**React-Redux的用法**

如何在react项目中使用redux。

库react-redux。

实际项目中，应该对于直接使用redux或者使用react-redux进行权衡。

后者虽然提供了便利，但是需要掌握额外的API，并且要遵守它的组建拆分规范。

React-redux将所有组件分成两大类：UI组件 (presentational component) 和容器组件 (container component)。

**UI组件**

UI组件的特征:

**只负责UI的呈现，不带有任何业务逻辑。**

**没有状态（即不使用this.state这个变量）。**

**所有数据都由参数 (this.props)提供。**

**不使用任何Redux的API。**

UI组件例子：

const Title = value => <h1>{value}</h1>

因为不含有状态，UI组件又称为“纯组件”，它像纯函数一样，纯粹由参数决定它的值。

**容器组件**

容器组件的特性。

负责管理数据和业务逻辑，不负责UI的呈现。

带有内部状态。

使用Redux的API。

总而言之，UI组件负责UI的呈现，容器组件负责管理数据和逻辑。

一个组件既有UI又有业务逻辑，将它拆分为下面的结构：

外面是一个容器组件，里面包了一个UI组件。前者负责与外部的通信，将数据传给后者，由后者渲染出视图。

React-Redux规定，所有的UI组件都由用户提供，容器组件则是由React-Redux自动生成。换言之，用户负责视图层，状态管理全部交给它。

**Connect()**

React-redux提供connect方法，用于从UI组件生成容器组件。Connect意为将这两种容器连起来。

import {connect} from 'react-redux';

const VisibleTodoList = connect()(TodoList);

/\*TodoList是 UI 组件，

VisibleTodoList就是由 React-Redux

通过connect方法自动生成的容器组件。\*/

但重要的是定义业务逻辑，否则就只是单纯的包装层。

为了定义业务逻辑，需要给出下面两个方面的信息。

**输入逻辑：外部的数据（即state对象）如何转换为UI组件的参数。**

**输出逻辑：用户发出的动作如何变为action对象，从UI组件传出去。**

完整connect的API如下：

import {connect} from 'react-redux';

const VisibleTodoList = connect(

mapStateToProps,

mapDispatchToProps,

)(TodoList)

/\*connect方法接受两个参数：

mapStateToProps和mapDispatchToProps。

它们定义了 UI 组件的业务逻辑。

前者负责输入逻辑，

即将state映射到 UI 组件的参数（props），

后者负责输出逻辑，

即将用户对 UI 组件的操作映射成 Action。\*/

**mapStateToProps()**

mapStateToProps是一个函数。它的作用就像它的名字那样，建立一个从（外部的）state对象到（UI组件的）props对象的映射关系。

作为函数，mapStateToProps执行后应该返回一个对象，里面的每一个键值对就是一个映射。

const mapStateToProps = (state)=>{

return{

todos:getVisibleTodos(state.todos,state.visibilityFilter)

}

}

/\*mapStateToProps是一个函数，

它接受state作为参数，返回一个对象。

这个对象有一个todos属性，代表 UI 组件的同名参数，

后面的getVisibleTodos也是一个函数，

可以从state算出 todos 的值。\*/

//下面就是getVisibleTodos的一个例子，用来算出todos。

const getVisibleTodos = (todos, filter) => {

switch (filter) {

case 'SHOW\_ALL':

return todos

case 'SHOW\_COMPLETED':

return todos.filter(t => t.completed)

case 'SHOW\_ACTIVE':

return todos.filter(t => !t.completed)

default:

throw new Error('Unknown filter: ' + filter)

}

}

mapStateToProps会订阅store，每当state更新的时候，就会自动执行，重新计算UI组件的参数，从而触发UI组件的重新渲染。

mapStateToProps组件的第一参数总是state对象，还可以使用第二个参数，代表组件的props对象。

// 容器组件的代码

// <FilterLink filter="SHOW\_ALL">

// All

// </FilterLink>

const mapStateToProps = (state, ownProps) => {

return {

active: ownProps.filter === state.visibilityFilter

}

}

/\*使用ownProps作为参数后，

如果容器组件的参数发生变化，也会引发 UI 组件重新渲染。\*/

Connect方法可以省略mapStateToProps参数，那样的话UI组件就不会订阅store，就是说store的更新不会引起UI组件的更新。

**mapDispatchToProps()**

mapDispatchToProps是connect函数的第二个参数，用来建立UI组件的参数到store.dispatch方法的映射。也就是说，它定义了哪些用户的操作应该当作action，传给store。它可以是一个函数，也可以是一个对象。

mapDispatchToProps如果是一个函数，会得到dispatch和ownProps（容器组件的props对象）两个参数。

const mapDispatchToProps = (

dispatch,

ownProps

)=>{

return {

onClick:()=>{

dispatch({

type:'SET\_VISIBILITY\_FILTER',

filter:ownProps.filter;

})

}

};

};

从上面代码可以看到，mapDispatchToProps作为函数，应该返回一个对象，该对象的每个键值对都是一个映射，定义了UI组件的参数怎样发出action。

如果mapDispatchToProps是一个对象，它的每个键名也是对应UI组件的同名参数，键值应该是一个函数，会被当作action creator，返回的action会由redux自动发出。举例来说：

const mapDispatchToProps = {

onClick:(filter)=>{

type:'SET\_VISIBILITY\_FILTER',

filter:filter

};

}

**Provider组件**

Connect方法生成容器组件以后，需要让容器组件拿到state对象，才能生成UI组件的参数。

一种解决方法是将state对象作为参数，传入容器组件。但是这样做比较麻烦，尤其是容器组件可能在很深的层级，一级级将state传下去就很麻烦。

React-Redux提供provider组件，可以让容器组件拿到state。

import {Provider} from 'react-redux'

import {createStore} from 'redux'

import todoApp from './reducer'

import App from './component/App'

let store = createStore(todoApp);

render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

/\*Provider在根组件外面包了一层，这样一来，

App的所有子组件就默认都可以拿到state了。\*/

Provider原理来自于react组件的context属性：

class Provider extends Component{

getChildContext(){

return{

store:this.props.store

};

}

render(){

return this.props.children;

}

}

Provider.childContextTypes = {

store:React.ProTypes.object

}

/\*store放在了上下文对象context上面。

然后，子组件就可以从context拿到store\*/

class VisibleTodoList extends Component{

componentDidMount(){

const {store} = this.context;

this.unsubscribe = store.subscribe(()=>

this.forceUpdate()

);

}

render(){

const props = this.props;

const {store} = this.context;

const state = store.getState();

//.....

}

}

VisibleTodoList.contextTypes = {

store:React.PropTypes.object

}

/\*React-Redux自动生成的容器组件的代码，

就类似上面这样，从而拿到store。\*/

**实例：计算器**

class Counter extends Component{

render(){

const {value,onIncreaseClick} = this.props;

return(

<div>

<span>{value}</span>

<button onClick={onIncreaseClick}>Increase</button>

</div>

)

}

}

///纯UI组件,该组件UI有两个参数value和onIncreaseClick。

//value需要从state计算得到，onIncreaseClick需要向外发出action

//value到state的映射

function mapStateToProps(state){

return{

value:state.count;

}

}

function mapDispatchToProps(dispatch){

return{

onIncreaseClick:()=>dispatch(increaseAction)

}

}

//action creator

const increaseAction = {type:'increase'}

//connect生成容器组件

const App = connect(

mapStateToProps,

mapDispatchToProps)(Counter)

//定义这个组件的reducer

function counter(state={count:0},action){

const count = state.count

switch(action.type){

case 'increase':

return {count:count+1}

default:

return state

}

}

import { loadState, saveState } from './localStorage';

const persistedState = loadState();

const store = createStore(

todoApp,

persistedState

);

store.subscribe(throttle(() => {

saveState({

todos: store.getState().todos,

})

}, 1000))

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);