Redux

React只是DOM的一个抽象层，并不是web应用的完整解决方案，没办法些大型应用。

2014 flux

2015 flux+函数式编程 Redux

Redux是一个有用的架构，但不是必须的。大多数情况。React就够了。

UI层非常简单，没有很多互动，Redux是不必要的，用了反而增加了复杂性。

用户的使用方式非常简单

用户之间没有协作

不需要与服务器大量交互，也没有使用webSocket

视图层只从单一来源获取数据

使用Redux的场景：

用户的使用方式负责

不同身份的用户有着不同的使用方式（如普通用户与管理员）

多个用户之间相互协作

与服务器大量交互，或者使用了webSocket

View要从多个来源获取数据

多交互，多数据源

从组件的角度来看：

某个组件的状态，需要共享

某个状态需要在任何地方都可以拿到

一个组件需要改变全局状态

一个组件需要改变另一个个组件的状态

一种机制：可以在同一个地方查询状态，改变状态，传播状态的变化。

Redux只是web架构的一种解决方案。

设计思想

Web应用是一个状态机，视图与状态是一一对应的。

所有的状态，保存在一个对象里面。

Store

Store就是保存数据的地方，你可以把它看作一个容器。整个应用只能由一个store

生成store

import {createStore} from 'redux';

const store = createStore(fn);

Redux提供createStore这个函数来生成store。createStore函数接受另一个函数做为参数，返回新生的store对象。

State

Store对象包含所有数据。如果想得到某个时点的数据，就要对store生成快照，这种时点的数据集合，就叫做state。

当前时刻的state，可以通过store.getState()得到

import {createStore} from 'redux'

const store = createStore(fn);

const state = store.getState();

Redux规定，一个state对应一个view。只要state相同，view就相同。反之亦然。由state状态就能知道view是什么样

Action

State的变化会导致view的变化。但是用户接触不到state，只能接触到view。所以state的变化必须是view变化所导致的。Action就是view 发出的通知，表示state应该要发生变化了。

Action是一个对象。其中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:'ADD\_TODO', //名称

payload:'Learn Redux' //所携带信息

};

Action描述当前发生的事情。改变state的唯一方法，就是使用action。它会运送数据到store。

Action Creator

View要发送多少种信息，就会有多少种Action。定义一个函数来生成action，函数名为action creator

const ADD\_TODO = '添加 TODO';

function addTodo(text){

return{

type:ADD\_TODO,

text

}

}

const action = addTodo('learn redux')

addTodo函数就是一个action creator。

**store.dispatch()**

store.dispatch()是view发出action的唯一方法。

import {createStore} from 'redux';

const store = createStore(fn);

store.dispatch({

type:'ADD\_TODO'，

payload:'learn redux'

})

//store.dispatch()是view发出actionde 唯一方法

该代码， store.dispatch接受一个action对象作为参数，将它发送出去。

结合action creator方法。

function addTodo(text){

return{

type:ADD\_TODO,

text

}

}

store.dispatch(addTodo('learn redux'));

Reducer

Store收到action后，必须给出一个新的state，这样view才会发生变化。这种state的计算过程叫做reducer。

Reducer是一个函数，它接受action和当前state作为参数，返回一个新的state。

const defaultState = 0;

const reducer = (state = defaultState,action) => {

switch:(action.type){

case 'ADD':

return state+action.payload;

default:

return state;

}

}

const state = reducer(1,{

type:'ADD',

payload:2

});

上面代码中，reducer函数收到名为ADD的action以后，就返回一个新的state，作为加法的计算结果。其他运算的逻辑实现类似。

实际应用中，reducer函数不用手动调用，store.dispatch方法会触发reducer的自动执行。为此，store需要知道reducer函数，做法就是在生成store的时候将reducer传入createStore方法

import {createStore} from 'redux';

const store = createStore(reducer);

createStore接受reducer作为参数，声称一个新的store。以后每当store.dispatch发送过来一个新的action，就会自动调用reducer，得到新的state。

函数为什叫做reducer。因为它可以作为数组的reduce方法的参数。

const actions = [

{type:'ADD',payload:0},

{type:'ADD',payload:1},

{type:'ADD',payload:2},

];

const total = actions.reduce(reducer,0);//3

//数组的reduce方法接受 Reducer 函数作为参数，

/就可以直接得到最终的状态3。

纯函数

Reducer函数最重要的特征，它是一个纯函数。相同输入必定得到相同输出。

纯函数是函数式编程的概念，必须遵守以下的一些约束。

**不得改写参数。**

**不能调用系统I/O的APT**

**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**

// State 是一个对象

function reducer(state, action) {

return Object.assign({}, state, { thingToChange });

// 或者

return { ...state, ...newState };

}

// State 是一个数组

function reducer(state, action) {

return [...state, newItem];

}

因此reducer函数内部不能改变state。最好把state对象设置为只读，你没法改变它。要得到新的state，唯一办法就是生成一个新对象。这样的好处是，任何时候，与某个view对应的state总是一个不变的对象。

**store.subscribe**

store允许使用store.subscribe方法设置监听函数，一旦state发生变换，该函数也随之执行。

import {createStore} from 'redux';

const store = createStore(reducer);

store.subscribe(listener);

显然，把view的更新函数 (对于react项目，就是组件的render方法或者是setState方法)放入listen，就会实现view的自动渲染。

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

let unsubscribe = store.subscribe(

()=>

console.log(store.getState())

);

unsubscribe();

//unsubscribe = store.subscribe()

Store的实现

Store提供了三种方法。

* store.getState()
* store.dispatch()
* store.subscribe()

/\*store.getState();

store.dispatch()

store.subscribe();\*/

import {createStore} from 'redux';

let {subscribe,dispatch,getState} = createStore(reducer);

createStore方法还可以接受第二个参数，表示state的最初状态。这通常是服务器给出的。

let store = createStore(todoApp, window.STATE\_FROM\_SERVER)

window.STATE\_FROM\_SERVER就是整个应用的状态初始值。注意，如果提供了这个参数，它会覆盖reducer函数的默认初始值。

下面是一个createStore方法的简单实现。

const createStore = (reducer) => {

let state;

let listeners = [];

const getState = ()=>state;

//获取当前时刻状态

const dispatch = (action)=>{

state = reducer(state,action);

listeners.forEach(listener => listener();)

}

//view 发出action的唯一方法

const subscribe = (listener) => {

listeners.push(listener);

return ()=>{

listeners = listeners.filter(l => l !== listener);

}

};

//监听state的变化

dispatch({});

//????????????????????

return {getState,dispatch,subscribe};

};

**Reducer的拆分**

Reducer负责生成state。由于整个应用只有一个state对象，包含所有数据。对于大型应用，state必定十分庞大，reducer若不拆分也会十分庞大。

const chatReducer = (state = defaultState, action = {}) => {

const { type, payload } = action;

switch (type) {

case ADD\_CHAT:

return Object.assign({}, state, {

chatLog: state.chatLog.concat(payload)

});

case CHANGE\_STATUS:

return Object.assign({}, state, {

statusMessage: payload

});

case CHANGE\_USERNAME:

return Object.assign({}, state, {

userName: payload

});

default: return state;

}

};

该代码，三种action分别改变state的三个属性。

* ADD\_CHAT：chatLog属性
* CHANGE\_STATUS：statusMessage属性
* CHANGE\_USERNAME：userName属性

属性之间没有联系，可以将reducer拆分，不同的函数负责处理不同的属性，最终把它们合并成一个大的reducer即可。

const chatReducer = (state = defaultState, action = {}) => {

return {

chatLog: chatLog(state.chatLog, action),

statusMessage: statusMessage(state.statusMessage, action),

userName: userName(state.userName, action)

}

};

//////////////////////////////////////////////////

//Redux 提供了一个combineReducers方法，用于 Reducer 的拆分

import { combineReducers } from 'redux';

const chatLog = (state1,action)=>{

//....

return newstate1;

}

const statusMessage = (state2,action)=>{

//...........

return newstate2;

}

//............

const chatReducer = combineReducers({

chatLog,

statusMessage,

userName

})

//这种写法有一个前提，就是 State 的属性名必须与子 Reducer 同名

//////////////////////////////////////////////////////////////

//不同名的写法

const reducer = combineReducers({

a: doSomethingWithA,

b: processB,

c: c

})

// 等同于

function reducer(state = {}, action) {

return {

a: doSomethingWithA(state.a, action),

b: processB(state.b, action),

c: c(state.c, action)

}

}

export default todoApp;

/\*combineReducers()做的就是产生一个整体的 Reducer 函数。

该函数根据 State 的 key 去执行相应的子 Reducer，

并将返回结果合并成一个大的 State 对象。\*/

combineReducers()简单实现

const combineReducer = reducers => {

return (state={},action)=>{

return Object.keys(reducers).reduce(

(nextState,key)=>{

nextState[key] = reducers[key](state[key],action);

return nextState;

},

{}

);

}

}

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);

**工作流程**



//首先，用户发出 Action。

store.dispatch(action);

//然后，Store 自动调用 Reducer，

//并且传入两个参数：当前 State 和收到的 Action。

//Reducer 会返回新的 State 。

let nextState = todoApp(previousState, action);

State 一旦有变化，Store 就会调用监听函数。

// 设置监听函数

store.subscribe(listener);

//listener可以通过store.getState()得到当前状态。

//如果使用的是 React，这时可以触发重新渲染 View。

function listerner() {

let newState = store.getState();

component.setState(newState);

}

**实例：计算器**

const Counter = ({value})=>(<h1>{value}</h1>);

const render = ()=>{

ReactDOM.render(

<Counter value={store.getState()} />,

document.getElementById('root')

);

};

store.subscribe(render);

render();

///////////////////////////////////////////////////////////////////

const Counter = ({ value }) => (

<h1>{value}</h1>

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

<button onClick={onDecrement}>-</button>

);

const reducer = (state = 0, action) => {

switch (action.type) {

case 'INCREMENT': return state + 1;

case 'DECREMENT': return state - 1;

default: return state;

}

};

const store = createStore(reducer);

const render = () => {

ReactDOM.render(

<Counter

value={store.getState()}

onIncrement={() => store.dispatch({type: 'INCREMENT'})}

onDecrement={() => store.dispatch({type: 'DECREMENT'})}

/>,

document.getElementById('root')

);

};

render();

store.subscribe(render);