

18 | Navigation: 页面之间怎么跳转?

2022-05-09 蒋宏伟

《React Native 新架构实战课》

课程介绍 >



讲述: 蒋宏伟

时长 28:55 大小 26.48M

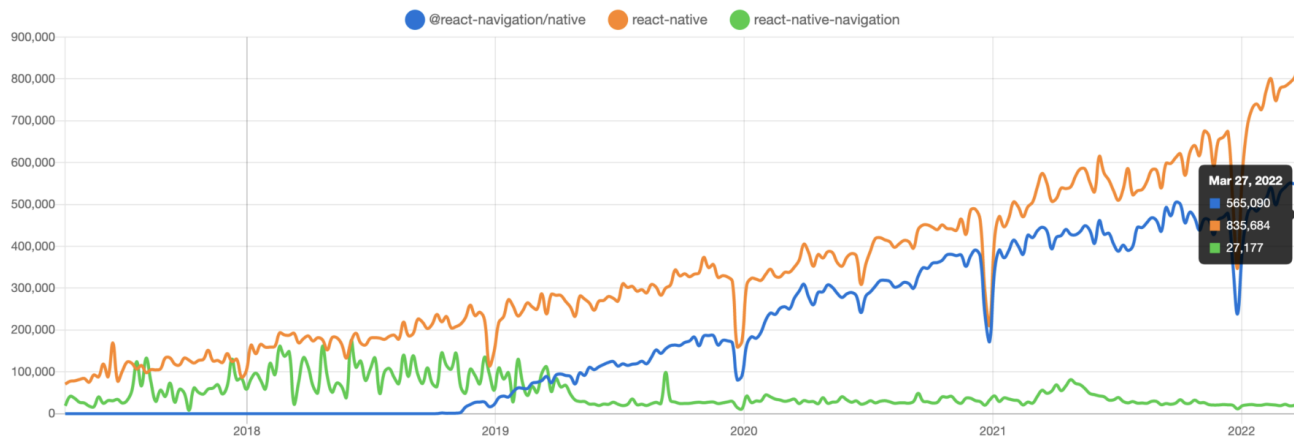


你好，我是蒋宏伟。今天我要给你介绍的是导航。

导航是用来管理页面之间的链接的。你平时用的 App，比如微信、抖音、京东，都有很多个页面，这些页面之间会有跳转、返回、切换等链接操作，这些链接操作就是导航。我们开发 React Native App 也一样，需要使用导航来链接各个页面。

尽管导航是开发 React Native App 必不可少的工具之一，但 React Native 框架并未将其内置，需要开发者自己进行集成。在 2018 年之前，业内用得比较多的导航是 React Native Navigation，在 2018 年之后大家用得更多的是 React Navigation。它们的名字很相似，不过你可千万不要搞混了，目前官方推荐的、主流的导航是 **React Navigation**，而不是 **React Native Navigation**。

你可以看一下，[🔗 React Navigation](#)、[🔗 React Native Navigation](#) 和 React Native 三个库的 npm 下载量：



这张图中，蓝色线条代表的是 React Navigation，绿色线条代表的是 React Native Navigation，橙色线条代表的是 React Native。从三个库的下载量中你可以看出，目前 React Navigation 导航已经成为主流，把 React Native Navigation 导航远远地甩在了后面，并且每十次 React Native 框架的下载，有七到八次都会下载 React Navigation 导航，由此可见，React Navigation 确实是非常受欢迎的。

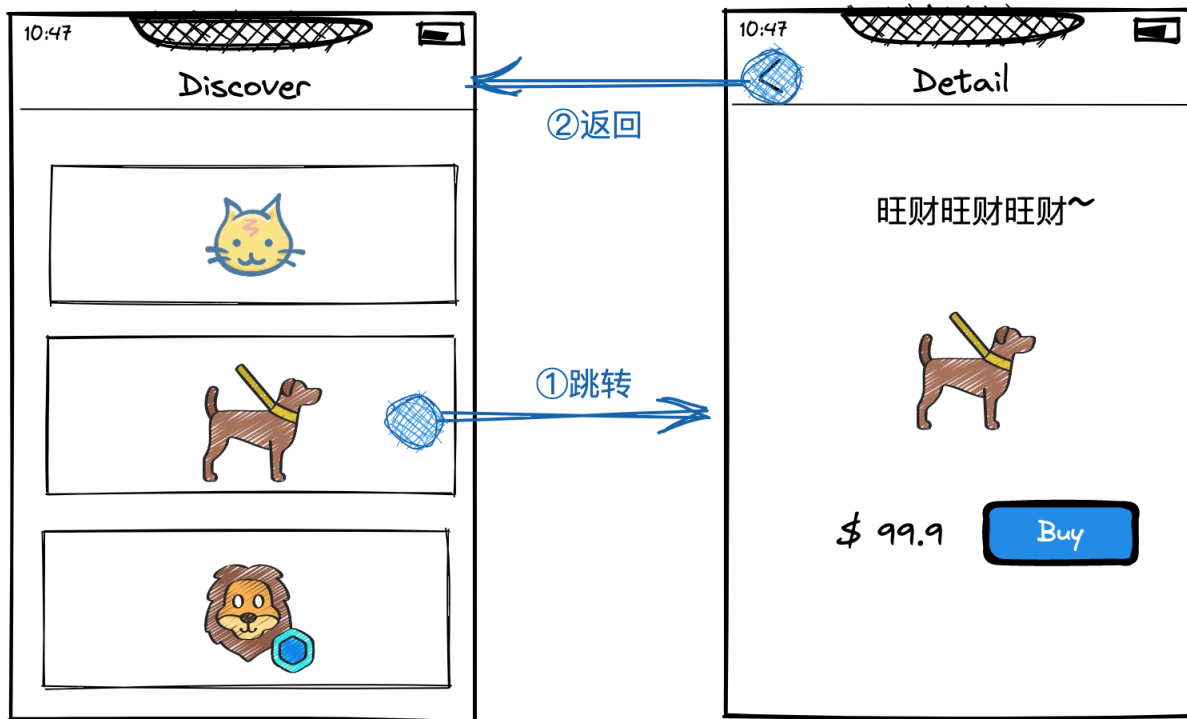
所以今天，我们就来看看怎么用 React Navigation 导航把各个页面链接起来。

导航基础

我们先从最常见的导航形式跳转开始。要实现一个基础的跳转导航，一共需要三步：

1. 创建“导航地图”；
2. 携带参数跳转页面；
3. 页面接收和解析参数。

接下来，我们一起实现一个最基础的导航功能，包括导航跳转和导航返回，示意图如下：



图中的第一个页面是 **Discover** 页面，第二个页面是 **Detail** 页面。**Discover** 页面包括 3 个列表项目：小猫、小狗和狮子。点击小狗列表项时，就会跳转到 **Detail** 页面，**Detail** 页面显示了小狗的详细介绍，这只小狗叫做“旺财”。点击 **Detail** 页面左上角的返回键时，就会返回到 **Discover** 页面。

假设现在由你来创建发现页 **Discover** 和另外一个详情页 **Detail**，并实现跳转和返回的导航，你应该怎么做呢？

在跟着我具体实现时，需要你先集成 **React Navigation**，这里的具体步骤你可以参考 [🔗 React Navigation 官方文档](#) 自行集成。接着，你就可以跟着我，按照创建“导航地图”、携带参数跳转页面、页面接收和解析参数这三步来实现应用的跳转。

第一步就是创建“导航地图”。

这点和我们日常生活中是一样的。日常生活中，我们要实现地点和地点之间的连接，前提是要有包含这两个地点的地图，比如你可以借助景区地图帮你导航到你想去的地方。**App** 中的页面导航也是如此，要实现页面和页面之间的链接，前提是要有包含该 **App** 所有页面的“导航地图”。

在 React/React Native 中，所有的页面都是由组件创建出来的，因此首先你需要创建两个组件，一个是 **Discover** 组件，另一个是 **Detail** 组件，组件内容比较简单，我就用伪代码代替了：

 复制代码

```
1 function Discover() {
2   return <Text>发现视图</Text>
3 }
4 function Detail() {
5   return <Text>详情视图</Text>
6 }
```

有了组件后，如何使用组件创建页面，又如何把这些页面链接成“导航地图”呢？

这时候你就需要使用到 **React Navigation** 提供的容器组件 **NavigationContainer**，以及创建导航的方法 **createNativeStackNavigator**，示例代码如下：

 复制代码

```
1 import { NavigationContainer } from '@react-navigation/native';
2 import { createNativeStackNavigator } from '@react-navigation/native-stack';
3
4 const Stack = createNativeStackNavigator();
5
6 function App() {
7   return (
8     <NavigationContainer>
9       <Stack.Navigator initialRouteName="Discover">
10         <Stack.Screen name="Discover" component={Discover} />
11         <Stack.Screen name="Detail" component={Detail} />
12       </Stack.Navigator>
13     </NavigationContainer>
14   );
15 }
```

React Navigation 导航分了好几个子库。要创建页面，我们首先要从 **@react-navigation/native** 库中引入 **NavigationContainer** 容器组件，并将其放在最外层包裹住整个 **App** 的 **JSX** 元素。然后从 **@react-navigation/native-stack** 库中引入 **createNativeStackNavigator** 方法，并使用它来创建原生堆栈导航 **Stack**。

原生堆栈导航 **Stack**，是用来创建页面和收集该导航下有哪些页面的。创建页面用的是 **Stack.Screen** 组件，收集页面用的是 **Stack.Navigator** 组件。

在上述代码中，我们使用 **Stack.Screen** 创建了两个页面，名字是“Discover”的页面是由 **Discover** 组件创建的，名字是“Detail”的页面是由 **Detail** 组件所创建的。虽然页面和组件的名字叫法相同，比如都叫 **Discover** 或 **Detail**，但是它们的数据类型不同。页面的名字是字符串类型，创建页面的组件是函数类型。

又因为这里两个 **Stack.Screen** 页面都是 **Stack.Navigator** 元素的子元素，这就相对于告诉了 **Stack.Navigator**，它有两个页面，分别是 **Discover** 页面和 **Detail** 页面。并且我将 **Stack.Navigator** 的 **initialRouteName** 设置成了 **Discover**，目的是告诉导航展示的默认页面是 **Discover** 页面。

完成页面的创建和页面的声明后，你的 App 才算点亮了“导航地图”。

需要提醒你的是，在上述示例中，页面名字和函数组件名字用的叫法是一样的，这只是为了好理解，实际上页面名字和组件名字可以取不同的名字，甚至你可以通过同一个组件来创建多个页面。示例代码如下：

 复制代码

```
1 <Stack.Navigator initialRouteName="Discover1">
2   <Stack.Screen name="Discover1" component={Discover} />
3   <Stack.Screen name="Discover2" component={Discover} />
4   <Stack.Screen name="Discover3" component={Discover} />
5 </Stack.Navigator>
```

如上所示，你可以使用 **Discover** 函数组件同时创建三个不同名字的页面 **Discover1**、**Discover2**、**Discover3**。

在有了导航地图后，要做的第二步是携带参数跳转页面。

首先我们要思考的是：如何从一个页面跳转到另外一个页面呢？

实现页面之间跳转，最常用的方法就是使用 **navigation.navigate** 函数，示例代码如下：

```
1 // 函数组件默认是没有 navigation 对象的
2 // 当函数组件通过 Stack.Screen 生成页面时，才会有 navigation 对象
3 function Discover({navigation}) {
4   return (
5     <Pressable onPress={() => {
6       navigation.navigate('Detail');
7     }}>
8     <Image source={require('./images/dog.png')} />
9   </Pressable>
10 );
11 }
12
13 function Detail() {
14   return (
15     <View>
16       <Text>旺财旺财旺财~</Text>
17       <Image source={require('./images/dog.png')} />
18     </View>
19   )
20 }
```

在上述代码中，在点击按钮后会执行回调函数，回调函数中会执行跳转代码 `navigation.navigate('Detail')`，从而实现了从 **Discover** 页面到 **Detail** 页面的跳转。

其中的关键是 `navigation` 对象，这个 `navigation` 对象是从哪里来的呢？在你将 **Discover** 组件绑定到 `Stack.Screen` 组件的 `component` 参数上执行后，该函数组件 **Discover** 就自动获取到导航对象 `navigation` 了，该对象的 `navigate` 方法可以实现从一个页面到另一个页面的跳转。

在这个示例中，**Detail** 页面的参数是完全写死的，详情页的文案是“旺财”，图片也是小狗的图片，而实际上我们要的详情页会有不同的参数，可能是小狗旺财，也可能是小猫 **Kitty**，还可能是狮子辛巴。

这也意味着，**Detail** 页面的数据是动态的，这些动态的数据需要上一个页面 **Discover** 给它带过来。

接下来的第三步就是，携带自定义参数跳转和接收自定义参数。

在两个页面之间运转参数的载体是路由 `route`。所以接下来，我们就修改一下上面的跳转示例，看下 `route` 是如何帮助我们携带自定义参数跳转的。


```
1  const ALL_NTF = [  
2    { /* ... */ },  
3    {  
4      describe: '旺财旺财旺财~',  
5      image: require('./images/dog.png'),  
6    },  
7    { /* ... */ },  
8  ];  
9  
10 function Discover({navigation}) {  
11   return (  
12     { /* ... */  
13       <Pressable onPress={() => {  
14         navigation.navigate('Detail', ALL_NTF[1]);  
15       }}>  
16         <Image source={ALL_NTF[1].image} />  
17       </Pressable>  
18     { /* ... */ }  
19   );  
20 }  
21  
22 function Detail({route}) {  
23   const { describe, image } = route.params  
24   return (  
25     <View>  
26       <Text>{describe}</Text>  
27       <Image source={image} />  
28     </View>  
29   )  
30 }
```

你可以看到，这里有一份自定义数据和当前页面 **Discover**，还有要跳转的页面 **Detail**。在当前页面 **Discover** 上，**navigation** 对象上的 **navigate** 方法接受的第一个参数是要跳转的页面名字，第二个参数是跳转时要携带的自定义参数。在这里，要跳转的页面是 **Detail** 页面，要携带的自定义参数是“旺财”的相关对象。

“旺财”对象会被挂在 **route** 的 **params** 属性上，因此你可以在 **Detail** 页面使用 **route.params** 来获取“旺财”对象。

route 对象和 **navigation** 对象类似，函数组件默认是没有这两个对象的。当你使用 **Stack.Screen** 创建页面时，用来创建页面的函数组件就会同时获取到 **navigation** 对象和 **route** 对象。其中 **navigation** 对象的主要作用是跳转，**route** 对象的主要作用是携带自定义参数。

那如何从 **Detail** 页面返回到 **Discover** 页面呢？

返回的相关工作，**React Navigation** 会帮你处理，它会给页面创建导航栏，并帮助页面处理返回相关手势动画。在导航栏中会有个回退到上一个页面的返回按钮，除此之外，它还支持 **iOS** 侧滑手势返回上个页面，以及 **Android** 点击底部虚拟回退按钮返回上级页面。

通过创建“导航地图”、携带参数跳转页面、页面接收和解析参数这三步，你就能够实现最基础的页面跳转和返回了。

不过，如果你要更好地用好 **React Navigation**，其中的关键是要理解它的两个配置项和导航路由对象。接下来我们就从它的两个配置项开始讲起。

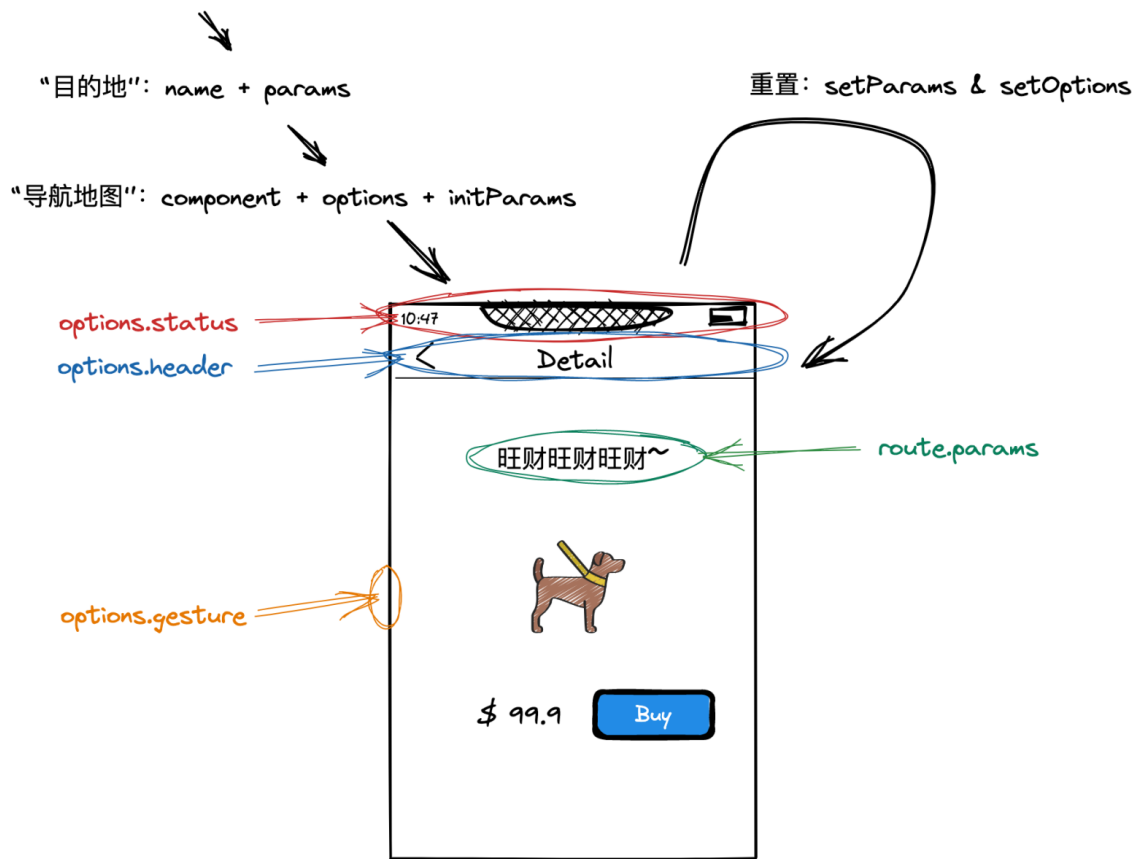
自定义参数 **params**

使用 **React Navigation** 创建出来的页面，有两类属性值比较常用，它们是：

- **params**：它是开发者自定义参数，通常用来渲染页面主体的数据，它是挂在 **route** 上的对象；
- **options**：它是导航相关的配置属性，包括手机顶部的状态栏、页面的标题栏、导航相关手势等等。

虽然 **params** 和 **options** 一个是自定义数据，一个是导航属性，但它们在使用方法上有很多相似之处，所以接下来我们会一起对比着学习。

它们既可以在全局的“导航地图”中进行配置，也可以在当前页面调用相关方法进行重置，示意图如下：



从上图中你可以看到，当你调用 `navigation.navigate` 方法时，就相当于告诉导航你“目的地”的名字 `name` 和要携带参数 `params`。这时导航框架会在它内部的“导航地图”中，也就是 `<Stack.Navigator>{ /* ... */ }</Stack.Navigator>` 中，寻找名字是 `name` 的页面，然后继续找到相关的组件 `component`，并配合导航配置项 `options` 和初始化自定义参数 `initParams`，一起把页面渲染出来。

在一些场景下，`params` 和 `options` 并不是固定的，当前页面也可以根据实际情况使用 `setParams` 和 `setOptions` 方法，对二者进行重新设置。我们这里分析的过程就是页面的 `params` 和 `options` 参数的配置和重置的全过程。

那么，现在来看一下 `params` 配置和重置的方法。

`params` 的配置方法比较简单，在 `Stack.Screen` 上有一个配置属性 `initialParams`，示例代码如下：

```
1 // 数据
2 const ALL_NTF = [
3   {title: 'Kitty',...},
```

复制代码

```

4   {title: '旺财', ...},
5   {title: 'Simba', ...},
6 ];
7
8 // 页面声明
9 <Stack.Screen name="Detail" initialParams={ALL_NTF[0]} component={Detail} />
10
11 // （1）跳转页面，携带 Params
12 navigation.navigate('Detail', ALL_NTF[1]);
13 // （2）跳转页面，不携带 Params，使用默认的 initialParams
14 navigation.navigate('Detail');

```

在这里的示例代码中，我们演示了两种跳转情况：第一种情况是携带“旺财”相关的 `params` 参数进行跳转；第二种情况是没有携带任何 `params` 参数，直接跳转到了 `Detail` 页面。第一种情况，展示的当然是“旺财”页面。不过，第二种没有携带任何参数跳转时，我们需要一个默认参数来渲染 `Detail` 页面，没有参数就会报错。

那要如何设置默认的 `params` 参数呢？在页面声明时，我给 `Stack.Screen` 元素设置了默认属性 `initialParams`，`initialParams` 的值是“小猫 Kitty”，所以，即便遇到不携带任何自定义参数跳转的情况，也会展示“小猫 Kitty”而不会报错。

在页面跳转的过程中，`initialParams` 对象和 `params` 对象会进行对象合并，而不是覆盖，演示代码如下：

 复制代码

```

1 // 对象合并
2 跳转时:  params {price: 99.9 }
3 配置的:  initialParams {symbol: '$'}
4 获取后:  route.params {symbol: '$', price: 99.9 }

```

在该示例中，跳转时传入的参数是价格 99.9，这个价格没有任何货币单位。我们可以在 `initialParams` 中配置它的默认单位 \$，这样在组件函数中获取到的 `route.params` 就既包括了价格 99.9 也包括了货币单位 \$。

因此，`initialParams` 属性的作用是，给页面设置默认的且可覆盖的 `params` 自定义参数。

我们再假设一种情况，如果在用户选择点击“切换成¥”按钮时，要把 \$99.9 按 6.3 的汇率换算成改成 ¥629，我们应该如何重置 `params` 呢？这里重置 `params` 参数，用到的是

navigation.setParams 方法，示例代码如下：

 复制代码

```
1 // 初始化 params: {symbol: '$', price: 99.9, image: 'dog.png' }
2 // 重置后 params: {symbol: '¥', price: 629.37, image: 'dog.png' }
3 function Detail({route, navigation}) {
4   const { price, symbol, image} = route.params
5   return (
6     <>
7       <Image source={image} />
8       <Text>{symbol}{price}</Text>
9       <Text onPress={() =>{
10         if (symbol === '¥') return
11         navigation.setParams({
12           symbol:'¥',
13           price: price * 6.3
14         })
15       }}>切换成¥</Text>
16     </>
17   )
18 }
```

在上述代码中，我们使用 `route.params` 参数将当前页面渲染了出来，包括图片、金额和“切换成¥”的按钮。在你点击“切换成¥”的按钮时，会调用 `navigation.setParams` 方法，将现金符号 `symbol` 和现金价格 `price` 重新设置一次。

初始化时，页面显示的现金符号是 \$，现金价格是 99.9，图片是“小狗旺财”的图片；重新设置后，现金符号是 ¥，现金价格是 629，但图片 `image` 的参数是不变的，它还显示的是“小狗旺财”的图片。

由此可以看出，使用 `navigation.setParams` 重置自定义 `params` 参数时，会将新旧 `params` 对象进行合并，并使用合并后的 `params` 重新渲染页面。

导航配置 options

导航页面中，第二类比较常用的是导航配置属性 `options`。

`options` 具体的配置非常多，如果你有配置的需要，最好查查 [🔗 文档](#)。不过为了让你对 `options` 有个整体的了解，我把一些常用的配置项都列了出来，分成了 3 类：

header 类:

- **title**: 它是字符串，用于设置导航标题；
- **headerBackTitleVisible**: 它是布尔值，用于决定返回按钮是否显示回退页面的名字。默认是 **true** 显示，大多数应用是不显示，因此最好设置为 **false**（iOS 专属）；
- **headerShown**: 它是布尔值，用于决定是否隐藏导航头部标题栏；
- **header**: 它接收一个返回 **React** 元素的函数作为参数，返回的 **React** 元素就是新的导航标题栏。

status 类:

控制屏幕顶部状态栏用的，也可使用 **React Native** 框架提供的 `<StatusBar />` 组件进行代替。

- **statusBarHidden**: 它是布尔值，它决定了屏幕顶部状态栏是否隐藏。

手势动画类:

- **gestureEnabled**: 它是布尔值，它决定了是否可用侧滑手势关闭当前页面（iOS 专属）；
- **fullScreenGestureEnabled**: 它是布尔值，它决定了是否使用全屏滑动手势关闭当前页面（iOS 专属）；
- **animation**: 它是字符串枚举值，它控制了打开或关闭 **Stack** 页面的动画形式，默认“**default**”是页面从右到左地推入动画，也可以设置成其他类型的动画，比如“**slide_from_bottom**”是页面从下到上的推入动画和从上到下的推出动画；
- **presentation**: 它是字符串枚举值，它控制了页面的展现形式，其主要作用是设置页面弹窗。常用的配置值是“**transparentModal**” 它会将页面展示为一个透明弹窗。

那么，这些 **options** 配置项具体怎么用呢？

我给你举个例子，比如你想让详情页的图片展示更加有沉浸感，你就可以把详情页的 **header** 给隐藏了，并让它支持全屏返回。示例代码如下：

```
1 <Stack.Screen name="Detail" component={Detail} options={{
2   headerShown: false,
3   fullScreenGestureEnabled: true
4 }} />
```

[复制代码](#)

如上所示，`Stack.Screen` 元素提供了 `options` 参数，该 `options` 参数可以控制页面导航属性。这里我先是使用了 `headerShown` 隐藏了标题栏，并且使用了 `fullScreenGestureEnabled` 属性使其支持全屏返回。

`options` 参数既可以配置，也可以动态设置。比如，点击某个按钮动态更新 `title` 标题的文案。又比如，有时候在大团队中一个人只负责一个部分，负责该页面的同学不方便修改其他团队维护的全局配置，就可以在当前页面初始化时动态设置 `options` 参数。

以上是通过配置设置 `options` 参数的过程，那如何在当前页面修改 `options` 配置呢？在当前页面重置 `options` 参数用的方法就是 `setOptions`，示例代码如下：

```
1 function Detail({ navigation }) {
2
3   // React.useEffect() 异步副作用回调，执行 setOptions 会导致闪屏，不推荐使用。
4
5   // 页面初始化时，同步设置
6   React.useLayoutEffect(() => {
7     navigation.setOptions({
8       headerShown: false,
9       fullScreenGestureEnabled: true,
10    });
11  }, [navigation])
12
13  // 点击按钮后，异步设置
14  const handlePress = () => {
15    navigation.setOptions({
16      title: '新标题',
17    });
18  }
19
20  return (
21    <Text onPress={handlePress}>设置新标题</Text>
22  );
23 }
```

[复制代码](#)

你可以看到，使用 `navigation.setOptions` 设置导航相关属性有两种形式，一种是同步设置，另一种是异步设置。

在初始化时，为了页面不抖动，我们必须使用同步的方法渲染页面。比如要隐藏头部和设置全局返回手势，如果是放在 `React.useEffect` 这种在页面渲染完成后再异步执行的副作用函数中，就会导致先渲染一次有头部的页面，然后再渲染一次没有头部的页面，头部的消失就会影响到整体页面的高度的变化，这时页面看起来就是抖动的。

React 提供了同步执行的副作用函数 `React.useLayoutEffect`，把 `navigation.setOptions` 放在这里面执行，页面初始化的时候会同步地把头部隐藏起来，这样就不会出现页面抖动的现象了。

而异步设置 `options` 参数的场景，多用在有交互的场景，比如点击某个按钮，改变标题的文案。如示例代码所示，在你点击“设置新标题”的按钮后，就会调用 `navigation.setOptions` 将标题文案重新设置。

各类导航

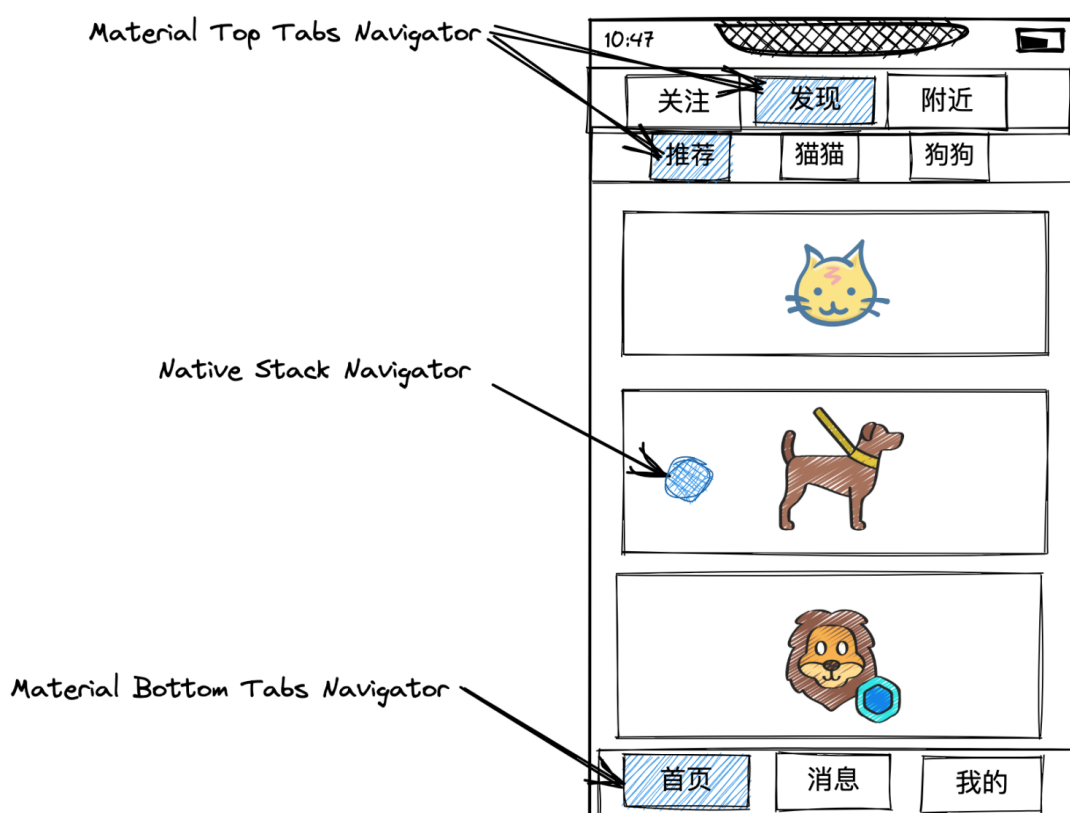
在上述导航示例中，我用的例子都是大家用得最多的 `Native Stack Navigator`。而实际上，除了 `Native Stack Navigator` 这类导航之外，还有 5 类导航：

- **Stack Navigator**: `Stack Navigator` 和 `Native Stack Navigator` 都属于**堆栈导航**，也就是每跳转一次在堆栈的最上面增加一个新页面，每回退一次在堆栈的最上面减少一个老页面。不同的是，`Stack Navigator` 底层使用的是 `Gesture` 手势库和 `Reanimated` 动画库实现的堆栈导航，而 `Native Stack Navigator` 使用的是 iOS 原生 `UINavigationController` 和 Android 原生 `Fragment` 实现的堆栈导航。**一般情况下，我不推荐你使用 `Stack Navigator`，`Native Stack Navigator` 的功能更多，性能也更强大**，[🔗 具体见文档和动图](#)。
- **Drawer Navigator**: **抽屉导航**，也就是从侧边栏推出的导航页面。底层也是用的 `Gesture` 手势库和 `Reanimated` 动画库实现的，类似微信首页侧滑查看收起的小程序或公众号文章，这就属于抽屉导航，[🔗 具体见文档和动图](#)。
- **Bottom Tabs Navigator**: **底部标签导航**。基本上每个 App 底部有好几个 Tab，这种多 Tab 的页面切换的效果在 `React Native` 中就可以用它来实现，[🔗 具体见文档和动图](#)。
- **Material Bottom Tabs Navigator**: 带 Material 样式的底部标签导航，[🔗 具体见文档和动图](#)。

- Material Top Tabs Navigator: 带 Material 样式的顶部标签导航，它是基于 react-native-tab-view 实现的，你可以把 Material 样式换成你自己的样式，常见的多列表 Tabs 就可以用它来实现，🔗具体见文档和动图。

虽然，各类导航实现的效果各不相同，但是它们的使用方法都是大同小异的，都包括创建“导航地图”、携带参数跳转页面、页面接收和解析参数这三步，它们最常用的参数也是 `params` 和 `options`。

那这些导航具体怎么用呢？我用简单电商 App 为例，讲讲常见导航的单个用法和搭配用法，其中涉及了三类导航：Material Top Tabs Navigator、Native Stack Navigator 和 Material Bottom Tabs Navigator。你先看下这张示意图：



你可以看到，这里我们一共用了 4 个导航，从上到下依次是两个顶部标签导航 Material Top Tabs Navigator、堆栈导航 Native Stack Navigator 和底部标签导航 Material Bottom Tabs Navigator。

页面顶部用的标签导航是可以左右滑动切换页面的，而且需要支持双层顶部标签切换。也就是说，我们需要优先切换第二层标签页，如果第二层标签页顶到头了，就切换第一层标签页。比

如推荐标签页在第二层标签页中的最左边已经顶到头了，但是在第一层标签页中，它的左边还有关注标签页，因此你还可以滑向左边的页面，左滑看到就是关注标签页了。

页面主题用的是堆栈导航，中间三个图标是可以点击跳转到详情页的。页面底部用的是底部标签导航，底部标签页只支持点击切换，不支持左右滑动。

那如何使用 **React Navigation** 实现这个 **App** 呢？答案就是使用**导航嵌套**来实现。

现在我先用底部标签导航和堆栈导航示范一下如何实现导航嵌套，示例代码如下：

 复制代码

```
1 import { createNativeStackNavigator } from '@react-navigation/native-stack';
2 import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
3
4 const Stack = createNativeStackNavigator();
5 const Tab = createBottomTabNavigator();
6
7
8
9 function App() {
10   return (
11     <NavigationContainer>
12       <Stack.Navigator initialRouteName="TabHome">
13         <Stack.Screen name="TabHome" component={TabHome} />
14       </Stack.Navigator>
15     </NavigationContainer>
16   );
17 }
18
19 function TabHome() {
20   return (
21     <Tab.Navigator initialRouteName="Home" >
22       <Tab.Screen name="Home" component={Home} />
23     </Tab.Navigator>
24   );
25 }
26
27 function Home() {return <Text>我是首页</Text>}
```

你可以看到，首先我分别使用 `createNativeStackNavigator` 和 `createBottomTabNavigator` 导航创建函数，创建了堆栈导航 **Stack** 和底部导航 **Tab**。接着又创建了“堆栈导航地图” **App**、“底部导航地图” **TabHome**、首页组件 **Home**，这三个函数组件对应的元素关系如下所示：

```
1 - App("导航地图")
2   - TabHome("导航地图")
3     - Home(真正的页面)
```

“堆栈导航地图” App 是根视图，该根视图下面只有一个页面就是“底部导航地图” TabHome，而 TabHome 视图中的子视图只有 Home 视图。

无论是 Stack.Screen 还是 Tab.Screen，这些“导航地图”中的 Screen 的 component 属性既接受普通页面函数作为参数，也可以接受“导航地图”函数作为参数。“导航地图”的 Screen 接收“导航地图”作为 component 参数，就是实现导航嵌套的方法，唯一需要保证的是嵌套的最内层必须是普通页面。

当然，上述示例只有一个 Home 的标签页面，你还可以继续在“导航地图”上进行扩展，多加几个标签页面和堆栈页面，添加完成时候示意结构如下：

```
1 - App("导航地图")
2   - TabHome("导航地图")
3     - Home(标签页面)
4     - Message(标签页面)
5     - My(标签页面)
6     - Page1(堆栈页面)
7     - Page2(堆栈页面)
8     - ...(堆栈页面)
```

如上所示，添加的 Message 和 My 标签页面会在原来的 Home 标签页组成有三个底部 Tab 的 App，这个 App 中还有若干个可以使用 navigation.navigate 方法进行跳转的堆栈页面。

我认为，以上这种使用堆栈“导航地图” Stack 作为根元素，使用底部标签“导航地图” Tab 作为子元素的嵌套方案是实现类似微信、淘宝这种多底部标签 App 的最佳实践。

你可能会问，既然“导航地图”可以相互嵌套，那为什么不使用底部标签“导航地图” Tab 作为根元素，来嵌套堆栈“导航地图” Stack，而是反过来呢？

我们还是以微信、淘宝，这类多底部标签的 App 为例分析一下。这几个 App 都有好几个底部标签页，每个标签页中都有可继续跳转的子页面。如果我们使用 Tab “导航地图”作为根元素，

那么这几个 **Tab** “导航地图”的子元素必须是 **Stack** “导航地图”，示意图如下：

 复制代码

```
1 - App("Tab 导航地图")
2   - Home("Stack 导航地图")
3       - Page1
4       - Page2
5   - Message("Stack 导航地图")
6       - Page3
7       - Page4
```

在这个方案中，**App**“Tab 导航地图”有两个子标签 **Home** 和 **Message**，这两个子标签都是“**Stack** 导航地图”。在 **Home** 子标签下有两个普通页面 **Page1** 和 **Page2**，在 **Message** 子标签下另外两个普通页面 **Page3** 和 **Page4**。

这就是使用“**Tab** 导航地图”嵌套“**Stack** 导航地图”的方案，你可以先思考一下，这个方案有没有什么问题？

第一个问题是，它有两个“**Stack** 导航地图”，管理起来比较麻烦，我们声明页面之前需要考虑该页面应该在哪个 **Tab** 中打开。

第二个问题更严重，如果想从 **Home** 的 **Page1** 页面跳转到 **Message** 的 **Page3** 下面，用户必须点开过 **Message** 标签页，不然就会出现报错。因为在进入 **Message** 标签页之前，它下面的 **Page3**、**Page4** 页面是没有初始化声明过的，直接跳过去会出现报错。

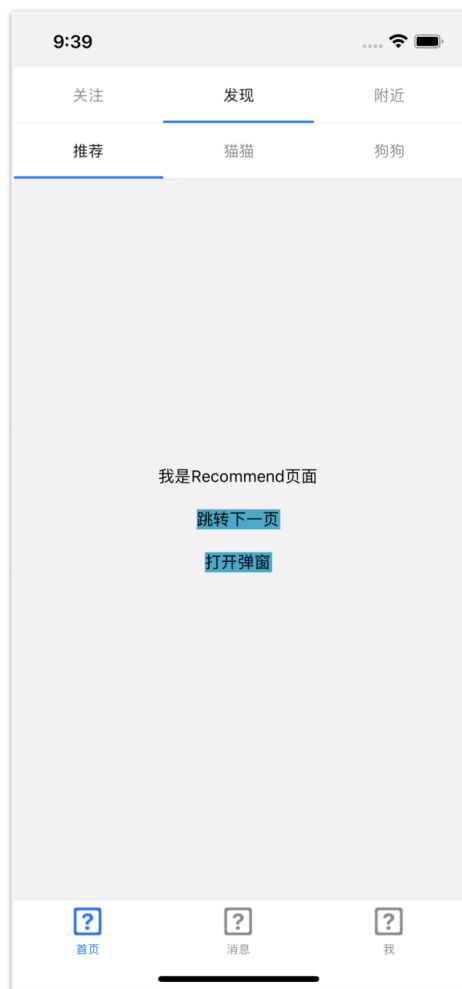
鉴于以上两个问题的存在，我推荐你使用 **Stack** 作为根导航，来嵌套其他 **Tab** 类型的导航。

我们要实现的简易电商 **App** 的架子，也就是两个顶部标签导航、一个底部标签导航和一个堆栈导航的 **App**，它的具体实现思路和最终实现的截屏，我都放在了下面。你对照着看下，理解一下它的实现思路：

 复制代码

```
1 - App("Stack 导航地图")
2   - BottomTabHome("Bottom Tab 导航地图")
3       - TopTabHome("Top Tab 导航地图")
4           - 关注
5       - TopTabDiscover("Top Tab 导航地图")
6           - 推荐
```

```
7      - 猫猫
8      - 狗狗
9      - 附近
10     - 消息
11     - 我
12 - Page1
13 - Page2
14 - PageN...
```



附加材料

最终的代码有点长，我就不贴出来了，你可以在 [GitHub](#) 上查到。

- 最新资料以 [🔗 React Navigation 官网](#) 的 V6 版本为准，它的中文网的示例有点老，我就不推荐了。
- React Navigator 也提供了 10+ 个 [🔗 官方 Demo](#)，你也可以参考着学一下。
- 本文的示例代码见 [🔗 GitHub](#)。

总结

今天这一讲，我们介绍了搭建 React Native App 必备的工具 React Navigator。

在 React Navigator 之中，最常用的导航是原生堆栈导航 Native Stack Navigator，通常情况下都是使用它作为最外层的导航，来包裹其他的底部标签导航和顶部标签导航，实现常见 App 的多标签导航效果。

用好 React Navigator 的关键是，**理解它的两个配置项和导航路由对象**。

在声明页面时，你可以通过配置的方式填写默认的自定义参数 `params` 和配置项 `options`。而在你进入页面后，可以通过导航对象 `navigation` 和路由对象 `route` 来控制页面。导航对象提供的主要功能有跳转回退、监听页面的生命周期和 `setParams`、`setOptions` 功能；路由对象上主要是页面名字 `name`、自定义参数 `params`。

作业


1. 请你使用 Native Stack Navigator 实现一个通用的弹窗功能。该弹窗样式包括一个可自定义的标题和一个可自定义的按钮，它可以在任意页面中调用显示，且当用户点击按钮或半透明背景时，该弹窗整体消失。



2. React Navigation 目前已经更新到 v6 版本了，你在使用老版本的 React Navigation 时遇到过哪些问题呢？欢迎留言，我们大家一起讨论。

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 0  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 17 | Gesture（下）：如何解决多视图多手势的冲突问题？

下一篇 19 | Redux：大型应用应该如何管理状态？

精选留言 (4)

 写留言



潇潇暮雨

2022-05-09

老师，可以讲一下原生页面与RN页面相互跳转吗

作者回复: RN 和 Native 相互跳，用的是 Native 导航。

在混合应用中，RN 只是 Native 的一个页面容器，比如在 Android 中为 Activity/Fragment，在 iOS 中为 ViewController。

跳转是用的 Native 通过 JSI/JS Bridge 暴露给 JS 的 Module/Component 跳转的。



 2



abc 

2022-05-10

一直纠结tab的最佳实践路由，官方文档也不给一个建议~今天学习到了



风之化身

2022-05-09

感觉v6版的 react-navigation 的TS类型系统做的也很好，可以讲讲最佳实践。我们团队用的 v3 版本：1、类型系统不太满意；2、navigation.setParams 对引用类型修改会影响到上一个页面传递过来的





007

2022-05-09

老师，能讲一下Modal页面。还有自定义导航动画么？

作者回复: 你好同学，你可参考以下资料

官方 Modal 的示例、文档和代码

示例: <https://reactnavigation.org/docs/modal/>

文档: <https://reactnavigation.org/docs/native-stack-navigator/#presentation>

代码: <https://github.com/react-navigation/react-navigation/blob/main/example/src/Screens/ModalStack.tsx>

官方动画的文档:

文档: <https://reactnavigation.org/docs/native-stack-navigator/#animation>

