React全家桶02

回顾

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-router
2. react-router中文文档

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

全局提供store，index.js

```
import React from 'react'
import ReactDom from 'react-dom'
import App from './App'
import store from './store/ReactReduxStore'

import { Provider } from 'react-redux'
ReactDom.render(
  <Provider store={store}>
    <App/>
  </Provider>,
  document.querySelector('#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>
        {/* <button onClick=
{asyAdd}>asyAdd</button> */}
      </div>
    );
  }
}

const mapStateToProps = state => {
  return {
    num: state,
  };
};
const mapDispatchToProps = {
  add: () => {
    return { type: "add" };
  },
  minus: () => {
    return { type: "minus" };
  },
  //Actions must be plain objects. Use custom
middleware for async actions.
  // asyAdd: () => {
    //   //console.log("omh", dispatch);
    //   setTimeout(() => {
    //     return { type: "add" };
```

```
    //   }, 1000);
    // },
};

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

实现react-redux

实现kReact-redux.js

```
import React from 'react'
import PropTypes from 'prop-types'
import {bindActionCreators} from './kkb-redux'

export const connect = (mapStateToProps =
state=>state, mapDispatchToProps = {}) =>
(WrapComponent) => {
  return class ConnectComponent extends
React.Component{
        // class组件中声明静态contextTypes可以获取
上下文Context
    static contextTypes = {
```

```
    store: PropTypes.object
  }
  constructor(props, context){
    super(props, context)
    this.state = {
      props:{}
    }
  }
  componentDidMount(){
    const {store} = this.context
    store.subscribe(()=>this.update())
    this.update()
  }
  update(){
    const {store} = this.context
          // state => ({num: state.counter})
    const stateProps =
mapStateToProps(store.getState())
          // {add:()=>({type:'add'})}
          // {add:(...args) =>
dispatch(creator(...args))}
    const dispatchProps =
bindActionCreators(mapDispatchToProps,
store.dispatch)
    this.setState({
      props:{
        ...this.state.props, // 之前的值
        ...stateProps, // num: state.counter
```

```
          ...dispatchProps // add:(...args) =>
dispatch(creator(...args))
        }
      })
    }
    render(){
      return <WrapComponent
{...this.state.props}></WrapComponent>
    }
  }
}

export class Provider extends React.Component{
  static childContextTypes = {
    store: PropTypes.object
  }
  getChildContext() {
    return { store: this.store }
  }
  constructor(props, context) {
    super(props, context)
    this.store = props.store
  }
  render() {
    return this.props.children
  }
}
//实现bindActionCreators
function bindActionCreator(creator, dispatch){
```

```
  return (...args) =>
dispatch(creator(...args))
}
export function
bindActionCreators(creators,dispatch){
    // {add:()=>({type:'add'})}
    // {add:(...args) =>
dispatch(creator(...args))}
  return
Object.keys(creators).reduce((ret,item)=>{
    ret[item] =
bindActionCreator(creators[item],dispatch)
    return ret
  },{})
}
```

用hooks实现:

```
import React, { useContext, useState, useEffect
} from "react";

const Context = React.createContext();

export function Provider({ store, children }) {
  return <Context.Provider value={store}>
{children}</Context.Provider>;
}

export const connect = (
```

```
  mapStateToProps = state => state,
  mapDispatchToProps = {},
) => Cmp => props => {
  const store = useContext(Context);
  const getMoreProps = () => {
    const stateProps =
mapStateToProps(store.getState());
    const dispatchProps = bindActionCreators(
      mapDispatchToProps,
      store.dispatch,
    );
    return { ...stateProps, ...dispatchProps };
  };
  const [moreProps, setMoreProps] =
useState(getMoreProps());
  useEffect(() => {
    store.subscribe(() => {
      setMoreProps({ ...moreProps,
...getMoreProps() });
    });
  }, []);
  return <Cmp {...props} {...moreProps} />;
};

//这两个函数可在redux源码中查看，不懂的地方课
console.log打印查看，建议打印下actionCreators和key
function bindActionCreator(creator, dispatch) {
  return (...args) =>
dispatch(creator(...args));
```

```
}
function bindActionCreators(actionCreators,
dispatch) {
  const boundActionCreators = {};
  for (const key in actionCreators) {
    boundActionCreators[key] =
bindActionCreator(actionCreators[key],
dispatch);
  }
  return boundActionCreators;
}
```

react-router

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>
        <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 => (
        <FadeIn>
          <Component {...routeProps} />
        </FadeIn>
      )}
    />
  );
}

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

component: component

只在当location匹配的时候渲染。

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。

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

class Foo extends Component {
  componentDidMount() {
    console.log("Foo componentDidMount");
  }
  componentWillUnmount() {
    console.log("Foo componentWillUnmount");
  }
  render() {
    const { counter } = this.props;
    return <div>Foo: {counter}</div>;
  }
}

export default class RouterPage extends
Component {
  constructor(prop) {
    super(prop);
```

```
    this.state = { counter: 1 };
  }
  render() {
    const { counter } = this.state;
    return (
      <div>
        <button onClick={() => this.setState({
counter: counter + 1 })}>
          {counter}
        </button>
        <BrowserRouter>
          {/* 渲染component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函数，导致生成的组件的
tyoe总不相同，会产生重复的卸载和挂载。
        所以请正确使用Route中的component和render。
         */}
          {/* <Route component={() => <Foo
counter={this.state.counter} />} /> */}

          {/* 以下才是正确使用 */}
          <Route render={() => <Foo counter=
{this.state.counter} />} />
        </BrowserRouter>
      </div>
    );
  }
}
```

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=
{Search} />
          </BrowserRouter>
        </div>
      );
    }
  }
```

嵌套

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 Detail() {
  return (
    <div>
      <h1>Detail</h1>
    </div>
  );
}

function Search({ match, history, location }) {
  const { id } = match.params;
  return (
    <div>
      <h1>Search: {id}</h1>
      <nav>
        <Link to="/search/add">新增</Link>
        <Link to={"/search/detail/" + id}>详情
</Link>
      </nav>
      <Route path="/search/add" component={()
=> <h1>add</h1>} />
      <Route path={"/search/detail/:" + id}
component={Detail} />
```

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

class PrivateRoute extends Component {
  render() {
    const { path, component, isLogin } = this.props;
    if (isLogin) {
      return <Route path={path} component={component} />;
    } else {
      return (
        <Redirect
          to={{
            pathname: "/login",
            state: { redirect: path },
          }}
        />
      );
    }
  }
}

export default connect(state => state.user)(PrivateRoute);
```

创建LoginPage.js

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

class LoginPage extends Component {
  render() {
    const { isLogin, login, location } =
this.props;
    const { redirect = "/" } = location.state
|| {};
    if (isLogin) {
      return <Redirect to={redirect} />;
    }
    return (
      <div>
        <h3>LoginPage</h3>
        <button onClick={login}>login</button>
      </div>
    );
  }
}

export default connect(
  state => state.user,
  {
    login: () => ({
      type: "loginSuccess",
    }),
  },
```

```
)(LoginPage);
```

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

```
import { createStore, combineReducers } from
"redux";

const initalUserInfo = {
isLogin: false,
user: {
 name: "小明",
 },
};
function loginReducer(state = {
...initalUserInfo }, action) {
switch (action.type) {
   case "getUserInfo":
      return { ...initalUserInfo };
    case "loginSuccess":
      return { ...state, isLogin: true };
    case "loginFailure":
      return { ...state, isLogin: true };
    default:
```

```
      return { ...state };
  }
}
const store = createStore(
combineReducers({
 user: loginReducer,
 }),
);

export default store;
```

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"),
);
```

作业：UserPage可以再设置一个退出登录

```
import React, { Component } from "react";
```

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

class UserPage extends Component {
  render() {
    const { logout } = this.props;
    return (
      <div>
        <h1>UserPage</h1>
        <button onClick={logout}>退出登
录</button>
      </div>
    );
  }
}

export default connect(
  state => state.user,
  {
    logout: () => ({
      type: "loginFailure",
    }),
  },
)(UserPage);
```

与HashRouter对比：

1. HashRouter最简单，不需要服务器端渲染，靠浏览器的#

的来区分path就可以，BrowserRouter需要服务器端对不同的URL返回不同的HTML，后端配置可参考。
2. BrowserRouter使用HTML5历史API（pushState，replaceState和popstate事件），让页面的UI同步与URL。
3. HashRouter不支持location.key和location.state，动态路由跳转需要通过?传递参数。
4. Hash history 不需要服务器任何配置就可以运行，如果你刚刚入门，那就使用它吧。但是我们不推荐在实际线上环境中用到它，因为每一个 web 应用都应该渴望使用`browserHistory`。

MemoryRouter

把 URL 的历史记录保存在内存中的`<Router>`（不读取、不写入地址栏）。在测试和非浏览器环境中很有用，如React Native。

](https://facebook.github.io/react-native/)。

拓展

react-router秉承一切皆组件，因此实现的核心就是BrowserRouter、Route、Link

实现BrowserRouter

BrowserRouter：历史记录管理对象history初始化及向下传递，location变更监听

创建测试页面MyRouterPage.js，

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

export default class MyRouterPage extends
Component {
  render() {
    return (
      <div>
        <h3>MyRouterPage</h3>
        <BrowserRouter>
          <Link to="/">首页</Link>
          <Link to="/user">用户中心</Link>
          <Route path="/" exact component=
{HomePage} />
          <Route path="/user" component=
{UserPage} />
        </BrowserRouter>
      </div>
    );
  }
}
```

my-react-router-dom.js，首先实现BrowserRouter

```
import { createBrowserHistory } from "history";

const RouterContext = React.createContext();

class BrowserRouter extends Component {
  constructor(props) {
    super(props);

    this.history =
createBrowserHistory(this.props);

    this.state = {
      location: this.history.location
    };

    this.unlisten =
this.history.listen(location => {
      this.setState({ location });
    });
  }

  componentWillUnmount() {
    if (this.unlisten) this.unlisten();
  }
```

```
  render() {
    return (
      <RouterContext.Provider
        children={this.props.children || null}
        value={{
          history: this.history,
          location: this.state.location
        }}
      />
    );
  }
}
```

实现Route

路由配置，匹配检测，内容渲染

```
export function Route(props) {
  const ctx = useContext(RouterContext);
  const { path, component: Cmp } = props;
  const { location } = ctx;
  let match = path === location.pathname;
  return match ? <Cmp /> : null;
}
```

实现Link

Link.js: 跳转链接，处理点击事件

```
export class Link extends Component {
  handleClick(event, history) {
    event.preventDefault();
    history.push(this.props.to);
  }

  render() {
    const { to, children } = this.props;

    return (
      <RouterContext.Consumer>
        {context => {
          return (
            <a
              {...rest}
              onClick={event =>
this.handleClick(event, context.history)}
              href={to}
            >
              {children}
            </a>
          );
        }}
      </RouterContext.Consumer>
    );
```

```
    }
}
```

回顾

React全家桶02

- 课堂目标
- 资源
- 知识要点
- 使用react-redux
- 实现react-redux

- react-router
  - 安装
  - 基本使用
  - Route渲染内容的三种方式
    - children：func
    - render：func
    - component: component
  - 动态路由
  - 嵌套
  - 404页面
  - 路由守卫
  - 与HashRouter对比：
  - MemoryRouter
- 拓展
  - 实现BrowserRouter

- 实现Route
- 实现Link

回顾