHT
5
6  const animatedStyle = useAnimatedStyle(() => {
7    return {
8      transform: [{translateY: refreshY.value}],
9    };
10 });
11
12 return (
13   <Animated.View style={[{height: wrapperHeight}, animatedStyle]}>
14     {/* ... */}
15   </Animated.View>
16 );

```

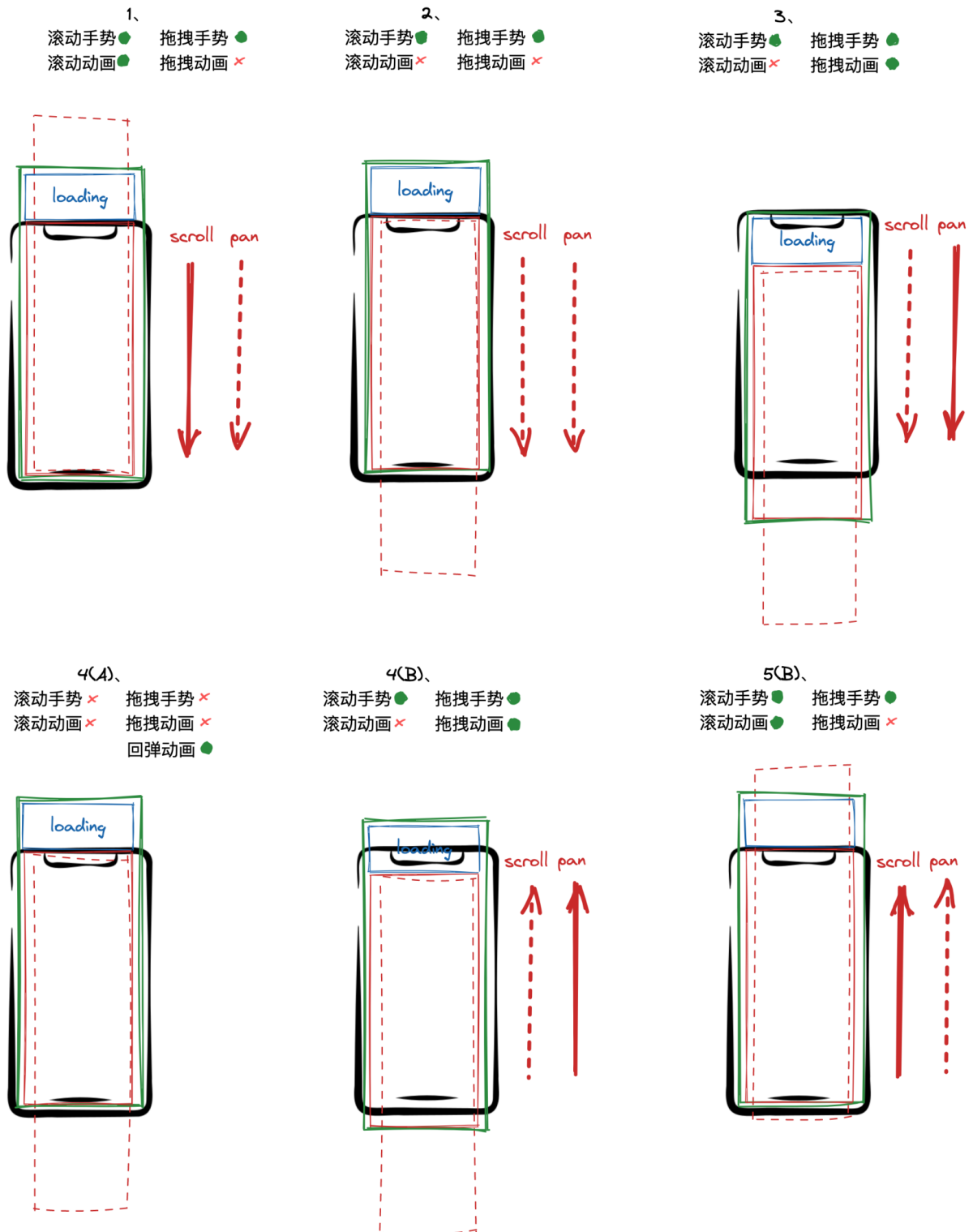
拖拽偏移量是用来控制整体视图 `Animated.View` 的纵轴偏移量的，包括 `ScrollView` 和 `Loading`。默认 `refreshY` 的值为 `-30`，也就是 `LOADING_HEIGHT` 的负值，此时正好把 `Loading` 隐藏在屏幕外。

和 scrollY 只读不写不同，**refreshY 的值会跟着拖拽手势的变化而变化**。当你拖拽下拉时，refreshY 的值从 -30 逐渐变大；当 refreshY 的值变为 0 时，Loading 字样会完全出现；继续向下拖拽，refreshY 会继续变大，页面继续下移，松手时，重新变为 -30。

了解了整体的 JSX 结构和共享值后，接下来我们面临的难题就是，回弹滚动究竟如何实现呢？

一图胜千言，我还画一张原理示意图，相信你看一下就能明白：

Android 弹性滚动原理示意图



示意图一共分为 6 步，有两种可能结果。

首先，在初始化时，你先将 `ScrollView` 内容滚动到最底部，然后手势向下往内容的最顶部方向滚动。此时，因为你的手是一直放在 `ScrollView` 视图上的，所以 `ScrollView` 视图会同时响应滚动手势和拖拽手势。但是滚动手势也就意味着触发了滚动动画，而拖拽手势只是触发了拖拽回调 `onChange`，但回调中并未改变共享值 `refreshY`，也未执行拖拽动画。

绑定手势的相关代码如下：

 复制代码

```
1  const scrollGesture = Gesture.Native()
2  const panGesture = Gesture.Pan()
3    .onChange(e => {
4    // 拖拽动画
5    if (scrollY.value === 0 || refreshY.value !== -LOADING_HEIGHT) {
6      refreshY.value = Math.max(-LOADING_HEIGHT, refreshY.value + e.changeY) ;
7    }
8  })
9    .onEnd(() => {})
10
11  const animatedStyle = useAnimatedStyle(() => {
12    return {
13      transform: [{translateY: refreshY.value}],
14    };
15  });
16
17  return (
18    <GestureDetector gesture={Gesture.Simultaneous(scrollGesture, panGesture)}>
19      <Animated.ScrollView />
20    </GestureDetector>
21  )
```

接着，你继续不松手地向下滚动 **ScrollView** 的内容，直到内容滚动到了最顶部。此时 **scrollY** 的值正好为 0，**refreshY** 的值也正好为 **-LOADING_HEIGHT**，此时拖拽动画处于还未触发但即将触发的临界点。

然后你再继续不松手地向下滚动和拖拽。这时你要注意一下 **ScrollView** 的滚动条，你会发现滚动条是一直显示的，但没有位置和长度的变化了，这代表 **ScrollView** 内容触顶了，并且不能回弹滚动了。而这时拖拽手势改变了共享值 **refreshY**，并开始执行拖拽动画。拖拽动画让外层容器 **Animated.View** 改变了它的 **translateY** 偏移量，所以 **Animated.View** 和其内部的 **Loading** 文字会一起往下移，于是你就在手机屏幕上看到了 **Loading** 文字。

接下来你会有两种选择，**A** 选择是松手刷新页面，**B** 选择是往反方向滚动取消刷新。

我们先来看选择了 **A** 会怎么样。如果你选择了松手刷新页面，那么拖拽手势和滚动手势会同时结束，此时只需要在手势结束回调中将 **Animated.View** 的 **translateY** 偏移量设置为默认值即可。这样所有视图都会恢复默认的位置，而 **Loading** 文字也会消失在屏幕中。

松手触发拖拽手势 onEnd 回调的代码如下：

 复制代码

```
1  const panGesture = Gesture.Pan()
2  // ...
3  .onEnd(() => {
4    // 松手时，如果容器整体偏离正常位置
5    if (refreshY.value !== -LOADING_HEIGHT) {
6      // 则使用弹性动画 withSpring，回弹至原位置
7      refreshY.value = withSpring(-LOADING_HEIGHT, {
8        stiffness:300,
9        overshootClamping: true
10     })
11   }
12 })
```

你可以看到，在你松手时，panGesture 手势的内部状态会由 **ACTIVE** 变为 **END**，并触发 onEnd 回调。在 onEnd 回调中，只有容器整体偏离正常位置，也就是 refreshY 的值不等于 -LOADING_HEIGHT，才会使用弹性动画 withSpring 将 refreshY 重置为 -LOADING_HEIGHT。

其中，弹性动画的 **stiffness** 指的是“弹簧硬度”，硬度越大弹簧弹的速度越快，“弹簧硬度”默认是 100，这里设置成了 300 的意思是希望回弹的速度快一点。弹性动画的 **overshootClamping** 指的是“夹住过冲”，默认 overshootClamping 的值是 **false**，这时弹簧会沿着它的默认形变的中心线来回反复地弹。我这里将它设置为了 **true**，也就是说，弹簧在回到默认形变的中心线的时候就会停下来，整体容器的松手动画会直接停在屏幕上边缘。

那如果你的选择是 **B** 方案呢？

如果你选择了往反方向滚取消刷新，那么滚动手势和拖拽手势还会同时响应，并且拖拽动画还会继续执行，拖拽动画向反方向改变 Animated.View 的 translateY 偏移量。具体代码你可以看下这里：

 复制代码

```
1  // 拖拽动画
2  if (scrollY.value === 0 || refreshY.value !== -LOADING_HEIGHT) {
3    refreshY.value = Math.max(-LOADING_HEIGHT, refreshY.value + e.changeY) ;
4  }
```


这里你可以注意下执行拖拽动画的判断条件。因为我设置了只要 `scrollY` 的值为 0 或者 `refreshY` 的值不为 `-LOADING_HEIGHT` 时，也就是 `ScrollView` 内容顶到头或者整体视图不在正确的位置上，就可以触发拖拽手势，因此这时我们是可以反方向向上拖拽的。

但这里有个坑，向上拖拽时 `ScrollView` 内容也是可以向上滚动的，但我们这里并没有禁止 `ScrollView` 内容的滚动。这个坑怎么处理呢？我们后面再聊，我们先接着看最后一步。

如果你继续不松手的向上滚动，滚动手势和拖拽手势还会同时响应，但因为整体视图回到默认位置了所以拖拽动画不会执行了，而是滚动动画开始执行，`ScrollView` 的内容开始向上滚动。

整个回弹下拉刷新的手势动效是连贯的，整个过程中都不需要通过松手来切换拖拽动画和滚动动画，这就是 `Gesture` 手势库和 `Reanimated` 动画库的强大之处。

实现 Android 回弹下拉的核心代码，我放到这里，你可以仔细看看：

 复制代码

```
1  const LOADING_HEIGHT = 30
2
3  function PanAndScrollView() {
4    const refreshY = useSharedValue(-LOADING_HEIGHT);
5    const scrollY = useSharedValue(0);
6    const {height: windowHeight} = useWindowDimensions()
7    const wrapperHeight = windowHeight + LOADING_HEIGHT
8
9    const scrollGesture = Gesture.Native()
10
11    const panGesture = Gesture.Pan()
12      .onChange(e => {
13        // 滚动到顶部或者容器整体偏离正常位置时，可触发手势动画
14        if (scrollY.value === 0 || refreshY.value !== -LOADING_HEIGHT) {
15          refreshY.value = Math.max(-LOADING_HEIGHT, refreshY.value + e.changeY)
16        }
17      })
18      .onEnd(() => {
19        // 松手时，如果容器整体偏离正常位置
20        if (refreshY.value !== -LOADING_HEIGHT) {
21          // 则使用弹性动画 withSpring，回弹至原位置
22          refreshY.value = withSpring(-LOADING_HEIGHT, {
23            stiffness: 300,
24            overshootClamping: true
25          })
26        }
27      })
28
```

```

29     const animatedStyle = useAnimatedStyle(() => {
30         return {
31             transform: [{translateY: refreshY.value}],
32         };
33     });
34
35     const scrollHandler = useAnimatedScrollHandler({
36         onScroll: e => {
37             // 记录偏移量，只读不写
38             scrollY.value = e.contentOffset.y;
39         },
40     });
41
42     return (
43         <Animated.View style={[{height: wrapperHeight}, animatedStyle]}>
44             <Text style={{height: LOADING_HEIGHT}}>loading...</Text>
45             <GestureDetector gesture={Gesture.Simultaneous(scrollGesture, panGesture)}>
46                 <Animated.ScrollView
47                     bounces={false}
48                     onScroll={scrollHandler}
49                     scrollEventThrottle={1}>
50                     {Array(100).fill(1).map((_, index) => (<Text key={index}>{index}<
51                         </Animated.ScrollView>
52                     </GestureDetector>
53                 </Animated.View>
54             );
55     }

```

多视图多手势的冲突问题

刚刚我们实现的 Android 回弹下拉刷新功能，大体上是能用的，但是它还有两个小的体验问题。

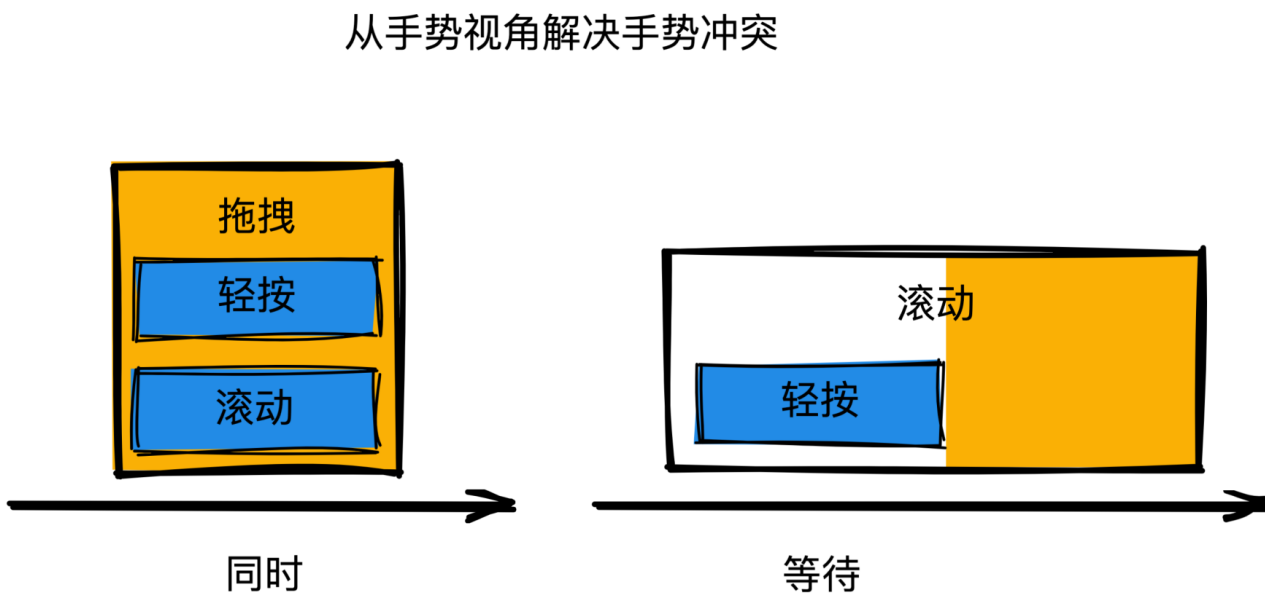
首先是 Loading 本身不能不响应拖拽手势，这就限制了回弹下拉刷新功能的通用性。如果你想把 Loading 替换成类似淘宝二楼的效果，用刚刚我们实现的下拉刷新组件来做就会有 Bug。要知道，二楼视图的高度可要比只有 30 像素的 Loading 高很多，用户很容易拖拽到二楼视图，如果用户拖拽后发现没有反应，肯定会感觉到很奇怪。

另外一个体验方面的问题是，在 B 方案中，也就是**不松手而是反向滚动或拖拽这个步骤**时，滚动动画和拖拽动画都没有禁止，二者可能会同时触发，这会导致出现两个叠加视图偏移问题。

要让 ScrollView 视图和 Loading 视图同时监听滚动和拖拽手势，要让执行拖拽动画时不能执行滚动动画，就涉及多视图多手势的冲突问题了。

那么，多视图多手势冲突问题该怎么解决？

解决多视图多手势的冲突问题，我们首先要学会站在单个手势的视角来解决这个问题，我给你画了一个示意图，你先看看：



示意图中左边的部分，就是站在拖拽手势的视角来解决冲突问题的。拖拽手势是这么想的：既然你想在响应我拖拽手势的同时响应轻按、滚动手势，那我可以提供一个方法函数，你把轻按、滚动手势都告诉我吧。

咦？这里多了个轻按手势，它是用来干嘛的呢？我们先保留一个悬念，你一会就知道了。

然后我们再看示意图中右边的部分，这是站在滚动手势的视角来解决冲突问题的。滚动手势是这么想的：你想让我滚动手势和拖拽手势同时响应，但不想让滚动动画和拖拽动画同时执行，但我滚动手势并不知道拖拽动画是否能执行呀！

那么这要怎么办呢？

我们可以换个思路。我滚动手势虽然不知道动画逻辑是什么时候执行的，但能够知道其他手势什么时候执行完成呀。要不这样？你先创建一个假的轻按手势，当拖拽动画不可执行时，你就主动把轻按手势结束。这样我收到轻按手势结束的通知时，就知道拖拽动画不可执行了，这时我再把滚动动画由不可执行的状态变为可执行的状态。

以上就是我们站在手势的视角，解决两个下拉刷新体验问题的核心思路，这也是为什么前面的示意图中会多一个轻按手势的原因。

那具体怎么实现呢？

我先带你看下新的 JSX 结构的实现，代码如下：

 复制代码

```
1 <GestureDetector gesture={panGesture}>
2   <Animated.View style={[{height: wrapperHeight}, animatedStyle]}>
3     <Text>loading...</Text>
4     <GestureDetector gesture={Gesture.Simultaneous(scrollGesture, tapGesture)}>
5       <Animated.ScrollView/>
6     </GestureDetector>
7   </Animated.View>
8 </GestureDetector>
```

这段代码就是站在视图的视角，把手势和视图绑定在一起了。这里有三个手势，分别是 `panGesture`、`scrollGesture`、`tapGesture`。当你手指触碰到外层容器 `Animated.View` 时，`panGesture` 就会响应。这样无论你是触碰到它的子容器 `Text`，还是 `ScrollView`，都能触发 `panGesture` 手势。

而 `scrollGesture` 手势只能在触碰到 `ScrollView` 视图时进行响应，而且我还配了一个控制滚动动画是否执行的 `tapGesture` 手势。

接下来的代码，就是站在 `panGesture` 手势的视角，让它支持和 `scrollGesture`、`tapGesture` 这两个手势同时响应，示例代码如下：

 复制代码

```
1 const tapGesture = Gesture.Tap()
2 const scrollGesture = Gesture.Native()
3 const panGesture = Gesture.Pan()
4   .simultaneousWithExternalGesture(scrollGesture, tapGesture)
```

上述代码中，`panGesture` 手势调用了 `simultaneousWithExternalGesture` 方法，方法入参是 `scrollGesture`、`tapGesture`。这段代码的意思是，在响应我 `panGesture` 手势时，可以同时响应 `scrollGesture`、`tapGesture` 手势。

Gesture 手势库中的 9 个手势，每个手势对象上都有 `simultaneousWithExternalGesture` 方法，该方法接收若干个其他手势作为参数，作用是让该手势能和若干个其他手势同时进行响应。

然后我们再站在 `scrollGesture` 手势的视角，让它在整体视图没有回归到正常位置的时候，不执行滚动动画，示例代码如下：

 复制代码

```
1 // hack: 使用 tapGesture 手势作为控制 scrollGesture 是否执行动画的开关
2 // 并不是真的要响应 Tap 手势
3 const tapGesture = Gesture.Tap()
4 .onTouchesMove((_, manager) => {
5     // 如果 ScrollView 容器没有顶到屏幕顶部
6     if (LOADING_HEIGHT + refreshY.value === 0) {
7         // 则设置 Tap 手势内部状态为 FAILED
8         manager.fail();
9     } else {
10        // 其他情况则设置 Tap 手势内部状态为 ACTIVE
11        // 因为 Tap 手势实际触发了，所以内部也会调用
12        // 这里又显式调用了一次，为的是让大家看得更明白一些。
13        manager.activate();
14    }
15 })
16 .maxDuration(1000000);
17
18 const scrollGesture = Gesture.Native()
19 // 当 Tap 手势内部状态为 ACTIVE 时，滚动动画不执行
20 // 当 Tap 手势内部状态为 FAILED 时，滚动动画执行
21 .requireExternalGestureToFail(tapGesture);
```

这段代码虽然很简单，但其实是一种 **hack** 方法，为了让你看得更明白一些，我又加了很多注释。

代码中先创建了一个 `tapGesture` 手势，在手势的 `onTouchesMove` 回调中执行了控制其内部状态的逻辑，只有当外层容器的偏移量 `refreshY` 和 `LOADING_HEIGHT` 的高度抵消时，整体视图才回归到正常位置，此时将 `tapGesture` 内部状态设置为 `FAILED`。

并且，我还调用了 `tapGesture` 的 `maxDuration` 方法，这个值的默认值只有 500ms。我将其最大响应时间设置为 1000000ms，大概是 16 分钟。在这 16 分钟内，**Gesture** 手势库不会直接结束 **Tap** 事件，只有在我主动调用 `FAILED` 时，或者手指离开屏幕时才会主动结束，这样就保证了我的 **hack** 逻辑正常执行了。

接着我又在 `scrollGesture` 手势中调用了 `requireExternalGestureToFail` 方法，该方法的入参是 `tapGesture`，其作用是当 `tapGesture` 手势的内部状态置为 `FAILED` 时，开始执行滚动动画。

其中，`requireExternalGestureToFail` 方法在 `Gesture` 手势库中的 9 个手势对象上都能调用。该方法接收若干个其他手势作为参数，只有在其他若干个手势都失败后，该手势才会变为 **ACTIVE** 响应，在滚动手势上表现为执行滚动动画。

这样我就通过 `hack` 的手段，解决了滚动动画和手势动画叠加导致的视图异常偏移的潜在问题。

优化后的 `Android` 回弹滚动示例代码如下：

 复制代码

```
1  const LOADING_HEIGHT = 200;
2
3  function PanAndScrollView() {
4    const refreshY = useSharedValue(-LOADING_HEIGHT);
5    const scrollY = useSharedValue(0);
6    const {height: windowHeight} = useWindowDimensions();
7    const wrapperHeight = windowHeight + LOADING_HEIGHT;
8
9    const tapGesture = Gesture.Tap()
10     .onTouchesMove((_, manager) => {
11       if (LOADING_HEIGHT + refreshY.value === 0) {
12         manager.fail();
13       } else {
14         manager.activate();
15       }
16     })
17     .maxDuration(1000000);
18
19    const scrollGesture = Gesture.Native()
20     .requireExternalGestureToFail(tapGesture);
21
22    const panGesture = Gesture.Pan()
23     .onChange(e => {
24       if (scrollY.value === 0 || refreshY.value !== -LOADING_HEIGHT) {
25         refreshY.value = Math.max(-LOADING_HEIGHT, refreshY.value + e.changeY);
26       }
27     })
28     .onEnd(() => {
29       if (refreshY.value !== -LOADING_HEIGHT) {
30         refreshY.value = withSpring(-LOADING_HEIGHT, {
31           stiffness: 300,
32           overshootClamping: true,
33         });
34     });
```



```

34     }
35   })
36   .simultaneousWithExternalGesture(scrollGesture, tapGesture);
37
38   const animatedStyle = useAnimatedStyle(() => {
39     return {
40       transform: [{translateY: refreshY.value}],
41     };
42   });
43
44   const scrollHandler = useAnimatedScrollHandler({
45     onScroll: e => {
46       scrollY.value = e.contentOffset.y;
47     },
48   });
49
50   return (
51     <GestureDetector gesture={panGesture}>
52       <Animated.View style={[{height: wrapperHeight}, animatedStyle]}>
53         <Text style={{height: LOADING_HEIGHT,}}>
54           loading...
55         </Text>
56         <GestureDetector
57           gesture={Gesture.Simultaneous(scrollGesture, tapGesture)}>
58           <Animated.ScrollView onScroll={scrollHandler} scrollEventThrottle={1
59             {Array(100).fill(1).map((_, index) => (<Text key={index}>{index}</
60             </Animated.ScrollView>
61           </GestureDetector>
62         </Animated.View>
63       </GestureDetector>
64     );
65   }

```

到这里，我们就通过站在单个手势的视角拆解问题，解决了多视图多手势的冲突。我们现在再回到 **Gesture** 第一讲中我提到的 3 个曾经困扰过我的问题：**Android** 回弹下拉刷新、类似抖音评论区的手势动效、类似淘宝首页的手势动效。

通过这三讲的学习，相信你已经知道怎么去解决第一个问题了。剩下的两个问题，我也找到了类似的解决方案，你可以看下我 [🔗 GitHub](#) 上的代码。有什么问题，请在评论区给我留言。

- [🔗 Android 回弹下拉刷新](#)（本节课案例）；
- [🔗 类似抖音评论区的手势动效](#)（改自 **Gesture** 手势库提供的 `bottom_sheet` 示例）；
- [🔗 类似淘宝首页的手势动效](#)（找到一个开源库，但用的是 **Reanimated v2 + Gesture v1** 实现的）。

如果你觉得光有图片，没有视频，入门比较费劲，你可以搭配 [🔗 《Introduction to Gesture Handler 2 \(React Native\)》](#) 视频教程一起学习。

总结

在平时和大家交流的时候，我发现不仅仅是我遇到了 **React Native** 的手势冲突问题，大家也经常遇到手势冲突的问题，但不知道怎么解决。

从技术上，解决思路有三个要点：

首先，手势动画不分家，将 **Reanimated v2 + Gesture v2** 搭配起来用，它俩的回调都是放在 **UI 线程**同步执行的，性能和体验上会更好。

其次，**Gesture** 手势库提供可扩展性强的、功能丰富的“**1 + 8**”种手势，**1** 是给你自定义的原始手势，**8** 是已经封装好的 **8** 种常用的手势。

最后，**Gesture** 手势库创新地站在组件角度、站在手势角度给出了手势冲突的解决方案，这两种解决方案完全可以替代 **React Native** 框架原有的、站在事件角度的捕获冒泡方案，而且解决了多视图多手势冲突的问题。

另外，我觉得当你遇到业内已有类似解决方案的问题时，不妨多看看 **Github** 社区上的代码，很多社区库在 **Github** 上都提供了 **Example** 示例，这些示例非常有价值，我实现 **Android** 回弹下拉刷新和类似抖音评论区的手势动效这两个 **demo**，都是从 **Gesture** 手势库的 [🔗 Example 示例](#)中找到的灵感。

有空的时候，你也可以多看看这些优秀开源库的示例，相信这对你的成长会特别有帮助。


作业

1. 请你参考这一讲中的 **Android** 下拉刷新 **Demo**，实现一个类似淘宝二楼的手势效果。
2. 请你说说你在开发 **React Native** 时遇到过哪些手势动效问题？学完这三讲后你有没有新的解决思路？

欢迎在评论区写下你的观点和想法。我是蒋宏伟，咱们下节课见。

分享给需要的人，Ta订阅超级会员，你最高得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | Gesture（中）：如何解决单视图多手势的冲突问题？

下一篇 18 | Navigation：页面之间怎么跳转？

精选留言 (2)

 写留言



abc 
2022-05-10

有没有可能跟RecyclerView组件结合起来，实现android高性能的长列表弹性滚动

作者回复: 可以的。RecyclerView 提供了相关属性，可以把默认的 ScrollView 替换成 Animated.ScrollView。



风星舞
2022-05-07

请问GitHub仓库中运行android时的这个 yarn install-android-hermes 命令是什么作用？

作者回复: 这个是开启 Hermes 的命令；现在可以直接通过配置同时开启新架构的 Hermes 和 Fabric 了。

On Android you can set newArchEnabled=true inside the android/gradle.properties file.



