

React全家桶02

React全家桶02

课堂目标

资源

知识要点

使用react-redux

API

```
<Provider store>
```

```
connect([mapStateToProps],
```

```
[mapDispatchToProps], [mergeProps],
```

```
[options]))
```

参数

实现react-redux

react-router简介

安装

基本使用

Route渲染内容的三种方式

children: func

render: func

component: component

注意

使用Router

动态路由

嵌套路由

404页面

课堂目标

1. 掌握react-redux
2. 掌握Router使用
3. 掌握路由守卫逻辑

资源

1. [React Redux API](#)
2. [react-redux](#)
3. [react-router-这个英文文档很好](#)

知识要点

使用react-redux

每次都重新调用render和getState太low了，想用更react的方式来写，需要react-redux的支持。

```
npm install react-redux --save
```

提供了两个api

1. Provider 为后代组件提供store
2. connect 为组件提供数据和变更方法

API

<Provider store>

<Provider store> 使组件层级中的 `connect()` 方法都能够获得 Redux store。正常情况下，你的根组件应该嵌套在 <Provider> 中才能使用 `connect()` 方法。

```
connect([mapStateToProps],  
[mapDispatchToProps], [mergeProps],  
[options])
```

连接 React 组件与 Redux store。

返回一个新的已与 Redux store 连接的组件类。

参数

- `mapStateToProps(state, [ownProps])`:
`stateProps` (*Function*)

该回调函数必须返回一个纯对象，这个对象会与组件的 props 合并。

如果定义该参数，组件将会监听 Redux store 的变化，否则不监听。

ownProps 是当前组件自身的 props，如果指定了，那么只要组件接收到新的 props，mapStateToProps 就会被调用，mapStateToProps 都会被重新计算，mapDispatchToProps 也会被调用。注意性能！

- mapDispatchToProps(dispatch, [ownProps]): dispatchProps (Object or Function):

如果你省略这个 mapDispatchToProps 参数，默认情况下，dispatch 会注入到你的组件 props 中。

如果传递的是一个对象，那么每个定义在该对象的函数都将被当作 Redux action creator，对象所定义的方法名将作为属性名；每个方法将返回一个新的函数，函数中 dispatch 方法会将 action creator 的返回值作为参数执行。这些属性会被合并到组件的 props 中。

如果传递的是一个函数，该函数将接收一个 dispatch 函数，然后由你来决定如何返回一个对象。

ownProps 是当前组件自身的 props，如果指定了，那么只要组件接收到新的 props，mapDispatchToProps 就会被调用。注意性能！

- mergeProps(stateProps, dispatchProps, ownProps): props (Function)

如果指定了这个参数, `mapStateToProps()` 与 `mapDispatchToProps()` 的执行结果和组件自身的 `props` 将传入到这个回调函数中。该回调函数返回的对象将作为 `props` 传递到被包装的组件中。你也许可以用这个回调函数, 根据组件的 `props` 来筛选部分的 `state` 数据, 或者把 `props` 中的某个特定变量与 `action creator` 绑定在一起。如果你省略这个参数, 默认情况下返回 `Object.assign({}, ownProps, stateProps, dispatchProps)` 的结果。

全局提供store, index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
// import {Provider} from "react-redux";
import {Provider} from "../kReactRedux";

import store from "../store/";

// 把Provider放在根组件外层, 使子组件能获得store
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

获取状态数据, ReactReduxPage.js

```
import React, { Component } from "react";
import { connect } from "react-redux";

class ReactReduxPage extends Component {
  render() {
    const { num, add, minus, asyAdd } =
this.props;
    return (
      <div>
        <h1>ReactReduxPage</h1>
        <p>{num}</p>
        <button onClick={add}>add</button>
        <button onClick={minus}>minus</button>
      </div>
    );
  }
}

const mapStateToProps = state => {
  return {
    num: state,
  };
};

const mapDispatchToProps = {
  add: () => {
```

```

    return { type: "add" };
  },
  minus: () => {
    return { type: "minus" };
  }
};

export default connect(
  mapStateToProps, //状态映射 mapStateToProps
  mapDispatchToProps, //派发事件映射
)(ReactReduxPage);

```

connect中的参数：state映射和事件映射

详细使用

```

import React, {Component} from "react";
import {connect} from "react-redux";
import {bindActionCreators} from "redux";

// connect帮组子组件与store链接，其实就是高阶组件，这里返回的是一个新的组件
export default connect(
  // mapStateToProps Function (state, ownProps)
  state => ({count: state}),
  // !谨慎使用ownProps，如果它发生变化，
  // mapStateToProps就会执行，里面的state会被重新计算，容易影响性能
  // (state, ownProps) => {

```

```

    // console.log("ownProps", ownProps); //sy-
log
    // return {
    //     count: state
    // };
    // }

    // mapDispatchToProps Object/Function 如果不定
义 默认把props注入组件
    // 如果是对象的话, 原版的dispatch就没有被注入了
    // {
    //     add: () => ({type: "ADD"})
    // }

    // Function (dispatch, ownProps)
    // !谨慎使用ownProps, 如果它发生变化,
mapDispatchToProps就会执行, 容易影响性能
    // (dispatch, ownProps) => {
    //     console.log("ownProps", ownProps); //sy-
log
    dispatch => {
        let res = {add: () => ({type: "ADD"}),
minus: () => ({type: "MINUS"})};
        res = bindActionCreators(res, dispatch);

        return {dispatch, ...res};
    },
    // mergeProps Function

```


// 如果指定了这个参数, `mapStateToProps()` 与
`mapDispatchToProps()` 的执行结果和组件自身的
`props` 将传入到这个回调函数中。

```
(stateProps, dispatchProps, ownProps) => {  
  console.log("mergeProps", stateProps,  
dispatchProps, ownProps); //sy-log  
  return {omg: "omg", ...stateProps,  
...dispatchProps, ...ownProps};  
}  
)(  
  class ReactReduxPage extends Component {  
    render() {  
      console.log("props", this.props); //sy-  
log  
      const {count, dispatch, add, minus} =  
this.props;  
      return (  
        <div>  
          <h3>ReactReduxPage</h3>  
          <p>{count}</p>  
          <button onClick={() =>  
dispatch({type: "ADD"})}>  
            add use dispatch  
          </button>  
          <button onClick={add}>add</button>  
          <button onClick=  
{minus}>minus</button>  
        </div>  
      );  
    }  
  }  
)
```

```
    }  
  }  
);
```

实现react-redux

实现kReact-redux.js

```
import React, {Component} from "react";  
  
const ValueContext = React.createContext();  
  
// connect  
export const connect = (  
  mapStateToProps = state => state,  
  mapDispatchToProps  
) => WrappedComponent => {  
  return class extends Component {  
    static contextType = ValueContext;  
    constructor(props) {  
      super(props);  
      this.state = {  
        props: {}  
      };  
    }  
    componentDidMount() {  
      this.update();  
    }  
  }  
}
```

```

    const {subscribe} = this.context;
    subscribe(() => {
      this.update();
    });
  }

  update = () => {
    const {getState, dispatch} =
this.context;

    const stateProps =
mapStateToProps(getState());
    let dispatchProps;
    console.log("mapDispatchToProps",
mapDispatchToProps); //sy-log
    if (typeof mapDispatchToProps ===
"object") {
      dispatchProps =
bindActionCreators(mapDispatchToProps,
dispatch);
    } else if (typeof mapDispatchToProps ===
"function") {
      dispatchProps =
mapDispatchToProps(dispatch, this.props);
    } else {
      dispatchProps = {dispatch};
    }
    this.setState({
      props: {
        ...stateProps,

```

```

        ...dispatchProps
      }
    });
  };

  render() {
    return <WrappedComponent {...this.props}
    {...this.state.props} />;
  }
};

// Provider
// /context
export class Provider extends Component {
  render() {
    return (
      <ValueContext.Provider value=
    {this.props.store}>
      {this.props.children}
      </ValueContext.Provider>
    );
  }
}

// let creators = {
//   add: () => ({type: "ADD"}),
//   minus: () => ({type: "MINUS"})
// };

```

```
function bindActionCreators(creator, dispatch) {
  return (...args) =>
    dispatch(creator(...args));
}

export function bindActionCreatorsCreators(creators,
dispatch) {
  const obj = {};
  for (const key in creators) {
    obj[key] = bindActionCreators(creators[key],
dispatch);
  }
  return obj;
}
```

react-router简介

react-router包含3个库，react-router、react-router-dom和react-router-native。react-router提供最基本的路由功能，实际使用的时候我们不会直接安装react-router，而是根据应用运行的环境选择安装react-router-dom（在浏览器中使用）或react-router-native（在rn中使用）。react-router-dom和react-router-native都依赖react-router，所以在安装时，react-router也会自动安装，创建web应用，使用：

安装

```
npm install --save react-router-dom
```

基本使用

react-router中奉行一切皆组件的思想，路由器-**Router**、链接-**Link**、路由-**Route**、独占-**Switch**、重定向-**Redirect**都以组件形式存在

创建RouterPage.js

```
import React, { Component } from "react";
import { BrowserRouter, Link, Route } from
"react-router-dom";
import HomePage from "./HomePage";
import UserPage from "./UserPage";

export default class RouterPage extends
Component {
  render() {
    return (
      <div>
        <h1>RouterPage</h1>
        <BrowserRouter>
          <nav>
            <Link to="/">首页</Link>
            <Link to="/user">用户中心</Link>
```

```
        </nav>
        { /* 根路由要添加exact, 实现精确匹配 */ }
        <Route exact path="/" component=
{HomePage} />
        <Route path="/user" component=
{UserPage} />
    </BrowserRouter>
</div>
);
}
}
```

Route渲染内容的三种方式

Route渲染优先级：children>component>render。

三者能接收到同样的[route props]，包括match, location and history，但是当不匹配的时候，children的match为null。

这三种方式互斥，你只能用一种，它们的不同之处可以参考下文：

children：func

有时候，不管location是否匹配，你都需要渲染一些内容，这时候你可以用children。

除了不管location是否匹配都会被渲染之外，其它工作方法与render完全一样。

```
import React, { Component } from "react";
import ReactDOM from "react-dom";
import { BrowserRouter as Router, Link, Route }
from "react-router-dom";

function ListItemLink({ to, name, ...rest }) {
  return (
    <Route
      path={to}
      children={({ match }) => (
        <li className={match ? "active" : ""}>
          <Link to={to} {...rest}>
            {name}
          </Link>
        </li>
      )}
    />
  );
}

export default class RouteChildren extends
Component {
  render() {
    return (
      <div>
        <h3>RouteChildren</h3>
      </div>
    );
  }
}
```



```

    <Router>
      <ul>
        <ListItemLink to="/somewhere"
name="链接1" />
        <ListItemLink to="/somewhere-else"
name="链接2" />
      </ul>
    </Router>
  </div>
);
}
}

```

render: func

但是当你用`render`的时候，你调用的只是个函数。但是它和component一样，能访问到所有的[route props]。

```

import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter as Router, Route } from
"react-router-dom";

// 方便的内联渲染
ReactDOM.render(
  <Router>
    <Route path="/home" render={() =>
<div>Home</div>} />

```

```

    </Router>,
    node
  );

  // wrapping/composing
  //把route参数传递给你的组件
  function FadingRoute({ component: Component,
    ...rest }) {
    return (
      <Route
        {...rest}
        render={routeProps => (
          <Component {...routeProps} />
        )}
      />
    );
  }

  ReactDOM.render(
    <Router>
      <FadingRoute path="/cool" component=
        {Something} />
    </Router>,
    node
  );

```

component: component

只在当location匹配的时候渲染。

```
import React, {Component, useEffect} from
"react";
import {BrowserRouter as Router, Route} from
"react-router-dom";

export default class RouteComponePage extends
Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    const {count} = this.state;
    return (
      <div>
        <h3>RouteComponePage</h3>
        <button
          onClick={() => {
            this.setState({count: count + 1});
          }}>
          click change count {count}
        </button>
      </div>
    );
  }
}
```

{/* 渲染component的时候会调用
React.createElement，如果使用下面这种匿名函数的形
式，每次都会生成一个新的匿名的函数，
 导致生成的组件的type总是不相同，这个时候会
产生重复的卸载和挂载 */}

 {/* 错误举例 课下自己尝试下 观察下child的
didMount和willUnmount函数 */}

 {/* <Route component={() => <Child
count={count} />} /> */}

 {/* <Route component={() =>
<FunctionChild count={count} />} /> */}

 {/* 下面才是正确的示范 */}

 {/* <Route render={() => <Child
count={count} />} /> */}

 <Route render={() => <FunctionChild
count={count} />} />

 {/* children 呢 */}

 {/* <Route children={() => <Child
count={count} />} /> */}

 <Route children={() => <FunctionChild
count={count} />} />

 </Router>

 </div>

);

}

}

```
class Child extends Component {
  componentDidMount() {
    console.log("componentDidMount"); //sy-log
  }

  componentWillUnmount() {
    console.log("componentWillUnmount"); //sy-log
  }

  render() {
    return <div>child-{this.props.count}</div>;
  }
}

// hook的例子 如果刚接触react不懂 下周我们再学 可以先忽略
function FunctionChild(props) {
  useEffect(() => {
    return () => {
      console.log("WillUnmount"); //sy-log
    };
  }, []);
  return <div>child-{props.count}</div>;
}
```

注意

当你用 `component` 的时候，Router 会用你指定的组件和 `React.createElement` 创建一个新的 [React element]。这意味着当你提供的是一个内联函数的时候，每次 render 都会创建一个新的组件。这会导致不再更新已经现有组件，而是直接卸载然后再去挂载一个新的组件。因此，当用到内联函数的内联渲染时，请使用 `render` 或者 `children`。

Route 核心渲染代码如下：

```
return (
  <RouterContext.Provider value={props}>
    {props.match
      ? children
        ? typeof children === "function"
          ? __DEV__
            ? evalChildrenDev(children, props, this.props.path)
              { children(props) } children是获取了props后执行
            : children
          : component
        ? React.createElement(component, props) component是createElement后执行
        : render
        ? render(props) render是获取了props后执行
        : null
      : typeof children === "function"
        ? __DEV__
          ? evalChildrenDev(children, props, this.props.path)
            : children(props)
          : null}
    </RouterContext.Provider>
  );
```

使用Router

动态路由

使用:id的形式定义动态路由

定义路由:

```
<Route path="/search/:id" component={Search} />
```

添加导航链接:

```
<Link to={"/search/" + searchId}>搜索</Link>
```

创建Search组件并获取参数:

```
import React, { Component } from "react";
import { BrowserRouter, Link, Route } from
"react-router-dom";
import HomePage from "./HomePage";
import UserPage from "./UserPage";

function Search({ match, history, location }) {
  const { id } = match.params;
  return (
    <div>
      <h1>Search: {id}</h1>
    </div>
  );
}

export default class RouterPage extends
Component {
  render() {
    const searchId = "1234";
    return (
```

```

<div>
  <h1>RouterPage</h1>
  <BrowserRouter>
    <nav>
      <Link to="/">首页</Link>
      <Link to="/user">用户中心</Link>
      <Link to={"/search/" + searchId}>搜索</Link>
    </nav>
    { /* 根路由要添加exact, 实现精确匹配 */ }
    <Route exact path="/" component=
{HomePage} />
    <Route path="/user" component=
{UserPage} />
    <Route path="/search/:id" component=
{SearchComponent} />
  </BrowserRouter>
</div>
);
}
}

```

嵌套路由

Route组件嵌套在其他页面组件中就产生了嵌套关系

修改Search，添加新增和详情


```

function DetailComponent(props) {
  return <div>DetailComponent</div>;
}

function SearchComponent(props) {
  console.log("props", props); //sy-log
  const {id} = props.match.params;
  return (
    <div>
      <h3>SearchComponent</h3>
      <p>{id}</p>
      <Link to={"/search/" + id + "/detail"}>详情</Link>
      <Route path="/search/:id/detail"
component={DetailComponent} />
    </div>
  );
}

```

404页面

设定一个没有path的路由在路由列表最后面，表示一定匹配

```
{/* 添加Switch表示仅匹配一个*/}  
<Switch>  
  {/* 根路由要添加exact, 实现精确匹配 */}  
  <Route exact path="/" component={HomePage} />  
  <Route path="/user" component={UserPage} />  
  <Route path="/search/:id" component={Search}  
  />  
  <Route render={() => <h1>404</h1>} />  
</Switch>
```

路由守卫

思路：创建高阶组件包装Route使其具有权限判断功能

创建PrivateRoute

```
import React, {Component} from "react";  
import {Route, Redirect} from "react-router-dom";  
  
export default class PrivateRoute extends  
Component {  
  render() {  
    const {isLogin, path, component} =  
this.props;  
    if (isLogin) {  
      // 登录
```

```

        return <Route path={path} component=
{component} />;
    } else {
        // 去登录，跳转登录页面
        return <Redirect to={{pathname: "/login",
state: {redirect: path}}}} />;
    }
}
}

```

创建LoginPage.js

```

import React, {Component} from "react";
import {Redirect} from "react-router-dom";

export default class LoginPage extends
Component {
    render() {
        const {isLogin, location} = this.props;
        const {redirect = "/" } = location.state ||
{};
        console.log("props", this.props); //sy-log
        if (isLogin) {
            // 已经登录
            return <Redirect to={redirect} />;
        } else {
            return (
                <div>
                    <h3>LoginPage</h3>

```

```
        <button onClick={() =>
        {}}>click</button>
      </div>
    );
  }
}
}
```

在RouterPage.js配置路由， RouterPage

```
<Route exact path="/login" component=
{LoginPage} />
<PrivateRoute path="/user" component={UserPage}
/>
```

整合redux， 获取和设置登录态， 创建./store/index.js

//自己实现

src/index.js

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import { Provider } from "react-redux";
import store from "./store";

ReactDOM.render(
  <Provider store={store}>
    <App />,
  </Provider>,
  document.getElementById("root"),
);
```

//UserPage可以再设置一个退出登录。

回顾

React全家桶02

课堂目标

资源

知识要点

使用react-redux

API

```
<Provider store>
```

```
connect([mapStateToProps],  
[mapDispatchToProps], [mergeProps],  
[options])
```

参数

实现react-redux

react-router简介

安装

基本使用

Route渲染内容的三种方式

children: func

render: func

component: component

注意

使用Router

动态路由

嵌套路由

404页面

路由守卫

回顾

作业

下节课内容

作业

1. 实现react-redux。
2. 用redux与router实现路由守卫，实现登录和退出登录。

下节课内容

1. Router实现
2. React实现：实现createElement、Component、render三个核心api。

