

15 | Gesture（上）：如何实现一个拖拽动效？

2022-05-02 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 22:53 大小 20.97M



你好，我是蒋宏伟。

我刚开始做 React Native 开发的时候，曾经被手势问题困扰过好几次。

第一次，我想在 Android 上实现类似 iOS 的下拉刷新效果。你可能知道，iOS 的 `ScrollView` 组件是有回弹属性 `bounces` 的。当开启回弹效果时，`ScrollView` 的内容区顶到头还可以继续往下拉，但 Android 的 `ScrollView` 组件就没有 `bounces` 属性，实现不了这种带回弹的下拉刷新效果。

第二次，我是想实现类似抖音评论区的手势动效。这个手势动效在上下方向存在三个手势，分别是最外层视频区域的上下切换动画、评论框的上下拖拽动画和评论内容的上下滚动动画。这种多视图、多手势的动效，本身就非常复杂，而且当时 React Native 框架自带的手势动画模块的能力太弱，也实现不了。

第三次，我想实现类似淘宝首页的手势动效。淘宝首页头部区域是由轮播图、金刚区等组成的固定内容区域，底部区域是由多 Tab、多长列表组成的可左右切换、可上下滚动的区域，实现难度非常高。

我提到的这三个手势动效的需求，都需要手势和动画搭配在一起才能实现。

但当初我用的是 React Native 的 0.44 版本，因为社区的 Gesture 手势库 [@react-native-gesture-handler](#) 和 Reanimated 动画库 [@react-native-reanimated](#) 都还不太成熟，所以我选择了 React Native 框架自带的手势模块 [@PanResponder](#) 和动画模块 [@Animated](#) 进行开发。但是仅仅只是如何解决手势冲突这个问题，就把我拦住了，只能降级处理。

但今时不如往日，在现在的 React Native 的 0.68 版本，也就是新架构预览版中，社区的 Gesture v2 手势库和 Reanimated v2 动画库都已提前适配，这几个曾经困扰我的难题都可以实现出来了。

要解决这三个问题，最重要的就是要**解决手势冲突的问题**。这一次，我们就围绕着这个核心，带你由浅到深，吃透手势的使用及其原理，日后你再遇到这些复杂的需求就都能迎刃而解了。

不过，由于要讲的内容比较多，我会分成三讲，也就是通过三个关卡带你逐级突破手势难题。今天的第一关就是：手势的基础原理以及如何进行基本手势，比如拖拽动效的开发。

手势基础

学习手势，我们当然要先了解手势的基本原理。

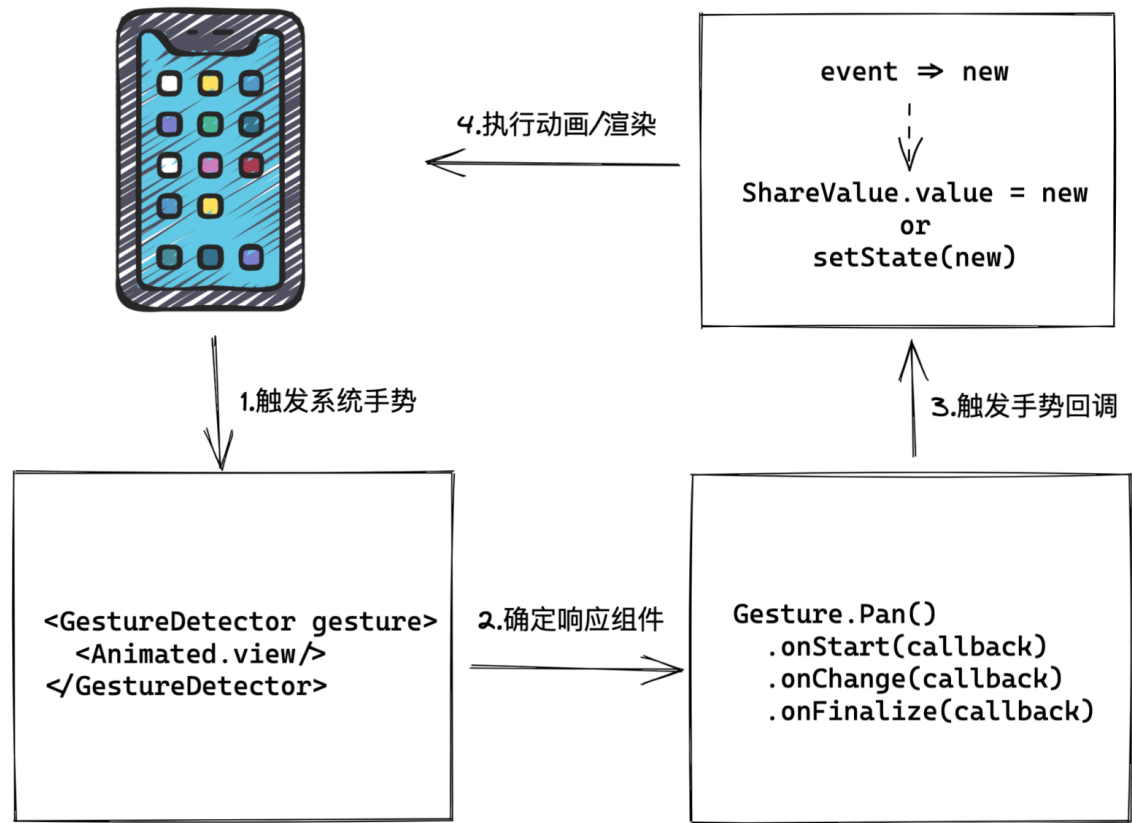
Pressable 点按手势就是最基础的手势。还记得我在 Pressable 一讲中，和你提到的点按手势的基本原理吗？响应 Pressable 手势要经过如下过程：

整个点按事件的响应过程是硬件和软件相互配合的过程。Pressable 组件响应的整体流程，是从触摸屏识别物理手势开始，到系统和框架 Native 部分把物理手势转换为 JavaScript 手势事件，再到框架 JavaScript 部分确定响应手势的组件，最后到 Pressable 组件确定是点击还是长按。

这里，框架的 JavaScript 部分指的就是 React Native 自带的 PanResponder 手势系统，它是运行在 JavaScript 线程的。

而今天我要和你介绍的手势库 **Gesture**，虽然在声明的初始化过程是运行在 **JavaScript** 线程中的，但声明之后的手势回调函数的执行都是默认运行在 **UI** 线程的，并且它和 **Reanimated** 可以很好地结合起来，一起使用。因此手势动效的全过程都是可以运行在 **UI** 线程的，不受 **JavaScript** 线程性能瓶颈的约束。

Gesture 手势库实现人机交互的原理图如下：



图标来源于 www.flaticon.com

你可以看到，使用 **Gesture** 手势库进行人机交互的基础流程一共分为 4 步：

1. **Gesture** 手势库收到系统手势事件；
2. **Gesture** 手势库确定需要响应哪些组件；
3. **Gesture** 手势库触发相关手势回调函数；
4. 通过 **Gesture** 回调函数返回 `event` 参数，使用回调返回值 `event` 可以更新共享值或状态，执行动画或渲染。

原理讲起来可能会比较抽象，下面我把原理和最基础的轻按手势结合起来，用实际案例帮你弄明白它的原理。

轻按手势

在 **Gesture** 手势库接收到系统的手势后，它需要判断页面中有没有哪个视图需要响应该手势，如果没有任何组视图需要响应该手势，那么什么都不会发生。

因此，你需要先告诉 **Gesture** 手势库，你有个视图需要响应手势。这一步是通过 **GestureDetector** 组件来实现的，代码如下：

 复制代码

```
1 import { GestureDetector } from 'react-native-gesture-handler';
2
3 <GestureDetector gesture={singleTap}>
4   <View />
5 </GestureDetector>
```

在这段代码中，**GestureDetector** 组件的作用是将单击手势 **singleTap** 绑定到 **View** 视图上。

GestureDetector 组件是从 **react-native-gesture-handler** 库中导出的复合组件。

GestureDetector 复合组件并不会真正渲染到屏幕上，它的作用是将配置好的手势绑定宿主组件视图上，这个宿主组件是 **GestureDetector** 组件内部第一个会真正渲染到屏幕上的视图。

那示例中的 **singleTap** 手势是什么呢？创建 **singleTap** 手势的代码如下：

 复制代码

```
1 import { Gesture } from 'react-native-gesture-handler';
2
3 const singleTap = Gesture.Tap()
4   .onStart(() => console.log('开始触发轻按事件'))
```

如代码所示，**Gesture** 对象是 **react-native-gesture-handler** 手势库提供的手势对象，在 **Gesture** 对象下，有一个 **Tap** 方法，调用该方法就可以生成一个轻按手势 **singleTap**。

并且它还支持链式调用，你可以用 **.onStart** 把触发轻按回调函数绑定上去，并在该回调中打印相关日志。你对屏幕中 **View** 视图点击一次，就会触发 **onStart** 的回调函数，并打印“开始触发轻按事件”的日志。

但是，前面我们说过，**Gesture** 手势库是跑在 UI 线程中的，因此 `console.log` 实际上是在 **Reanimated** 创建的 **JavaScript** 虚拟机中执行的，日志是从 UI 线程中打印出来的。

虽然我们可以在 UI 线程中打印 `console` 日志，但是并不能在 UI 线程调用 `setState` 的渲染方法，如果调用就会报错。触发报错的示例代码如下：

 复制代码

```
1 const [logs, setLogs] = useState<string[]>([]);
2
3 const singleTap = Gesture.Tap()
4   .onStart(() => setLogs(logs.concat('开始触发轻按事件'))))
5
6 // 报错
7 Tried to synchronously call function {setLogs} from a different thread.
```

这段代码中，我们先定义了日志状态 `logs` 及其更新函数 `setLogs`，然后在 `onStart` 的回调函数中调用了 `setLogs` 更新日志状态。当你点击触发 `onStart` 的回调函数后，就会出现一个报错，报错的内容是你尝试在另一个线程，也就是 UI 线程，同步调用 `setLogs` 更新函数。

这是因为，渲染过程是跑在 **JavaScript** 线程中的，而 UI 线程的 **JavaScript** 虚拟机和 **JavaScript** 线程的 **JavaScript** 虚拟机的上下文是隔离的，UI 线程拿不到渲染相关的上下文。因此，如果你在 `onStart` 的回调函数中调用 `setState`，就会报错。

既然 `setState` 相关函数放到 UI 线程执行会报错，那我们就放回 **JavaScript** 线程执行呗？

确实可以。修改后的方案如下：

 复制代码

```
1 import {runOnJS} from 'react-native-reanimated';
2
3 const [logs, setLogs] = useState<string[]>([]);
4 const singleTap = Gesture.Tap()
5   .onStart(() => runOnJS(setLogs)(logs.concat('开始触发轻按事件'))))
```

在该方案中，`onStart` 的回调函数依旧是在 UI 线程执行的，但是从 **react-native-reanimated** 动画库中引入的 `runOnJS` 方法，可以把 `setLogs` 方法放回 **JavaScript** 线程执行。

具体方法是这样的：`runOnJS` 是一个 [柯里化](#) 的函数，它接收的第一个参数是要放回 JavaScript 线程调用的函数，这里是 `setLogs` 函数；它接收到的第二个入参是 `logs.concat('开始触发轻按事件')` 执行后的返回值，这个返回值是一个数组，把相关函数及其入参放回 JavaScript 线程后，接着就会在 JavaScript 线程执行调用。

因此，在学习 **Gesture** 手势库的时候，你需要注意区分 **UI 线程** 和 **JavaScript 线程**。Gesture 手势库和 **Reanimated** 动画库搭配使用时，Gesture 的手势回调函数是在 **Reanimated** 动画库创建的 UI 线程的 JavaScript 虚拟机中执行的。

但 UI 线程的 JavaScript 虚拟机和 JavaScript 线程的 JavaScript 虚拟机是两个不同的虚拟机。UI 线程的 JavaScript 虚拟机只有执行相关动画、手势回调函数的上下文，而 JavaScript 线程的 JavaScript 虚拟机拥有的是执行页面渲染的上下文，因此手势回调函数必须调用 `runOnJS` 回到 JavaScript 线程的 JavaScript 虚拟机中执行 `setState` 渲染页面。

我把使用 **Gesture** 手势库实现轻按手势的完整示例代码放在这里了，你可以仔细看下：

 复制代码

```
1 import React, {useState} from 'react';
2 import { Text, View, SafeAreaView } from 'react-native';
3 import {runOnJS} from 'react-native-reanimated';
4 import {Gesture, GestureDetector} from 'react-native-gesture-handler';
5
6 export default function App() {
7   const [logs, setLogs] = useState<string[]>([]);
8
9   const singleTap = Gesture.Tap()
10     // 渲染
11     .onStart(() => runOnJS(setLogs)(logs.concat('开始触发轻按事件')));
12   // 动画或日志
13   // .onStart(() => console.log('开始触发轻按事件'));
14
15   return (
16     <SafeAreaView>
17       <GestureDetector gesture={singleTap}>
18         <View style={{width: 100,height: 100,backgroundColor: 'red'}} />
19       </GestureDetector>
20       {logs.map((log, index) => (
21         <Text key={index}>{log}</Text>
22       ))}
23     </SafeAreaView>
24   );
25 }
```

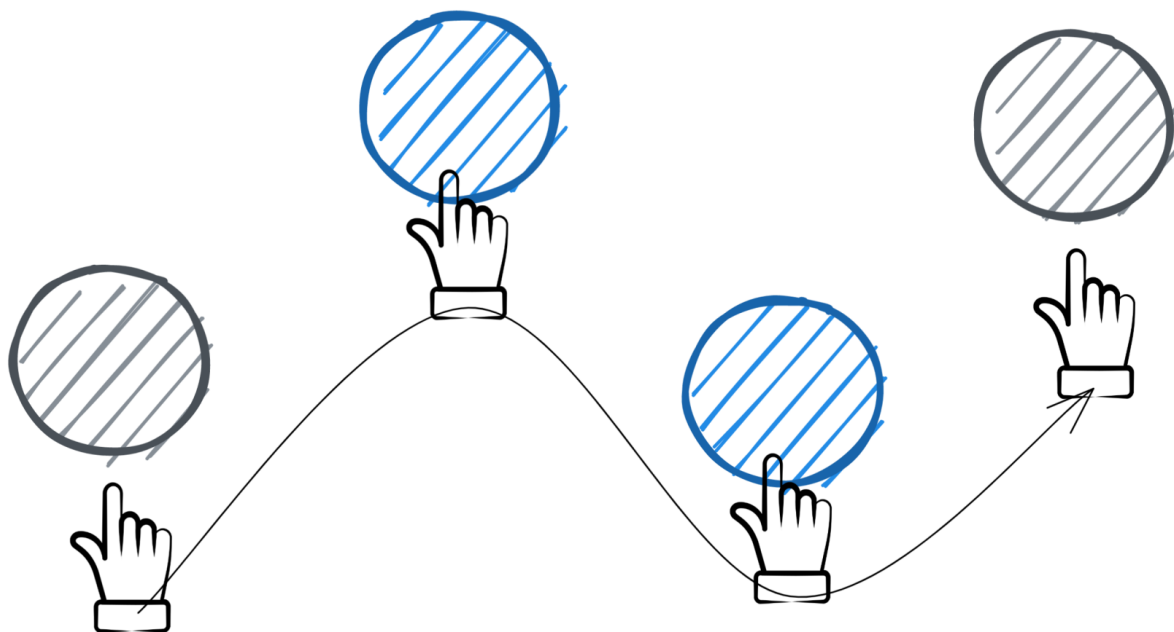
拖拽动效

刚才的轻按手势只是小试牛刀，其实这种基础的轻按手势，我们使用 **Pressable** 组件也可以实现。**Gesture** 手势库真正厉害的地方，在于能和 **Reanimated** 动画库配合着一起使用，它们二者一结合，就能玩出 **React Native** 老版本玩不出来的手势动画效果。

接着，我们再来看第二个案例，拖拽动效。通过拖拽案例，你可以学会如何将 **Gesture** 和 **Reanimated** 搭配在一起使用。

现在，你要实现的拖拽动效是这样的：你可以拖动屏幕上的一个圆形视图，让跟随你的手指一起移动。并且，在你触碰到该圆形视图时，圆形视图是蓝色的，当你手指离开该圆形视图时，圆形视图是灰色的。

拖拽动效的示意图如下：



图标来源于 www.flaticon.com

那我们应该如何实现这样的拖拽动效呢？

通过 **Gesture** 手势库进行人机交互一共分为 4 步，除了第 1 步只需 **Gesture** 手势库底层参与外，其他 3 步都涉及代码，那我就分 3 步来实现拖拽动效。

第一步是，将拖拽手势绑定到动画组件上，示例代码如下：

```

1 import Animated from 'react-native-reanimated';
2 import {Gesture, GestureDetector} from 'react-native-gesture-handler';
3
4 const dragGesture = Gesture.Pan()
5
6 <GestureDetector gesture={dragGesture}>
7   <Animated.View style={[{
8     width: 100,
9     height: 100,
10    borderRadius: 100,
11    }, animatedStyles]}/>
12 </GestureDetector>

```

因为涉及动画，所以我们需要使用动画视图，也就是 **Animated.View** 视图。该 **View** 视图的样式属性，包括一个固定的圆形样式和一个动画样式 **animatedStyles**。

其中，**animatedStyles** 的具体实现代码如下：

```

1 const isPressed = useSharedValue(false);
2 const offset = useSharedValue({ x: 0, y: 0 });
3
4 const animatedStyles = useAnimatedStyle(() => {
5   return {
6     transform: [
7       { translateX: offset.value.x },
8       { translateY: offset.value.y },
9     ],
10    backgroundColor: isPressed.value ? 'blue' : '#ccc',
11  };
12 });

```

这段代码中主要用到了两个共享值：一个是 **isPressed**，它用来判断圆形视图是否被点击了；另一个是 **offset**，用来确定圆形视图的 **x/y** 坐标。默认情况下，圆形视图未被点击，则 **isPressed** 为 **false**，且在 **x=0 y=0** 的坐标点上。

而具体的动画样式 **animatedStyles** 是使用 **isPressed** 和 **offset** 衍生得到的，默认情况下它的 **translateX=0 translateY=0**，且 **backgroundColor** 为 **#ccc** 灰色。

到这里，我们就完成了第一步，也就是将拖拽手势绑定到动画组件上。

接下来的第二步是，实现拖拽手势 `dragGesture` 和其手势回调。

前面我和你介绍了轻按手势 `Gesture.Tap` 和相关的 `onStart` 回调。现在，我要用到的是拖拽手势 `Gesture.Pan` 及其相关的三个回调 `onBegin`、`onChange` 和 `onFinalize`。

你可能会问：轻按手势是 `Gesture.Tap`，拖拽手势 `Gesture.Pan`，但 `onStart`、`onBegin`、`onChange` 和 `onFinalize` 这 4 个手势回调之间有什么关系啊？为什么实现拖拽动效需要的是 `onBegin`、`onChange` 和 `onFinalize` 这三个手势回调呢？除了这 4 个手势回调，还有什么其他手势回调吗？

这些问题很有价值，轻按手势和拖拽手势虽然有所不同，但是对应的手势回调大体上都是相同的，因此我用拖拽手势回调进行讲解，其中细节差异你可以查一下 [🔗 相关文档](#)。

拖拽手势相关的回调一共 10 个，示例代码如下：

 复制代码

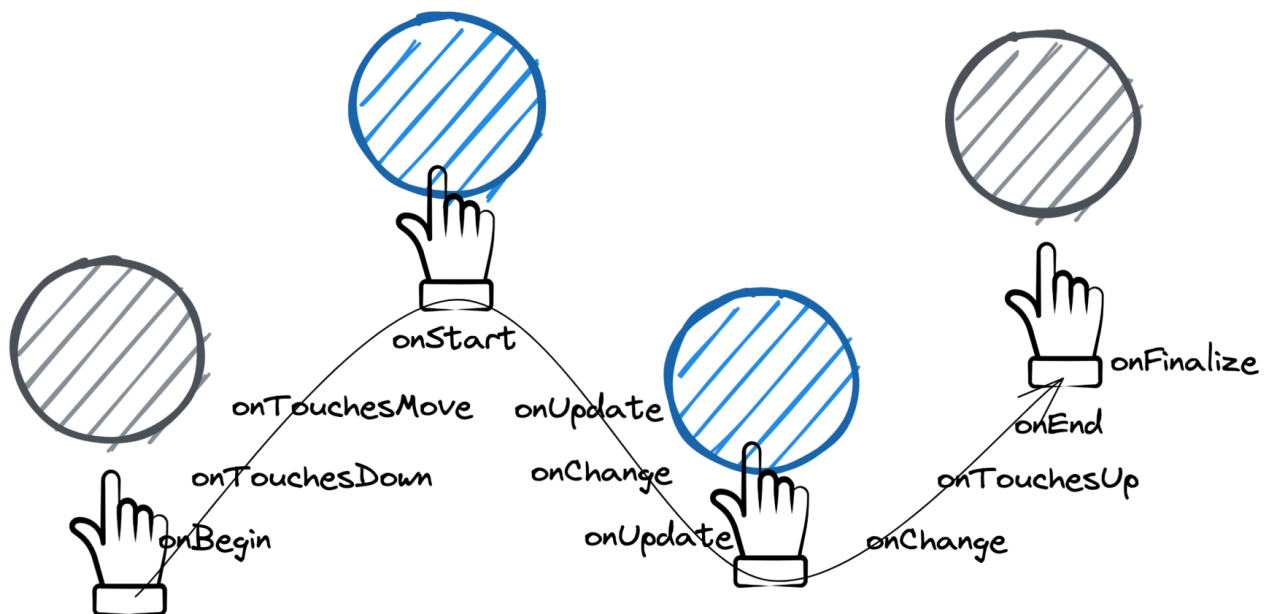
```
1 const dragGesture = Gesture.Pan()  
2   .onBegin(() => console.log('onBegin'))  
3   .onTouchesDown(() => console.log('onTouchesDown'))  
4   .onTouchesMove(() => console.log('onTouchesMove'))  
5   .onStart(() => console.log('onStart'))  
6   .onUpdate(() => console.log('onUpdate'))  
7   .onChange(() => console.log('onChange'))  
8   .onTouchesUp(() => console.log('onTouchesUp'))  
9   .onEnd(() => console.log('onEnd'))  
10  .onTouchesCancelled(() => console.log('onTouchesCancelled'))  
11  .onFinalize(() => console.log('onFinalize'))
```

这些手势回调是按顺序从上到下依次触发的：

- `onBegin`：开始识别到手势，但此时拖拽并未发生。也就是说，这时你的手指是触碰到 `View` 视图，不过手指并未移动；
- `onTouchesDown`：手指按下触摸到视图时会触发。你可以理解为在手指触摸到视图时，先触发了 `onBegin`，紧接着就触发了 `onTouchesDown`；
- `onTouchesMove`：手指移动后会触发；

- **onStart**: 当手指移动距离超过 `Float.MIN_VALUE` 的阈值时，也就是精度为 0.000000 的距离时，就会触发该回调，此时**拖拽事件正式触发**；
- **onUpdate**: 在手指移动的过程中 x/y 坐标系等参数会更新，参数更新后 **onUpdate** 回调就会触发；
- **onChange**: 在手指移动的过程中 x/y 坐标系等参数会更新，参数更新后 **onChange** 回调会紧接着 **onUpdate** 触发。**onChange** 和 **onUpdate** 的区别是，**onChange** 的参数是以上一次回调的参数作为基准进行更新的，而 **onUpdate** 是以手势触发 **onStart** 时的参数为基准进行更新的；
- **onTouchesUp**: 当手指离开屏幕时，触发 **onTouchesUp** 回调；
- **onEnd**: 当手指离开屏幕时，会先触发 **onTouchesUp** 回调，然后紧接着触发 **onEnd** 回调。需要注意的是，**onEnd** 回调是和 **onStart** 配套出现的，如果没有触发 **onStart** 回调，那也不会触发 **onEnd** 回调；
- **onTouchesCancelled**: 一般是在系统弹窗中断手势的情况下触发，较为少见；
- **onFinalize**: 只要手势结束，最终都会触发 **onFinalize** 回调。

这 10 个手势回调，可能稍微有点难记，我建议你自己亲自动手试试，我也给你画了一张示意图，帮你加深理解：



图标来源于 www.flaticon.com

回到我们要实现的拖拽动效，它的一个功能是手指触碰到视图，视图变蓝，手指离开视图，视图变灰，另一个功能是手指移动时视图跟随着手指一起移动。

手指触碰和离开视图时是不会触发 `onStart` 和 `onEnd` 回调的，只会触发 `onBegin` 和 `onFinalize` 回调，因此我选择了 `onBegin` 和 `onFinalize` 来处理手指触碰和手指离开这两个事件。

我们再来看视图跟随手指移动的功能。`onUpdate` 返回的是以 `onStart` 触发位置为基准的参数，而 `onChange` 返回的参数不仅包括 `onUpdate` 参数，还多了以上一次 `onChange` 触发位置为基准的偏移量 `changeX` 和 `changeY`，我需要根据上一次手势的偏移量 `changeX` 和 `changeY` 来计算视图新的偏移量，因此我选择使用 `onChange` 而不是 `onUpdate`。

实现拖拽动效的最后一步是，更新动画的共享值。

因为，`Animated.View` 视图的衍生样式值已经设置好了，只要动画的共享值一更新，衍生样式值就会更新，`Animated.View` 就会跟着手指动起来。

更新动画共享值的代码如下：

 复制代码

```
1  const dragGesture = Gesture.Pan()
2    .onBegin(() => {
3      isPressed.value = true;
4    })
5    .onChange((e) => {
6      offset.value = {
7        x: e.changeX + offset.value.x,
8        y: e.changeY + offset.value.y,
9      };
10   })
11   .onFinalize(() => {
12     isPressed.value = false;
13   });
```

这段代码比较简单，在 `onBegin` 回调中将 `isPressed.value` 标记为 `true`，在 `onFinalize` 回调中将 `isPressed.value` 标记为 `false`，这就实现了手指触碰视图置蓝，离开视图置灰的效果。

然后在 `onChange` 回调中，获取上一次 `Animated.View` 视图的偏移位置 `offset.value.x` 和 `offset.value.y`，并将其分别和相对上一次位置的偏移量 `e.changeX` 和 `e.changeY` 进行相加，就能计算出当前 `Animated.View` 的偏移位置 `offset.value`。当 `offset.value` 共享值更新时，

Animated.View 的衍生样式值 `translateX` 和 `translateY` 也会随之更新，该视图在屏幕上的位置也会发生相应地移动。

到这里，我们就把拖拽动效实现了。这里我们再整合下实现拖拽动效的主要代码，你可以仔细看看以加深印象：

📄 复制代码

```
1 function Ball() {
2   const isPressed = useSharedValue(false);
3   const offset = useSharedValue({ x: 0, y: 0 });
4
5   const animatedStyles = useAnimatedStyle(() => {
6     return {
7       transform: [
8         { translateX: offset.value.x },
9         { translateY: offset.value.y },
10      ],
11      backgroundColor: isPressed.value ? 'blue' : '#ccc',
12    };
13  });
14
15  const dragGesture = Gesture.Pan()
16    .onBegin(() => {
17      isPressed.value = true;
18    })
19    .onChange((e) => {
20      offset.value = {
21        x: e.changeX + offset.value.x,
22        y: e.changeY + offset.value.y,
23      };
24    })
25    .onFinalize(() => {
26      isPressed.value = false;
27    });
28
29  return (
30    <GestureDetector gesture={dragGesture}>
31      <Animated.View style={[styles.ball, animatedStyles]} />
32    </GestureDetector>
33  );
34 }
35
36 export default function Example() {
37   return (
38     <View style={styles.container}>
39       <Ball />
40     </View>
41   );
42 }
```

总结

今天这一讲，我们将 **Gesture** 手势库和 **Reanimated** 动画库搭配一起使用，实现一个最基础的拖拽动效。

实现一个拖拽动效主要分为三步：

- 第一步，创建 **Gesture.Pan** 手势并将拖拽手势绑定到动画组件上；
- 第二步，在 **Gesture.Pan** 拖拽手势的 10 个手势回调中，选择 **onBegin** 和 **onFinalize** 手势回调响应拖拽开始和拖拽完成，选择 **onChange** 响应拖拽移动；
- 第三步，在相应的拖拽回调中同步更新动画组件的共享值，也就是 **x、y** 轴坐标，实现基础的拖拽动效。

手势相关的问题是我们很容易遇到的，而且学习曲线很陡峭，所以我把手势分为了上中下三篇。今天的内容还是相对基础的，在了解了手势基础的原理和使用后，下一讲我们将在此基础上，继续讲解手势进阶的内容，特别是如何解决手势冲突的问题。


作业

1. 请你实现一个类似微信消息的左滑删除的功能；
2. 请你聊一下，你遇到过哪些需要手势和动画配合才能实现的功能，你是原来都是怎么实现的？

欢迎在评论区和我探讨。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

上一篇 14 | Reanimated: 如何让动画变得更流畅?

下一篇 16 | Gesture (中): 如何解决单视图多手势的冲突问题?

精选留言 (3)

写留言



Hadwz

2022-05-06

老师，用Gesture 手势库和 Reanimated，实现在ScrollView里上下拖拽一个元素，并在元素到达列表的顶部/底部的时候，滚动ScrollView，让列表和元素同时移动，有什么实现思路吗？

作者回复: 你遇到的问题，大家也经常遇到，就是手势冲突的问题，在 Gesture 中下两讲中有回答。



风之化身

2022-05-05

讲的蛮好，建议录个小视频看下最终效果会更直观点～



ZouLe

2022-05-02

上 Reanimated v2 和 Gesture v2 就不能用原来的浏览器debug模式开发了，这个转换跨度还是蛮大的 😊

作者回复: 是的。主要还是因为 JSI 和 Hermes 的原因。其实 Flipper 功能更强大。

