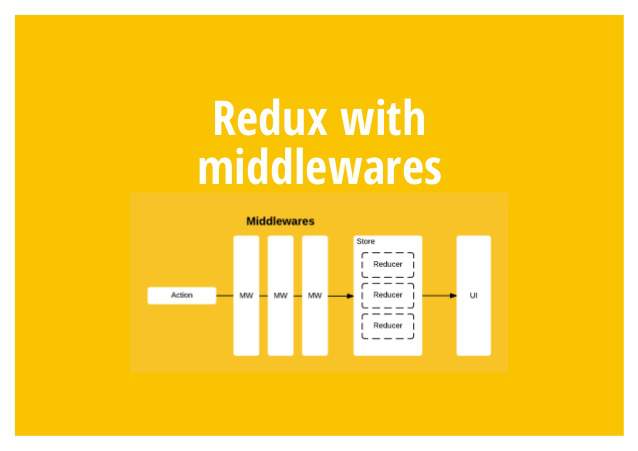
React基本做法：用户发出action，reducer函数算出新的state，view重新渲染

**中间件与异步操作**

Action发出以后，reducer立即算出state，叫做同步；action发出以后，过一段时间再执行reducer，这就是异步。

Reducer在异步操作结束后自动执行要用到新的工具**中间件**：**middleware**



目前的三个部分：

Reducer：纯函数，只承担计算state的功能，不适合承担其他功能，也承担不了，理论上纯函数不能进行读写操作。

View：与state一一对应，可以看作state的视觉层，不适合承担其他功能。

Action：存放数据的对象，消息的载体，只能被别人操作，自己不能进行任何操作。

在发送action的这个步骤，即store.dispatch()方法添加功能。

let next = store.dispatch;

store.dispatch = function dispatchAndLog(action){

console.log('dispatching',action);

next(action);

console.log('next state',store.getState());

//console.log为增加的功能

}

对store.dispatch进行了重定义，在发送action前后添加了打印功能。这就是中间件的雏形。

**中间件就是一个函数，对store.dispatch方法进行了改造，在发出action和执行reducer这两个步骤之间，添加了启发功能**

**中间件的用法**

常用中间件都有现成的，重点在如何使用中间件。

/\*有现成的redux-logger模块。这里介绍怎么使用中间件。\*/

import {applyMiddleware,createStore} from 'redux';

import createStore from 'redux-logger';

const logger = createLogger();

const store = createStore(

reducer,

applyMiddleware(logger)

);

/\*redux-logger提供一个生成器createLogger，

可以生成日志中间件logger。然后，

将它放在applyMiddleware方法之中，

传入createStore方法，就完成了store.dispatch()的功能增强。\*/

/\*createStore方法可以接受整个应用的初始状态作为参数，

那样的话，applyMiddleware就是第三个参数了。\*/

const store = createStore(

reducer,

initial\_state,

applyMiddleware(logger)

);

//中间件的次序有讲究。

const store = createStore(

reducer,

applyMiddleware(thunk, promise, logger)

);

/\*applyMiddleware方法的三个参数，就是三个中间件。

有的中间件有次序要求，使用前要查一下文档。

比如，logger就一定要放在最后，否则输出结果会不正确。\*/

**applyMiddleware**

redux原生方法，作用是将所有中间件组成一个数组，依次执行。

export default function applyMiddleware(...middlewares){

return (createStore)=>(reducer,preloadedState,enhancer)=>{

var store = createStore(reducer,preloadedState,enhancer);

var dispatch = store.dispatch;

var chain = [];

var middlewareAPI = {

getState: store.getState;

dispatch: (action)=>dispatch(action)

};

chain = middlewares.map(middleware=>middleware(middlewareAPI));

dispatch = compose(...chain)(store.dispatch)

return{...store,dispatch}

}

}

/\*所有中间件被放进了一个数组chain，

然后嵌套执行，最后执行store.dispatch。

可以看到，中间件内部（middlewareAPI）

可以拿到getState和dispatch这两个方法。\*/

**异步操作的基本思路**

同步操作只要发出一种action即可，而异步操作要发出三种

操作发起时的action

操作成功时的action

操作失败时的action

三种action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: 'Oops' }

{ type: 'FETCH\_POSTS', status: 'success', response: { ... } }

// 写法二：名称不同

{ type: 'FETCH\_POSTS\_REQUEST' }

{ type: 'FETCH\_POSTS\_FAILURE', error: 'Oops' }

{ type: 'FETCH\_POSTS\_SUCCESS', response: { ... } }

除了action种类不同，异步操作的state也要进行改造，反应不同的操作状态。

//除了 Action 种类不同，

//异步操作的 State 也要进行改造，反映不同的操作状态

let state = {

// ...

isFetching: true,

didInvalidate: true,

lastUpdated: 'xxxxxxx'

};

/\*State 的属性isFetching表示是否在抓取数据。didInvalidate

表示数据是否过时，lastUpdated表示上一次更新时间。\*/

**异步操作思路：**

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，出发state，更新为“操作结束状态”，view再一次重新渲染。

**Redux-thunk中间件**

异步操作至少发出两个action。用户触发第一个action；在操作结束后，系统自动送出第二个action。

奥妙在action creator之中。

class AsyncApp extends Component{

componentDidMount(){

const{dispatch,selectedPost} = this.props

dispatch(fetchPosts(selectedPost))

}

//.......

}

/\*上面代码是一个异步组件的例子。

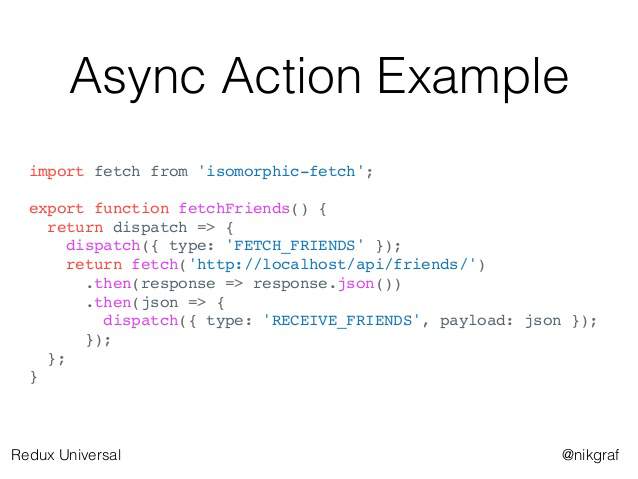
加载成功后（componentDidMount方法），

它送出了（dispatch方法）一个 Action，

向服务器要求数据 fetchPosts(selectedSubreddit)。

这里的fetchPosts就是 Action Creator。\*/

下面就是fetchPosts的代码，关键之处就在里面。



const fetchPosts = postTitle => (dispatch,getState)=>{

dispatch(requestPosts(postTitle));

return fetch('/some/API/${postTitle}.json')

.then(response=>response.json())

.then(json=>dispatch(receivePosts(postTitle,json)));

};

// 使用方法一

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

// 使用方法二

store.dispatch(fetchPosts('reactjs')).then(() =>

console.log(store.getState())

);

/\*上面代码中，fetchPosts是一个Action Creator（动作生成器），

返回一个函数。这个函数执行后，

先发出一个Action（requestPosts(postTitle)），

然后进行异步操作。拿到结果后，先将结果转成 JSON 格式，

然后再发出一个 Action（ receivePosts(postTitle, json)）。\*/

注意点：

**fetchPosts返回了一个函数，而普通的anction creator默认返回一个对象。**

(dispatch,getState)=>{

dispatch(requestPosts(postTitle));

return fetch('/some/API/${postTitle}.json')

.then(response=>response.json())

.then(json=>dispatch(receivePosts(postTitle,json)));

};

**返回的函数的参数是dispatch和getState两个redux方法，普通的action creator的参数是action的内容。**

**在返回的函数之中，先发出一个action （requestPosts(postTitle)），表示操作开始。**

dispatch(requestPosts(postTitle));

**异步操作结束之后，再发出一个action（receivePosts(postTitle,json)）表示动作结束。**

dispatch(receivePosts(postTitle,json)

这样解决了自动发送第二个action的问题。但是action是有store.dispatch方法发送的。而store.dispatch方法正常情况下，参数只能是对象，不能是函数。

中间件redux-thunk来解决这个问题。

import {createStore,applyMiddleware} from 'redux';

**import thunk from 'redux-thunk';**

import reducer from './reducers';

// Note: this API requires redux@>=3.1.0

const store = createStore(

reducer,

**applyMiddleware(thunk)**

);

/\*上面代码使用redux-thunk中间件，

改造store.dispatch，使得后者可以接受函数作为参数。\*/

异步操作的第一种解决方案就是，写出一个返回函数的action creator，然后使用redux-thunk中间件改造store.dispatch。

**Redux-promise中间件**

既然actioncreator可以返回函数，当然也可以返回其他值。另一种异步操作的解决方案，就是让action creator返回一个**promise对象。**

需要使用redux-promise中间件。

import {createStore,applyMiddleware} from 'redux';

import promiseMiddleware from 'redux-promise';

import reducer from './reducers';

const store = createStore(

reducer,

applyMiddleware(promiseMiddleware)

);

//这个中间件使得store.dispatch方法可以接受 Promise 对象作为参数

有了以上的两个中间件，action creator有两种写法。

写法一，返回值是一个promise对象。

const fetchPosts =

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

dispatch(requestPosts(postTitle));

return fetch('/some/API/${postTitle}.json')

.then(response=>{

type:'FETCH\_POSTS',

payload:response.json()

});

});

//写法一，返回值是一个 Promise 对象。

/\*fetch('/some/API/${postTitle}.json')

.then(response=>{

type:'FETCH\_POSTS',

payload:response.json()

});\*/

写法二，action对象的payload属性是一个promise对象。这需要从redux-action模块引入createAction方法，而写法也要有如下变化。

import {createAction} from 'redux-action';

class AsyncApp extends Component{

componentDidMount(){

const {dispatch,selectedPost} = this.props;

//发出同步action

dispatch(requestPosts(selectedPost));

//发出异步 Action

dispatch(createAction(

'FETCH\_POSTS',

fetch('/some/API/${postTitle}.json')

.then(response=>response.json())));

}

}

/\*，第二个dispatch方法发出的是异步 Action，

只有等到操作结束，这个 Action 才会实际发出。\*/

/\*createAction的第二个参数必须是一个 Promise 对象。\*/

/\*fetch('/some/API/${postTitle}.json')

.then(response=>response.json())\*/

Redux-promise源码

export default function promiseMiddleware({ dispatch }) {

return next => action => {

if (!isFSA(action)) {

return isPromise(action)

? action.then(dispatch)

: next(action);

/\*如果 Action 本身是一个 Promise，

它 resolve 以后的值应该是一个 Action 对象，

会被dispatch方法送出（action.then(dispatch)），

但 reject 以后不会有任何动作；\*/

}

return isPromise(action.payload)

? action.payload.then(

result => dispatch({ ...action, payload: result }),

error => {

dispatch({ ...action, payload: error, error: true });

return Promise.reject(error);

}

)

: next(action);

/\*如果 Action 对象的payload属性是一个 Promise 对象，

那么无论 resolve 和 reject，dispatch方法都会发出 Action。\*/

};

}