🡪

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。

🡪

ComponentB = connect(mapStateToProps, mapDispatchToProps)(ComponentA)

这是将一个UI组件ComponentA连接到Redux最常用语句。通过语句，将UI组件ComponentA经过mapStateToProps,以及mapDispatchToProps映射，生成容器组件ComponentB.容器组件也可以称为HOC(High-level Component).页面渲染时候，直接渲染ComponentB,他的展示层将由UI组件ComponentA决定，并且经过映射，ComponentA在Redux dataStore的状态改变时候，显示最新状态。Component也可以通过function props来改变Redux dataStore的状态。

语句可以分为三个部分。

第一部分为左边ComponentB为生成的容器组件，也是HOC(High Order Component)组件。用于最终渲染在页面上。这个HOC将获取mapStateToProps, mapDispatchToProps的props, 并且由于HOC组建的特性，他将这些props添加给其底层的UI组件。

第二部分Connect(mapStateToProps, mapDispatchToprops)为connector连接器，用于将一个UI组件连接到Redux dataStore中。编程时候，为了减少代码重复，我们可以编码一些通用的connector （通用的connector一般就是一个方法，在其中定义如何mapStateToProps, mapDispatchToProps,甚至mergeProps, 调用connector方法的时候，传入需要连接的UI组件以及一些额外参数，connect方法返回一个HOC容器组件。获得该容器组件后，我们可以用于渲染）,为传入的UI组件提供连接Redux的dataStore.

第三部分ComponentA为我们编写的UI组件。

🡪

**Connect()方法**

Redux将组件分为两种类型。容器组件以及UI组件。

UI组件用于接受props, 从而显示数据并且根据props中的回调函数对用户的操作进行反馈。UI组件不与dataStore进行直接连接。它的props由容器组件提供。

容器组件直接和dataStore相连，并且将相连以后获取的data作为props传给UI组件。

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dataStore中的state映射到 UI 组件(TodoList)的属性（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。最后，我们需要指定用户的哪个组件需要进行Connect方法连接。此处为TodoList。 从而connect方法生成VisibleTodoList的容器组件包裹着我们用户提供的UI组件TodoList。在页面渲染时，只需要渲染生成的容器组件便可。

Connect方法完整模式为connect (mapStateToProps, mapDispatchToProps, mergeProps, options)

一般情况下我们只需要传入mapStateToProps,以及mapDispatchToProps,特殊情况下我们还以传入mergeProps以及options.

mergeProps效果是指定一个方法，该方法接受dataProps, functionProps, ownProps参数。分别对应着mapStateToProps的prop对象, mapDispatchToProps的prop对象, 以及容器组件的prop对象. 此函数要求返回一个对象，该对象是最终UI组件接受的prop整体对象。用处在于我们可以在dataProps, functionProps, ownProps的基础上，根据我们的需求最终改变传给UI组件的props.

例如：我们可以覆盖某些props.例如mapStateToProps中有prop Name, ownProps中也有prop Name,则如果mergeProps方法中最终返回的对象为{…dataProps, functionProps, ownProps}则ownProps中的Name会覆盖mapStateToProps中的Name,使得最终UI组件获得prop Name值为ownProps中的prop Name值。

例如：我们可以添加某prop.例如

const mergeProps = (dataProps, functionProps, ownProps) => {

let routedDispatchers = { //构建一个routedDispatchers prop对象，使用functionProps

cancenCallback: ()=> ownProps.history.push(`/${dataType}`), //添加cancelCallbackProp

saveCallback: (data) =>{ //基于functionProps改写saveCallback Prop

functionProps.saveCallback(data)

ownProps.history.push(`/${dataType}`)

}

}

return Object.assign({}, dataProps, routedDispatchers, ownProps); //最后传给UI组建的整体Prop对象

}

Options参数，options参数要求传入一个对象。

|  |  |
| --- | --- |
| Name | Description |
| pure | 默认为true.true的意思是当dataStore的中的组件prop对应的state变化时候，才会引起组件的刷新。否则阻止无必要的更新。当设置为false时候，表示该组件还用到了其余非prop所对应的state, 从而不会阻止任何更新。 |
| areStatePropsEqual | 当pure设置为true时，可以传入自定义方法用于覆盖mapStateToProps中比较是否相等的方法。 |
| areOwnProsEqual | 当pure设置为true时，可以传入自定义方法用于覆盖mapDispatchToProps中比较是否相等的方法。 |
| areMergedPropsEqual | 当pure设置为true时，可以传入自定义方法用于覆盖mergeProps中比较是否相等的方法。 |
| areStatesEqual | 当pure设置为true时，可以传入自定义方法用于覆盖mergeProps, mapDispatchToProps, mergeProps中比较是否相等的方法。 |

P.s connect方法用于将**一个**React组件连接到React-Redux中。**在一个APP中如果有多个组件，每个组件都需要从store中获取state作为自身的props，或者dispatch action去改变dataStore中的state,则每一个组件都需要执行connect方法**。**虽然我们通过<Provider>对整个App进行包裹，这样只是为所有在APP中所有的组件提供了访问Redux dataStore。但是每个组件的data prop, action prop如果需要从dataStore中获取，还需要每个组件都执行connect方法，进行与dataStore中的映射。**

**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。**并且key将会是未来组件的prop, value是Redux datastore中某个state的值。**

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: state.todos # 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.

mapStateToProps会让组件订阅 Redux State Store，每当组件props所对应的dataStore中的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ownProps，代表**未来容器组件的props对象**。引入该参数的目的是，当我们编码通用连接器connector时，将接受各式各样的UI组件。为了让UI组件减少不必要的更新，也就是减少从Redux dataStore映射到UI组件的props, 提供灵活性, 接受未来容器组件的props对象。当页面渲染未来容器组件的时候，为容器组件指定不同的props, 从而可以根据未来容器组件的props对象，改变连接器connector中的mapStateToProps, mapDispatchToProps的映射行为。

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function 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. Dispatch将作为一个function props,注入到组件当中。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，例如在函数体中获取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function props,对应的值是一个将会dispatch action的函数，也就是action creator函数。当组件中元素被点击或交互以后，调用action creator，发出一个action让reducer进行处理改变redux中的状态。进而由于redux状态的改变，改变UI的表现形式，而不是直接像以前Jquery那样触发事件以后直接修改UI。

🡪手动指定可以让一个props事件，触发多个action.然后通过dispatch(Action)传递到reducer中处理。

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，键值应该是预先定义的Action Creator 函数**，会被定义在redux的Action Creator函数JS文件中。定义的action creator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。所以通常对App进行包裹。

**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')

)

🡪

Reducers

Reducers的作用有两个，一个处理接收到的action, 并且返回新的dataStore状态。二是处理为dataStore指定初始状态。指定dataStore的初始状态时候，一般有两种方式。

第一种使用Javascript默认参数的方法例如：

export default function(storeData=initialData.modelData, action){ //通过指定storeData=initialData.modelData当App开始运行时候,无action时候，默认StoreData便等于initialData.modelData

switch(action.type){

case STORE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].concat([action.payload])

}

case UPDATE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].map(p => p.id === action.payload.id? action.payload: p)

}

case DELETE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].filter(p => p.id !== action.payload)

}

}

}

第二种使用switch的default默认方式：

export default function(storeData, action){

switch(action.type){

case STORE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].concat([action.payload])

}

case UPDATE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].map(p => p.id === action.payload.id? action.payload: p)

}

case DELETE:

return {

...storeData,

[action.dataType]: storeData[action.dataType].filter(p => p.id !== action.payload)

}

default:

return storeData || initialData.modelData; //不指定默认storeData而是在APP开始后，由于无action所以默认处理返回initialData.modelData

}

}

🡪

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. 如果有多个js文件，便代表有多个reducer. 每当一个action发出，每一个reducer都会接收到该action,直到其中一个reducer返回一个新的state对象。并且由于每个reducer都会提供初始状态，所以相当于每个reducer都会提供一部分初始状态给dataStore. 含有多个reducer时候，使用react-redux提供的combineReducer来合成所有的reducer.combineReducer接受一个对象作为参数，该对象的key为datastore中对应的属性，值为负责的reduer.

每个Reducer方法都接受两个参数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。

🡪

React中UI组件通过连接Redux, 可以获取props. 这些props有可能是数据有可能是Action Creators. 这些放在Redux Datastore中的内容我们称为modal data.

与此同时，React组件内部还可以保留自身的state，props以及方法，用于控制组件不同显示方式等，我们称之为state data.甚至我们可以把这些state data也放入redux中。

🡪

React中的UI组件连接Redux时，我们可以检查该UI组件需要哪些Props.有一些是数据props, 有一些是function props. 这些props应该由Redux提供，数据props应该出现在mapStateToProps的key中以及function props应该出现在mapDispatchToProps的Key中。

🡪

Redux datasotre额外API

dataStore = createStore (reducer, [initialState], enhancer)

createStore方法接受三个参数。

**第一个参数reducer**,对于这个参数我们可以使用reducer enhancer函数。Redux只允许使用一个reducer.所以如果有多个reducer则我们需要使用Redux自带的combineReducers函数先将他们合成一个reducer对象，再传入createStore中。同理，当合成一个reducer后，可以在整个整体的reducer上采用reducerEnhancer对其进一步加工。

例如：

自定义一个ReducerEnhancer函数：

export function customReducerEnhancer(originalReducer){

let initialState = null;

return (storeData, action)=> {

if (action.type === STORE\_RESET && initialData != null){

return initialState

}

else{

const result = originalReducer(storeData, action);

if(initialState === null){

initialState = result

}

return result

}

}

}

应用ReducerEnhancer函数:

const enhancedReducer = customReducerEnhancer( //应用自定义的ReducerEnhancer函数

combineReducers({ //也应用了Redux自带的ReducerEnhancer函数combineReducers

modelData:modelReducer,

stateData:stateReducer

})

)

**第二个参数初始化状态。**

初始化状态可以在createStore的第二个参数指定，也可以在Reducer中进行指定。如果都指定则以Reducer中指定的初始化状态为准。所以一般我们不再createStore中指定，而是在Reducer中指定。

**第三个参数enhancer function**

第三个参数enhancer function包括使用中间件applyMiddleware,或者一些自定义的enhancer function.需要注意的是createStore只允许使用一个enhancer function.所以如果只是采用中间件，那么直接采用createStore(reducer,{}, applyMiddleware(…))便可。但是如果我们除了applyMiddleware这个enhancer function以外，还需要应用其他的enhancer function时候，我们需要使用Redux自带的compose函数，将多个enhancer function合成一个以后，再传入createStore函数（此规定和只能传入一个Reducer到createStore相似）。

例如：

经过createStore方法后返回的对象dataStore拥有：

|  |  |
| --- | --- |
| 属性 | 效果 |
| getState() | 以对象的形式返回dataStore中的整体状态 |
| subscribe(回调函数) | 每当Redux的dataStore发生改变时侯，将调用Subscribe中传入的回调函数。并且Subscribe(回调函数)， 调用该方法可以取消订阅消息。 |
| dispatch(action对象) | 传递一个action对象到Redux的Reducer中 |