第 10 章

导航与 Modal

在本书第三章中，我们已经讨论了最基本的，完全由开发者代码控制的导航。这种导航，功能简单，并且随着应用的页面增多，控制导航的代码会变得非常大，容易出错。因此在 RN 开发中，开发者通常使用导航组件来实现导航功能。

RN 的官方导航框架有三种，分别是 NavigatorIOS、Navigator 与 react-navigation。

NavigatorIOS 组件有以下几个缺点：

- 轻量、受限的 API 设置，使其相对 Navigator 来说不太方便定制。
- 由开源社区主导开发 —— React Native 的官方团队并没有在自己的应用中使用该组件。这样导致目前有一些积压的 bug，而且没有人负责处理。
- 封装了 UIKit，因此和其他原生应用的表现完全一致。依赖 Objective-C 和 JavaScript，因此只能使用苹果开发好的动画和行为，并且这个组件只支持 iOS。
- 默认包含一个导航栏，这个导航栏不是 React Native 视图组件，因此只能稍微修改样式。

考虑到上述不足，因此不建议读者在 React Native 开发中使用 NavigatorIOS 组件。

从 RN 0.44 版本开始，Navigator 组件被 RN 官方不建议使用，并且被移至名为'react-native-deprecated-custom-components'包中。因为在 RN 0.44 前，这是绝大多数 RN 应用使用的导航框架。开发者在老的代码中可能还会遇到它，在本章最后，我们还是介绍一下它。

react-navigation 组件是目前 RN 官方建议使用的导航组件。它分为三类导航方式：栈式导航、标签导航与抽屉式导航。栈式导航与标签导航之间可以互相嵌套调用。

注意：因为第二次印刷需要与第一次印刷页数一样的关系，本章不能完全印刷在书中，请读者访问本书配套的 Github 网页，下载完整的第十章。

10.1 栈式导航—StackNavigator

栈式导航就好象将一个个界面放入在一个栈中一样。只有栈顶的界面（最后入栈）对用户是可见的。开发者可以进行入栈、出栈等操作。

让我们在第三章例程的基础上，使用 react-navigation 组件来控制页面导航。首先，开发者需要安装 react-navigation 组件。进入命令行窗口，在 RN 项目目录下输入命令：

npm install –save react-navigation

随后，修改 index.XXXX.js 如例程 10-8-1。

例程 10-1-1：

```
import React, { Component } from 'react';
import {
   AppRegistry
} from 'react-native';
import { StackNavigator } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
const SimpleApp = StackNavigator({
     Home: { screen: LoginLeaf },
     Wait: { screen: WaitingLeaf },
});
AppRegistry.registerComponent('LearnRN', () => SimpleApp);
```

可以看到，新的 index.XXXX.js 被大幅简化。在导入需要的组件后，只是调用 StackNavigator 接口生成一个对象，然后直接将这个对象传入 AppRegistry API 的注册组件接口。

修改 LoginLeaf.js 如例程 10-1-2。

例程 10-1-2：

```
import React, { Component } from 'react';
import {
   StyleSheet, Text, View, Dimensions, TextInput, Alert
} from 'react-native';
let widthOfMargin = Dimensions.get('window').width * 0.05;
export default class LoginLeaf extends Component {
   static navigationOptions = {     //这里定义导航的选项，现在只是简单的定义了标题
      title: '登录',
   };
   constructor(props) {
      super(props);
      this.state = {
                inputedNum: '',
                inputedPW: ''
      };
      this.updateNum = this.updateNum.bind(this);
      this.jumpToWaiting = this.jumpToWaiting.bind(this);
   }
   updateNum(newText) {
      this.setState((state) => {
         return {
            inputedNum: newText,
         };
      });
   }
   updatePW(newText) {
```

```
        this.setState(() => {
            return {
                inputedPW: newText,
            };
        });
    }
    render() {
        return (
            <View style={styles.container}>
                <TextInput style={styles.textInputStyle}
                           placeholder={'请输入手机号'}
                           onChangeText ={this.updateNum}/>
                <Text style={styles.textPromptStyle}>
                    您输入的手机号：{this.state.inputedNum}
                </Text>
                <TextInput style={styles.textInputStyle}
                           placeholder={'请输入密码'}
                           secureTextEntry={true}
                           onChangeText={this.updatePW.bind(this)}/>
                <Text style={styles.bigTextPrompt}
                      onPress={()=>this.userPressConfirm()}>
                    确  定
                </Text>
                <Text style={styles.bigTextPrompt}
                    onPress={()=>this.userPressAddressBook()}>
                    通讯录
                </Text>
            </View>
        );
    }
    jumpToWaiting() {
        this.props.navigation.navigate( 'Wait',   //导航跳转命令
                                    {             //传递属性
                                        phoneNumber:this.state.inputedNum,
                                        userPW:this.state.inputedPW
                                    }
         );
    }
    userPressConfirm() {                          //修改 userPressConfirm 函数
        Alert.alert(
            '提示',
            '确定使用'+this.state.inputedNum+'号码登录吗？',
            [
                {text: '取消', onPress:(()=>{}), style: 'cancel' },   //按下取消无操作
                {text: '确定', onPress: this.jumpToWaiting }
            ]
        );
    }
    userPressAddressBook() {

    }
}
```

```
let styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: 'white',
    },
    textInputStyle: {
        margin: widthOfMargin,
        backgroundColor: 'gray',
        fontSize: 20
    },
    textPromptStyle: {
        margin: widthOfMargin,
        fontSize: 20
    },
    bigTextPrompt: {
        margin: widthOfMargin,
        backgroundColor: 'gray',
        color: 'white',
        textAlign: 'center',
        fontSize: 30
    }
});
```

修改 WaitingLeaf.js 如例程 10-1-3。

例程 10-1-3：

```
import React, { Component } from 'react';
import {
    StyleSheet, Text, View
} from 'react-native';
import PropTypes from 'prop-types';
export default class WaitingLeaf extends Component {
    static navigationOptions = {
        title: '登录中',
    };
constructor(props) {
     super(props);
        this.onGobackPressed = this.onGobackPressed.bind(this);
}
    render() {
        const { params } = this.props.navigation.state;
        return (
            <View style={styles.container}>
                <Text style={styles.textPromptStyle} >
                    登录使用手机号：{params.phoneNumber}
                </Text>
                <Text style={styles.textPromptStyle}>
                    登录使用密码：{params.userPW}
                </Text>
                <Text style={styles.bigTextPrompt}
                       onPress={this.onGobackPressed}>
                    返　回
                </Text>
              </View>
```

```
        );
    }
    onGobackPressed() {
     this.props.navigation.goBack();              //弹出当前界面，返回上一个界面
    }
}
WaitingLeaf.propTypes ={                          //注意此时属性类型检查将不再生效
          phoneNumber: PropTypes.string,          //因为传递的属性不在 this.props 下
          userPW: PropTypes.string
       }

let styles = StyleSheet.create({
    container: {
       flex: 1,
       justifyContent: 'center',
       alignItems: 'center',
       backgroundColor: '#F5FCFF',
    },
    textPromptStyle: {
       fontSize: 20
    },
    bigTextPrompt: {
       width: 300,
       backgroundColor: 'gray',
       color: 'white',
       textAlign: 'center',
       fontSize: 60
    }
});
```

修改后的 RN 项目现在就可以运行了。运行时的手机界面如图 10-1-1。

图 10-1-1　例程运行结果

在项目中，没有导入 BackHandler 组件对 android 平台的返回键按下事件进行捕捉与处理。react-navigation 组件将自动帮助开发者完成这个任务。

细心的读者会发现，当从等待界面通过按返回按钮或者按导航栏的返回图标返回到登录页面时，原来用户在登录页面输入的手机号与密码还在。这也是使用导航组件相比第三章中简单导航的一大区别。

例程 10-1-1 的形式也许会让读者感到困惑，原来熟悉的 RN 程序入口怎么变成这个样子了。那么例程 10-1-4 与例程 10-1-1 功能完全相同，会解开读者的疑惑。

例程 10-1-4：

```
import React, { Component } from 'react';
import {
   AppRegistry, BackHandler, Platform
} from 'react-native';
import { StackNavigator } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
const SimpleApp = StackNavigator({
    Home: { screen: LoginLeaf },
    Wait: { screen: WaitingLeaf },
});
export default class LearnRN extends Component {
  render() {
    return (
     <SimpleApp />
```

```
    );
  }
}
AppRegistry.registerComponent('LearnRN', () => LearnRN);
```

以例程 10-1-4 为基础，读者可以在其中加入 RN 的各个生命周期函数与自己需要的其它处理逻辑。

10.1.1 StackNavigator 接口

在例程 10-1-1 中，我们已经使用了最简单形式的 StackNavigator 接口。现在让我们看看它的定义。它的定义是：

StackNavigator(RouteConfigs, StackNavigatorConfig)

它接收两个 JS 对象类型的参数：RouteConfigs 与 StackNavigatorConfig。

RouteConfigs 用来配置 RN 应用的路由信息。它的结构是：

```
{
    路由名称：路由配置，
    ……          //可以有任意个，示例：Home: { screen: LoginLeaf },
}
```

需要注意的是路由名称不是字符串，不需要用单引号或者双引号包围。

每个路由配置是一个 JS 对象，它的结构是：

```
{
   screen: ProfileScreen,          //对应 RN 组件的名称，必须有
   path: 'people/:username',       //可选项，供深度链接提取活动与路径参数
   navigationOptions: 回调函数,    //可选项，当有时函数返回的 navigationOptions
                                   //将覆盖组件提供的 navigationOptions
}
```

当一个 RN 组件通过 StackNavigator 接口加载时，它会自动得到一个名为 navigation 的属性。这个属性是一个对象，对象的成员函数可以让开发者操纵路由。

StackNavigator 接口的第二个参数 StackNavigatorConfig 也是一个 JS 对象。它的结构是：

```
{
   initialRouteName: ,             //设置默认路由名称，必须是 RouteConfigs 中定义的
   initialRouteParams: ,           //初始路由参数
   navigationOptions: ,            //所有界面的默认导航选项设置
   paths: ,                        //一个 JS 对象，对象中的每个元素代表着一条开发
                                   //者希望重写的路由配置中的 path
   mode: 'card'/'modal',           //定义渲染与切换的风格
   headerMode: 'float'/'screen'/'none',     //定义导航栏如何渲染
   cardStyle: ,                    //使用它重写或者扩展导航栈中的每个界面的默认 style
   transitionConfig: ,             //一个返回对象的回调函数，它将替代默认的界面切换操作
```

```
    onTransitionStart: ,            //界面切换发生前的回调函数
    onTransitionEnd: ,              //界面切换完成后的回调函数
}
```

需要注意的是 StackNavigatorConfig 这个参数可以没有。如果提供了这个参数，这个参数中的任意一个成员都可以没有。

mode 取值为 card 与 modal 之一。card 是默认值，它代表界面切换将使用标准的 iOS 与 Android 风格。modal 只在 iOS 平台生效，它让界面从屏幕底部向上升起。

headerMode 取值为 float、screen、none 三者之一。float 代表导航栏始终在屏幕顶部分（iOS 风格）。screen 代表导航栏可以随着屏幕的上下划动而消失、重现（Android 风格）。none 表示导航栏将不会呈现。

在例程 10-1-1 中添加代码如例程 10-1-5。

例程 10-1-5：

```
……
const SimpleApp = StackNavigator(
      {
          Home: { screen: LoginLeaf },
          Wait: { screen: WaitingLeaf }
      },
      {
          headerMode: 'none'
      }
);
……
```

就可以让导航栏隐藏掉，新的代码界面与第三章代码效果一样。其它本小节讨论的参数，读者可以自行尝试效果。

10.1.2　栈式导航屏幕导航选项

在例程 10-1-2 中与例程 10-1-3 中，第个组件都有一个静态成员变量 navigationOptions。它是组件的屏幕导航选项。它的结构是：

```
{
    title: ,              //字符串，导航栏标题，没有提供 headerTitle 时将使用此值
    header: ,             //React 元素或者一个返回 React 元素的函数 ,当它为 null 时导航栏将隐藏
    headerTitle: ,        //字符串或者 React 元素用来显示导航栏标题，默认值为 title
    headerBackTitle: ,    //显示在返回按钮上的字符串，当它为 null 时，将没有返回标签
    headerTruncatedBackTitle: ,//当 headerBackTitle 无法适应屏幕时使用的字符串，默认值 Back
    headerRight: ,                    //React 元素，用来显示在导航栏的右侧
    headerLeft: ,                     //React 元素，用来显示在导航栏的左侧
    headerStyle: ,                    //导航栏的样式
    headerTitleStyle: ,               //导航栏标题样式
    headerBackTitleStyle: ,           //导航栏返回区域样式
```

```
    headerTintColor: ,                    //导航栏染色值
    headerPressColorAndroid: ,            //水滴效果颜色，只对 Android 5.0 或以上机型生效
    gesturesEnabled: ,     //是否可以使用手势退出界面，默认值：true(iOS)，false(Android)
}
```

屏幕导航选项在每个界面中定义，只对那个界面生效。

例如我们希望在导航栏右侧设置一个按钮，可以写：

```
    ……
    static navigationOptions = {
        headerRight: <Button title="Info" />,
        ……
```

设置导航栏选项时，可以使用传入的属性。使用方法参见例程 10-1-6。

例程 10-1-6：

```
……
export default class WaitingLeaf extends Component {
    static navigationOptions = ({ navigation }) => ({
        title: `Logining with ${navigation.state.params.inputedNum}`,
    });
……
```

10.1.3 栈式导航导航属性

除了在使用 navigator 函数跳转界面时携带属性外，开发者还可以设置一个通用的界面属性。设置方法见例程 10-1-7。

例程 10-1-7：

```
……
const SimpleAPP = StackNavigator({
  // config
});

<SimpleAPP
  screenProps={
                  prop1: value1,
                  prop2: value2,
                  ……
              }
/>
……
```

开发者在各个界面中都可以通过“this.props.screenProps.属性名称”来获取相应的属性值。本书的例程 10-8-2 示例了界面属性的使用。

10.2 界面导航属性

屏幕导航属性用来操纵界面之间的跳转。它虽然被放在 10.2 中讨论，但它除了可以在 10.1 中讨论的 StackNavigator 中使用外，还可以在马上要讨论的 TabNavigator 与 DrawerNavigator 中使用。

10.2.1 组件路径参数

当一个 RN 组件通过 StackNavigator 接口加载时，它会自动得到一个名为 navigation 的属性。这个属性下面有个名为 state 的变量记录了当前组件的路径参数。它的结构：

```
{
    routeName: 'XXXX',                    //路径名称，字符串类型
    key: 'XXXX',                          //路径的唯一 key 值，字符串类型
    params: { ……}                         //路径携带的参数，对象类型
}
```

其中，routeName 与 params 都是开发者在代码中定义的，但 key 值是 react-navigation 组件产生的，不受开发者控制。

开发者可以简单的在 WaitingLeaf 组件的 render 函数首部加入 console.log(this.props.navigation.state)语句，打印出组件路径参数。

10.2.2　导航操控

自动得到的 navigation 属性有四个公开函数，可以用来操控导航。它们是：navigate, setParams, goBack 与 dispatch。在 10.8 节中，我们已经使用了 navigate 函数与 goBack 函数。

navigate 函数的完整形式是：navigate(routeName, params, action)，通常开发者不需要使用第三个参数。

setParams 函数用来修改 10.2.1 中提到的组件路径参数中的 params 对象值。

dispatch 函数的使用在 10.3 中讨论。

goBack 函数的作用之一是：关闭当前组件，回到上一个组件。

在一个组件中，goBack 函数只在调用第一次时生效，如果连续调用，将不会生效。但在 APP 开发中，开发者经常发现自己需要按顺序关闭多个组件。比如一个 APP 目前有五个界面，按先后呈现顺序是 A,B,C,D,E。在 E 界面用户按下一个按钮后，开发者希望关闭 E 与 D 界面，回到 C 界面。

为了实现这个功能，开发都需要在进入 D 界面时，得到 D 组件的组件路径参数中的 key 值，并且将这个 key 值传入到 E 组件中。当希望回到 C 界面时，开发者调用 goBack 函数时传递入 D 组件的 key 值，表示希望从 D 组件开始执行 goBack 操作。这样就可以达到期望的效果。

当 goBack 函数被调用时，被弹出组件的 componentWillUnmount 函数将被调用。随后这个组件有可能在任何时候被 JS 的垃圾回收机制收回。

10.3　导航活动

开发者可以创建一个导航活动，然后使用 navigation 的 dispatch 函数来执行这个导航活动。它虽然被放在 10.3 中讨论，但它除了可以在 10.1 中讨论的 StackNavigator 中使用外，还可以在马上

要讨论的 TabNavigator 与 DrawerNavigator 中使用。

开发者可以创建的导航活动有：

navigate --导航至另一个组件，作用与 navigate 函数相同，事实上开发者调用这个函数后，react-navigation 组件会创建这个活动并执行（下面几个活动同理）；

back --弹出当前组件，并返回上一组件，作用与 goBack 函数相同；

setParams--设置当前组件组件路径参数中的 params 对象值，作用与 setParams 函数相同；

init --用来初始化第一个组件如果它的状态没有定义；

reset --将当前整个导航路径重置为指定的导航路径；

这五个活动中，前三个与对应的函数作用相同。它们的主要作用是在导航组件外对导航进行操作。如果需要使用，请阅读 react-navigation 官方文档。

init 活动不建议使用，或者说开发者在启动第一个组件前应当设置好它的状态。因此下面将重点讨论 reset 活动。例程 10-3-6 示例了 reset 活动的使用方法。请读者将 WaitingLeaf 组件中的 onGobackPressed 函数修改为如例程 10-3-1 所示。

例程 10-3-1：

```
……
onGobackPressed() {
       const resetAction = NavigationActions.reset({
          index: 5,
          actions: [
             NavigationActions.navigate(
                {
                   routeName: 'Home'
                }
             ),
             NavigationActions.navigate(
                {
                   routeName: 'Wait',
                   params: {
                             phoneNumber: 1,
                             userPW:1
                          }
                }
             ),
             NavigationActions.navigate(
                {
                   routeName: 'Wait',
                   params: {
                             phoneNumber: 2,
                             userPW:2
                          }
                }
             ),
             NavigationActions.navigate(
                {
```

```
                routeName: 'Wait',
                params: {
                        phoneNumber: 3,
                        userPW:3
                    }
            }
        ),
        NavigationActions.navigate(
            {
                routeName: 'Wait',
                params: {
                        phoneNumber: 4,
                        userPW:4
                    }
            }
        ),
        NavigationActions.navigate(
            {
                routeName: 'Wait',
                params: {
                        phoneNumber: 5,
                        userPW:5
                    }
            }
        )
    ]
  })
  this.props.navigation.dispatch(resetAction);
}
……
```

修改后，在等待界面按下返回按钮（非导航栏左侧的返回箭头）时，reset 活动将被执行，它会重置当前的整个导航路径，然后将当前导航栈设置为一个登录面与五个等待页。如果开发者实现 componentWillUnmount 函数，将会发现在 reset 时，原来导航栈中的两个组件的 componentWillUnmount 函数都会被执行。

10.4　标签导航--TabNavigator

标签导航是 APP 中常见的导航方式。使用 TabNavigator 接口可以很方便地实现标签导航。

修改 index.XXXX.js 如代码 10-4-1。

例程 10-4-1：

```
import React, { Component } from 'react';
import {
   AppRegistry, BackHandler, Platform
} from 'react-native';
import { StackNavigator, TabNavigator } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
import Mine from './Mine';
const MainScreenNavigator = StackNavigator({
```

```
    Login: { screen: LoginLeaf },
    Wait: { screen: WaitingLeaf },
});
const SimpleApp = TabNavigator({
    Home: { screen: MainScreenNavigator, },
    Mine: { screen: Mine, },
});
AppRegistry.registerComponent('LearnRN', () => SimpleApp);
```

请读者注意在例程 10-4-1 中，先是建立了一个栈式导航，然后将这个栈式导航嵌套到一个标签导航中。这就是导航的嵌套。

新建 Mine.js 如代码 10-4-2。

例程 10-4-2：

```
import React, { Component } from 'react';
import {
   StyleSheet, Text, View
} from 'react-native';
import { NavigationActions } from 'react-navigation';
export default class Mine extends Component {
   static navigationOptions = {
      title: '登录中',
   };
   render() {
    console.log( 'list state of route.');
    console.log( this.props.navigation.state );
      return (
         <View style={styles.container}>
            <Text style={styles.textPromptStyle} >
               我的
            </Text>

         </View>
      );
   }
   componentWillUnmount() {
    console.log( 'Mine  will close.');
   }
}
let styles = StyleSheet.create({
   container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: '#F5FCFF',
   },
   textPromptStyle: {
      fontSize: 20
   },
   bigTextPrompt: {
      width: 300,
      backgroundColor: 'gray',
      color: 'white',
```

```
        textAlign: 'center',
        fontSize: 60
    }
});
```

修改后，运行例程。运行结果如图 10-4-1。

图 10-4-1　例程运行结果

例程 10-4-1 中因为没有对导航栏进行设置，在页面中会同时出现标签导航的导航栏与栈式导航的导航栏。这些设置工作比较简单，不再讨论。

10.4.1　TabNavigator 接口

TabNavigator 接口的定义是：

TabNavigator(RouteConfigs, TabNavigatorConfig)

其中，RouteConfigs 的类型，结构与 StackNavigator 接口中的类型、结构完全一样，不再重述。需要注意的是，在例程 10-11-1 中，我们把一个 StackNavigator 作为一个路由配置交给了 TabNavigator。以这种方式实现了不同导航方式的嵌套使用。

TabNavigator 接口的第二个参数 TabNavigatorConfig 是一个 JS 对象，它的结构：

```
{
    tabBarComponent: ,                  //用来渲染标签栏的组件
    tabBarPosition： 'top'/'bottom',    //设定标签栏在屏幕最上方还是最下方
    swipeEnabled: true/false,           //是否允许通过手势划动来切换标签页
    animationEnabled: true/false,       //标签页切换时是否有动画效果
    lazy: true/false,                   //是否只渲染需要显示的标签
    tabBarOptions: ,                    //JS 对象，用来配置标签栏
```

```
    initialRouteName: ,                    //初始标签的名称
    order： ,                              //在个存放标签名称的数组，用来决定标签顺序
    paths                                  //一个 JS 对象，对象中的每个元素代表着
                                           //一条开发者希望重写的路由配置中的 path

    backBehavior: ,            //默认情况下，按 Android 手机的返回键将返回 initialRouteName
                               //指向的标签页。backBehavior 设为非‘initialRoute’值
                               //将让返回键直接结束应用
}
```

上述对象结构中的第一个参数的示例可以参考“项目名称”\node_modules\react-navigation\src\views\TabView\下的 TabBarBottom.js 与 TabBarTop.js。其中，TabBarBottom 组件是在 iOS 平台运行时 tabBarComponent 的默认值 ,TabBarTop 组件是在 Android 平台运行时的 tabBarComponent 的默认值。因此可以知道，标签导航的标签栏位置默认在两个平台是不同的。

TabNavigatorConfig 中的 tabBarOptions 是一个 JS 对象，在 iOS 平台，它的结构：

```
{
    activeTintColor            //当标签栏激活时的染色
    activeBackgroundColor      //仅 iOS 平台有效，当标签栏激活时，标签栏背景色
    inactiveTintColor          //当标签栏非激活时的染色
    inactiveBackgroundColor    //仅 iOS 平台有效，当标签栏非激活时，标签栏背景色
    showLabel                  //是否显示标签栏，默认值为 true
    style                      //标签栏样式
    labelStyle                 //标签样式
}
```

而在 Android 平台，除了上面列出的结构，还有：

```
{
    showIcon                   //是否显示标签图标，默认值为不显示（false）
    upperCaseLabel             //是否让标签英文全大写，默认值为 true
    pressColor                 //按下时涟漪效果的颜色（Android 操作系统大于等于 5.0）
    pressOpacity               //按下时标签透明度（Android 操作系统小于 5.0）
    scrollEnabled              //是否允许标签栏横向滚动
    tabStyle                   //标签页样式
    indicatorStyle             //标签指示器样式
    iconStyle                  //标签图标样式
}
```

10.4.2 标签导航屏幕导航选项

在例程 10-11-1 中与例程 10-11-2 中，第个组件都有一个静态成员变量 navigationOptions。它是组件的屏幕导航选项。它的结构是：

```
{
     title: ,                //字符串，导航栏标题，没有提供 tabBarLabel 时将使用此值
     tabBarVisible ,         //布尔值，是否显示标签栏，默认值为 true
     tabBarIcon ,            //React 元素，用来渲染图标
     tabBarLabel ,           //一个字符串或者一个 React 元素用来显示在标签中。
}
```

屏幕导航选项在每个界面中定义，只对那个界面生效。

通常在这里定义标签导航栏的图标。定义标签导航栏图标的代码参见例程 10-4-3。

例程 10-4-3：

```
……
static navigationOptions = {
    tabBarLabel: 'Home',
    tabBarIcon: ({ tintColor }) => (
      <Image
        source={require('./anIcon.png')}
        style={[styles.icon, {tintColor: tintColor}]}
      />
    ),
  };
……
```

需要注意的是对于 Android 平台，默认是不显示导航栏图标。即使如例程 10-4-3 一样定义了导航栏图标，还需要在 TabNavigatorConfig 的 tabBarOptions 中将 showIcon 设为 true，才会显示图标。

10.4.3　标签导航导航属性

标签导航中导航属性的使用方法与 10.1.3 中栈式导航导航属性使用方法相同，不再重述。

10.5　抽屉式导航—DrawerNavigator

抽屉导航也是常用的一种导航方式，DrawerNavigator 接口可以很方便的实现抽屉式导航。

修改 index.XXXX.js 如例程 10-5-1。

例程 10-5-1：

```
import React, { Component } from 'react';
import {
    AppRegistry, BackHandler, Platform, Button
} from 'react-native';
import { DrawerNavigator } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
import Mine from './Mine';

const SimpleApp = DrawerNavigator(
{
     Login: { screen: LoginLeaf },
```

```
    Wait: { screen: WaitingLeaf },
    Mine: { screen: Mine, },
}
);
AppRegistry.registerComponent('LearnRN', () => SimpleApp);
```

修改 LoginLeaf.js 如例程 10-5-2。

例程 10-5-2：

```
import React, { Component } from 'react';
import {
   StyleSheet, Text, View, Dimensions, TextInput, Alert, Image
} from 'react-native';
let widthOfMargin = Dimensions.get('window').width * 0.05;
export default class LoginLeaf extends Component {
   static navigationOptions = {
      drawerLabel: 'Login',
      drawerIcon: ({ tintColor }) => (
         <Image
            source={require('./chats-icon.png')}
            style={[styles.icon, {tintColor: tintColor}]}
         />
      ),
   };
   constructor(props) {
      super(props);
      this.state = {
                inputedNum: '',
                inputedPW: ''
      };
      this.updateNum = this.updateNum.bind(this);

   }
   updateNum(newText) {
      this.setState((state) => {
         return {
            inputedNum: newText,
         };
      });
   }
   updatePW(newText) {
      this.setState(() => {
         return {
            inputedPW: newText,
         };
      });
   }
   render() {
      return (
         <View style={styles.container}>
            <TextInput style={styles.textInputStyle}
                       placeholder={'请输入手机号'}
                       onChangeText ={this.updateNum}/>
            <Text style={styles.textPromptStyle}>
```

```
                您输入的手机号：{this.state.inputedNum}
            </Text>
            <TextInput style={styles.textInputStyle}
                       placeholder={'请输入密码'}
                       secureTextEntry={true}
                       onChangeText={this.updatePW.bind(this)}/>

            <Text style={styles.bigTextPrompt}
                  onPress={ ()=>this.userPressConfirm()}>
                确　定
            </Text>
            <Text style={styles.bigTextPrompt}
                  onPress={()=>this.userPressAddressBook()}>
                通讯录
            </Text>
            </View>
        );
    }
    userPressConfirm() {
        this.props.navigation.navigate('DrawerOpen');  //打开抽屉导航页
    }
}       //这个反大括号标识了 LearnRN 类定义的结束

let styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: 'white',
    },
    textInputStyle: {
        margin: widthOfMargin,
        backgroundColor: 'gray',
        fontSize: 20
    },
    textPromptStyle: {
        margin: widthOfMargin,
        fontSize: 20
    },
    icon: {
        width: 24,
        height: 24,
    },
    bigTextPrompt: {
        margin: widthOfMargin,
        backgroundColor: 'gray',
        color: 'white',
        textAlign: 'center',
        fontSize: 30
    }
});
```

修改 WaitingLeaf.js 如例程 10-5-3。

例程 10-5-3：

```
import React, { Component } from 'react';
import {
   StyleSheet, Text, View, Image
} from 'react-native';
import PropTypes from 'prop-types';
export default class WaitingLeaf extends Component {
   static navigationOptions = {
      drawerLabel: 'Waiting',
         drawerIcon: ({ tintColor }) => (
            <Image
               source={require('./notif-icon.png')}
               style={[styles.icon, {tintColor: tintColor}]}
            />
         ),
   };
   constructor(props) {
      super(props);
   }
   render() {
      return (
         <View style={styles.container}>
            <Text style={styles.textPromptStyle} >
               登录使用手机号：{this.props.phoneNumber}
            </Text>
            <Text style={styles.textPromptStyle}>
               登录使用密码：{this.props.userPW}
            </Text>
            <Text style={styles.bigTextPrompt}
                  onPress={()=>this.onGobackPressed()}>
               返  回
            </Text>
          </View>
      );
   }
   onGobackPressed() {
        this.props.onGobackPressed();
   }
}
WaitingLeaf.propTypes ={
          phoneNumber: PropTypes.string,
          userPW: PropTypes.string
      }

let styles = StyleSheet.create({
   container: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
      backgroundColor: '#F5FCFF',
   },
   textPromptStyle: {
      fontSize: 20
   },
```

```
    icon: {
        width: 24,
        height: 24,
    },
    bigTextPrompt: {
        width: 300,
        backgroundColor: 'gray',
        color: 'white',
        textAlign: 'center',
        fontSize: 60
    }
});
```

修改 Mine.js 如例程 10-5-4。

例程 10-5-4：

```
import React, { Component } from 'react';
import {
    StyleSheet, Text, View, Image
} from 'react-native';
import { NavigationActions } from 'react-navigation';
export default class Mine extends Component {
    static navigationOptions = {
        drawerLabel: 'Mine',
            drawerIcon: ({ tintColor }) => (
                <Image
                    source={require('./notif-icon.png')}
                    style={[styles.icon, {tintColor: tintColor}]}
                />
            ),
    };
    render() {
        console.log( 'list state of route.');
        console.log( this.props.navigation.state );
        return (
            <View style={styles.container}>
                <Text style={styles.textPromptStyle} >
                    我的
                </Text>

            </View>
        );
    }
    componentWillUnmount() {
        console.log( 'Mine  will close.');
    }
}
let styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    textPromptStyle: {
```

```
        fontSize: 20
    },

    icon: {
        width: 24,
        height: 24,
    },
    bigTextPrompt: {
        width: 300,
        backgroundColor: 'gray',
        color: 'white',
        textAlign: 'center',
        fontSize: 60
    }
});
```

运行代码，运行结果如图 10-5-1。

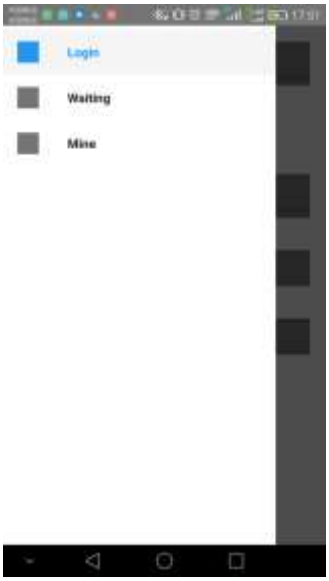

图 10-5-1　例程运行结果

对于抽屉导航而言，navigate 函数有两个特别的参数，分别是'DrawerOpen'，用来弹出抽屉页，以及'DrawerClose'，用来关闭抽屉页。抽屉导航，并不象其它两个导航一样会默认加上导航栏。因此开发者需要在页面上自己编写触发抽屉页打开或者关闭的 UI 元素与逻辑。

10.5.1　DrawerNavigator 接口

DrawerNavigator 接口的定义是：

DrawerNavigator(RouteConfigs, TabNavigatorConfig)

其中，RouteConfigs 的类型，结构与 StackNavigator 接口中的类型、结构完全一样，不再重述。需要注意的是，在例程 10-4-1 中，我们把一个 StackNavigator 作为一个路由配置交给了 TabNavigator。以这种方式实现了不同导航方式的嵌套使用。

DrawerNavigator 接口的第二个参数 TabNavigatorConfig 是一个 JS 对象，它的结构：

```
{
    drawerWidth: ,              //抽屉的宽度
    drawerPosition： ,          //取值在'left'与'right 中二选一，默认为 left
    contentComponent： ,        //RN 组件用来渲染抽屉
    contentOptions： ,          //抽屉内容设置
}
```

DrawerNavigator 中的 contentOptions 是一个 JS 对象，它的结构：

```
    {
        activeTintColor： ,                 //当标签栏激活时的染色
```

```
        activeBackgroundColor: ,            //仅 ios 平台有效，当标签栏激活时，标签栏背景色
        inactiveTintColor： ,               //当标签栏非激活时的染色
        inactiveBackgroundColor: ,     //仅 ios 平台有效，当标签栏非激活时，标签栏背景色
        style： ,                           //抽屉面样式
        labelStyle： ,                      //标题样式
    }
```

10.5.2　提供订制化的 contentComponent

contentComponent 用来显示在弹出的抽屉页上。例程 10-5-5 简单地订制化了一个 contentComponent。

例程 10-5-5，index.XXXX.js：

```
import React, { Component } from 'react';
import {
   AppRegistry, BackHandler, Platform, View, Text, StyleSheet
} from 'react-native';
import { DrawerNavigator, DrawerItems } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
import Mine from './Mine';


const CustomDrawerContentComponent = (props) => (    //订制的 contentComponent
<View style={styles.container}>
    <DrawerItems {...props} />
    <Text>
        My custom drawer content
    </Text>
</View>
);

const SimpleApp = DrawerNavigator(
{
    Login: { screen: LoginLeaf },
    Wait: { screen: WaitingLeaf },
    Mine: { screen: Mine, },
},
{
        contentComponent: CustomDrawerContentComponent  //设置订制的 contentComponent
    }
);
let styles = StyleSheet.create({
   container: {
      flex: 1,
      backgroundColor: 'white',
   }
```

```
    });
AppRegistry.registerComponent('LearnRN', () => SimpleApp);
```

运行后的效果如图 10-5-2。

图 10-5-2　例程运行结果

10.5.3　屏幕导航选项

在例程 10-5-1 中与例程 10-5-2 中，第个组件都有一个静态成员变量 navigationOptions。它是组件的屏幕导航选项。它的结构是：

```
{
    title: ,              //字符串，导航栏标题，没有提供 drawerLabel 时将使用此值
    drawerIcon: ,         //抽屉图标
    drawerLabel: ,        //字符串或者 React 元素用来显示在抽屉的边栏上
}
```

屏幕导航选项在每个界面中定义，只对那个界面生效。

10.5.4　导航属性

标签导航中导航属性的使用方法与 10.1.3 中栈式导航导航属性使用方法相同，不再重述。

10.6　Modal 组件

Modal 组件用来在 RN 的某个 View 上呈现内容。如果读者觉得这句话不好理解，想一想 Alert API 提供的对话框。Modal 组件的功能与它类似，区别在于 Alert 呈现的对话框格式固定，而利用 Modal 组件，我们可以在某个 View 上按开发者的需求呈现任意 UI 界面，并且处理这个 UI 界面的交互事件。前面介绍过的各个 UI 组件都可以包含在 Modal 组件中，在需要时呈现给用户。当业务逻辑需要时，我们又可以将 Modal 组件撤出屏幕，让原来被遮盖的 UI 界面显露出来并正常工作。

为什么在本章介绍 Modal 组件。因为在移动应用开发中，Modal 组件最常用的功能是与 Navigator 组件联合使用。我们将马上看到它们联合使用的例程。

Modal 组件的属性

Modal 组件本身没有样式设置，它只有属性。它的属性有：

animationType，这是一个字符串类型的属性，取值只能是 none、slide、fade 三者之一。它用来控制 Modal 组件的呈现方式。none 代表没有动画效果，直接呈现。Slide 代表从底部滑上来。fade 代表淡入淡出。

onShow，这是一个回调函数类型的属性。当 Modal 组件在屏幕上完成显示后，这个回调函数将被执行。

transparent，这是一个布尔类型的属性。它决定 Modal 组件是否是透明的。当它为 true 时，用户通过 Modal 组件能看到原来的 View 的内容（有半透明效果）。就好像 Modal 组件浮在原来的 View 上一样。

visible，这是一个布尔类型的属性。它决定 Modal 组件何时显示、何时隐藏。

onRequestClose，这是 Android 平台独有的回调函数类型的属性。当 Modal 在显示时，用户按下返回键后，这个函数将被调用。在 Android 平台，这个属性必须要有值，或者说开发者必须要写一个函数来处理当 Modal 显示在界面时且返回键被按下时的业务逻辑。

onOrientationChange，这是 iOS 平台独有的回调函数类型的属性。当 Modal 在显示时，如果手机的放置方向改变，这个回调函数将被调用。

supportedOrientations，这是 iOS 平台独有的字符串类型的属性。它的取值只能是 portrait、portrait-upside-down、landscape、landscape-left、landscape-right 之一。它用来声明 Modal 组件支持的手机放置方向。有 iOS 开发经验的人应该都知道 iPhone 手机的四种放置状态，分别为 Portrait、PortraitUpsideDown、LandscapeLeft 和 LandscapeRight。而多出的 landscape 代表 LandscapeLeft 和 LandscapeRight 都可以。

10.7　Modal 组件与 Navigator 组件的配合

移动应用中，典型的业务逻辑是在 A 界面上，用户点击某个区域，RN 应用与网络服务器交

换数据，然后跳转到 B 界面向用户呈现新的数据。

其中从 A 界面跳转到 B 界面一般是由 Navigator 组件来完成的。但这里有一个问题，因为 RN 应用与网络服务器交换数据有时延。如果在交换数据时，用户又进行了一个操作，这个操作要求跳转到 C 界面，RN 应用怎么处理呢？

解决的办法之一是不让用户再进行操作。当用户执行一个操作后，将界面锁住，向用户显示等待信息，然后开始与网络服务器交换数据，直到数据交换完成并跳转到新的界面，才解锁界面，允许用户继续操作。

可以有很多种方法锁住界面，但最简洁高效的做法是让 Navigator 组件与 Modal 组件配合实现。

10.8 Modal 组件例程

在第 3 章中，我们完成了从登录界面到注册结果界面的转换。现在让我们假装这个界面跳转过程中有一个网络登录的延时，在延时时使用 Modal 组件锁住屏幕，同时向用户呈现等待提示。

10.8.1 实现自定义 Modal 组件

使用 Modal 组件的第一步是实现自定义的 Modal 组件界面。在本例程中，我们希望在屏幕中出现一个文本“登录中，请稍候”，同时下方显示一个活动指示器。实现代码参见代码 10-7-1。

代码 10-8-1，WaitingModal：

```
import React, { Component } from 'react';
import {
   StyleSheet, Text, View, Modal, ActivityIndicator          //导入 Modal 组件
} from 'react-native';
export default class WaitingModal extends Component {
   render() {
      return (
         <Modal transparent={true}                          //设置 Modal 透明
            onRequestClose={() => {}}                      //忽略用户按下 Android 返回键事件
            visible={this.props.show}>                     //让父组件传入的属性决定 Modal 是否显示
            {/*下面的部分是自定义 Modal 组件真正的 UI 界面代码*/}
               <View style={ styles.mainViewStyle}>
                  <View style={ styles.contentViewStyle}>
                     <Text style={ styles.textStyle }>
                        {this.props.prompt}
                     </Text>
                     <ActivityIndicator animating={true}
                        color={'blue'}
                        size={'large'}/>
                  </View>
               </View>
         </Modal>
```

```
        );
    }
}
const styles = StyleSheet.create({
    mainViewStyle:{
        flex:1,
        alignItems: 'center',
        justifyContent: 'center',
        backgroundColor: 'rgba(0,0,0,0.75)'    //设置主 View 的半透明效果，Modal 才能透明
    },
    contentViewStyle:{
        backgroundColor:'white'    //Modal 显示内容背景色为白色，不让背景透过来，让用户看不清
    },
    textStyle:{
        fontSize: 30,
        margin:30
    }
});
```

读者可以看到，除了 render 函数返回的 JSX 元素的外层是一个 Modal，其他与实现一个普通的 UI 界面没有区别。此例程实现的 UI 很简单。在真正的应用开发中，视业务需要，图片、按钮等都可以用来拼装自定义 Modal 组件。Modal 组件也可以像一个普通的组件一样拥有自己的状态机变量，进行各种业务逻辑处理。一句话，它是一个 RN 组件，普通 RN 组件能做的它都能做，并且还能踩在别的 RN 组件的头上。

在这个自定义组件中，提示文字我们没有写死在代码中，而是让它的父组件从外层传入，这样可以使它更通用一些。

10.8.2　使用自定义 Modal 组件

接下来，我们修改本书 10.1 中的例程 10-1-4，让自定义的 Modal 组件与原来例程中的导航组件协同工作。修改后的 index.××××.js 参见代码 10-8-2。

代码 10-8-2，index.XXXX.js：

```
import React, { Component } from 'react';
import {
    AppRegistry, View, Modal
} from 'react-native';
import { StackNavigator } from 'react-navigation';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
import WaitingModal from './WaitingModal';          //导入自定义 Modal
const SimpleApp = StackNavigator({
     Home: { screen: LoginLeaf },
     Wait: { screen: WaitingLeaf },
});
export default class LearnRN extends Component {
```

```
constructor(props) {
    super(props);
    this.state={
        showWaitingModal:false,                          //定义供自定义 Modal 使用的状态机变量
        modalPrompt:''
    };
    this.setWaitingModal = this.setWaitingModal.bind( this );
}
setWaitingModal( show, aPrompt) {                  //用来控制显示自定义 Modal 的回调函数
    this.setState({showWaitingModal:show, modalPrompt: aPrompt});
}
    render() {
    return (
        <View style={{flex:1}} >
            //下面的语句挂接了导航的通用界面属性
            <SimpleApp screenProps={{ setWaitingModal:this.setWaitingModal }} />
            <WaitingModal show={this.state.showWaitingModal}     //挂接自定义 Modal
                          prompt={this.state.modalPrompt}/>
        </View>
    );
  }
}
AppRegistry.registerComponent('LearnRN', () => LearnRN);
```

修改例程 10-1-2 为例程 10-8-3：

代码 10-8-3，LoginLeaf.js：

```
……
   jumpToWaiting() {       //定时器到时时，关闭自定义 Modal，并跳转页面
    this.props.screenProps.setWaitingModal( false, '' );
      this.props.navigation.navigate( 'Wait',
                                  {
                                     phoneNumber:this.state.inputedNum,
                                     userPW:this.state.inputedPW
                                  }
       );
}
showWaitingModalBeforeJump() {      //当用户按下确定时，启动一个定时器，并显示自定义 Modal
    this.props.screenProps.setWaitingModal( true, '请等待...' );
    this.aTimer = window.setTimeout( this.jumpToWaiting, 3000);

}
   userPressConfirm() {                          userPressConfirm 函数
      Alert.alert(
          '提示',
          '确定使用'+this.state.inputedNum+'号码登录吗？',
          [
              {text: '取消', onPress:(()=>{}), style: 'cancel' },
              {text: '确定', onPress: this.showWaitingModalBeforeJump }     //修改
```

```
            ]
        );
    }
```

……

原来的 WaitingLeaf 函数不需要任何修改。

执行修改后的代码，按下“确定”键后，手机屏幕显示如图 10-8-1 所示。

图 10-8-1　修改代码后的显示效果

10.8.3　Modal 组件与 Alert 组件

RN 开发者在开发中需要注意 Modal 组件与 Alert API 冲突的情况。当 APP 在显示一个 Modal 组件时，不要再调用 Alert API。因为 Modal 组件会屏蔽掉 Alert 弹出的询问框（或者确认框），让用户看不到弹出询问框，更别说做出选择，而我们的业务逻辑在等待用户选择的结果，造成的结果就是应用卡死。这是 RN 的一个 bug，也许后期会修复这个 bug。

10.8.4　总结

读者也许会想，费了半天劲，就实现了一个等待提示，直接在 LoginLeaf.js 中做一个多简单快捷。在本例程中，因为结构简单，确实可以直接在 LoginLeaf.js 中做。但在正式移动应用中，一个应用内有非常多个“界面跳转中”需要加入等待提示。在每个跳转前的界面中都加入跳转等待 UI 界面会让代码非常臃肿，同时也不方便控制。而本章中使用一个自定义 Modal 组件为任意界面提供跳转服务就显示出它简洁高效的优点了。

10.9 Navigator 导航组件

从 RN 0.44 版本开始，Navigator 组件已经被不建议使用。但因为很多以前的项目用到了这个组件。因此本章还是对此组件进行讨论。请读者自己决定是否学习这个组件。如果不需要阅读、维护 RN 旧版本的代码，可以不阅读本章的后续部分。

如果需要在 RN 0.44 版本后的 RN 项目中使用 Navigator 组件，需要先在窗口模式下，进入项目目录运行：

```
npm install --save react-native-deprecated-custom-components
```

然后在项目中这样导入 Navigator 组件：

```
import {
    Navigator
} from 'react-native-deprecated-custom-components';
```

这样就可以继续使用 Navigator 组件。

如果你初始化的是 0.44 版本之前的 RN 项目，则直接导入 Navigator 组件：

```
import { Navigator } from 'react-native';
```

10.9.1 Navigator 组件的回调函数

开发者可以通过指定 Navigator 组件的 configureScene 属性来定制场景切换时的动画效果。这是一个回调函数类型的属性，它接收场景切换的数据，然后返回开发者希望的场景切换动画效果。

configureScene 指定的回调函数被执行时，会收到一个导航路径参数，开发者可以根据导航路径中的各信息（由开发者在要求切换场景时提供）来决定场景切换使用何种效果。返回值可以是：PushFromRight、FloatFromRight、FloatFromLeft、FloatFromBottom、FloatFromBottomAndroid、FadeAndroid、HorizontalSwipeJump、VerticalUpSwipeJump、VerticalDownSwipeJump、PushFromLeft、HorizontalSwipeJumpFromLeft。configureScene 属性的具体使用与效果将在代码 10-4-1 中展示。

onDidFocus 属性用来指定一个回调函数。当导航组件导入初始场景后，或者每一个新的场景切换完成时，这个回调函数将被调用。它可以接收一个保存有新场景路径信息的参数。

现在 React Native 不建议开发者使用 onDidFocus 属性，而是鼓励开发者使用 navigationContext.addListener('didfocus', callback)事件监听器来实现相同的功能。

onWillFocus 属性用来指定一个回调函数。在导航组件准备进行场景切换前，这个回调函数将被调用。

现在 React Native 同样不建议开发者使用 onWillFocus 属性，而是鼓励开发者使用 navigationContext.addListener('willfocus', callback)事件监听器来实现相同的功能。

renderScene 属性用来指定一个回调函数。导航组件必须要提供这个属性。它用来为特定路径实现界面的渲染。当它被调用时，会提供一个 router 对象（导航路径）与一个 navigator 对象（导航器本身）。

10.9.2　其他属性

sceneStyle，style 类型的属性。如果开发者提供了这个属性，指定的样式将被应用到每一个切换的场景中。

initialRoute，类类型的属性。在 3.2 节中我们已经看到了它的用法。如果给 Navigator 组件提供了 initialRouteStack 属性，那么 initialRoute 必须是 initialRouteStack 中的一个元素；如果提供了 initialRouteStack 但没有提供 initialRoute，那么 initialRouteStack 中的最后一个元素将默认为 initialRoute。

initialRouteStack，类类型的属性，用来在 Navigator 组件初加载时提供导航路径。如果没有向 Navigator 组件提供 initialRoute 属性，就必须要提供 initialRouteStack 属性；如果提供了 initialRoute 但没有提供 initialRouteStack，那么 React Native 会生成一个只有 initialRoute 元素的数组作为 initialRouteStack。

navigationBar，该属性返回一个可以渲染的节点，这个节点可以用作所有界面的通用导航栏。我们将在 10.11 节中详细讨论它的用法。

如果当前组件是由父 Navigator 组件导航而产生的，则可以对其指定 navigator 属性，用来执行组件导航相关操作。

10.10　导航器

在本书 3.4 节中，我们已经讨论了导航器的 push、pop 和 replace 函数的使用。导航器用于各界面导航的函数有：

- push()函数，用来将一个界面放在导航栈最上方；
- pop()函数，用来弹出一个界面，让原来导航栈的下一个界面显示；
- replace()函数，用来将当前的界面替换为指定的界面；
- getCurrentRoutes()函数，用来得到当前的路径列表；
- jumpBack()函数，退回到上一个界面而不卸载当前界面；
- jumpForward()函数，沿界面路径向前跳一个界面而不卸载当前界面；
- jumpTo(route)函数，跳转到某个界面而不卸载任何界面；
- push(route)函数，导航组件在路径列表最前端添加一个新的界面，并马上跳转至这个界面；
- pop()函数，导航器退回一个界面并卸载原界面；
- replace(route)函数，导航器将当前界面用一个新的界面替代；
- replaceAtIndex(route, index)函数，使用一个新的界面替代路径列表中的第 index 个界面（下标从 0 开始），但不改变当前显示界面；

- replacePrevious(route)函数，将当前导航路径的上一个界面使用指定的界面替代；
- immediatelyResetRouteStack(routeStack)函数，使用给定的路径列表替换当前的路径列表；
- popToRoute(route)函数，导航器将退回到指定的界面，并在这个过程中将回退过的界面都一一卸载；
- popToTop()函数，导航器会回到界面路径列表中的第一个界面，并且卸载其他所有界面；
- popN()函数，接收一个数值型参数，导航器会退回多个界面并卸载退回的多个界面。

使用旧 Navigator 组件实现导航的例程参见例程 10-10-1。

代码 10-10-1：index.XXXX.js

```
import React, { Component } from 'react';
import {
   AppRegistry, Navigator, BackAndroid, Platform
} from 'react-native';
import LoginLeaf from './LoginLeaf';
import WaitingLeaf from './WaitingLeaf';
export default class NaviModule extends Component {
     constructor(props) {
         super(props);
         this.renderScene = this.renderScene.bind(this);
         this.configureScene = this.configureScene.bind(this);
         this.handleBackAndroid = this.handleBackAndroid.bind(this);
     }
     configureScene(route) {              //决定场景切换使用何种效果
         return Navigator.SceneConfigs.FadeAndroid;
     }
    renderScene(router, navigator) {          //为特定路径实现界面的渲染
       this.navigator = navigator;
       switch (router.name) {
           case "login":
              return < LoginLeaf navigator={navigator}/>;
           case "waiting":
              return <WaitingLeaf phoneNumber={router.phoneNumber}
                 userPW={router.userPW} navigator={navigator} />
       }
    }
    render() {
       return (
          <Navigator                          //初始化 Navigator 组件
             initialRoute={{name: 'login'}}
             configureScene={this.configureScene}
             renderScene={this.renderScene} />
       );
    }
    handleBackAndroid() {
       if ( this.navigator.getCurrentRoutes().length > 1) {
          this.navigator.pop();
          return true;
       }
       return false;
    }
```

```
    componentDidMount() {
        if ( Platform.OS === "android" ) BackAndroid.addEventListener(
                'hardwareBackPress', this.handleBackAndroid);
    }
    componentWillUnmount() {
        if ( Platform.OS === "android" ) BackAndroid.removeEventListener(
                'hardwareBackPress', this.handleBackAndroid);
    }
}
AppRegistry.registerComponent('LearnRN', () => NaviModule);
```

修改第三章实现的 LoginLeaf.js 文件的部分代码，如代码 10-10-2 所示。

代码 10-10-2：

```
……              //确定 Text 组件上面的代码保持不变
                    <Text style={styles.bigTextPrompt}
                        onPress={ ()=>this.userPressConfirm()}>
                        确　定
                    </Text>
                    <Text style={styles.bigTextPrompt}
                             onPress={()=>this.userPressAddressBook()}>
                        通讯录
                    </Text>
                </View>
            );
     }
     userPressConfirm () {
          this.props.navigator.push({
              phoneNumber: this.state.inputedNum,
              userPW: this.state.inputedPW,
              name: 'waiting',
          });
     }
     userPressAddressBook() { }
}               //这个反大括号标识了 LearnRN 类定义的结束
……              //下面的代码保持不变
```

WaitingLeaf.js 文件的内容如代码 10-10-3 所示。

代码 10-10-3：

```
import React, { Component } from 'react';
import {
    StyleSheet, Text, View
} from 'react-native';
export default class WaitingLeaf extends Component {
constructor(props) {
     super(props);
}
    render() {
       return (
          <View style={styles.container}>
              <Text style={styles.textPromptStyle} >
                  登录使用手机号：{this.props.phoneNumber}
```

```
                </Text>
                <Text style={styles.textPromptStyle}>
                    登录使用密码：{this.props.userPW}
                </Text>
                <Text style={styles.bigTextPrompt}
                       onPress={()=>this.onGobackPressed()}>
                    返  回
                </Text>
              </View>
        );
    }
    onGobackPressed() {
     this.props.navigator.pop();              //调用 pop 函数实现回退导航
    }
}
let styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    },
    textPromptStyle: {
        fontSize: 20
    },
    bigTextPrompt: {
        width: 300,
        backgroundColor: 'gray',
        color: 'white',
        textAlign: 'center',
        fontSize: 60
    }
});
```

代码修改后，运行效果与第三章运行的效果相似。细心的读者会注意到，从等待界面退回时，登录界面的输入还在，而第三章的例程将不会在。

10.11 NavigationBar

使用 Navigator 组件时，我们可以为 Navigator 组件管理的所有界面指定一个导航栏，这个导航栏将显示在指定的位置。如果没有指定这个位置，导航栏将默认显示在手机屏幕的最上方。

从本质上说，Navigator 组件的导航栏是三个显示区域，开发者可以在这三个显示区域中显示任何 React Native 组件，如文字、图片、按钮、输入框等。开发者需要能：

- 设置三个区域的大小；
- 控制在这三个区域中显示的内容；
- 如果三个区域中有列表或者输入框，要能够控制用户按下按钮或者输入文字后的业务逻辑。

当开发者决定使用 NavigationBar 来进行界面导航时，大部分应用界面的导航栏都具有相同的格式（同样大小的按钮、标题栏等），只是按钮的图片或者标题栏中的文字各有不同。如果各应用界面的导航栏有不同的格式，这些导航的元素就应当在各个界面中被单独实现，而不是使用 NavigationBar 来实现。

当大部分应用界面的导航栏有相同的格式时，相比 5.5.1 节中的自定义导航栏，使用 NavigationBar 实现导航栏能让程序代码更精练、集中，应用逻辑更好处理。

但在实际应用中，仍然有个别界面的导航栏有特殊需求，不能套用统一的导航栏格式，NavigationBar 也支持这种需求，只是实现代码会复杂些。

给 Navigator 组件指定导航栏的示例如下：

```
navigationBar={
          <Navigator.NavigationBar routeMapper={ NavigationBarRouteMapper } />
      }
```

其中的 Navigator.NavigationBar 是一个可显示的 React Native 组件，它必须有一个 routeMapper 属性。

开发者必须将一个对象指定给 routeMapper 属性。这个对象可以有三个成员变量：LeftButton、RightButton 和 Title。其中，Title 成员变量必须要有，其他两个视开发者需要来提供。这三个成员变量要求都是函数类型的，Navigator 组件渲染导航栏时，使用这三个函数的返回值渲染导航栏的对应区域。

每个函数可以接收 4 个参数。示例如下：

```
LeftButton( route, navigator, index, navState )
```

在三个成员函数返回的可渲染节点的样式中设置三个区域的大小。这三个函数返回的可渲染节点就是三个区域中显示的内容。

不同的页面需要控制这三个区域中显示不同的内容，开发者需要将不同页面待显示的不同内容（文字、图片）通过 route 传入这三个函数中，然后这三个函数从 route 的成员变量中取出传入的供显示的不同内容，最后渲染显示。

对按钮或输入框的处理，通常都需要调用父组件的函数，这就需要将这个父组件的函数以某种方式传入 routeMapper 属性中。开发者无法直接给 routeMapper 属性再传值，但可以放在 route 中，由 Navigator 组件在渲染时交给 routeMapper 属性。而 route 中的成员变量，都是由开发者提供的，并且对每个事件只能提供一个回调函数（准确地说，是最近一次提供的回调函数会覆盖上一次提供的回调函数）。这也正是 NavigationBar 组件目前使用不多的原因。我们将在例程中看到这一点。

10.12　导航例程

代码 10-12-1、10-12-2、10-12-3 综合展示了 configureScene 的各种效果，以及 NavigationBar

的使用方法。

代码 10-12-1，index.android.js 或者 index.ios.js

```
import React, { Component } from 'react';
import {
   AppRegistry, Navigator, PixelRatio
} from 'react-native';
let pixelRatio = PixelRatio.get();
import ImageDisplayer from './ImageDisplayer';
import NavigationBarRouteMapper from './NavigationBarRouteMapper';
export default class LearnRN extends Component {
   constructor(props) {
      super(props);
      this.touchtime=0;
      this.switchSceneStyle=Navigator.SceneConfigs.PushFromRight;
      this.initialRoute = {
                          UIIndex:0,
                          cbForLeftButton: this.callbackforLeftButton
                       };
      this.state = { textPrompt: '', UIIndex:0 };
      this.renderScene = this.renderScene.bind(this);
      this.changeStateVarBeforeRoute = this.changeStateVarBeforeRoute.bind(this);
      this.configureScene = this.configureScene.bind(this);
   }
   renderScene(router, navigator) {
      return <ImageDisplayer navigator={navigator}
                textForLeftButton={'新文字'}
                cbForLeftButton={this.callbackforLeftButton}
                UIIndex={this.state.UIIndex}
                textPrompt={this.state.textPrompt}
                callback={this.changeStateVarBeforeRoute}/>;
   }
   callbackforLeftButton(aNumber) {
      console.log('call back function received number:'+aNumber);
   }
   configureScene(route) {
      return this.switchSceneStyle;       //设置当前切换使用何种效果
   }
   render() {
      return ( <Navigator initialRoute={this.initialRoute}
                    configureScene = {this.state.switchSceneStyle}
                    navigationBar = {<Navigator.NavigationBar
                           routeMapper={ NavigationBarRouteMapper } />}
                    renderScene = {this.renderScene}>
             </Navigator>
      );
   }
   changeStateVarBeforeRoute() { //子组件用这个函数通知父组件准备切换场景
      this.touchtime++;              //点击次数成员变量加 1
      let textPrompt;
      switch (this.touchtime % 12) { //通过点击次数模 12 来达到每次切换效果的不同
         case 0:                     //按模 13 后的不同结果设置不同的值
            textPrompt='PushFromRight';
```

```
                this.switchSceneStyle = Navigator.SceneConfigs.PushFromRight;
                break;
            case 1:
                textPrompt='FloatFromRight';
                this.switchSceneStyle = Navigator.SceneConfigs.FloatFromRight;
                break;
            case 2:
                textPrompt='FloatFromLeft';
                this.switchSceneStyle = Navigator.SceneConfigs.FloatFromLeft;
                break;
            case 3:
                textPrompt='FloatFromBottom';
                this.switchSceneStyle = Navigator.SceneConfigs.FloatFromBottom;
                break;
            case 4:
                textPrompt='FloatFromBottomAndroid';
                this.switchSceneStyle = Navigator.SceneConfigs.FloatFromBottomAndroid;
                break;
            case 5:
                textPrompt='FadeAndroid';
                this.switchSceneStyle = Navigator.SceneConfigs.FadeAndroid;
                break;
            case 6:
                textPrompt='SwipeFromLeft';
                this.switchSceneStyle = Navigator.SceneConfigs.SwipeFromLeft;
                break;
            case 7:
                textPrompt='HorizontalSwipeJump';
                this.switchSceneStyle = Navigator.SceneConfigs.HorizontalSwipeJump;
                break;

            case 8:
                textPrompt='HorizontalSwipeJumpFromRight';
                this.switchSceneStyle = Navigator.SceneConfigs.HorizontalSwipeJumpFromRight;
                break;
            case 9:
                textPrompt='HorizontalSwipeJumpFromLeft';
                this.switchSceneStyle = Navigator.SceneConfigs.HorizontalSwipeJumpFromLeft;
                break;
            case 10:
                textPrompt='VerticalUpSwipeJump';
                this.switchSceneStyle = Navigator.SceneConfigs.VerticalUpSwipeJump;
                break;
            case 11:
                textPrompt='VerticalDownSwipeJump';
                this.switchSceneStyle = Navigator.SceneConfigs.VerticalDownSwipeJump;
        }
        this.setState({ textPrompt, UIIndex:this.state.UIIndex+1});
    }
}
AppRegistry.registerComponent('LearnRN', () => LearnRN);
```

NavigationBarRouteMapper.js 定义了例程中需要使用的导航栏。

代码 10-12-2，NavigationBarRouteMapper.js：

```
import React from 'react';
import { StyleSheet, Text } from 'react-native';
var NavigationBarRouteMapper = {       //定义设置导航栏的变量
    LeftButton(route, navigator, index, navState) {    //定义左侧区域如何显示
        let pString;
        if (route.textForLeftButton !== undefined) pString = route.textForLeftButton;
        else pString = '上一个';
        if (index > 0) {
            return ( <Text style = { styles.buttonStyle }
                        onPress = { () => {
                               try {
                                   route.cbForLeftButton(route.UIIndex);
                                   navigator.jumpBack();
                               } catch (error) {
                                   //用户退回到尽头后还企图退回，这里 catch 的 error 不需要处理
                               }
                        }
                    }>
                         {pString}
                </Text>
            );
        }
        else {
            return (
                <Text style={[styles.buttonStyle,{color:'red'}]}>
                       {pString}
                </Text>
            );
        }
    },
    Title(route, navigator, index, navState) {    //定义导航栏中部区域如何显示
        return (
            <Text style={styles.titleStyle}>
                   第{route.UIIndex}个界面
            </Text>
        )
    },
    RightButton(route, navigator, index, navState) {    //定义导航栏右侧区域如何显示
        if (navState.sceneConfigStack.length === index + 1) {
            console.log( ' no onpress for rigth button.');
            return (
                <Text style={[styles.buttonStyle,{color:'red'}]}>
                       下一个
                </Text>
            )
        }
        return ( <Text style = {styles.buttonStyle}
                     onPress = {() => {
                           if (navState.sceneConfigStack.length === index + 1) {
                               console.log('Can not jump forward.');
```

```
                        }
                        else navigator.jumpForward()
                        }
                    }>
                    下一个
                </Text>
        )
    }
};
var styles = StyleSheet.create({
    buttonStyle: {
        fontSize: 20,
        margin: 10,
        backgroundColor: 'grey',
        width: 70
    },
    titleStyle: {          //导航栏中央区域文字显示样式
        fontSize: 20,
        margin: 10,
        left:10,
        textAlign: 'center'
    }
});
export default NavigationBarRouteMapper;
```

ImageDisplayer.js 是本例程中真正的界面显示组件。读者可以看到，同一个组件也可以多次被推入导航路径中。

代码 10-12-3，ImageDisplayer.js

```
import React, { Component } from 'react';
import {
    AppRegistry, StyleSheet, Text, View, PixelRatio, TouchableOpacity, Image
} from 'react-native';
var Image1 = require('./image/image1.jpg');
var Image2 = require('./image/image2.jpg');
let pixelRatio = PixelRatio.get();
export default class ImageDisplayer extends Component {
    changeScene = ()=>{
    //因为场景切换发生在一个修改状态机变量的过程中，不能在切换过程中再次调用 setState
    //修改状态机变量，所以在这里通过一个回调函数在场景切换之前通知父组件修改状态机变量
        this.props.callback();
        console.log( 'before push.');
        this.props.navigator.push({          //切换场景
            name: "1",
            UIIndex:this.props.UIIndex + 1,   //这个 Index 被显示在导航栏上，说明是第几个界面
            cbForLeftButton:this.props.cbForLeftButton    //左导航栏按钮事件处理函数
        });
    }
    render() {
        return (    //ImageDisplayer 显示一张图片和一段文字，文字是上一次场景切换效果的名称
            <View style={styles.container}>
```

```
                <TouchableOpacity onPress={this.changeScene}>
                {/*注意问号三元运算符在 JSX 代码中的使用技巧，不熟悉三元运算符请阅读附录 A.8*/}
                    {
                        ( this.props.UIIndex%2 === 0 )?
                        (
                               <Image style={styles.imageStyle}
                               source={Image1}/>
                        ):(
                               <Image style={styles.imageStyle}
                               source={Image2}/>
                        )
                    }
                </TouchableOpacity>
                <Text style={styles.textStyle}>
                    {this.props.textPrompt}
                </Text>
            </View>
        );
    }
}
var styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
    },
    imageStyle: {       //显示图片用的样式
        width: 1080 / pixelRatio / 2,
        height: 1920 / pixelRatio / 2
    },
    textStyle: {       //显示文字用的样式
        fontSize: 30,
        top: 50,
        left: 5,
    }
});
```

代码运行结果是，用户每点击一次手机屏幕上的图片，就会使用一种切换模式进行页面切换，同时切换模式的名称也显示在手机上，方便读者理解每一种切换效果。在界面的上方有导航栏，导航栏中间的文字表明是第几个界面，数字会跟随用户的点击而改变。用户点击左侧按钮可以回到上一个界面（再点击再回，直到回到第 0 个界面），点击右侧按钮可以去下一个界面（如果它存在的话）。

如果需要使用 NavigationBar，则可以仔细研究此例程。

最后总结一下使用 NavigationBar 实现导航栏的特点：如果只是简单地在一系列预设好的界面路径中按序切换，那么使用 NavigationBar 是最合适不过了。如果导航栏功能复杂一些，建议开发者自定义导航栏。