React Navigation V5/V6 用法详解

马六甲的笔记 关注

0.38 2020.04.28 21:03:05 字数 7,873 阅读 4,975

React Navigation V5、V6版本使用方式基本相同，只是改了部分属性，参数。

安装

核心：提供底层 API，可在此基础上实现各种导航形式

```
yarn add react-native-screens react-native-safe-area-context @react-navigation/native
```

- react-native-screens: 用于原生层释放未展示的页面，改善 app 内存使用
- react-native-safe-area-context: 用于保证页面显示在安全区域（主要针对刘海屏）
- `@react-navigation/native`：为 React Navigation 的核心

修改 `MainActivity.java` 添加以下代码，否则 Android 下可能在某些情况下造成 App 崩溃，比如调整系统字体缩放/修改 APP 权限配置后再次返回 App，若没有以下修改，App 崩溃。即使设置了，还是有问题，无法保持最后查看的页面，App 会重新加载 js Bundle，返回到首页（是使用 `3.10.1` 版本在 debug 模式下发现的该问题，其他情况还需实际测试）

```
....
import android.os.Bundle;

public class MainActivity extends ReactActivity {


    ....
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(null);
    }
}
```

在 App.js 中添加以下代码，激活 `react-native-screens` 的原生端（最新版本默认已为激活状态，可省略该步骤）

```
import { enableScreens } from 'react-native-screens';
enableScreens();
```

Stack：相当于路由，如果不是仅需的 Tab 或 Drawer，必装

```
yarn add @react-native-masked-view/masked-view react-native-gesture-handler @react-navigation/stack
```

- @react-native-masked-view/masked-view: 用在头部导航栏中返回按钮的颜色设置
- react-native-gesture-handler: 用于支持手势切换页面
- `@react-navigation/stack`：Stack Navigator

安装完之后，在 js 入口文件，如 index.js 顶部添加 `import 'react-native-gesture-handler';`，少了这一句，可能会导致生产环境 app 出现闪退现象。

其他：官方提供的几种导航器，根据需要安装，也可以参考自行建构

- Drawer：`yarn add @react-navigation/drawer react-native-gesture-handler react-native-reanimated`
- Bottom Tabs：`yarn add @react-navigation/bottom-tabs`
- Material Bottom Tabs：`yarn add @react-navigation/material-bottom-tabs react-native-paper`

`react-native-vector-icons`

- Material Top Tabs：`yarn add @react-navigation/material-top-tabs react-native-tab-view react-native-pager-view`

安装所需导航器并安装相应依赖，有些依赖可能有重复，安装一次就行了，比如 `Drawer` 与 `Stack` 都依赖 `react-native-gesture-handler`，安装一次即可。

最后

`react-native-screens` 需要修改 Android 平台的 `MainActivity.java`，不再需要其他操作了

```
// 顶部添加
import android.os.Bundle;

// 主体 class 中添加
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(null);
}
```

对于 iOS，需要在项目根目录执行 `npx pod-install` 安装原生组件的依赖

使用

先看以下一段伪代码了解 `React Navigation` 的使用方法

```
import { NavigationContainer } from '@react-navigation/native';

import { createStackNavigator } from '@react-navigation/stack';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import { createMaterialTopTabNavigator } from '@react-navigation/material-top-tabs';

const Stack = createStackNavigator()
const Tab = createBottomTabNavigator();
const Top = createMaterialTopTabNavigator();

const First = () =>  (<Top.Navigator ...props>
      <Top.Screen  ...props/>
      <Top.Screen  ...props/>
</Top.Navigator>);

const Main = () => (<Tab.Navigator ...props>
    <Tab.Screen ...props component={First}/>
    <Tab.Screen ...props/>
</Tab.Navigator>);

const App = () => (<NavigationContainer ...props>
    <Stack.Navigator ...props>
        <Stack.Screen  ...props component={Main}/>
        <Stack.Screen  ...props/>
    </Stack.Navigator>
</NavigationContainer>);
```

1. 导航器总是使用 `<Nav.Navigator> <Nav.Screen/> </Nav.Navigator>` 格式组合页面，其中 `Nav` 可以使用官方提供的几个导航器 `stack`、`drawer`、`bottom-tabs`、`material-bottom-tabs`、`material-top-tabs`，当然也可参考自行建构。
2. 导航器本身可以作为另外一个导航器的 `Screen component`，即当作一个普通页面作为另外一个导航器的子页面。
3. 最终使用 `NavigationContainer` 包裹最顶层导航器。

   一般使用 `Stack` 作为顶层导航器（如上的例子），在 Tab 内打开页面会覆盖整个屏幕，退回后才能进行 Tab 切换。

   当然也可以使用 `Tab` 作为顶层导航器，切换页面时 TabBar 不会被覆盖，每个 Tab 都有独立的堆栈。

`React Navigation` 对 `NavigationContainer`、`Nav.Navigator`、`Nav.Screen` 提供了丰富的配置选项，做个简单介绍

一、NavigationContainer

该组件在 `@react-navigation/native` 中定义，一般情况，一个 APP 只有一个，参考 官方文档，该组件支持以下属性

1. theme

主题，该属性由 `@react-navigation/native` 缓存，但并未直接起作用，而是会下发到导航器，由导航器获取并加以利用，默认提供了 浅色 和 深色 两组属性 属性使用情况如下：

```
theme={
  dark: false,
  colors: {
    // 文字颜色：Header 标题 / BottomTab 未激活文字
    text: 'rgb(28, 28, 30)',

    // 激活颜色：BottomTab 激活文字 / iOS Header 返回上一页文字
    primary: 'rgb(99, 164, 252)',

    // 区块背景色：如 Header, TopTab, BottomTab 背景
    card: 'rgb(255, 255, 255)',

    // 边框颜色：如 Header, TopTab, BottomTab 边框
    border: 'rgb(216, 216, 216)',

    // 背景色：页面整体背景颜色
    background: 'rgb(242, 242, 242)',

    // 提醒色：BottomTab 角标背景
    notification: 'rgb(255, 59, 48)',
  },
}
```

2. ref

获取 `NavigationContainer` 实例，用于调用实例 api，可通过 console.log 打印可用 api

3. initialState

自定义传入变量，多用于 deepLink，该项暂未验证

4. onStateChange

导航状态变化的监听函数，可用于页面统计或其他操作

5. onReady

V6 版本新增，容器加载并渲染完毕时的回调，仅会触发一次。此时可安全的使用 `ref` 调用 API，也可在此时隐藏开屏页

6. linking

V6 版本新增，用于 deepLink

7. children

子组件（导航器 `Navigator` 组件），该项一般使用 jsx 直接插入，而不是通过 props 传递，比如上面的示例，`children` 为 `Stack.Navigator`

二、Nav.Navigator

导航器根组件，用于包裹导航器下的页面；通过阅读源码可知道所有导航器都支持4个属性：

- `@react-navigation/routers` 定义的 `initialRouteName`
- `@react-navigation/core` 定义的 `children` / `screenOptions` / `screenListeners`

1. initialRouteName

导航器默认要显示的 screen

2. children

导航器包裹的 screens，通常不会使用 Props 传递，而是在 jsx 中实现。

3. screenOptions

不同类型的导航器包裹的 screen 支持的属性是一样的，都会有一个 `options` 属性，此处设置的 `screenOptions` 与 `screen.options` 属性完全相同，作为所有 screen 的 `options` 默认值。

阅读 `@react-navigation/core` 源码和文档，这个参数的值可以是 `Object` 或 `Function({route, navigation}) => Object`，且未对 Object 字段做任何限制，而只是为导航器的实现提供了一个顶层 API，比如官方的两个实现支持不同的 options：

- `@react-navigation/stack` -> `Stack.Navigator` 的 screenOptions可用属性
- `@react-navigation/bottom-tabs` -> `Tab.Navigator` 的 screenOptions可用属性

```
// 直接设置为 Object
<Nav.Navigator
  screenOptions = {{
      title, header, headerShown, ........
  }}
>
  <Nav.Screen ...props />
</Nav.Navigator>


// 或通过函数返回，比如大部分 screen 所需属性相同，仅在函数内对特别的 screen 做处理
<Nav.Navigator
  screenOptions = { ({route, navigation}) => {
      return { title, header, headerShown, ........}
  }}
>
  <Nav.Screen ...props />
</Nav.Navigator>
```

在 V6 版本官方还提供了一个 `Nav.Group` 对具有相同属性的 screen 进行分组批量设置

```
<Nav.Navigator>

  <Nav.Group screenOptions={{}}>
    <Nav.Screen/>
    <Nav.Screen />
  </Nav.Group>

  <Nav.Group screenOptions={{}}>
    <Nav.Screen/>
    <Nav.Screen />
  </Nav.Group>

</Nav.Navigator>
```

4. screenListeners

监听导航器发送的事件消息，所有导航器共有的消息类型有 `focus` / `blur` / `beforeRemove` / `state` (文档)，不同导航器还有特有消息，如：

- `@react-navigation/stack` 的 events
- `@react-navigation/bottom-tabs` 的 events；

该属性的值与 `screenOptions` 有点类似，也可以指定为 `Object` 或 `Function`，如

```
// 直接设置为 Object
<Nav.Navigator
  screenListeners={{
    focus: () => {},
    state: (e) => { console.log('state changed', e.data);},
  }}
>
  <Nav.Screen ...props />
</Nav.Navigator>

// 或通过函数返回 要绑定的 监听函数
<Nav.Navigator
  listeners={({ navigation, route }) => ({
      return {
        focus: () => {},
```

```
        state: (e) => { console.log('state changed', e.data);},
      }
  }}
>
  <Nav.Screen ...props />
</Nav.Navigator>
```

以上四项为基础属性，适用于所有导航器，不同的导航器会在此基础中拓展额外的属性：

5. @react-navigation/stack

`detachInactiveScreens` / ~~`keyboardHandlingEnabled`~~ / ~~`mode`~~ / ~~`headerMode`~~ 属性（文档）

`Stack.Navigator` 的部分属性现在已经或即将转移到 `options` 中，具体支持哪些属性需以官方文档为准，下面介绍 `stack` 的 `options` 会提到目前的变化。

6. @react-navigation/bottom-tabs

`detachInactiveScreens` / `backBehavior`（继承自 `@react-navigation/routers`）/ `sceneContainerStyle` / `tabBar` / ~~`lazy`~~ / ~~`tabBarOptions`~~（V6版这两个属性移动到了 options 中配置）属性（文档）

三、Nav.Screen

导航器内的具体页面，该组件是在 `@react-navigation/core` 中实现的，与导航器类型无关，所有类型的导航器 screen 都支持且仅支持以下属性

1. name

页面名称，可用于导航跳转

2. options

与 `Nav.Navigator` 中的 `screenOptions` 相同，单独设置来覆盖 `screenOptions` 的配置，仅针对当前页面；同样的，可以设置为 `Object` 或通过 `Function` 返回；具体结构由 Screen 所属的 Navigator 类型决定。

3. listeners

与 `Nav.Navigator` 中的 `screenListeners` 相同，可以设置为 `Object` 或通过 `Function` 返回；仅对当前页面进行监听，不会覆盖 `Nav.Navigator` 中的设置，即二者都会被触发。

4. initialParams

传递给 `Screen` 组件的初始化 params，可在 screen 内获取，从而显示不同数据。

5. getId

属性值为 `Function(initialParams) => string`，返回一个唯一 ID，在多个页面有相同 `name` 属性时，可在使用 `navigate('ScreenName', params)` 时通过指定 `params.userId` 跳转到预期页面。

6. component / getComponent / children

`Screen` 绑定的组件，可通过 `component` 指定组件对象，`getComponent` 回调返回组件，或直接使用 `children` 定义组件；三者互斥，一般使用 `component` 属性来定义

```
<Nav.Screen component={Screen} />

<Nav.Screen getComponent={() => require('./Screen').default} />

<Nav.Screen>
  {(props) => <Screen {...props} />}
</Nav.Screen>
```

四、页面内

通过以上文档可以看出，页面的 `options` / `listeners` 都是由上层代码控制，不过 `React Navigation` 也提供了相关接口，可直接在 `Screen` 组件内部维护。

```
// 函数式组件：react navigation 会传递 navigation / route 两个参数
function Screen({ navigation, route }) {

  // 在页面显示之前设(重)置 options 值，相当于在 componentDidMount 阶段执行
  // useLayoutEffect 是阻塞同步的，即执行完此处之后，才会继续向下执行
  React.useLayoutEffect(() => {
     navigation.setOptions({
       title:'....'
     });
  }, [navigation]);

  // 绑定 listener，useEffect 是异步执行的，不会阻塞
  React.useEffect(() => {
     const unsubscribe = navigation.addListener('focus', () => {
       // do something
     });
     // 返回卸载监听的方法，以便在当前组件注射时取消监听
     return unsubscribe;
  }, [navigation]);

  // 页面内容
  return <ScreenContent />;
}



// Class 组件:navigation / route 以 props 参数传递
class Screen extends React.Component {
  componentDidMount() {
     const {navigation} = this.props;

     // render 后，未显示前，设置 options
     navigation.setOptions({
       title:'....'
     });

     // 绑定监听函数
     this._unsubscribe = navigation.addListener('focus', () => {
         // do something
     });
  }

  componentWillUnmount() {
      // 组件注销时，取消监听
      this._unsubscribe();
  }

  render() {
    return <ScreenContent />;
  }
}
```

在 react navigation V4 之前的版本，还提供了一个 `navigationOptions` 静态变量用于设置 options，V5 版本之后移除了该特性，如果要继续使用，可使用以下方法

```
class Home extends React.Component {
    static navigationOptions = {
       title:'....'
    };
}


function Screen() {
}
Screen.navigationOptions = {
    title:'....'
};

// V4 版本无需额外设置了，react navigation 默认已支持，对于 V5 之后版本
<Nav.Navigator>
  <Nav.Screen  component={Home}  options={Home.navigationOptions} />
  <Nav.Screen  component={Screen}  options={Screen.navigationOptions} />
</Nav.Navigator>
```

当然，也可以使用同样的方法自定义一个 static listener，但 react navigation 不推荐使用这种方式了，所以才会从内核中移除了该特性，因为这种方式有以下弊端（如果不在意以下弊端，仍然这么使用也是可以的）：

- 如果是高阶组件，静态属性需要额外的代码才能工作
- 无法使用 props 和 context 的能力，灵活性变差
- 无法自动进行类型检查，你需要手动给这个属性添加注解
- 在 Fast Refresh 下有问题，具体来说是修改它无法触发重新渲染

五、StackNavigator.Screen options

这是最常用的导航器，几乎是必备的，通常情况下，使用默认的配置即可取得不错的效果，不过 `Stack` 为了更加灵活，提供了很多 `options` 自定义属性，这里对其做一个整理（若无特殊说明，则代表为 V5/V6 都支持的属性）

最基本属性

- `title`：设置为 string，会作为 `stack` 导航器标题文字 `headerTitle` 的 fallback
- `keyboardHandlingEnabled`：切换页面时是否自动隐藏已打开的键盘，默认为 `true`（V6 新增，从 V5 版本的 `Stack.Navigator` 属性移动到了这里）
- `presentation`：页面模式，支持 `card`(默认) / `modal` / `transparentModal`（V6 新增，该属性相当于 V5 版本 `Stack.Navigator` 的 `mode` 属性转移到了这里），该值较为重要，会在下面单独说明。
- `detachPreviousScreen`：V6 新增，是否在页面切换后注销上一个页面以节省内存。默认情况下，若新页面为 modal 模式，该值为 `false`，否则为 `true`。即下一个页面未铺满全屏，当前页面仍会显示部分，就应该设置为 `false`。

  注意：该值只有在 `Stack.Navigator` 属性值 `detachInactiveScreens=true`(默认)时才生效，该值一般无需手动设置，会根据页面 `presentation` 等属性值自动设置合适的值。

与 Header 组件相关的属性

V6 版与 V5 版相比，将 Header 相关组件从 `stack` 包中提取出来组合为一个 `Elements` 包，该包提供了 `Header`、`HeaderBackground`、`HeaderTitle`、`HeaderBackButton`、`MissingIcon`、`PlatformPressable`、`ResourceSavingView` 组件和一些工具函数。`Header` 组件并未直接支持 left / right，`stack` 导航器使用该 `Header` 扩充了左侧组件，并额外支持一些其他属性。这样做的好处是，更利于自定义 Header，可利用 `Elements` 中组件自定义整个 Header，或仅自定义 Header 左(右)侧组件。以下说明中：

- 无特殊标记，则代表是 `Elements/Header` 组件直接支持的属性，V5/V6通用
- `+`：V6 版新增属性（仍是 `Elements/Header` 直接支持的属性）
- `*`：V5/V6通用（由 `stack` 扩充而非 `Elements/Header` 直接支持的属性）
- `*+`：V6 新增属性（由 `stack` 扩充而非 `Elements/Header` 直接支持的属性）

标题栏整体属性

- `headerStyle`：自定义标题栏样式，如果要改变高度，应直接使用 `height:Number` 设置，不要通过布局方式设置一个不确定的高度，该值对于页面切换时的 Header 动效非常重要。
- `headerTransparent`：标题栏是否透明，与 `headerStyle` 中直接设置 `backgroundColor` 的不同在于：这里设置透明，会使页面的 `marginTop` 为 0，此时需要定义 `headerBackground` 组件来遮挡。
- `headerBackground`：标题栏背景组件，配合 `headerTransparent` 使用的，可以用来实现诸如毛玻璃 Header 的效果。
- `headerStatusBarHeight`：手动设置 statusBar 高度，Header 组件会 paddingTop 这个值以保证在刘海屏机型也可以正常使用，默认会由系统自动获取。
- `headerPressColor` (V5版名称: ~~`headerPressColorAndroid`~~)：点击 Header 内按钮组件的水波纹颜色，仅对 Android 5 及以上
- + `headerPressOpacity`：点击 Header 内按钮组件的透明度，对 iOS 和 Android 5 以下
- `headerTintColor`：设置标题色调（该属性和 `headerPressColor` / `headerPressOpacity` 会传递给 Header 的各子组件使用，比如标题就会使用该属性设置文字颜色，按钮则使用到另两个属性）
- * `header`：自定义标题栏组件，定义为函数，返回一个 RN 组件；设置该属性，即不再使用默认 Header。
- * `headerShown`：是否显示标题栏
- *+ `headerMode`：标题栏显示模式，支持 "float"(iOS默认值)、"screen"(非 iOS 默认值)（该属性 V5

版是在 `Nav.Navigator` 属性中设置，V6 转移到了这里）

- ~~`safeAreaInsets`~~：安全区域设置（针对刘海屏机型），默认情况下会自动设置，但可以通过该属性通过 `{left, right, top, bottom}` 手动设置，自定义设置注意考虑横竖屏的情况。（该属性仅 V5 支持，V6 已移除，设置安全区域可参考 官方文档、用法说明）

标题组件

- `headerTitleAlign`：标题对齐方式，支持 `left` (Android 默认) / `center` (iOS 默认)
- `headerTitleAllowFontScaling`：标题文字是否随系统文字大小缩放
- `headerTitleStyle`：自定义标题文字的样式
- `headerTitleContainerStyle`：自定义标题文字所在 View 容器的样式
- `headerTitle`：标题，可直接设置文字，未设置则使用 `title` 属性；也可以设置为函数，返回一个组件，函数参数为 `{allowFontScaling, style, children}`，这三个参数是由上面属性结合而来。

左侧返回组件

- `headerLeft`：自定义 Header 左侧组件，props 会传递 options 设置
- `headerLeftContainerStyle`：自定义包裹 Header 左侧组件容器的样式
- * `headerBackImage`：返回键，设置为一个函数，返回“返回键”组件，函数参数为 `{tintColor:"标题颜色"}`
- * `headerBackTitle`：返回键右侧的文字
- * `headerTruncatedBackTitle`：返回键右侧文字过长，标题栏无法显示时的替代返回文字，默认："Back"
- * `headerBackAllowFontScaling`：返回文字是否随系统文字大小缩放
- * `headerBackTitleStyle`：自定义返回文字样式
- * `headerBackTitleVisible`：是否显示返回文字，Android 默认 false，iOS 默认 true
- * `headerBackAccessibilityLabel`：返回键的无障碍标签

右侧自定义组件

- `headerRight`：自定义 Header 右侧组件，指定为函数 或 RN组件，props 会传递 options 设置
- `headerRightContainerStyle`：自定义包裹 Header 右侧组件容器的样式

与页面组件相关的属性

- `cardStyle`：页面 Card 的样式
- `cardShadowEnabled`：是否在切换页面时显示页面边缘的阴影，默认为 `false`，启用阴影需要当前页面背景不能为透明 + `cardStyleInterpolator` 属性返回了 `shadowStyle` 样式，默认只有 `SlideFromRightIOS` 动效支持且仅支持 iOS，因为阴影组件 Animated.View 的默认样式是使用 shadowStyle 实现的，该类型 style 仅支持 iOS
- `cardOverlayEnabled`：是否在 Card 下方添加一个 overlay 组件（即在前一个 Card 的上方添加），iOS 默认为 `false`，`Android` 在 `presentation="transparentModal"` 为 `false`，否则为 `true`
- `cardOverlay`：函数，返回 `cardOverlayEnabled=true` 要覆盖的组件，该组件可用于页面切换时的效果设定，比如一个黑色的 view，切换过程中逐渐透明，甚至是毛玻璃组件，下方页面就呈现出一种逐渐显示的效果。

与页面切换手势相关的属性

- `gestureEnabled`：是否启用手势返回，iOS默认开启（不开启的话只能在页面上自定义返回按钮了），Android 默认是关闭的（Android 除了返回按钮，还有物理/虚拟返回键）
- `gestureDirection`：返回的手势滑动方向，支持以下值
  - `horizontal`：从左到右
  - `horizontal-inverted`：从右到左
  - `vertical`：从上到下
  - `vertical-inverted`：从下到上
- `gestureResponseDistance`：从边缘为起点，支持手势返回的距离，格式为 `{horizontal:50,`

`vertical:135}`；比如手势方向 `gestureDirection` 为 `horizontal`，那么只有在左边缘 50 以内的区域向右滑动才会响应。

- `gestureVelocityImpact`：触摸返回的手速设置，在手速低于该值时，滑动距离需大于滑动方向上尺寸的 50% 才会返回到上一页，否则弹回；高于所设置手速，即使滑动距离未达到50%，也会返回到上一页面；默认值为 0.3

与页面切换效果相关的属性

- `animationEnabled`：是否使用页面切换动效，在 Android 和 iOS 默认为 `true`，Web 为 `false`
- `animationTypeForReplace`：切换动画的方式：支持 "push"（默认）和 "pop"
- `transitionSpec`：切换页面的动效配置
- `cardStyleInterpolator`：切换页面时 Screen Card 的样式
- `headerStyleInterpolator`：切换页面时 Screen Header 的样式

切换动效

由以上属性可以看出，页面切换效果由以下属性共同构成：

- `transitionSpec`
- `cardStyleInterpolator`
- `headerStyleInterpolator`
- `gestureDirection`

前三个用于实现切换动效和样式，`gestureDirection` 用于在 `gestureEnabled=true`（iOS默认为 true）时配合动效，比如切换为上下展开收缩，`gestureDirection` 则应该支持上下滑动的手势。设置自定义动效可使用如下结构的代码：

```
const transition = {
     gestureDirection:"horizontal",
     transitionSpec: {},
     cardStyleInterpolator:() => {},
     headerStyleInterpolator:() => {},
}
<Stack.Navigator
    screenOptions={
       cardStyle:{},
       gestureEnabled:true,
       ...transition
    }
>
     <Stack.Screen />
</Stack.Navigator>
```

React Navigation 的设计初衷应该也在于此，所以已默认提供了几组属性，可以直接使用。

- `BottomSheetAndroid`：半透明到不透明，从底部滑入
- `FadeFromBottomAndroid`：半透明到不透明，从距离顶部一小段距离的位置滑至顶部
- `ModalFadeTransition`：无运动，仅半透明到不透明
- `ModalPresentationIOS`：无透明度变化，从底部滑倒接近顶部，以卡片形式弹窗，下方页面会缩小
- `ModalSlideFromBottomIOS`：无透明度变化，从底部滑倒顶部
- `RevealFromBottomAndroid`：无透明度变化，新页面从底部逐渐展开
- `ScaleFromCenterAndroid`：透明到不透明，从中心点爆炸式弹出
- `SlideFromRightIOS`：无透明度变化，从右侧滑入（只有该效果实现了 `cardShadowEnabled` 且仅支持 iOS）
- `ModalTransition`：iOS 为 `ModalPresentationIOS`，Android 为 `BottomSheetAndroid`
- `DefaultTransition`：iOS 为 `SlideFromRightIOS`，Android 为 `ScaleFromCenterAndroid`（API >= 29）、`RevealFromBottomAndroid`（API = 28）、`FadeFromBottomAndroid`（API < 28）

对于以上切换效果，有以下特点

- 对于 `headerMode="float"`，以上运动除 `ModalPresentationIOS` 会强制修改 `headerMode="screen"` 外，其他切换效果都是仅在页面内容区发生，而不是整个页面；页面 Header 会保持独立的运动，若前一个页面没有 Header，会从右侧滑入，否则会渐显式替换前一个 Header。
- iOS 默认启用了手势切换，所以 `IOS` 结尾的切换效果都没有透明度变化，适合手势切换，但也

同样可以用于 Android。但反过来则不行，非 `IOS` 结尾的切换效果由于有透明度变化，不适合用于手势切换。

使用方法：

```
import { TransitionPresets } from '@react-navigation/stack';

<Stack.Navigator
    screenOptions={
       cardStyle:{},
       ...TransitionPresets.SlideFromRightIOS,
    }
>
    <Stack.Screen />
</Stack.Navigator>
```

若对这些默认提供的效果都不满意，那只能自定义了。

1、`transitionSpec`

`transitionSpec` 需要提供 `open` / `close` 两个配置，每个配置需包含 `animation` / `config` 两个属性。其中 `config` 根据 `animation` 类型进行配置。可参考 timing 、spring

```
const config = {

  // 一般就两种
  animation: 'timing || spring',

  // 根据 animation 值提供配置
  config: {

    // animation="timing" 支持:
    duration:1000,
    easing: Easing.ease,

    // animation="spring" 支持:
    stiffness: 1000,
    damping: 500,
    mass: 3,
    overshootClamping: true,
    restDisplacementThreshold: 0.01,
    restSpeedThreshold: 0.01,

  },
};

const transitionSpec = {
    open: config,   // 新页面弹出时动效
    close: config,  // 新页面收回时动效，一般二者为同一个
};


// React Navigation 提供了几个默认的，可直接使用或作为参考
import { TransitionSpecs } from '@react-navigation/stack';
transitionSpec = TransitionSpecs.TransitionIOSSpec
transitionSpec = TransitionSpecs.FadeInFromBottomAndroidSpec
transitionSpec = TransitionSpecs.FadeOutToBottomAndroidSpec
transitionSpec = TransitionSpecs.RevealFromBottomAndroidSpec
```

2、`cardStyleInterpolator`

通过函数返回以下样式

- `containerStyle`：Card 所在 `Animated.View` 容器的样式
- `cardStyle`：Card 组件（`Animated.View`）样式
- `overlayStyle`：在 `cardOverlayEnabled=true` 时，由 `cardOverlay` 组件的样式
- `shadowStyle`：在 `cardShadowEnabled=true` 时，Card 边缘的 `Animated.View` 组件样式

```
cardStyleInterpolator = ({
    current, //当前页面值，如 current.progress 进度
    next,    //切换后的页面值，如 next.progress 进度
    index,   //card 在 stack 堆栈中的序号
    closing, //是关闭还是打开 1 or 0
    layouts  //布局尺寸 {screen}
}) => {
    return  {
```

```
            containerStyle:{},
            cardStyle:{},
            overlayStyle:{},
            shadowStyle:{},
        }
}

// React Navigation 提供了几个默认的，可直接使用或作为参考
import { CardStyleInterpolators } from '@react-navigation/stack';
cardStyleInterpolator = CardStyleInterpolators.forHorizontalIOS
cardStyleInterpolator = CardStyleInterpolators.forVerticalIOS
cardStyleInterpolator = CardStyleInterpolators.forModalPresentationIOS
cardStyleInterpolator = CardStyleInterpolators.forFadeFromBottomAndroid
cardStyleInterpolator = CardStyleInterpolators.forRevealFromBottomAndroid
```

3、`HeaderStyleInterpolators`

通过函数返回以下样式

- `leftLabelStyle`：Header 返回键旁边的"返回"文字所在 `Animated.Text` 的样式
- `leftButtonStyle`：Header 左侧返回键外层的 `Animated.View` 容器的样式
- `rightButtonStyle`：Header 右侧 `Animated.View` 容器的样式
- `titleStyle`：Header 标题所在 `Animated.View` 容器的样式
- `backgroundStyle`：Header 背景组件的样式

```
HeaderStyleInterpolators = ({
    current, //当前页面值，如 current.progress 进度
    next,    //切换后的页面值，如 next.progress 进度
    layouts  //布局尺寸：{screen, title, leftLabel}
}) => {
    return  {
          leftLabelStyle:{},
          leftButtonStyle:{},
          rightButtonStyle:{},
          titleStyle:{},
          backgroundStyle:{},
    }
}

// React Navigation 提供了几个默认的，可直接使用或作为参考
import { HeaderStyleInterpolators } from '@react-navigation/stack';
HeaderStyleInterpolators = HeaderStyleInterpolators.forUIKit
HeaderStyleInterpolators = HeaderStyleInterpolators.forFade
HeaderStyleInterpolators = HeaderStyleInterpolators.forStatic
```

以上三个属性可全部自定义，也可以部分自定义 + 部分使用 React Navigation 提供的预置，最后再添加一个 `gestureDirection` 属性就可构成一组自定义页面切换效果，非常的方便。

页面模式

以上便是 `Stack.Screen` 的 `options` 属性支持的所有配置，最后再对影响页面效果较大的 `headerMode`、`presentation` 配置稍作说明，`headerMode` 支持的两个值：

- `float`：此时页面 Header 与页面 Card 是分离的，有一个 Header 容器组件总是在顶部，所有页面的 Header 都在这个容器里，这种模式下，在切换页面时， Header 与 Card 可以独立执行各自的切换动效，比如模拟 iOS 原生效果。
- `screen`：每个页面的 Header 都在各自的 Card 顶部，即每个页面整体独立。切换页面时，是整个页面进行动效过渡。
- ~~`none`~~：该模式在 V6 版已移除，使用 `headerShown=false` 替代

`presentation` 配置更像一个快捷方式，修改该值，可能会自动设置 `cardOverlayEnabled`、`detachPreviousScreen`、`headerMode`、`gestureDirection`、`transitionSpec` 等属性的默认值用以配合效果，但如果这些值手动设置了值，将不会自动配置，而是使用手动设置的值，若设置为 `transparentModal`，默认 `cardStyle` 的背景将修改为透明。支持以下三个值：

- `card`：页面切换为模拟原生的效果，iOS 为 Header 渐隐渐显/Card左右显示隐藏，Android 为整体由下向上显示(默认值)
- `modal`：无论任何平台，都设置为页面整体由下向上滑动显示(与 `card` 模式下的 Android 由下向上的动效不同)
    - `headerMode` 自动设置为 `screen`

- 动效也会自动设置用以配合 modal 页面切换效果

- `transparentModal`：与 `modal` 类似
  - `headerMode` 自动设置为 `screen`
  - 屏幕背景会设置为透明，因此可以看到上一个页面
  - 自动设置 `detachPreviousScreen=false` 保持上一个页面的渲染状态
  - 设置上一个/当前页面的动效以配合效果

对于 `card`、`modal` 比较好理解，很容易适配到具体使用场景，`transparentModal` 值则更倾向于模拟弹窗效果，比如

```
<Stack.Navigator>
  <Stack.Screen name="Home" component={HomeStack} />
  <Stack.Screen
    name="Modal"
    component={ModalScreen}   // ModalScreen 为弹窗组件
    options={{
         presentation: 'transparentModal',
         headerShown: false,  // 不要显示 Header
         cardOverlayEnabled: true, // 弹窗下显示一个半透明 overlay 蒙层
    }}
  />
</Stack.Navigator>
```

如果需要对于 `ModalScreen` 自定义动画效果，可以借助 `useCardAnimation` 接口实现

```
import { Animated, View,  Text,  Pressable, Button, StyleSheet } from 'react-native';
import { useTheme } from '@react-navigation/native';
import { useCardAnimation } from '@react-navigation/stack';

function ModalScreen({ navigation }) {
  const { colors } = useTheme();
  const { current } = useCardAnimation();

  return (
    <View style={{ flex: 1,  alignItems: 'center', justifyContent: 'center'}}>
      <Pressable
        style={[StyleSheet.absoluteFill,  {backgroundColor: 'rgba(0, 0, 0, 0.5)' } ]}
        onPress={navigation.goBack}
      />
      <Animated.View
        style={{
          padding: 16,  width: '90%', maxWidth: 400, borderRadius: 3,
          backgroundColor: colors.card,
          transform: [
            {
              scale: current.progress.interpolate({
                inputRange: [0, 1],
                outputRange: [0.9, 1],
                extrapolate: 'clamp',
              }),
            },
          ],
        }}
      >
        <Text>
          Mise en place is a French term that literally means “put in place.” It
          also refers to a way cooks in professional kitchens and restaurants
          set up their work stations—first by gathering all ingredients for a
          recipes, partially preparing them (like measuring out and chopping),
          and setting them all near each other. Setting up mise en place before
          cooking is another top tip for home cooks, as it seriously helps with
          organization. It’ll pretty much guarantee you never forget to add an
          ingredient and save you time from running back and forth from the
          pantry ten times.
        </Text>
        <Button
          title="Okay" color={colors.primary} style={{ alignSelf: 'flex-end' }}
          onPress={navigation.goBack}
        />
      </Animated.View>
    </View>
  );
}
```

六、BottomTabsNavigator.Screen options

最基本属性

- `title`：设置为 string，会作为标题文字 `headerTitle` 的 fallback，底部 Tab 的 `tabBarLabel` 文字的 fallback
- `lazy`：选修卡页面是否为懒加载（即切换至页面时才渲染），默认为 `true`
- `unmountOnBlur`：页面失去焦点后是否自动卸载，若为 `true`，每次切换至页面都会重新加载，默认为 `false`

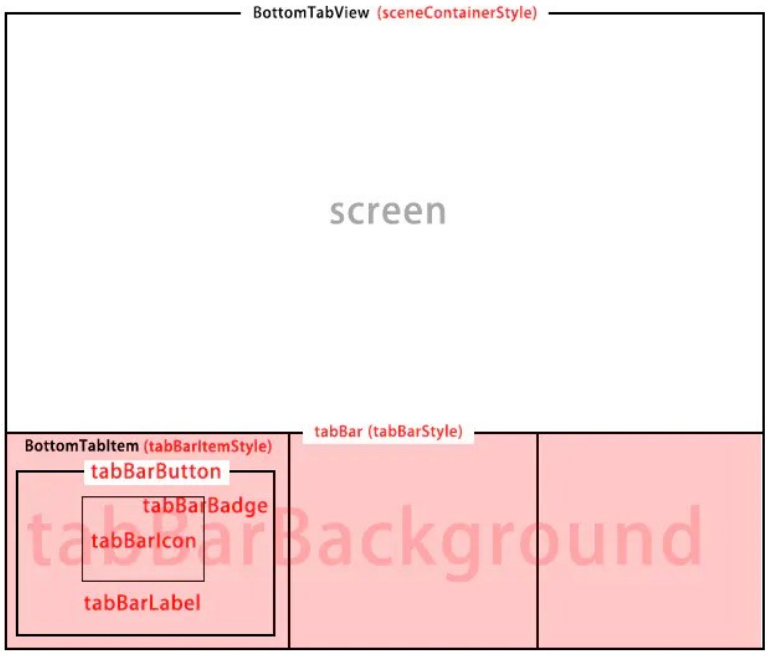
与 Header 组件相关的属性

参考 `StackNavigator options` 中与 Header 组件相关的属性，支持 `Elements/Header` 组件直接支持的所有属性，参考上面 `StackNavigator.Screen options` 所介绍的不含 `*` 的 Header 相关属性。除了这些属性外，额外扩展并支持以下属性

- *`header`：自定义标题栏组件，定义为函数，返回一个 RN 组件；设置该属性，即不再使用默认 Header。
- *`headerShown`：是否显示标题栏

与 TabBar 组件相关的属性

这些属性在 V5 版时，是设置在 `Navigator` 的 `tabBarOptions` 属性中，V6 版移动到了 `Screen` 的 `options` 中，为了更容易理解，可结合下图



BottonTab(红色为支持属性)

上图中除了 `sceneContainerStyle` / `tabBar` 是在 `BottomTabsNavigator.Navigator` 属性中设置的，其他都是在 `BottomTabsNavigator.Screen` 中设置的。默认的 `tabBar` 组件会利用下面要介绍的 `Screen options` 渲染为上图结构，如果自定义了 `tabBar` 组件，则可利用 `Screen options` 自行设计结构。既然提到了 `Navigator` 属性，顺带说下另外两个支持的属性：

- `detachInactiveScreens`：切换 Tab 后，是否回收未显示 Tab 页面内存，默认为 `true`
- `backBehavior`：在 Tab 页面按下物理(虚拟)返回键后的行为，支持以下值
  - `firstRoute`：跳转到第一个 Tab 页面(默认)
  - `initialRoute`：跳转到载入时的 Tab 页面(由 `initialRouteName` 指定的页面)
  - `order`：按照顺序依次跳转到前一个页面
  - `history`：按照浏览历史依次跳转到上一个访问的页面
  - `none`：什么都不做，通常会直接返回桌面

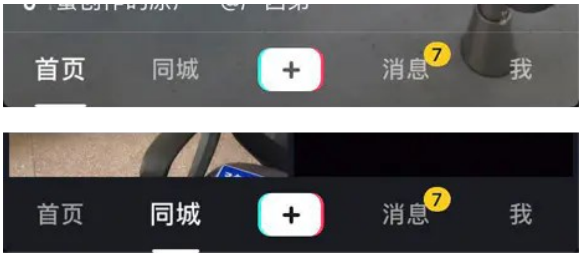
说完 `BottomTabsNavigator.Navigator` 的属性，下面说一下 `BottomTabsNavigator.Screen` 的 `options` 属性中与 `TabBar` 相关的属性，可结合上图进行理解。

- `tabBarBackground`：默认情况下，背景为 `tabBar` 的背景色，若指定了该组件，`tabBar` 背景色会

自动设置为透明，`tabBarBackground` 组件在 Z 轴上位于 `tabBar` 的下面，可以设置一些个性的 UI 效果，比如渐变色、图片、毛玻璃等。

- `tabBarStyle`：TabBar 整体容器的样式
- `tabBarShowLabel`：是否显示 TabBar 的文字
- `tabBarLabelPosition`：TabBar 文字显示的位置，默认会根据设备类型自动显示
  - `below-icon`：文字显示在图标下面（手机默认）
  - `beside-icon`：文字显示在图标右边（平板默认）
- `tabBarInactiveTintColor`：默认状态下文字颜色
- `tabBarActiveTintColor`：激活状态下文字颜色
- `tabBarInactiveBackgroundColor`：默认状态下背景颜色
- `tabBarActiveBackgroundColor`：激活状态下背景颜色
- `tabBarHideOnKeyboard`：在键盘展开时隐藏 TabBar，默认 `false`

对于 `StackNavigator`，每个页面都是独立的，所有属性都是对于所属页面而言的。而 `BottomTabsNavigator` 则不然，多个页面公用同一个 TabBar，以上是共用属性，每个处于激活的页面设置的属性都会影响整个 TabBar，比如下面这种效果



TabBar 效果

上面为共用属性，而以下属性则是每个页面的私有属性，即仅会影响所属页面的 TabItem。

- `tabBarItemStyle`：TabBar Item 容器的样式
- `tabBarButton`：设置 `tabBarLabel`，`tabBarIcon`，`tabBarBadge` 的容器组件，通常无需设置，可参考默认的 button
- `tabBarIcon`：TabBar 图标组件（会收到 `{ focused: boolean, color: string, size: number }` 参数）
- `tabBarIconStyle`：TabBar 图标样式
- `tabBarBadge`：TabBar 角标，可以是 `String` 或 `Number`
- `tabBarBadgeStyle`：TabBar 角标样式
- `tabBarLabel`：TabBar 要显示的文字（不设置会使用 `title` 属性），可以设置为 `String` 或返回 React 组件的函数（函数会收到 `{ focused: boolean, color: string }` 参数）
- `tabBarLabelStyle`：TabBar 文字的样式
- `tabBarAllowFontScaling`：TabBar 文字是否随系统字体大小缩放
- `tabBarAccessibilityLabel`：无障碍标签
- `tabBarTestID`：用于本地测试的 ID

结合【四、页面内】章节，可以使用 `React.useLayoutEffect` 在页面内设置 `options`，仅适合 TabBar 的共用属性，而不适合 TabBar 私有属性，毕竟不能让用户激活了 Tab 页面后，才能看到诸如 `tabBarBadge` / `tabBarLabel` 信息，这一点需要注意。

七、接口

1. 页面组件会收到 navigation 和 route 两个参数。

`navigation` 提供相关操作API，根据 Screen 组件所在导航器的不同，API 也会有所不同。

通用API，所有类型导航器都可使用

- `navigate`：跳转到指定页面
- `goBack`：关闭当前页面返回到上一页
- `reset`：重置导航器状态

- `setParams`：更新当前页面的 `route.params` 参数
- `setOptions`：更新当前页面的 `options` 选项配置
- `isFocused`：检测当前页面是否处于活动状态
- `dispatch`：发送 Action 给导航器，可参考 文档
- `getParent`：若当前导航器嵌套在另外一个导航器中，返回上级导航器，否则返回 `undefined`
- `getState`：获取导航器当前的状态，一般用不到，少数情况下可能用得到

stack 导航器独有

- `replace`：替换当前页面为指定页
- `push`：添加一个新页面到堆栈
- `pop`：从堆栈弹出当页面
- `popToTop`：返回到堆栈的起始页

tab 导航器独有

- `jumpTo`：跳转到 Tab 内的指定页面

`route` 属性提供当前页面的相关信息

- `key`：页面唯一值，通常为自动生成
- `name`：所定义的页面名称
- `path`：页面路径，通过 Link 打开的页面才会有这个属性
- `params`：页面导航时传递的参数

2. 非页面组件如何使用 navigation 和 route 属性

通常可以在页面内调用组件时，将 navigation 和 route 以 props 的方式传递给子组件，但这样对于嵌套较深的组件使用起来非常痛苦，另外子组件也要依赖父组件正确传递，`React Navigation` 提供了另外一种方法：

```
import * as React from 'react';
import { View, Text, Button } from 'react-native';
import { useNavigation, useRoute } from '@react-navigation/native';

function MyConmpoent() {
  const navigation = useNavigation();
  const route = useRoute();

  return <View>
     <Text>{route.params.caption}</Text>
     <Button
      title="Back"
      onPress={() => {
        navigation.goBack();
      }}
    />
   </View>;
}


// 对于 class 组件
class MyConmpoent extends React.Component {
  render() {
    const { navigation, route  } = this.props;
    return <View>
       <Text>{route.params.caption}</Text>
       <Button
         title="Back"
         onPress={() => {
           navigation.goBack();
        }}
       />
     </View>;
  }
}

// Wrap and export
export default function(props) {
   props.navigation = useNavigation();
   props.route = useRoute();
   return <MyConmpoent {...props}  />;
}
```

3. 其他可用的 Hook API

```
import * as React from 'react';
import { View, Text, Button } from 'react-native';

// 可用 Hook API
import {
  useNavigation,
  useIsFocused,
  useLinkTo,
  useLinkProps,
  useLinkBuilder,
  useScrollToTop,
  useTheme
} from '@react-navigation/native';

// function 组件
function MyConmpoent() {

  const theme = useTheme();
  const isFocused = useIsFocused();
  const state = useNavigationState(state => state);

  const linkTo = useLinkTo();
  const { onPress, ...props } = useLinkProps({ to, action });
  const buildLink = useLinkBuilder();

  const scrollRef = React.useRef(null);
  useScrollToTop(scrollRef);

  // code
}



// 对于 class 组件
class MyConmpoent extends React.Component {
  render() {
    const {theme, isFocused, state, linkTo, onPress, buildLink, scrollRef} = this.props;
    // code
  }
}

// Wrap and export
export default function(props) {
   props.theme = useTheme();
   props.isFocused = useIsFocused();
   props.state = useNavigationState(state => state);

   props.linkTo = useLinkTo();
   props.onPress = useLinkProps({ to, action }).onPress;
   props.buildLink = useLinkBuilder();

   const scrollRef = React.useRef(null);
   useScrollToTop(scrollRef);

   return <MyConmpoent {...props}  />;
}
```

将这些API分为三类，第一类有 `useTheme`，`isFocused`，`useNavigationState`，这三个使用 get 型 API 是可以直接获取的，比如 `navigation.isFocused()`，但在 render() 界面时依赖相关变量的话，这些 API Hook 就比较有用了，当这些相关变量发生变化，界面会自动更新。

第二类为 `useLinkTo`，`useLinkProps`，`useLinkBuilder`，这三个都与 `Link` 功能有关，以伪代码做个说明：

```
import { Link, useLinkTo, useLinkProps, useLinkBuilder} from '@react-navigation/native';



// Link 组件使用 Text 模拟，类似于 Html 的 a 标签，接受 to / action 两个参数
// to 指定目标页面， action 与 navigate.dispatch 接口参数同，不指定为 navigate action
function Componet() {
  return (
    <Link
      to={{ screen: 'Profile', params: { id: 'jane' } }}
      action={StackActions.replace('Profile', { id: 'jane' })}
    > Go </Link>
  );
}

```

```
// 可以使用 useLinkBuilder 生成 Link 组件的 to 参数
function Componet({ route }) {
  const buildLink = useLinkBuilder();
  return (
    <Link
      to={buildLink(route.name, route.params)}
      action={StackActions.replace('Profile', { id: 'jane' })}
    > Go </Link>
  );
}


// Link 组件使用 Text 模拟，可以使用 useLinkProps 自定义其他组件模拟
function LinkButton({ route }) {
  const { onPress, ...props } = useLinkProps({ to, action });
  return (
    <Button onPress={onPress}> Go </Button>
  );
}
// 这样就可以和使用 Link 一样的方式，使用自己创建的 'Link' 组件了
<LinkButton to={} action={}/>



// useLinkTo 与以上不同，更类似于 navigation.navigate ，用于跳转到指定页面
// 但提供的参数不同，这里需要提供 Deep Link 所设置的页面 path
function Screen() {
  const linkTo = useLinkTo();
  return (
    <Button onPress={() => linkTo('/profile/jane')}>
      Go to Jane's profile
    </Button>
  );
}
```

第三类是 `useScrollToTop` Hook，该 API 的作用是为了模拟原生 Bottom Tab 的效果，如果 Tab 页面是可滚动的(比如 `ScrollView`，`FlatList`)，在页面已处于激活状态的情况下，点击底部 Tab 图标，页面滚动到最顶部。

```
import * as React from 'react';
import { ScrollView } from 'react-native';
import { useScrollToTop } from '@react-navigation/native';

function Screen() {
  const ref = React.useRef(null);
  useScrollToTop(ref);

 // 如果希望点击底部 Tab 图标不是滚动到最顶部，可以这样来指定一个 offset 值
 // useScrollToTop(React.useRef({
 //    scrollToTop: () => ref.current?.scrollToOffset({ offset: -100 }),
 // }));

  return <ScrollView ref={ref}>{/* content */}</ScrollView>;
}
```

最后，除了以上 Hook API，`React Navigation` 还提供了一个 `useFocusEffect` Hook，该 API 与以上都不同，所以放到最后单独说一下。以上 API 都是返回值式的 Hook，该 Hook 则更类似于添加一个 listener 监听：

```
import { useFocusEffect } from '@react-navigation/native';

function Profile({ userId }) {
  const [user, setUser] = React.useState(null);

  // useFocusEffect 与 React.useEffect 类似，不同之处在于只会在页面激活时触发
  // 可使用 React.useCallback 包裹回调，这样回调只会在首次激活或依赖项发生变化才触发
  // 否则每次页面激活都会被触发
  useFocusEffect(
    React.useCallback(() => {
      const unsubscribe = API.subscribe(userId, user => setUser(user));
      return () => unsubscribe();
    }, [userId])
  );


  // 一般远程请求都是异步的，所以务必只请求一次
  //(因为该回调不一定仅触发一次，可能造成竞争请求)
  // 如果请求 API 未提供取消机制，需自行处理，如：
  useFocusEffect(
```

```
    React.useCallback(() => {

      let isActive = true;
      const fetchUser = async () => {
         try {
             const user = await API.fetch({ userId });
             if (isActive) {
                setUser(user);
             }
          } catch (e) {
            // Handle error
          }
      };
      fetchUser();
      return () => {
        isActive = false;
      };

    }, [userId])

  );

  return <ProfileContent user={user} />;
}




// 对于 class 组件，需采用类似于 StatusBar 的方法
function FetchUserData() {
  useFocusEffect(
       ....
  );
  return null;
}

class Profile extends React.Component {
  _handleUpdate = user => {
    // Do something with user object
  };
  render() {
    return (
      <>
        <FetchUserData />
        {/* 其他组件 */}
      </>
    );
  }
}
```