

## 16 | Gesture（中）：如何解决单视图多手势的冲突问题？

2022-05-04 蒋宏伟

《React Native 新架构实战课》

[课程介绍 >](#)



讲述：蒋宏伟

时长 21:57 大小 20.10M



你好，我是蒋宏伟。

通过上节课对轻按手势和拖拽动效这两个基础手势案例的学习，相信你现在已经能够完成一些基本手势需求的开发了。今天这节课，我们就再进一步，聊下怎么解决更进阶的手势问题。

在手势基础的学习中，我们给到的手势案例都是围绕着一个视图、一个手势展开的，处理起来很简单。但在真实的工作中，情况会更加复杂。比如说，我们会有稍微难一点的情况，也就是一个视图同时存在多个手势。还有更复杂的，就是同时有多个视图、多个手势，并且这些视图和手势环环相扣。

当然你也不用担心，这两种复杂的情况，我们的 **Gesture** 手势库都提供了相关的解决方案。不过俗语也有说，“一口吃不成大胖子。”所以今天这一讲，我们先来聊聊一个视图多个手势如何处理，下一讲再聊聊多个视图、多个手势如何处理。

但在展开讲解手势冲突问题之前，我需要带你补全 **Gesture** 手势的一些进阶知识。

## 手势进阶

我们要研究多个手势冲突的问题，大体上得遵循这样的流程：

- 首先你得知道 **Gesture** 手势库都能识别哪些手势；
- 然后手势实际是一系列连续的动作，而这一系列动作大致可以分为几个阶段，比如开始、进行中、完成和中途取消，**Gesture** 手势库又提供了哪些手势回调来识别手势的不同阶段；
- 最后，不同阶段的回调又都能提供什么参数，能让开发者来使用。

我们来看第一个问题，**Gesture** 手势库都能识别哪些手势呢？

**Gesture** 手势库一共支持“1 + 8”个手势，我画了一张示意图，你可以先看一下：



图标来源于 [www.flaticon.com](http://www.flaticon.com)

“1 + 8”中的“1”指的是 1 种原始手势，“8”指的是 8 种封装手势，它们都是：

轻按手势 Tap	默认耗时 500ms 以下的普通点击会触发该手势
长按手势 LongPress	默认耗时 500ms 以上的长按点击会触发该手势
拖拽手势 Pan	当手指在屏幕上移动时，会触发该手势
旋转手势 Rotation	当用户在屏幕上的两个手指的角度发生变化时，会触发该手势。例如两指旋转图片
缩放手势 Pinch	当用户在屏幕上的两个手指之间的距离发生变化时，会触发该手势。例如，两指之间的距离变小即缩小图片，两指之间距离变大即放大图片
快滑手势 Fling	当用户在屏幕中快速滑动（默认800ms 内滑动超过 160 像素）时会触发该手势。例如，轮播图既可以通过慢速滑动触发超过 1/2 页面宽度或高度的阈值进行翻页，也可以动过 Fling 快滑进行翻页
重按手势 ForceTouch	也叫做 3D Touch，是 iOS 独有的事件
原生手势 Native	虽然名字叫做 Native 手势，但我发现只在 ScrollView 滚动组件中用到过该手势。取名原生手势，是因为 ScrollView 的滚动偏移量并不是由 Gesture 手势系统控制的，而是由原生平台控制的
原始手势 Manual	在原始手势中，它会把所有的手指触控的点位和手势的内部状态都暴露给你，由你自己根据情况进行封装。当你遇到以上 8 种封装手势不能处理的场景时，就需要用到原始手势了

接着你需要知道的是，**Gesture 手势库的各个手势都支持哪些手势回调？**

刚才我和你介绍的 7 种新手势，虽然和之前学的 Tap、Pan 手势的触发条件不同，但整体上它们的手势回调还是 Pan 手势的那 10 种。这 10 种手势回调，也可以进一步分为三类。

**第一类是通用回调，包括：**

- onBegin;
- onTouchesDown;
- onTouchesMove;
- onTouchesUp;
- onFinalize。

无论哪种手势都会有以上回调，只要用户和相关视图发生了交互行为，即便该手势并未真正触发，但也会触发相关的通用回调。这是什么意思呢？

举个例子，比如拖拽手势 **Pan**，你点击相关视图的交互，只是手指按下、手指抬起，中间过程中手指不移动，这时拖拽手势不算触发吧？你得真正移动了手指拖拽手势才算触发，对不对？

这时拖拽手势并未真正触发，但却会触发 **onBegin**、**onTouchesDown**、**onTouchesUp**、**onFinalize** 这些通用回调事件。

又比如，用户两个手指触碰到了屏幕，但此时用户手指并未旋转，而且接着又离开了屏幕，因此不会触发旋转手势。但会依次触发 **onBegin**、**onTouchesDown**、**onTouchesUp** 回调和 **onFinalize** 回调。这就是“手势并未真正触发，但也会触发通用回调”的意思。

第二类是激活（**ACTIVE**）回调，包括：

- **onStart**;
- **onUpdate**;
- **onChange**;
- **onEnd**。

激活（**ACTIVE**）是手势内部的一种状态，它代表某个手势真正被触发了。

我们还是以 **Rotation** 手势为例分析一下。只有当用户两个手指触碰到屏幕且发生了旋转时，**Rotation** 手势内部状态才会变为 **ACTIVE**，此时才会触发 **onStart**、**onUpdate**、**onChange**。当 **ACTIVE** 被触发后，用户手指离开时，**Rotation** 手势的内部状态会由 **ACTIVE** 变为 **END**，此时才会触发 **onEnd** 回调。

注意，只有先变为 **ACTIVE** 状态，再由 **ACTIVE** 状态变为 **END** 状态的这一种情况，会触发 **onEnd** 回调。

第三类是系统取消回调，它是一个特例，只有一种，就是 **onTouchesCancelled**。

当触发 **onTouchesCancelled** 回调时，通常是操作系统把手势打断了。比如，你在旋转图片时，突然来了个电话，此时 **Rotation** 手势被打断，内部状态会从 **ACTIVE** 变为

CANCELLED，这时就不会触发 onEnd 回调了，而是触发 onTouchesCancelled 回调。

因此，即便手势触发成功，但当手势结束时不一定会调用 onEnd 回调，因为还有 onTouchesCancelled 回调这种情况，所以如果你想保证无论发生什么情况都有结束回调，你应该使用 onFinalize 回调代替 onEnd 回调。

关于手势进阶，你需要知道的第三个知识点是，**Gesture 手势库的手势回调都返回哪些参数？**

在学习 Pan 手势时，通过 onChange 回调返回的 changeX/changeY 只是手势回调返回参数的一种。实际上，不同的手势、不同的回调返回的值都有所区别，这类知识点非常零散。为了让你更好地记忆，我把它分为两类，分别是常用类回调参数和场景类回调参数。

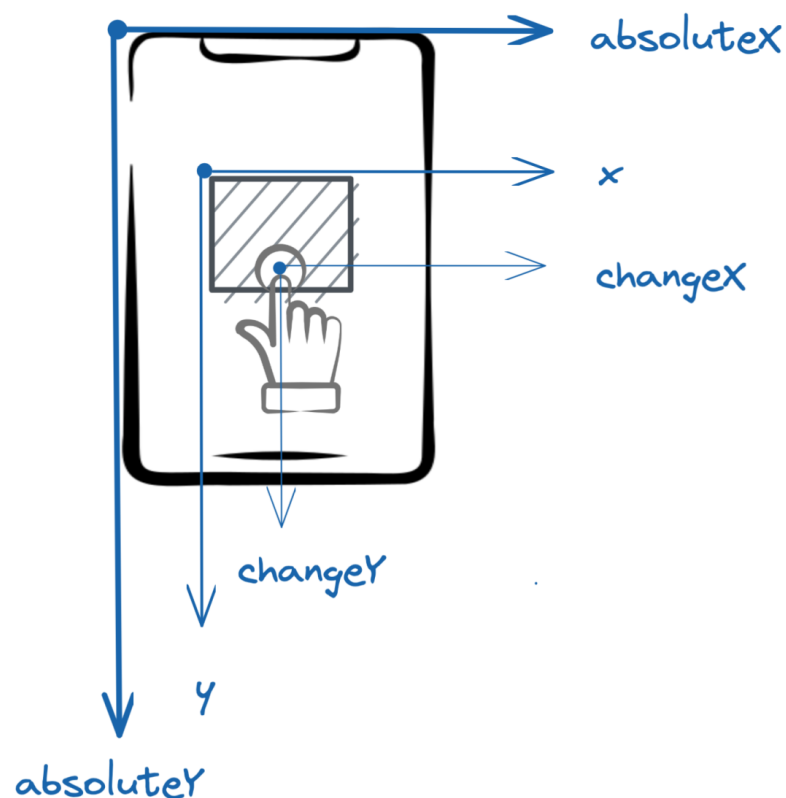
常用类回调参数：

state	手势的内部状态，它是一个枚举类型（在 TypeScript 中是枚举类型，在 Javascript 中是 number 类型）。手势正常的触发流程其内部状态的变化依次是， UNDETERMINED -> BEGAN ----> ACTIVE ----> END -> UNDETERMINED，其他失败流程会提前触发 FAILED 状态，被系统打断会触发 CANCELLED 状态
numberOfPointers	触碰到屏幕的手指数量，它是 number 类型
x/y	是以触发手势的视图的左上角为基点建立的坐标系的横轴和纵轴的坐标值，它是 number 类型
absoluteX/Y	是以 React Native 应用的根视图的左上角为基点，通常就是手机屏幕的左上角，建立的坐标系的横轴和纵轴的坐标值，它是 number 类型
changedX/Y	是以上一次 onChange 事件触发点为基点建立的坐标系的横轴和纵轴的坐标值，它是 number 类型



x/y、absoluteX/Y 和 changedX/Y 这三个坐标位置比较容易搞混，我给你画了一张示意图，你看一眼就会明白：





图标来源于 [www.flaticon.com](http://www.flaticon.com)

## 场景类回调参数：

<b>duration</b>	是使用 LongPress 长按手势场景下的回调参数。duration 代表的是 LongPress 长按手势的持续时间，单位是 ms
<b>rotation</b>	是使用 Rotation 旋转手势场景下的回调参数。rotation 代表的是以两个手指的中心点为基准点，进行旋转的弧度
<b>scale</b>	使用 Pinch 缩放手势场景下的回调参数。scale 代表的是相对于两个手指放大或缩小的倍数，初始值是 1
<b>allTouches</b>	使用 Manual 原始手势或 onBegin、onTouchesDown 等通用回调返回的参数。Manual 原始手势和通用回调都意味着，你可能有自定义手势需求，这时有多少个手指触碰到屏幕就会返回几个 touche，每个 touche 中又会包括 x/y 和 absoluteX/Y 坐标位置

这些场景类的回调参数，可能归纳得不全，但经常用的，我都帮你归类出来了。这 9 种手势事件、10 种回调函数、两类回调参数，都是基于 Gesture 手势 v2 版本来讲的，v1 版本太旧

了，我这里就不展开了。学习到这里，你应该对手势库有了一个大概的了解。

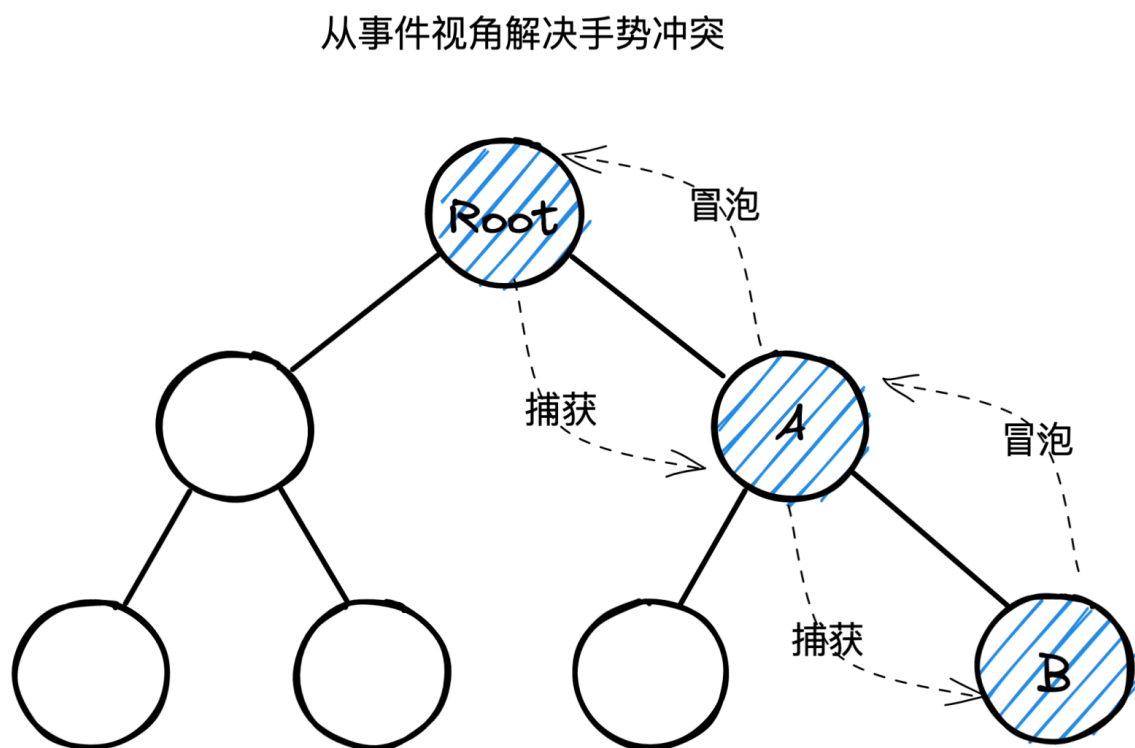
有了这些基础，你再来理解我们接下来要解决的手势冲突的问题，就会变得更简单了。

## 常规手势冲突解决方案：捕获冒泡机制

上一讲的开头，我和你列举了三个案例：Android 下拉刷新、类抖音的评论区拖拽效果、类淘宝首页的带头部的多 Tab 长列表效果。其实这些案例要解决本质问题都是**手势冲突**。要实现这些效果，我们要解决的是如何在多个视图之间处理多个手势，特别是 `ScrollView` 的滚动手势。

冒泡机制是一种常规的手势识别和分配机制，我们先来看看它是怎么处理手势冲突问题的。无论是在 Android、iOS 还是 Web 中，都有事件冒泡机制，事件冒泡机制是站在“事件视角”给不同视图分配不同的手势事件。

我画了一张**事件冒泡机制解决手势冲突**示意图，你可以看一下：



在“事件视角”的图中，如果用户点了屏幕上的某个点，站在底层框架视角看，用户屏幕中的视图，实际是由一棵视图树组成的，用户想点的既有可能是叶子视图 **B**，也有可能是叶子视图的父视图 **A**，还可以一层层继续往上找的父视图，直到找到根视图 **Root**。

那用户点的这一下，究竟想点的是哪个视图呢？底层框架本身并不知道。

底层框架不知道，它就需要一个个地问，“我这个框架收到了用户的一个点击手势，你这个视图是否需要处理它？”

那么，框架提问的顺序又是怎样的呢？

框架它会先根视图 **Root** 往下一直到叶子视图 **B**，然后再从叶子视图 **B** 往上一直到根视图 **Root**。从上往下问叫做捕获，从下往上再问一次叫做冒泡。

但为什么要先从上往下问一次，又从下往上再问一次呢？

根本原因是，框架不知道哪个视图处理事件优先级更高。比如框架它先问了叶子视图 **B**，如果这时它就直接把事件分发给叶子视图 **B** 处理了，等它问到根视图 **Root** 时，结果 **Root** 根视图告诉框架，这个事件应该由它 **Root** 根视图处理。

但这时候，手势事件已经被叶子视图 **B** 处理完了，**Root** 根视图就处理不了了，这就出现了优先级问题。

因此，框架和视图之间就约定，正常情况下，所有的视图都是在冒泡的流程中确定是否响应事件，如果有哪个视图想要拦截其他视图的事件，就可以在捕获流程中提前拦截。

**React Native** 框架自带的手势事件用的就是捕获冒泡机制。但是这套机制有两个弊端，一是理解起来费劲，需要开发者理解视图树和捕获冒泡的传导机制；二是它没办法处理一个事件要在两个组件上同时响应的情况。

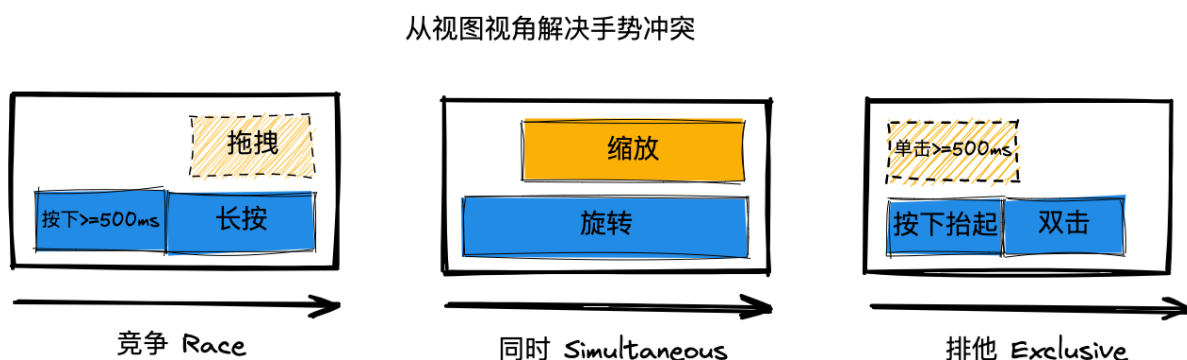
所以，**Gesture** 手势库提供了另外两种解决手势冲突的视角：一个是从单个视图的视角出发，来解决单视图、多手势之间的冲突问题，另一个是从单个手势的视角来解决多视图、多手势之间的冲突问题。这一讲中，我们从单个视图视角出发，看看如何解决单视图、多个手势之间的冲突。

## 单视图多手势冲突问题

首先，我们来分析下如何从单个视图视角出发解决多个手势之间的冲突。



Gesture 手势库，提供了 3 种解决单视图多手势冲突问题的 API，你先看下这张示意图：



示意图中的第一个例子是，通过 **Gesture.Race** 函数让同一个组件中的多个手势之间进行竞争。谁先触发就响应谁，通过竞争的方式解决了手势冲突的问题，示例代码如下：

复制代码

```
1 function RaceDemo() {
2   const pan = Gesture.Pan()
3   const longPress = Gesture.LongPress()
4
5   return (
6     <GestureDetector gesture={Gesture.Race(pan, longPress)}>
7       <View/>
8     </GestureDetector>
9   );
10 }
```

在上述代码中，我使用了手势竞争函数 **Gesture.Race**，**Gesture.Race** 函数接收了两个手势：**Pan** 拖拽手势和 **LongPress** 长按手势。

此时，如果你手指按下的时间超过 **500ms**，就会触发长按事件。一旦触发了长按事件，即便你再移动手指也不会再触发拖拽事件了。反之，如果你按下的时间小于 **500ms**，这时你移动了手指，就会触发拖拽事件，即便后续你按下的时间超过了 **500ms**，也不会再触发长按事件了。

其中，起关键作用的就是 **Gesture.Race** 函数。**Gesture.Race** 函数可以接收若干个手势事件，这些手势事件只要触发了一个，其他的手势事件都不会再触发。

示意图中的第二个例子是，通过 **Gesture.Simultaneous** 函数让同一个组件中的多个手势同时响应。多个手势可以同时响应，就没有手势冲突的问题了，示例代码如下：

 复制代码

```
1 function SimultaneousDemo() {
2   const pinch = Gesture.Pinch()
3   const rotation = Gesture.Rotation()
4
5   return (
6     <GestureDetector gesture={Gesture.Simultaneous(pinch, rotation)}>
7       <View/>
8     </GestureDetector>
9   );
10 }
```

在上述代码中，我使用了手势同时响应函数 **Gesture.Simultaneous**。**Gesture.Simultaneous** 函数接收了两个手势：**Pinch** 缩放手势和 **Rotation** 旋转手势。

此时，你既可以使用两个手指旋转视图，也可以使用两个手指对视图进行缩放。其中，起作用的就是 **Gesture.Simultaneous** 函数。**Gesture.Simultaneous** 函数可以接收若干个手势事件，并且这些手势事件会同时触发。

示意图中的第三个例子是，通过 **Gesture.Exclusive** 函数让组件决定多个手势的响应优先级。它解决的是响应优先级的问题，示例代码如下：

 复制代码

```
1 function ExclusiveDemo() {
2   const singleTap = Gesture.Tap()
3   const doubleTap = Gesture.Tap().numberOfTaps(2)
4
5   return (
6     <GestureDetector gesture={Gesture.Exclusive(doubleTap, singleTap)}>
7       <View/>
8     </GestureDetector>
9   );
10 }
```

在上述代码中，我使用了 **Gesture.Tap()** 创建了单击手势，使用 **Gesture.Tap().numberOfTaps(2)** 创建了双击手势，并且使用了 **Gesture.Exclusive(doubleTap, singleTap)** 把双击手势的优先级设置在了单击手势之前。

这样做的原因是：如果这两个手势使用的是 **Race** 竞争机制，那么单击手势永远会先响应，而双击手势永远不会响应；如果它们使用的是 **Simultaneous** 共存机制，那么短时间内有第二次点击，会同时触发单击手势和双击手势，不符合预期。

这里的解决方案就是，我们可以使用 **Gesture.Exclusive** 设置优先级。如果 500ms 内只有一次点击，那么会在 501ms 触发单击事件；如果 500ms 内有两次点击，那么会在第二次点击完成时触发双击事件。

其中，**Gesture.Exclusive** 函数的作用就是给它接收到的若干个手势事件排个优先级：第一个参数的手势事件大于第二个手势事件，第二个参数的手势事件大于第三个手势事件，以此类推。

还记得我在 **Pressable** 一讲中给你留了个作业吗？那个作业问的也是单击手势和双击手势之间的优先级的问题，那一讲的作业是：

在较老版本的手机浏览器中，点击事件存在 350ms 延迟；在微信聊天框中，点击对方的微信头像比点击右上角三个点的更多按钮，打开页面的速度慢一些；双击事件是常见的点按事件之一，**Pressable** 组件却没有提供；这三个现象涉及 **Web**、**Android**、**iOS** 和 **React Native** 这四个技术领域，但这三个现象其实都指向同一个答案。

这个答案就是，在较老版本的手机浏览器中，浏览器本身提供了双击放大页面的手势，而点击页面按钮是单击手势，因此页面按钮的单击需要等 350ms 才能响应；微信头像既有单击手势，也有双击手势，因此单击微信头像跳转页面的速度会变慢。

而 **Pressable** 组件不提供双击手势的原因，是因为当某个组件要同时处理双击手势和单击手势时，组件要优先响应双击事件再延长响应单击事件，这会导致单击手势的响应会变慢。既然会导致变慢，那么 **Pressable** 组件为了保证单击事件的优先响应，干脆就不提供了双击事件了。

概括而言，在单个视图中响应不同手势时会有冲突，**Gesture** 组件库提供了 3 种处理冲突的方式，分别是竞争 **Gesture.Race**、同时 **Gesture.Simultaneous**、排他 **Gesture.Exclusive**。在遇到单视图、多手势冲突问题时，你需要根据不同情况选择不同的处理方案。

## 总结

这一讲，我们介绍了如何解决单视图多手势的冲突问题，为此我们介绍了 **Gesture** 手势库的 9 个手势、3 类回调事件和 3 种手势冲突的解决方案。

9 个手势分别是：1 个最底层原始手势 **Manual**，以及 8 个封装好的上层手势。这 8 个手势包括轻按手势 **Tap**、长按手势 **LongPress**、拖拽手势 **Pan**、旋转手势 **Rotation**、缩放手势 **Pinch**、快滑手势 **Fling**、重按手势 **ForceTouch**、原生手势 **Native**。

3 类回调分别是：第一类上述 9 类手势都有的通用回调，包括 **onBegin**、**onTouchesDown**、**onTouchesMove**、**onTouchesUp** 和 **onFinalize**；第二类是满足各自触发条件时会触发的激活回调，包括 **onStart**、**onUpdate**、**onChange** 和 **onEnd**；第三类是系统取消回调 **onTouchesCancelled**。

3 种单视图多手势冲突的解决方案是：竞争 **Gesture.Race**、同时 **Gesture.Simultaneous**、排他 **Gesture.Exclusive**。

这一讲的知识点比较多，但它们都是处理复杂手势的基础，下一讲我会给你介绍更高阶的多视图多手势冲突问题的解决方案。

## 作业


1、请你实现一个同时支持单击、长按、拖拽、缩放和旋转的图片组件。

有啥问题欢迎在评论区留言。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 2  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。

## 精选留言 (1)

写留言



霍霍

2022-05-15

老师讲的很好，要是能有一些demo代码实例，让我们身临其境学习就更好了

作者回复: GitHub 上有代码，可以看看哦

