<http://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_one_basic_usages.html>

见素材redux

基本用法

二、Redux**设**计思想

（1）Web 应用是一个状态机，视图与状态是一一对应的。

（2）所有的状态，保存在一个对象里面。

三、基本概念和 API

3.1 **Store**

Store就是保存数据的地方。整个应用只能有一个Store。

Redux 提供createStore函数，用来生成 Store。

import { createStore } from 'redux';

const store = createStore(reducer);

3.2 **State**

Store快照生成某个时刻的数据，叫做State。

Redux 规定， 一个 State 对应一个 View。

当前时刻的 State，可以通过store.getState()得到。

const state = store.getState();

3.3 **Action**

View发出Action，使State改变。

Action 是一个对象。其中的type属性是必须的，表示 Action 的名称。其他属性可以自由设置。

const action = {

type: '',

};

3.4 Action **Creator**

View 要发送多少种消息，就会有多少种 Action。如果都手写，会很麻烦。可以定义一个函数来生成 Action，这个函数就叫 Action Creator。

3.5 store.**dispatch**()

store.dispatch()是View发出Action的唯一方法。接受一个 Action 对象作为参数

3.6 **Reducer**

Reducer 是一个函数，接受当前State和Action作为参数，返回一个新的 State。

const reducer = function (state, action) {

return newState;

};

store.dispatch()会触发 Reducer 的执行

3.7 **纯**函数

Reducer 函数是一个纯函数。即只要是同样的输入，必定得到同样的输出

纯函数是函数式编程的概念，遵守以下约束：

不能调用系统 I/O 的API

不得改写参数

不能调用Date.now()或者Math.random()等不纯的方法，因为每次会得到不一样的结果

3.8 store.**subscribe**()

store.subscribe()设置监听函数，一旦 State 发生变化，就执行这个函数。

store.subscribe()返回一个函数，调用这个函数就可以解除监听。

四、Store 的**实**现

createStore方法还可以接受第二个参数，表示 State 的最初状态，通常是服务器给出。如果接受第二个参数，它会覆盖 Reducer 函数的默认初始值。

let store = createStore(reducer, window.state\_from\_server)

五、Reducer 的**拆**分

Redux 提供combineReducers方法，用于 Reducer 的拆分

可以把所有子 Reducer 放在一个文件里面，然后统一引入。

import { combineReducers } from 'redux'

import \* as reducers from './reducers'

const reducer = combineReducers(reducers)

六、工作**流**程



中间件与异步操作

二、**中**间件的用法

applyMiddlewares()是 Redux 的原生方法，作用是将所有中间件组成一个数组，依次执行。有的中间件有次序要求。

import {createStore, applyMiddleware} from 'redux';

const store = createStore(

reducer,

window.state\_from\_server,

applyMiddleware(中间件1,中间件2,中间件3……)

);

三、applyMiddlewares()

四、**异**步操作的基本思路

同步操作只要发出一种 Action 即可，异步操作的差别是它要发出三种 Action：

操作发起时的 Action

操作成功时的 Action

操作失败时的 Action

三种 Action 可以有两种不同的写法：

// 写法一：名称相同，参数不同

{ type: 'fetch\_posts' }

{ type: 'fetch\_posts', status: 'error', error: '' }

{ type: 'fetch\_posts', status: 'success', response: { ... } }

// 写法二：名称不同

{ type: 'fetch\_posts' }

{ type: 'fetch\_posts\_failure', error: '' }

{ type: 'fetch\_posts\_success', response: { ... } }

整个异步操作的思路：

操作开始时，送出一个 Action，触发 State 更新为"正在操作"状态，View 重新渲染

操作结束后，再送出一个 Action，触发 State 更新为"操作结束"状态，View 再一次重新渲染

五、redux-**thunk** 中间件

import thunk from 'redux-thunk';

const store = createStore(

reducer,

applyMiddleware(thunk)

);

用于store.dispatch()。以下暂时不背

store.dispatch(fetchPosts('fetchPostsParam'));

const fetchPosts =

fetchPostsParam => (dispatch, getState) => {

dispatch(requestPosts(fetchPostsParam));

return fetch(`/someURL/${fetchPostsParam}`)

.then(response => dispatch(receivePosts(fetchPostsParam, response)));

};

};

六、redux-**promise** 中间件

import promiseMiddleware from 'redux-promise';

const store = createStore(

reducer,

applyMiddleware(promiseMiddleware)

);

用于store.dispatch()。以下暂时不背

const fetchPosts =

(dispatch, fetchPostsParam) => new Promise(function (resolve, reject) {

dispatch(requestPosts(fetchPostsParam));

return fetch(`/someURL/${fetchPostsParam}`)

.then(response => {

type: 'fetch\_posts',

payload: response

});

});

**React-Redux** 的用法

一、UI 组件

React-Redux 将所有**组**件分成两大类：UI 组件和容器组件。

UI 组件**特**征：

**负**责 UI 的呈现，不负责管理数据和业务逻辑

**没**有状态，即不使用this.state这个变量

**不**使用 Redux 的 API

**所**有数据都由参数this.props提供

二、容器组件

容器组件特征：

不负责 UI 的呈现，负责管理数据和业务逻辑，

有状态

使用 Redux 的 API

React-Redux **规**定，UI 组件由用户提供，容器组件由 React-Redux 生成。

三、**connect**()

React-Redux 提供connect方法，用于从 UI 组件生成容器组件。

import { connect } from 'react-redux'

const App = connect(

mapStateToProps,

mapDispatchToProps

)(App的UI组件)

四、map**StateTo**Props()

mapStateToProps将store的state映射到UI组件的props。

第一个参数总是store的state，还可以有第二个参数，代表容器组件的props。

当store的state或容器组件的props发生变化，都会引发 UI 组件重新渲染

五、map**DispatchTo**Props()

mapDispatchToProps将对UI组件的操作映射成store.dispatch

可以是一个函数。以下暂时不背

const mapDispatchToProps = (dispatch,ownProps) => {

return {

onClick: () => {

dispatch({

type: '',

filter: ownProps.filter

});

}

};

}

也可以是一个对象。以下暂时不背

const mapDispatchToProps = {

onClick: (filter) => {

type: '',

filter: filter

};

}

六、<Provider> 组件

React-Redux **提**供Provider组件，可以让容器组件拿到store的state。

import { Provider } from 'react-redux'

render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

七、实例：**计**数器，以下暂时不背

import React, { Component } from 'react'

import PropTypes from 'prop-types'

import ReactDOM from 'react-dom'

import { createStore } from 'redux'

import { Provider, connect } from 'react-redux'

// React component

class Counter extends Component {

render() {

const { value, onIncreaseClick } = this.props

return (

<div>

<span>{value}</span>

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

</div>

)

}

}

Counter.propTypes = {

value: PropTypes.number.isRequired,

onIncreaseClick: PropTypes.func.isRequired

}

// Action

const increaseAction = { type: 'increase' }

// Reducer

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

const count = state.count

switch (action.type) {

case 'increase':

return { count: count + 1 }

default:

return state

}

}

// Store

const store = createStore(counter)

// Map Redux state to component props

function mapStateToProps(state) {

return {

value: state.count

}

}

// Map Redux actions to component props

function mapDispatchToProps(dispatch) {

return {

onIncreaseClick: () => dispatch(increaseAction)

}

}

// Connected Component

const App = connect(

mapStateToProps,

mapDispatchToProps

)(Counter)

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

八、React-Router **路**由库

const Root = ({ store }) => (

<Provider store={store}>

<Router>

</Router>

</Provider>

);

设存状动创发减 纯签实拆流中异铛诺

结组特负没不所规连 状发到提计路