React Native Day5

课前回顾

- React Navigation介绍
- React Navigation概念与属性介绍
- 核心导航器的学习与使用

课堂目标

- 掌握react navigation 导航框架设计
- 了解redux在RN项目（使用react navigation）中的集成方式
- 掌握Fetch网络编程

App导航框架设计

仿主流APP设计一个导航框架

欢迎页面设计

```
import React, { Component } from "react";
import { Platform, StyleSheet, Text, View } from "react-native";
export default class WelcomePage extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to WelcomePage!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
```

```
  }
});
```

App主页设计

```
import React, { Component } from "react";
import { Platform, StyleSheet, Text, View } from "react-native";

export default class HomePage extends Component {
  constructor(props) {
    super(props);
    console.disableYellowBox = true;
  }

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to HomePage!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});
```

详情页设计

```
import React, { Component } from "react";
import { Platform, StyleSheet, Text, View } from "react-native";

export default class DetailPage extends Component {
```

```
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to DetailPage!</Text>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});
```

安装 react navigation 与第三方图标库 react-native-vector-icons

```
yarn add react-navigation
# or with npm
# npm install --save react-navigation


yarn add react-native-gesture-handler
# or with npm
# npm install --save react-native-gesture-handler


react-native link react-native-gesture-handler


yarn add react-native-vector-icons


react-native link react-native-vector-icons

###记得关闭模拟器，服务器，重新启动项目
```

设计欢迎页进入主页导航

```
##### AppNavigator.js

import {
  createStackNavigator,
  createAppContainer,
  createSwitchNavigator
} from "react-navigation";

import HomePage from "../Pages/HomePage";
import WelcomePage from "../Pages/WelcomePage";
import DetailPage from "../Pages/DetailPage";


//定义欢迎导航
const AppInitNavigator = createStackNavigator({
  WelcomePage: {
    screen: WelcomePage,
    navigationOptions: {
      header: null
    }
  }
});
//定义主页导航
const AppMainNavigator = createStackNavigator({
  HomePage: {
    screen: HomePage,
    navigationOptions: {
      header: null
    }
  },
  DetailPage: {
    screen: DetailPage
  }
});

export default createAppContainer(
  createSwitchNavigator({
    Init: AppInitNavigator,
    Main: AppMainNavigator
  })
);
```

App入口引用导航

```
import App from "./js/Navigator/AppNavigator";
```

欢迎页面5秒后进入主页

```
componentDidMount() {
    this.timer = setTimeout(() => {
      const { navigation } = this.props;
      navigation.navigate("Main");
    }, 1000);
  }
  componentWillUnmount() {
    this.timer && clearTimeout(this.timer);
  }
```

设计一个转场工具类 NavigationUtil.js

```
export default class NavigationUtil {
  //跳转到指定页面
  static goPage(props, page) {
    const navigation = NavigationUtil.navigation;
    navigation.navigate(page, {
      ...props
    });
  }
  //go Back
  static resetGoBack(props) {
    const { navigation } = props;
    navigation.goBack();
  }
  //回到主页
  static resetToHomePage(params) {
    const { navigation } = params;
    navigation.navigate("Main");
  }
}
```

欢迎页改造

```
import NavigationUtil from "../Navigator/navigationUtil";

componentDidMount() {
    this.timer = setTimeout(() => {
      navigationUtil.resetToHomePage({
        navigation: this.props.navigation
      });
    }, 1000);
  }
```

主页设计底部导航

```
import React, { Component } from "react";
import { Platform, StyleSheet, Text, View } from "react-native";
import {
  createAppContainer,
  createBottomTabNavigator,
} from "react-navigation";

import IndexPage from "./IndexPage";
import MyPage from "./MyPage";
import VideoPage from "./VideoPage";
import FontAwesome from "react-native-vector-icons/FontAwesome";

const TABS = {
  IndexPage: {
    screen: IndexPage,
    navigationOptions: {
      tabBarLabel: "首页",
      tabBarIcon: ({ tintColor, focused }) => (
        <FontAwesome name={"home"} size={26} style={{ color: tintColor }} />
      )
    }
  },
  VideoPage: {
    screen: VideoPage,
    navigationOptions: {
      tabBarLabel: "视频",
      tabBarIcon: ({ tintColor, focused }) => (
        <FontAwesome
          name={"video-camera"}
          size={26}
          style={{ color: tintColor }}
        />
      )
    }
```

```
  },
  MyPage: {
    screen: MyPage,
    navigationOptions: {
      tabBarLabel: "我的",
      tabBarIcon: ({ tintColor, focused }) => (
        <FontAwesome name={"user"} size={26} style={{ color: tintColor }} />
      )
    }
  }
};
export default class HomePage extends Component {
  constructor(props) {
    super(props);
    console.disableYellowBox = true;
  }
  _TabNavigator() {
    return createAppContainer(createBottomTabNavigator(TABS));
  }
  render() {
    const Tabs = this._TabNavigator();
    return <Tabs />;
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});
```

Index页面顶部导航设计

```
import React, { Component } from "react";
import { Button, Platform, StyleSheet, Text, View } from "react-native";
import {
  createAppContainer,
  createMaterialTopTabNavigator
```

```
} from "react-navigation";
import navigationUtil from "../Navigator/navigationUtil";
export default class IndexPage extends Component {
  constructor(props) {
    super(props);
    this.tabNames = [
      "ios",
      "android",
      "nodeJs",
      "Vue",
      "React",
      "React Native"
    ];
  }
  _genTabs() {
    const tabs = {};
    this.tabNames.forEach((item, index) => {
      tabs[`tab${index}`] = {
        screen: props => <IndexTab {...props} tabName={item} />,
        navigationOptions: {
          title: item
        }
      };
    });
    return tabs;
  }
  render() {
    const TabNavigator = createAppContainer(
      createMaterialTopTabNavigator(this._genTabs(), {
        tabBarOptions: {
          tabStyle: {},
          upperCaseLabel: false,
          scrollEnabled: true,
          style: {
            //选项卡背景色
            backgroundColor: "red"
          },
          indicatorStyle: {
            //指示器的样式
            height: 2,
            backgroundColor: "#fff"
          },
          labelStyle: {
            //文字的样式
            fontSize: 16,
            marginTop: 6,
            marginBottom: 6
          }
        }
```

```
      })
      //   createMaterialTopTabNavigator({
      //     IndexTab1: {
      //       screen: IndexTab,
      //       navigationOptions: {
      //         title: "Tab1"
      //       }
      //     },
      //     IndexTab2: {
      //       screen: IndexTab,
      //       navigationOptions: {
      //         title: "Tab2"
      //       }
      //     }
      //   })
    );
    return (
      <View style={{ flex: 1, marginTop: 30 }}>
        <TabNavigator />
      </View>
    );
  }
}

class IndexTab extends Component {
  render() {
    const { tabName } = this.props;
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to {tabName}</Text>
        <Button
          title={"go to DetailPage"}
          onPress={() => {
            navigationUtil.goPage(this.props, "DetailPage");
          }}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
```

```
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});
```

Redux 与 React Navigation结合集成

Redux + React Navigation有点复杂 因为Redux是自顶向下管理一套状态，React Navigation也是自顶向下管理一套状态甚至页面，这俩融合起来就有点困难了

Redux 不是必须的

要想撮合Redux和React Navigation(为了方便描述，后面就直接称呼为导航)，就得先各自了解

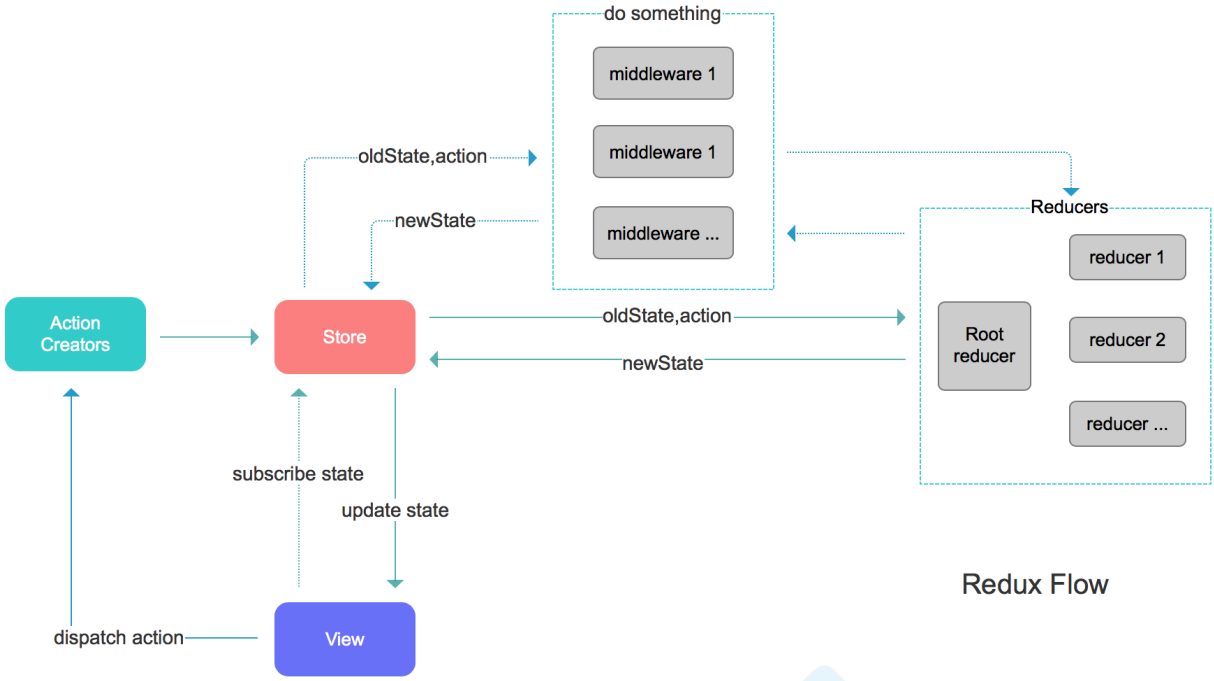
在使用 React Navigation 的项目中，想要集成 redux 就必须要引入 `react-navigation-redux-helpers` 这个库。

Redux

React或者说ReactNative原生使用state和props管理UI状态，但是俩属性非常琐碎，稍微复杂一点的UI就没法应付，所以Redux应运而生，但是Redux又很乱，我觉得乱主要是以下原因：

- 需要声明的东西多
- 需要声明的地方多
- API方法的命名很莫名其妙

流程 ， 概念



- 用户（操作View）发出Action，发出方式就用到了dispatch方法；
- 然后，Store自动调用Reducer，并且传入两个参数(当前State和收到的Action)，Reducer会返回新的State，如果有Middleware，Store会将当前State和收到的Action传递给Middleware，Middleware会调用Reducer 然后返回新的State；
- State一旦有变化，Store就会调用监听函数，来更新View；

梳理

- redux是一个存储状态，响应事件动作(action)的地方，所以定义redux实现的叫store
- store有一个初始状态(default state)，还有响应某个动作(action)的处理器(reducer)
- 然后UI视图将这个store及其状态(state)和方法(action)注册到视图组件的props，这样就可以在组件中取到这些状态和方法了。
- 当用户点击了某个操作等，会从props中拿到action并调用它，他会向store发送(dispatch)这个action的内容，
- 如果store中有中间件，会先逐个调用中间件来完成预处理
- 然后再调用各个reducer，来完成状态的改变。
- 状态改变以后，因为状态绑定了UI组件的props，所以react会自动刷新UI。

概念

reducer：名字起源于Array的reduce方法，作者估计向表达的是遍历的意义，但是这个名字实在是诡异，所以我给他起名叫做**处理器**，或者叫**事件触发器**，作用就是UI发来action以后，它根据action的类型，对状态进行修改。他是action的消费者，他是个函数(或者专业点叫纯函数)，但是他有个缺点，就是需要立即返回，如果是网络请求等异步操作，他就没法胜任了。

中间件：中间件的作用就是完成异步请求，或者完成其他一些需要封装起来的预处理，比如redux-logger，就是把action前后的状态打印出来的中间件，本质也是个函数，但是结构很诡异，诡异程度类似于C语言中的3级指针，这个指针还尼玛是函数指针。不过这种诡异我们不需要操心，只需要填写内容

第一步：安装redux,react-redux,react-navigation-redux-helpers

```
yarn add redux
yarn add react-redux //因为redux其实是可以独立运行的js项目，所以把他使用在react项目中，还需要使用react-redux
yarn add react-navigation-redux-helpers//在使用 React Navigation 的项目中，想要集成 redux 就必须要引入 react-navigation-redux-helpers 这个库
```

第二步：配置Navigation

- 引入redux和react-navigation-redux-helpers

```
import { connect } from "react-redux";

import {
  createReactNavigationReduxMiddleware,
  createReduxContainer
} from "react-navigation-redux-helpers";
```

- 使用createReduxContainer方法，将RootNavigator封装成高阶组件AppWithNavigationState，这个高阶组件完成了navigation prop的替换，改成了使用redux里的navigation

```
// 修改AppNavitor 为 RootNavigator 并不再默认导出
export const RootNavigator = createAppContainer(
  createSwitchNavigator({
    Init: AppInitNavigator,
    Main: AppMainNavigator
  })
);

const AppWithNavigationState = createReduxContainer(RootNavigator, "root");
```

- **创建导航中间件：createReduxContainer把导航状态放到props里只是能被各个组件访问到，但是React Navigation还不能识别，所以还需要最后一步——创建一个中间件，把需要导航的组件与导航reducer连接起来**

```
export const middleware = createReactNavigationReduxMiddleware(
  state => state.nav,
  "root"
);
```

- **然后使用Redux的connect函数再封装一个高阶组件，默认导出**

```
//State到Props的映射关系
const mapStateToProps = state => {
  return {
    state: state.nav
  };
};

//使用Redux的connect函数再封装一个高阶组件,连接 React 组件与 Redux store
export default connect(mapStateToProps)(AppWithNavigationState);
```

- 完整代码

```
import {
  createStackNavigator,
  createAppContainer,
  createSwitchNavigator
} from "react-navigation";

import HomePage from "../Pages/HomePage";
import WelcomePage from "../Pages/WelcomePage";
import DetailPage from "../Pages/DetailPage";

//引入redux
import { connect } from "react-redux";

import {
  createReactNavigationReduxMiddleware,
  // reduxifyNavigator,react-navigation-redux-helpers3.0变更,reduxifyNavigator被改名为createReduxContainer
  createReduxContainer
} from "react-navigation-redux-helpers";

export const rootCom = "Init"; //设置根路由
```

```
const AppInitNavigator = createStackNavigator({
  WelcomePage: {
    screen: WelcomePage,
    navigationOptions: {
      header: null
    }
  }
});
const AppMainNavigator = createStackNavigator({
  HomePage: {
    screen: HomePage,
    navigationOptions: {
      header: null
    }
  },
  DetailPage: {
    screen: DetailPage
  }
});

export const RootNavigator = createAppContainer(
  createSwitchNavigator({
    Init: AppInitNavigator,
    Main: AppMainNavigator
  })
);

/**
 * 1.初始化react-navigation与redux的中间件,
 * 该方法的一个很大的作用就是为reduxifyNavigator的key设置actionSubscribers(行为订阅者)
 */

//react-navigation-redux-helpers3.0变更,createReactNavigationReduxMiddleware的参数顺序发生了变化
export const middleware = createReactNavigationReduxMiddleware(
  state => state.nav,
  "root"
);

/* 2.将根导航器组件传递给 reduxifyNavigator 函数,
 * 并返回一个将navigation state 和 dispatch 函数作为 props的新组件;
 * 使用createReduxContainer方法，将RootNavigator封装成高阶组件
AppWithNavigationState
 * 这个高阶组件完成了navigation prop的替换，改成了使用redux里的navigation
 *
 * */

const AppWithNavigationState = createReduxContainer(RootNavigator, "root");
```

```
//State到Props的映射关系
const mapStateToProps = state => {
  return {
    state: state.nav
  };
};

//使用Redux的connect函数再封装一个高阶组件,连接 React 组件与 Redux store
export default connect(mapStateToProps)(AppWithNavigationState);
```

第二步：配置Reducer

```
import { combineReducers } from "redux";
import theme from "./theme";
import { rootCom, RootNavigator } from "../Navigator/AppNavigator";

//1.指定默认state
const navState = RootNavigator.router.getStateForAction(
  RootNavigator.router.getActionForPathAndParams(rootCom)
);

/**上面的代码创建了一个导航action(表示我想打开rootCom)，那么我们就可以通过action创建导航state，通过方*法getStateForAction(action, oldNavigationState)
*俩参数，一个是新的action，一个是当前的导航state，返回新的状态，当没有办法执行这个action的时候，就返回*null。
**/



/**
 * 2.创建自己的 navigation reducer,
 */
const navReducer = (state = navState, action) => {
  const nextState = RootNavigator.router.getStateForAction(action, state);
  // 如果`nextState`为null或未定义，只需返回原始`state`
  return nextState || state;
};

/**
 * 3.合并reducer
 * @type {Reducer<any> | Reducer<any, AnyAction>}
 */
const index = combineReducers({
  nav: navReducer,
```

```
  theme: theme
});

export default index;
```

第三步：配置store

```
import { applyMiddleware, createStore } from "redux";
import reducers from "../Reducer";
import { middleware } from "../Navigator/AppNavigator";

const middlewares = [middleware];
/**
 * 创建store
 */
export default createStore(reducers, applyMiddleware(...middlewares));
```

第四步：在组件中应用

```
import React, {Component} from 'react';
import {Provider} from 'react-redux';
import AppNavigator from './Navigator/AppNavigator';
import store from './Store'

type Props = {};
export default class App extends Component<Props> {
    render() {
        /**
         * 将store传递给App框架
         */
        return <Provider store={store}>
            <AppNavigator/>
        </Provider>
    }
}
```

经过上述4步呢，我们已经完成了react-navigaton+redux的集成，那么如何使用它呢?

案例：使用react-navigaton+redux 修改状态栏颜色

创建Actions

```
### Types.js
export default {
  THEM_CHANGE: "THEM_CHANGE",
  THEM_INIT: "THEM_INIT"
};
```

创建 Actions/theme

```
import Types from "../Types";

export function onThemeChange(theme) {
  return {
    type: Types.THEM_CHANGE,
    theme: theme
  };
}
```

创建Reducer/theme

```
import Types from "../../Actions/Types";

const defaultState = {
  theme: "blue"
};
export default function onAction(state = defaultState, action) {
  switch (action.type) {
    case Types.THEM_CHANGE:
      return {
        ...state,
        theme: action.theme
      };
    default:
      return state;
```

```
  }
}
```

在Reducer中聚合

```
const index = combineReducers({
  nav: navReducer,
  theme: theme
});
```

> 1.订阅state

```
import React, { Component } from "react";
import { Button, Platform, StyleSheet, Text, View } from "react-native";
import {
  createAppContainer,
  createMaterialTopTabNavigator
} from "react-navigation";
import IndexTab from "../Pages/IndexTab";
import { connect } from "react-redux";
import { onThemeChange } from "../Actions/theme";
import navigationUtil from "../Navigator/navigationUtil";
class IndexPage extends Component {
  constructor(props) {
    super(props);
    this.tabNames = [
      "ios",
      "android",
      "nodeJs",
      "Vue",
      "React",
      "React Native"
    ];
  }
  _genTabs() {
    const tabs = {};
    this.tabNames.forEach((item, index) => {
      tabs[`tab${index}`] = {
        screen: props => <IndexTab {...props} tabName={item} />,
        navigationOptions: {
          title: item
```

```
        }
      };
    });
    return tabs;
  }
  render() {
    const TabBackground = this.props.theme;
    console.log(this.props);
    const TabNavigator = createAppContainer(
      createMaterialTopTabNavigator(this._genTabs(), {
        tabBarOptions: {
          tabStyle: {},
          upperCaseLabel: false,
          scrollEnabled: true,
          style: {
            //选项卡背景色
            backgroundColor: TabBackground
          },
          indicatorStyle: {
            //指示器的样式
            height: 2,
            backgroundColor: "#fff"
          },
          labelStyle: {
            //文字的样式
            fontSize: 16,
            marginTop: 6,
            marginBottom: 6
          }
        }
      })
      //   createMaterialTopTabNavigator({
      //     IndexTab1: {
      //       screen: IndexTab,
      //       navigationOptions: {
      //         title: "Tab1"
      //       }
      //     },
      //     IndexTab2: {
      //       screen: IndexTab,
      //       navigationOptions: {
      //         title: "Tab2"
      //       }
      //     }
      //   })
    );
    return (
      <View style={{ flex: 1, marginTop: 30 }}>
        <TabNavigator />
```

```
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});

const mapStateToProps = state => ({
  theme: state.theme.theme
});

export default connect(mapStateToProps)(IndexPage);
```

在上述代码中我们订阅了store中的theme state，然后该组件就可以通过`this.props.theme`获取到所订阅的theme state了。

> 2.触发action改变state

```
import React, { Component } from "react";
import { Button, Platform, StyleSheet, Text, View } from "react-native";

import { connect } from "react-redux";
import { onThemeChange } from "../Actions/theme";

class IndexTab extends Component {
  render() {
    const { tabName } = this.props;
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Welcome to {tabName}</Text>
        <Button
          title={"go to DetailPage"}
          onPress={() => {
            navigationUtil.goPage(this.props, "DetailPage");
```

```
          }}
        />
        <Button
          title={"改变tab背景色"}
          onPress={() => {
            this.props.onThemeChange("#000");
            // navigationUtil.goPage(this.props, "DetailPage");
          }}
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  welcome: {
    fontSize: 20,
    textAlign: "center",
    margin: 10
  }
});
const mapStateToProps = state => ({});

const mapDispatchToProps = dispatch => ({
  onThemeChange: theme => dispatch(onThemeChange(theme))
});
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(IndexTab);
```

API

combineReducers(reducers)

随着应用变得越来越复杂，可以考虑将 reducer 函数 拆分成多个单独的函数，拆分后的每个函数负责独立管理 state 的一部分。

> 函数原型：`combineReducers(reducers)`

- 参数：reducers (Object): 一个对象，它的值（value）对应不同的 reducer 函数，这些 reducer 函数后面会被合并成一个。下面会介绍传入 reducer 函数需要满足的规则。
- 每个传入 combineReducers 的 reducer 都需满足以下规则：
  - 所有未匹配到的 action，必须把它接收到的第一个参数也就是那个 state 原封不动返回。
  - 永远不能返回 undefined。当过早 return 时非常容易犯这个错误，为了避免错误扩散，遇到这种情况时 combineReducers 会抛异常。
  - 如果传入的 state 就是 undefined，一定要返回对应 reducer 的初始 state。根据上一条规则，初始 state 禁止使用 undefined。使用 ES6 的默认参数值语法来设置初始 state 很容易，但你也可以手动检查第一个参数是否为 undefined。

返回值

(Function)：一个调用 reducers 对象里所有 reducer 的 reducer，并且构造一个与 reducers 对象结构相同的 state 对象。

`combineReducers` 辅助函数的作用是，把一个由多个不同 reducer 函数作为 value 的 object，合并成一个最终的 reducer 函数，然后就可以对这个 reducer 调用 createStore 方法。

合并后的 reducer 可以调用各个子 reducer，并把它们返回的结果合并成一个 state 对象。 由 `combineReducers()` 返回的 state 对象，会将传入的每个 reducer 返回的 state 按其传递给 `combineReducers()` 时对应的 key 进行命名。

> 提示：在 reducer 层级的任何一级都可以调用 combineReducers。并不是一定要在最外层。实际上，你可以把一些复杂的子 reducer 拆分成单独的孙子级 reducer，甚至更多层。

createStore

> 函数原型：`createStore(reducer, [preloadedState], enhancer)`

参数

- `reducer (Function):`：项目的根reducer。
- `[preloadedState] (any)`:这个参数是可选的, 用于设置 state 初始状态。这对开发同构应用时非常有用，服务器端 redux 应用的 state 结构可以与客户端保持一致, 那么客户端可以将从网络接收到的服务端 state 直接用于本地数据初始化。
- `enhancer (Function)`: Store enhancer 是一个组合 store creator 的高阶函数，返回一个新的强化过的 store creator。这与 middleware 相似，它也允许你通过复合函数改变 store 接口。

返回值

- `(Store)`: 保存了应用所有 state 的对象。改变 state 的惟一方法是 dispatch action。你也可以 subscribe 监听 state 的变化，然后更新 UI。

示例

```
import { createStore } from 'redux'
```

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.text])
    default:
      return state
  }
}

let store = createStore(todos, ['Use Redux'])

store.dispatch({
  type: 'ADD_TODO',
  text: 'Read the docs'
})

console.log(store.getState())
// [ 'Use Redux', 'Read the docs' ]
```

> 注意事项

- 应用中不要创建多个 store！相反，使用 combineReducers 来把多个 reducer 创建成一个根 reducer。
- 你可以决定 state 的格式。你可以使用普通对象或者 Immutable 这类的实现。如果你不知道如何做，刚开始可以使用普通对象。
- 如果 state 是普通对象，永远不要修改它！比如，reducer 里不要使用 Object.assign(state, newData)，应该使用 Object.assign({}, state, newData)。这样才不会覆盖旧的 state。如果可以的话，也可以使用 对象拓展操作符（object spread spread operator 特性中的 return { ...state, ...newData }。
- 对于服务端运行的同构应用，为每一个请求创建一个 store 实例，以此让 store 相隔离。dispatch 一系列请求数据的 action 到 store 实例上，等待请求完成后再在服务端渲染应用。
- 当 store 创建后，Redux 会 dispatch 一个 action 到 reducer 上，来用初始的 state 来填充 store。你不需要处理这个 action。但要记住，如果第一个参数也就是传入的 state 是 undefined 的话，reducer 应该返回初始的 state 值。
- 要使用多个 store 增强器的时候，你可能需要使用 compose

applyMiddleware

> 函数原型：`applyMiddleware(...middleware)`

使用包含自定义功能的 middleware 来扩展 Redux。

如何做到从不直接修改 state？

从不直接修改 state 是 Redux 的核心理念之一：为实现这一理念，可以通过一下两种方式：

1.通过Object.assign()创建对象拷贝, 而拷贝中会包含新创建或更新过的属性值

在下面的 todoApp 示例中, Object.assign() 将会返回一个新的 state 对象, 而其中的 visibilityFilter 属性被更新了:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

2. 通过通过ES7的新特性[对象展开运算符(Object Spread Operator)]

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return { ...state, visibilityFilter: action.filter }
    default:
      return state
  }
}
```

这样你就能轻松的跳回到这个对象之前的某个状态（想象一个撤销功能）。

总结

- Redux 应用只有一个单一的 store。当需要拆分数据处理逻辑时，你应该使用 reducer 组合 而不是创建多个 store；
- redux一个特点是：状态共享，所有的状态都放在一个store中，任何component都可以订阅store中的数据；
- 并不是所有的state都适合放在store中，这样会让store变得非常庞大，如某个状态只被一个组件使用，不存在状态共享，可以不放在store中；

RN网络编程

React Native 提供了和 web 标准一致的Fetch API，用于满足开发者访问网络的需求。

发起请求

要从任意地址获取内容的话，只需简单地将网址作为参数传递给 fetch 方法即可（fetch 这个词本身也就是`获取`的意思）

```
fetch('https://mywebsite.com/mydata.json');
```

Fetch 还有可选的第二个参数，可以用来定制 HTTP 请求一些参数。你可以指定 header 参数，或是指定使用 POST 方法，又或是提交数据等等：

```
fetch('https://mywebsite.com/endpoint/', {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    firstParam: 'yourValue',
    secondParam: 'yourOtherValue',
  }),
});
```

提交数据的格式关键取决于 headers 中的`Content-Type`。`Content-Type`有很多种，对应 body 的格式也有区别。到底应该采用什么样的`Content-Type`取决于服务器端，所以请和服务器端的开发人员沟通确定清楚。常用的'Content-Type'除了上面的'application/json'，还有传统的网页表单形式，示例如下：

```
fetch('https://mywebsite.com/endpoint/', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
  },
  body: 'key1=value1&key2=value2',
});
```

Fetch 方法会返回一个Promise，这种模式可以简化异步风格的代码

```
function getMoviesFromApiAsync() {
  return fetch('https://facebook.github.io/react-native/movies.json')
    .then((response) => response.json())
    .then((responseJson) => {
      return responseJson.movies;
    })
    .catch((error) => {
      console.error(error);
    });
}
```

你也可以在 React Native 应用中使用 ES2017 标准中的`async`/`await`语法:

```
// 注意这个方法前面有async关键字
async function getMoviesFromApi() {
  try {
    // 注意这里的await语句，其所在的函数必须有async关键字声明
    let response = await fetch(
      'https://facebook.github.io/react-native/movies.json',
    );
    let responseJson = await response.json();
    return responseJson.movies;
  } catch (error) {
    console.error(error);
  }
}

//fetch("https://api.douban.com/v2/movie/top250")
```

别忘了 catch 住`fetch`可能抛出的异常，否则出错时你可能看不到任何提示

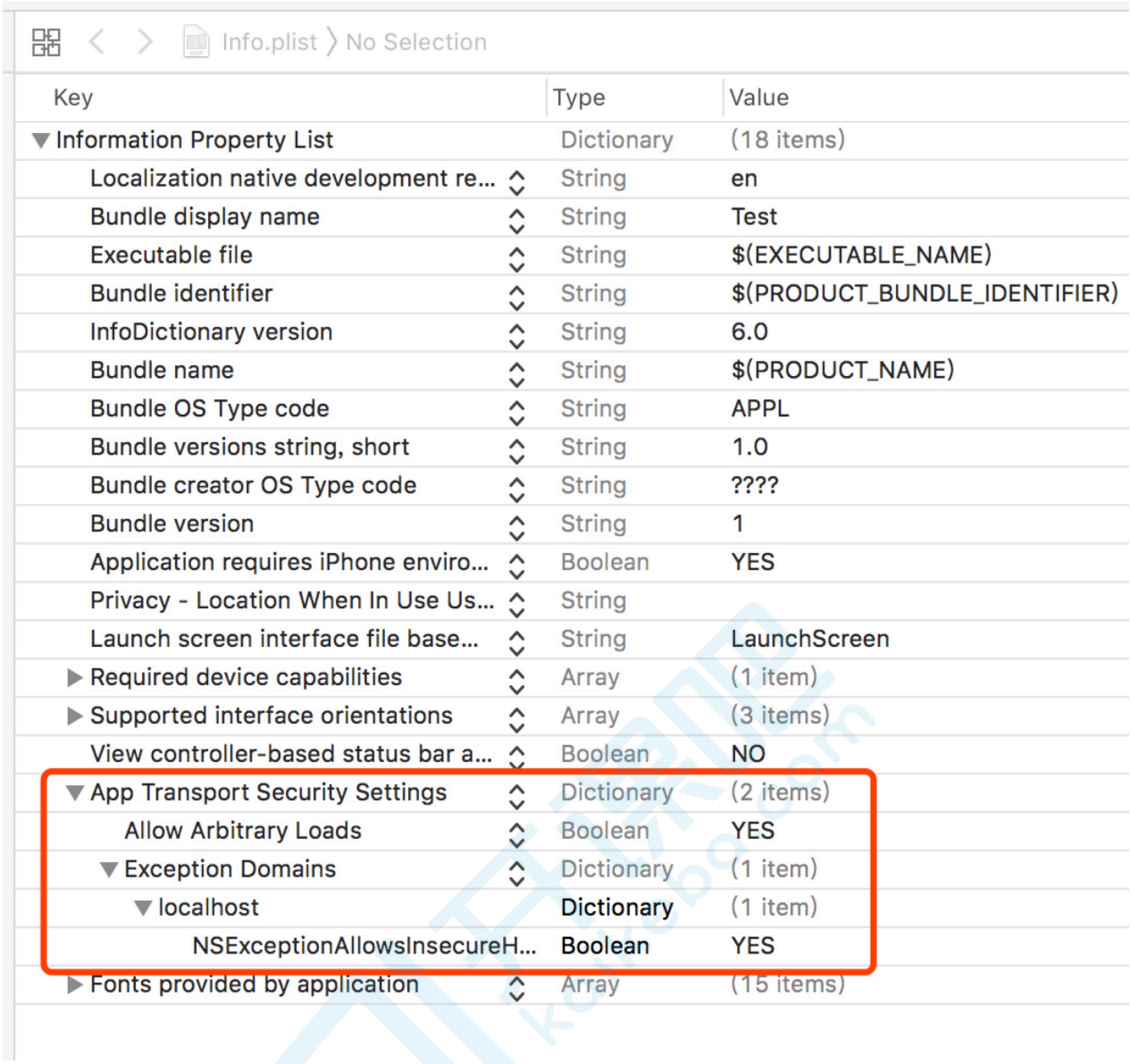
注意：使用 Chrome 调试目前无法观测到 React Native 中的网络请求，你可以使用第三方的react-native-debugger来进行观测。

关于RN中Http请求问题

IOS:

最新测试：RN 0.59.6项目已经默认开启支持http请求,不需要额外设置,查看步骤

通过xcode 进入项目的ios目录/项目目录 打开Info.plist文件



新项目已经默认设置

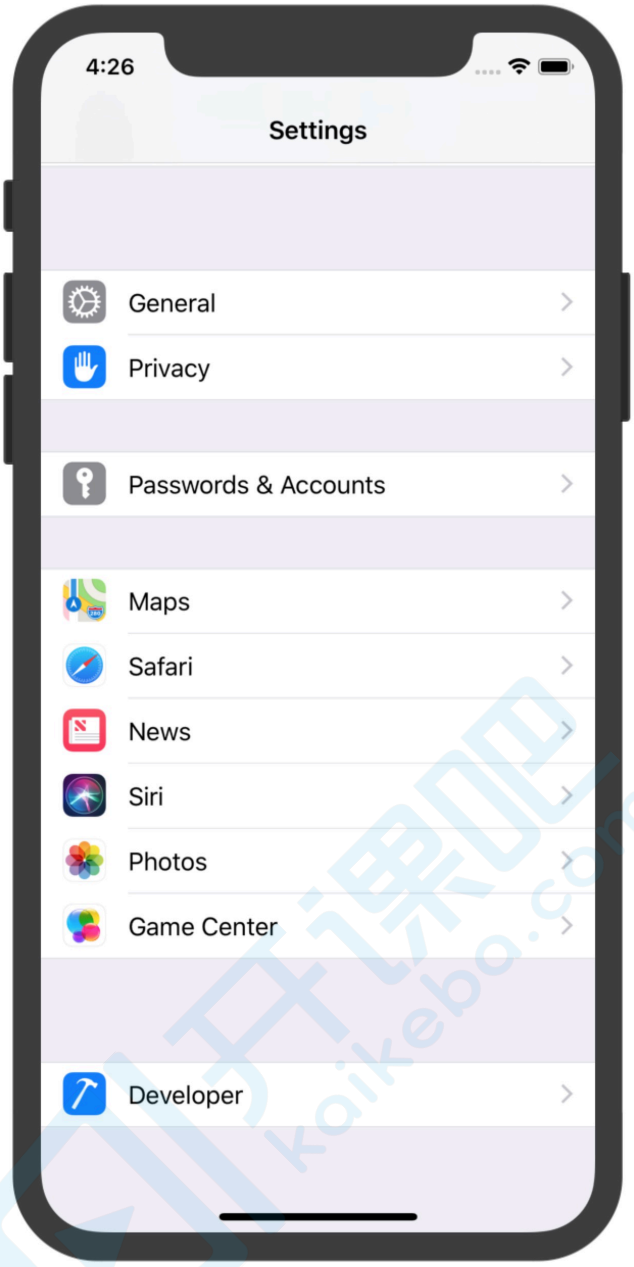
Allow Arbitrary Loads 字段 通过选择 后面的 YES 和 NO来控制App中是否允许http，设置改变后记得重启项目

已经设置没有问题，发现项目中的http请求，扔没有成功，也没有报错，比如在显示图片时。

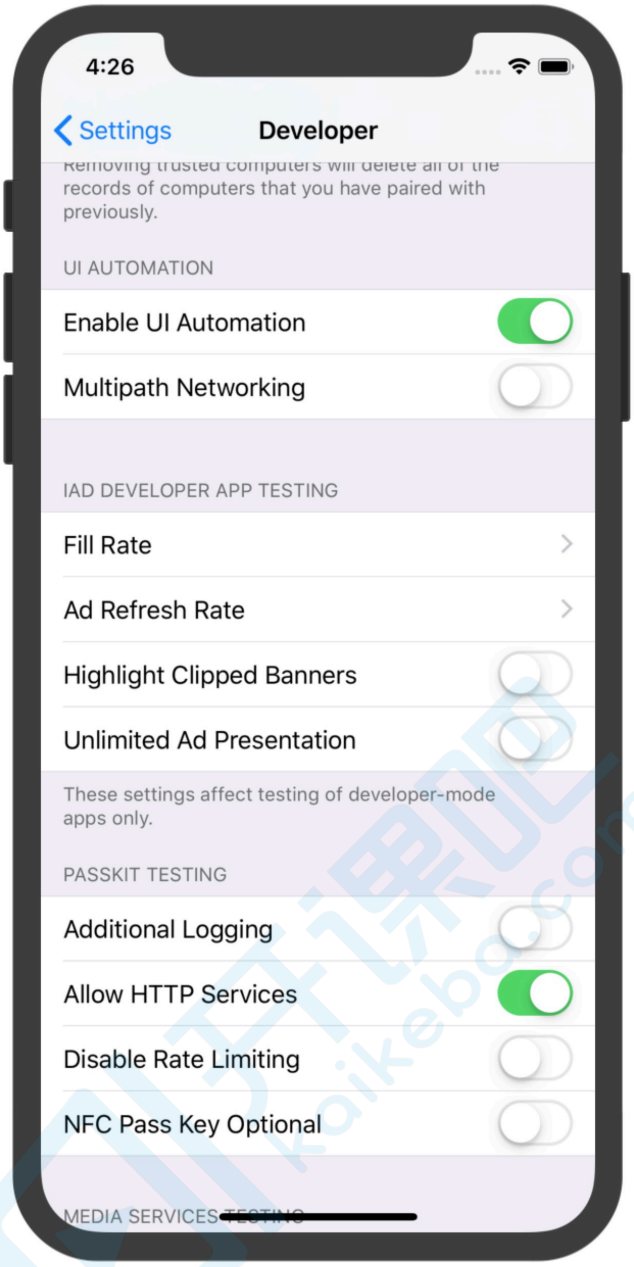
检查模拟器的设置，步骤

- 模拟器回到主页 ,点击Settings 进入设置

- 滑到底部，进入Developer



- 打开Allow HTTP Services

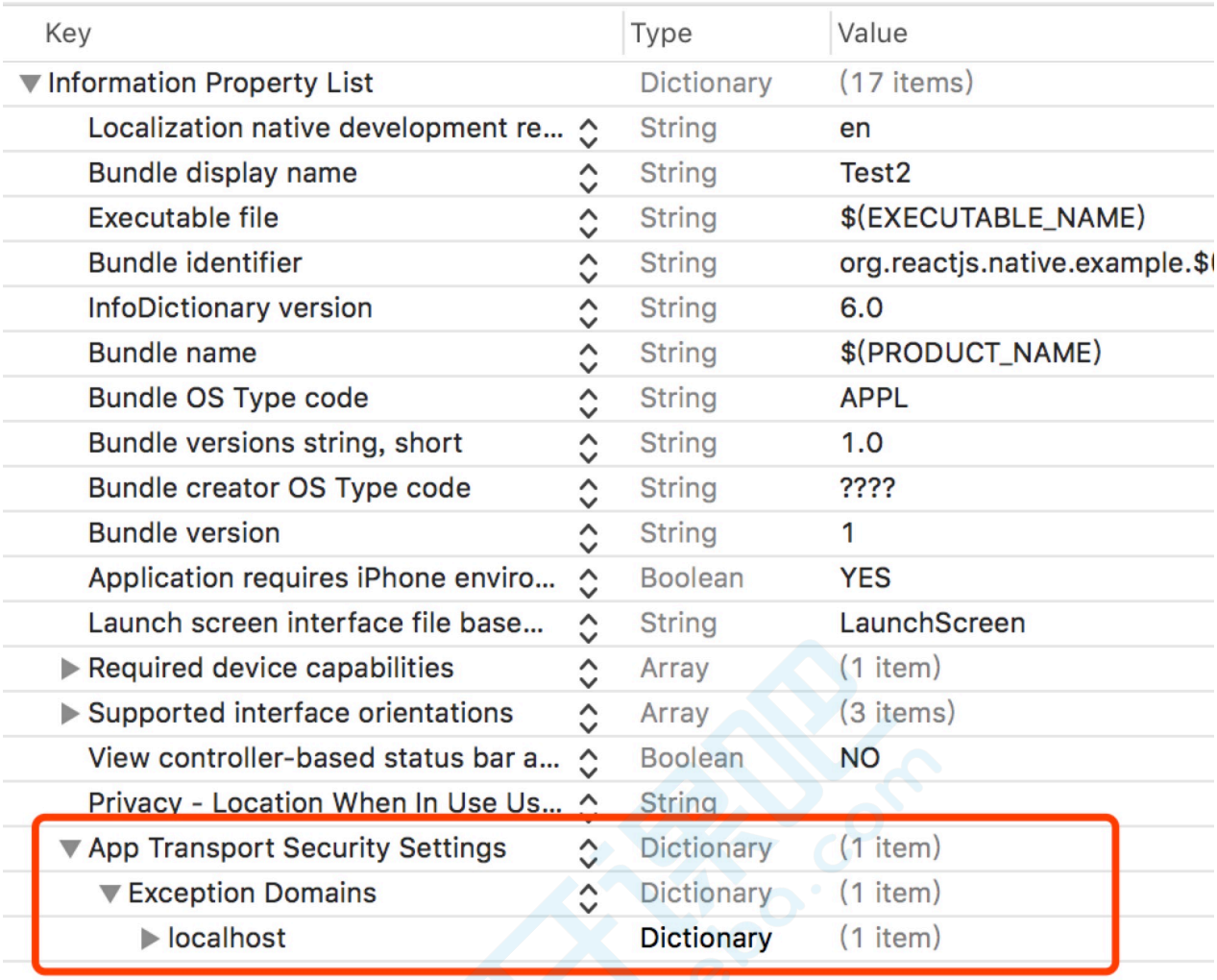


老版本项目中没有默认配置怎么办？以0.43版本为例

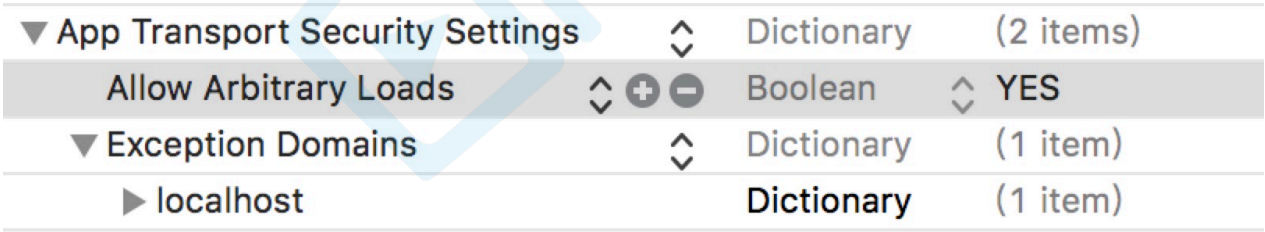
默认情况下，iOS 会阻止所有非 https 的请求。如果你请求的接口是 http 协议，**那么首先需要添加一个App Transport Security(新特性要求App内访问的网络必须使用`HTTPS`协议) 的例外**，从 Android9 开始，也会默认阻止 http 请求

Xcode 解决办法：

打开info.plist文件



在App Transport Security Settings添加Allow Arbitrary Loads 布尔类型，设置为YES



AndroidStudio

- 在 res 下新增一个 xml 目录，然后创建一个名为：network_security_config.xml 文件（名字自定），内容如下，大概意思就是允许开启http请求

```
<?xml version="1.0" encoding="utf-8"?>

<network-security-config>

 <base-config cleartextTrafficPermitted="true" />

</network-security-config>
```

- 在项目的AndroidManifest.xml文件下的application标签增加以下属性，应用以上配置

```
<application
 ...
    android:networkSecurityConfig="@xml/network_security_config"
```

上面介绍的方法虽然解决了网络访问的问题，但是苹果提供的安全保障也被关闭了。 不过，按照国内的现状，关闭这个限制也许是更实际的做法。 至于原因就太多了，第三方SDK（几乎都是访问`HTTP`），合作伙伴接入（不能要求它们一定要支持HTTPS）。 如果你的App没有受到这些原因的限制，还是更建议你增加`HTTPS`支持，而不是关闭限制。 请大家根据项目的实际情况作调整。

出于安全考虑我们提倡使用`HTTPS`，退而求其次