

# React全家桶02

---

## React全家桶02

课堂目标

资源

知识要点

Hooks API

useReducer

useEffect

useLayoutEffect

使用react-redux

API

```
<Provider store>
```

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

```
[mapDispatchToProps], [mergeProps], [options]))
```

参数

实现react-redux

react-redux hooks API及实现

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

404页面

路由守卫

回顾

## 课堂目标

---

1. 掌握所有Hooks方法
2. 掌握高阶组件
3. 掌握react-redux使用和原理
4. 掌握Router使用

## 资源

---

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

## 知识要点

---

### Hooks API

#### useReducer

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

`useState` 的替代方案。它接收一个形如 `(state, action) => newState` 的 reducer，并返回当前的 state 以及与其配套的 `dispatch` 方法。（如果你熟悉 Redux 的话，就已经知道它如何工作了。）

```
import React, {useReducer, useEffect,
useLayoutEffect} from "react";
import {counterReducer} from "../store";

const init = initArg => {
  return initArg + 1;
};

export default function HooksPage(props) {
  const [state, dispatch] =
    useReducer(counterReducer, 0, init);

  useEffect(() => {
    console.log("useEffect"); //sy-log
  });

  useLayoutEffect(() => {
    console.log("useLayoutEffect"); //sy-log
  });

  console.log("---"); //sy-log
  return (
    <div>
      <h3>HooksPage</h3>
      <p>{state}</p>
      <button onClick={() => dispatch({type:
"ADD"})}>add</button>
    </div>
  );
}
```

```
);  
}
```

## useEffect

```
useEffect(didUpdate);
```

该 Hook 接收一个包含命令式、且可能有副作用代码的函数。

在函数组件主体内（这里指在 React 渲染阶段）改变 DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的 bug 并破坏 UI 的一致性。

使用 `useEffect` 完成副作用操作。赋值给 `useEffect` 的函数会在组件渲染到屏幕之后延迟执行。你可以把 effect 看作从 React 的纯函数式世界通往命令式世界的逃生通道。

默认情况下，effect 将在每轮渲染结束后执行，但你可以选择让它在只有某些值改变的时候才执行。

官网地址：<https://zh-hans.reactjs.org/docs/hooks-reference.html#useeffect>

## useLayoutEffect

其函数签名与 `useEffect` 相同，但它会在所有的 DOM 变更之后同步调用 effect。可以使用它来读取 DOM 布局并同步触发重渲染。在浏览器执行绘制之前，`useLayoutEffect` 内部的更新计划将被同步刷新。

尽可能使用标准的 `useEffect` 以避免阻塞视觉更新。

## 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。

```
yarn add react-redux
```

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";  
import {bindActionCreators, connect} from  
"../kReactRedux";  
  
// connect用于连接React组件与store， 返回一个新的已经与  
// store连接的组件类（HOC）  
export default connect(  
  // mapStateToProps Fucntion  
  // !慎重定义ownProps，因为你一旦定义ownProps，那么每当  
  // ownProps发生改变的时候，当前的mapStateToProps都会被调用，  
  // !这里的state也会被重新计算，容易影响性能  
  state => {  
    // console.log("mapStateToProps"); //sy-log  
    return {  
      count: state  
    };  
  },  
  // mapDispatchToProps Object Fucntion  
  // Object 此时props中没有dispatch，但是有action  
  // creators，内部实现dispatch  
  // {  
  //   add: () => ({type: "ADD"}),  
  //   minus: () => ({type: "MINUS"})  
  // }
```

```

// }
// Fucntion 参数是dispatch与ownProps
// !慎重定义ownProps, 因为你一旦定义ownProps, 那么每当
ownProps发生改变的时候, 当前的mapStateToProps都会被调用,
容易影响性能
(dispatch, ownProps) => {
  console.log("mapDispatchToProps--", ownProps);
//sy-log
  let creators = {
    add: () => ({type: "ADD"}),
    minus: () => ({type: "MINUS"})
  };
  creators = bindActionCreators(creators,
dispatch);
  return {dispatch, ...creators};
}
)((
class ReactReduxPage extends Component {
  add = () => {
    this.props.dispatch({type: "ADD"});
  };
  render() {
    console.log("props", this.props); //sy-log
    const {count, dispatch, add, minus} =
this.props;
    return (
      <div>
        <h3>ReactReduxPage</h3>
        <p>omg:{count}</p>
        <button onClick={this.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, {useContext, useReducer,
useLayoutEffect} from "react";

const Context = React.createContext();

export const connect = (
  mapStateToProps = state => state,
  mapDispatchToProps
) => WrappendComponent => props => {
  const store = useContext(Context);
  const {dispatch, getState, subscribe} = store;
  const stateProps = mapStateToProps(getState());
  let dispatchProps = {dispatch};
  const [ignored, forceUpdate] = useReducer(x => x +
1, 0);

  if (typeof mapDispatchToProps === "function") {
    dispatchProps = mapDispatchToProps(dispatch);
  } else if (typeof mapDispatchToProps === "object")
  {
```

```

    dispatchProps =
bindActionCreators(mapDispatchToProps, dispatch);
}

useLayoutEffect(() => {
  const unsubscribe = subscribe(() => {
    forceUpdate();
  });
  return () => {
    if (unsubscribe) {
      unsubscribe();
    }
  };
}, [store]);
return <WrappendComponent {...props}
{...stateProps} {...dispatchProps} />;
};

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

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

function bindActionCreators(creators, dispatch) {
  const obj = {};
  for (let key in creators) {
    obj[key] = bindActionCreator(creators[key],
dispatch);
  }
}

```

```
    return obj;
}
```

## react-redux hooks API及实现

useSelector 获取store state

useDispatch 获取dispatch

```
import React, {useCallback} from "react";
import {useSelector, useDispatch} from "react-
redux";

export default function ReactReduxHookPage({value})
{
    const dispatch = useDispatch();
    const add = useCallback(() => {
        dispatch({type: "ADD"});
    }, []);

    const count = useSelector(({count}) => count);

    return (
        <div>
            <h3>ReactReduxHookPage</h3>
            <p>{count}</p>
            <button onClick={add}>add</button>
        </div>
    );
}
```

实现：

```
export function useSelector(selector) {
  const store = useStore();
  const {getState, subscribe} = store;
  const selectedState = selector(getState());

  const [ignored, forceUpdate] = useReducer(x => x +
1, 0);

  useEffect(() => {
    const unsubscribe = subscribe(() => {
      forceUpdate();
    });

    return () => {
      if (unsubscribe) {
        unsubscribe();
      }
    };
  }, [store]);

  return selectedState;
}

export function useDispatch() {
  const store = useStore();
  return store.dispatch;
}

export function useStore() {
  const store = useContext(Context);
  return store;
}
```

```
}
```

## 拓展

### function组件中有类似 `forceUpdate` 的东西吗？

如果前后两次的值相同，`useState` 和 `useReducer` Hook [都会放弃更新](#)。原地修改 `state` 并调用 `setState` 不会引起重新渲染。

通常，你不应该在 React 中修改本地 `state`。然而，作为一条出路，你可以用一个增长的计数器来在 `state` 没变的时候依然强制一次重新渲染：

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

可能的话尽量避免这种模式。

## 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

```
yarn add 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>
        </BrowserRouter>
      </div>
    );
  }
}
```



```
        </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 => (  

```

```

        <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>
    <Router>
      { /* 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>;
  }
}

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  
        : component  
      ? React.createElement(component, props)  
      : render  
      ? render(props)  
      : null  
      : typeof children === "function"  
      ? __DEV__  
        ? evalChildrenDev(children, props, this.props.path)  
        : children(props)  
      : null}  
    </RouterContext.Provider>  
  );
```

## 使用Router

## 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

```
import React from "react";  
import {Route, Redirect} from "react-router-dom";  
import {connect} from "react-redux";  
  
export default connect(({user}) => ({isLogin:  
user.isLogin}))(  
  function PrivateRoute({isLogin, component:  
Component, ...rest}) {  
    return (  
      <Route  
        {...rest}  
        render={props =>  
          isLogin ? (  
            <Component {...props} />  
          ) : (  

```



```

        <Redirect
          to={{
            pathname: "/login",
            state: {redirect:
props.location.pathname}
          }}
        />
      )
    }
  />
);
}
);

```

## 创建LoginPage.js

```

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

export default connect(({user}) => ({isLogin:
user.isLogin}), {
  login: () => ({type: "LOGIN_SUCCESS"})
})(
  class LoginPage extends Component {
    render() {
      const {isLogin, location, login} = this.props;
      const {redirect = ""} = location.state || {};
      if (isLogin) {
        return <Redirect to={redirect} />;
      }
      return (
        <div>

```

```

        <h3>LoginPage</h3>
        <button onClick={login}>login</button>
      </div>
    );
  }
}
);

```

在RouterPage.js配置路由，RouterPage

```

<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 userInit = {
  isLogin: false,
  username: ""
};

// 定义修改规则
export const loginReducer = (state = {userInit},
{type}) => {
  switch (type) {
    case "LOGIN_SUCCESS":
      return {...state, isLogin: true, username:
"xiaoming"};
    default:
      return state;
  }
};

```

```
const store = createStore(combineReducers({user:  
loginReducer}));  
  
export default store;
```

# 回顾

---

## React全家桶02

课堂目标

资源

知识要点

Hooks API

useReducer

useEffect

useLayoutEffect

使用react-redux

API

```
<Provider store>
```

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

```
[mapDispatchToProps], [mergeProps], [options])
```

参数

实现react-redux

react-redux hooks API及实现

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

404页面

路由守卫

回顾

作业

下节课内容

## 作业

---

2. 掌握react-redux。
3. 实现路由守卫的登录和退出登录。

## 下节课内容

---

实现react-router：实现Router、Link、Route、Switch、Redirect和react-router hooks方法。