🡪

react-redux在redux的基础上提供了额外的API与React协同工作。例如connect() 方法，Provider组件，mapStatetoProps(), mapDispatchtoProps()等。

🡪

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 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。

**UI 组件**有以下几个特征。

--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

下面就是一个 UI 组件的例子。

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

因为不含有状态，UI 组件又称为"纯组件"，即它纯函数一样，纯粹由参数决定它的值。

**容器组件**的特征恰恰相反。

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

如果一个组件既有 UI 又有业务逻辑，那怎么办？回答是，将它拆分成下面的结构：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并且将数据传给后者UI组件，由后者UI组件渲染出视图。

React-Redux 规定，所有的 UI 组件都由用户提供也就是由我们编程实现，容器组件则是由使用connect返回的组件，由React-Redux 自动生成。也就是说，用户负责视觉层，状态管理则是全部交给它。

🡪

Connect()方法

React-Redux 提供connect方法, connect的意思，就是将这两种组件连起来, 生成容器组件并且映射到（To）UI组件的props.

例如：

import { connect } from 'react-redux'

const VisibleTodoList = connect (

mapStateToProps,

mapDispatchToProps) (TodoList)

上面代码中，connect方法接受两个参数，这两个参数是方法的引用：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(TodoList)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，dispatch出去给store中的reducer进行处理。Connect方法生成VisibleTodoList为容器组件包裹着我们用户提供的UI组件TodoList。

P.s connect方法用于将**一个**React组件连接到React-Redux中。**所以如果有多个组件，多个组件都需要从store中获取state，或者dispatch action去改变state,则每一个组件都需要执行connect方法**。

**mapstatetoProps参数🡪只能为方法**

mapStateToProps是一个函数。它的作用就是像它的名字那样，建立一个从Redux的state库中到（UI组件的）props对象的映射关系。

我们需要自行定义这个函数，要求是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。

这个函数的第一个参数将代表整个Redux中的state.我们定义的这个函数将会从参数接收的整个Redux中的state，通过返回对象表示组件对store中有关系的state.

例如：

const mapStateToProps = (state) => { # 此时参数中传入的state是整个Redux中的state

return {

todos: getVisibleTodos(state.todos, state.visibilityFilter) # 返回的对象中表示未来的组件只需要其中的todos。

}

}

返回的对象中todos将作为UI组件中的同名属性this.props.todos. getVisibleTodo是一个功能函数用于执行逻辑算出结果。

mapStateToProps会让组件订阅 Redux State Store，每当组件所注册的state更新的时候，就会自动执行一次mapstatetoProps，重新计算 UI 组件的参数，从而UI组件获取新的props, 从而触发 UI 组件的重新渲染。基础此，所定义的mapStateToProps函数执行速度要快。

mapStateToProps的第一个参数总是state对象，还可以使用第二个参数，代表容器组件的props对象。

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参数🡪可为函数可为对象**

mapDispatchToProps是connect函数的第二个参数，用来建立 UI 组件的props将触发store.dispatch映射。

**首先connect方法可以不传入mapDispatchToProps,不使用mapDispatchToProps**

例如：

connect () (MyComponent)

// which is equivalent with

connect (

null,

null

) (MyComponent)

// or

connect (mapStateToProps /\*\* no second argument \*/) (MyComponent)

如果不传入mapDispatchToProps的情况下，进行和Redux的connect. 组件将会获取props.dispatch。所以如下组件编写

function Counter ({count, dispatch}) {

return (

<div>

<button onClick= {() => dispatch ({type: 'DECREMENT’})} >-</button>// 此按钮触发的事件将dispatch一个type=DESCREMNT的action

<span>{count}</span>

<button onClick= {() => dispatch ({type: 'INCREMENT’})} >+</button>

<button onClick= {() => dispatch ({type: 'RESET’})} >reset</button>

</div>

)

}

**如果使用mapDispatchToProps的话：**

它可以是一个函数，也可以是一个对象。函数形式可以提供更灵活的个性化设置，还可以在函数体中获取ownProps，对象的形式更加直接，简单使用。

**mapDispatchToProps(函数形式 – 手动定义)**

函数形式的mapDispatchToProps提供两个参数，第一个是dispatch,第二个是可选参数ownProps.

函数形式的mapDispatchToProps应该返回一个对象，该对象的key将作为以后组件的props,对应的值是一个将会dispatch action的函数。（因为组件中的按钮onclick将会调用this.props.key，那么这个触发事件理应是一个可以出发dispatch action的函数）

例如：

// 定义mapDispatchToProps

const mapDispatchToProps = dispatch => {

return {

// dispatching plain actions

increment: () => dispatch ({type: 'INCREMENT’}),

decrement: () => dispatch ({type: 'DECREMENT’}),

reset: () => dispatch ({type: 'RESET’})

}

}

function Counter ({count, increment, decrement, reset}) {

return (

<div>

<button onClick={decrement}>-</button> //组件使用绑定并发出dispatch action

<span>{count}</span>

<button onClick={increment}>+</button>

<button onClick={reset}>reset</button>

</div>

)

}

**mapDispatchToProps(函数形式 – 使用bindActionCreators定义)**

bindActionCreators由Redux提供，帮助我们更方便的进行映射。

bindActionCreators接收两个参数，第一个参数是action creator(返回action的方法)或者对象。第二个参数是dispatch.

例如：

//在mapDispatchToProps应用

import { bindActionCreators } from 'redux'

// ...

const increment = () => ({ type: 'INCREMENT' })

const decrement = () => ({ type: 'DECREMENT' })

const reset = () => ({ type: 'RESET' })

function mapDispatchToProps(dispatch) {

return bindActionCreators({ increment, decrement, reset }, dispatch)

}

// component receives props.increment, props.decrement, props.reset

connect (

null,

mapDispatchToProps

) (Counter)

**mapDispatchToProps(对象形式)**

如果mapDispatchToProps是一个对象，它的每个键名也是对应 UI 组件的同名props参数，键值应该是一个函数，会被当作 Action creator 函数，所以这个函数返回的 Action将是一个对象起码包含type的键，并且Action会由 Redux 自动发出。

例如：

const mapDispatchToProps = {

onClick: (filter) => {

type: 'SET\_VISIBILITY\_FILTER',

filter: filter

};

}

**并且如果我们直接传入一个对象，每个对象的元素都是*action creator*，connect方法将自动为我们调用bindActionCreators。(经常使用的方式)**

例如：

import {increment, decrement, reset} from "./counterActions";//引入我们编写的三个action creator方法

export default connect (

mapState,

{increment, decrement, reset} //直接作为mapDispatchToProps便可以

) (Counter);

🡪

<Provider>组件

<Provider>组件为**最外层的用户自定义组件**包裹一层。

**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。**

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 './reducers'

import App from './components/App'

let store = createStore(todoApp);

render (

<Provider store={store}> //store=store可以使用接下来所有组件以及子组件都能获取store中state

<App /> //这是通过connect方法使得我们编写的UI组件与Redux联系以后产生的容器组件

</Provider>,

document.getElementById('root')

)

🡪

使用React-Redux的React项目文件夹结构例子：

建立components文件夹，里面每一个js文件是每一个React控件，并且在每一个React控件中如果控件需要改变Redux中的state或者自身根据Redux的state改变时候，需要调用connect(mapStatesToProps, mapDispatchToProps)(本组件名)方法，连接Redux.本组件便可以通过this.props.xxx调用 mapStatesToProps以及mapDispatchToProps中的属性以及action creator方法。

mapDispatchToProps由于是action creator所以我们经常从actions.js定义并且引入到控件中使用。并且在connect方法作为mapDispatchToProps一部分调用中传入。

从actions.js引入的action creator在控件中使用有两种风格。

如果是使用extends React.Component产生的控件那么使用action creator的时候，由于使用的React-Redux所以调用为this.props.[action creator].

如果是使用纯函数申明的控件，由于没有this关键字使用，所以采用直接调用方法。不管采用哪种风格的控件申明，最后都会使用在connect方法作为mapDispatchToProps一部分调用中传入。

建立Redux文件夹，集中放置与React-Redux相关的文件。

包括reducers文件夹，文件夹中每一个js文件是一个reducer. Reducer方法接受两个参数state现在初始的状态以及action接受的action, 根据一定逻辑后以对象的形式返回新的状态。

包括actions.js文件，其中写明了action creator. Action creator方法用于返回一个对象表示action, 将对象必须包含type关键字。

包含store.js文件，包含了使用createStore方法建立项目中唯一一个React-Redux的Store.

包含index.js文件，其中将会包括对最外层的组件使用Provider控件（react-redux引入）进行包括，并且使用我们在store.js中建立的store。

🡪