# [用react+redux编写一个页面小demo](https://blog.csdn.net/yuzhongzi81/article/details/51880577)

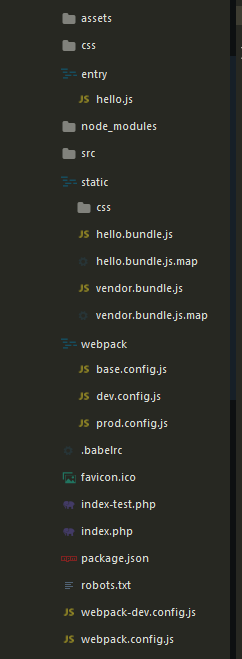
2016年07月11日 14:36:10 [萌萌的卡洛奇](https://me.csdn.net/yuzhongzi81" \t "https://blog.csdn.net/yuzhongzi81/article/details/_blank) 阅读数：8842更多

个人分类： [react](https://blog.csdn.net/yuzhongzi81/article/category/6306165" \t "https://blog.csdn.net/yuzhongzi81/article/details/_blank)

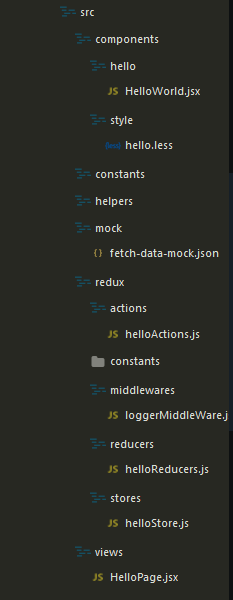
版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/yuzhongzi81/article/details/51880577

## **初步目录结构**

### **整体目录结构**



### **src目录结构**



****entry**** 存放页面的入口文件

****src**** 页面的源文件

****static**** 页面源文件打包后生成的文件

****webpack**** webpack打包文件

****package.json**** package.json文件

****.babelrc**** 支持es6语法

其中 src 中子目录结构如下：

components 页面组件

constants 页面需要用到的一些常量

helpers 工具方法

mock 模拟json数据

redux redux数据控制目录

views 页面视图文件，主要视图全在这个文件

## **react配合redux编写页面流程**

#### ****entry/hello.js****

import React from 'react';

import { render } from 'react-dom';

import HelloPage from '../src/views/HelloPage';

import helloStore from '../src/redux/stores/helloStore';

import { Provider } from 'react-redux';

render(

<Provider store={helloStore}>

<HelloPage />

</Provider>,

document.getElementById('hello')

);

HelloPage.jsx是视图文件，通过react-redux中的Provider将store绑定到视图中

#### ****src/redux/actions/helloAction.js****

import fetch from 'isomorphic-fetch';

*// 纯事件定义*

export const ADD\_COUNT = 'ADD\_COUNT';

export const ADD\_PERSON = 'ADD\_PERSON';

export const DELETE\_PERSON = 'DELETE\_PERSON';

*// async// 异步的请求定义*

export const FETCH\_START = 'FETCH\_START';

export const FETCH\_SUCCESS = 'FETCH\_SUCCESS';

export const FETCH\_FAUILE = 'FETCH\_FAUILE';

*// pure functions*

export function addCount() {

return {

type : ADD\_COUNT

}

}

export function addPerson(person) {

return {

type : ADD\_PERSON,

person

}

}

export function deletePerson(idx) {

return {

type : DELETE\_PERSON,

idx

}

}

export function refreshStart() {

return {

type : FETCH\_START

}

}

export function refreshSuccess(list) {

return {

type : FETCH\_SUCCESS,

list

}

}

export function refreshFauile() {

return {

type : FETCH\_FAUILE

}

}

*// 定义的非纯函数，提供异步请求支持// 需要在sotre中使用thunkMiddleware*

export function refresh() {

return dispatch => {

dispatch(refreshStart());

return fetch(`src/mock/fetch-data-mock.json`)

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

.then(json => {

setTimeout(() => {

dispatch(refreshSuccess(json && json.data.list));

}, 3000);

});

}

}

action中主要定义事件类型，基本上都是一些常量。另外如果要进行一些非常量返回，比如异步请求，则需要输出一个函数，这个函数通常带有dispatch这个对象，用于对action的重新包装，其实类似于后台语言中的“拦截器”，返回函数之后，需要在store中配置thunkMiddleware。

#### ****src/redux/reducers/helloReducers.js****

import { combineReducers } from 'redux';

import { ADD\_COUNT, ADD\_PERSON, DELETE\_PERSON, FETCH\_START, FETCH\_SUCCESS, FETCH\_FAUILE } from '../actions/helloActions';

*// store中可以定义页面中的初始状态*const initialState = {

count : 0, *// count = 0*

loaded : false, *// 异步请求是否加载*

personList : [ *// 人员列表*

{"name" : "lily", "age" : 21}

]

};

*// count的初始状态以及处理之后返回的state值*function count(state = initialState.count, action) {

switch (action.type) {

case ADD\_COUNT :

return state + 1;

default :

return state;

}

}

function personList(state = initialState, action) {

switch (action.type) {

case ADD\_PERSON :

return Object.assign({}, ...state, {

personList : [...state.personList, action.person]

});

case DELETE\_PERSON :

return Object.assign({}, ...state, {

personList : state.personList.filter((s, i) => {

return action.idx !== i;

})

});

case FETCH\_START :

case FETCH\_SUCCESS :

case FETCH\_FAUILE :

return fetchDataFromServer(state, action);

default :

return state;

}

}

function fetchDataFromServer(state, action) {

if (action.type === FETCH\_SUCCESS) {

console.log(action);

return Object.assign({}, ...state, {

personList : [...state.personList, ...action.list],

loaded : true

});

}

return state;

}

const helloReducers = combineReducers({

count,

personList

});

export default helloReducers;

reducer中是对action发起的数据进行处理，这其中可能action只是发出了一个纯指令，带参数或者不带参数根据业务的需求来定，总一个准则就是，初始的state经过处理之后会返回新的state。即：( state, action ) => newState。同时另外注意一点的是，初始的state值是不会变得，需要操作则会另外创建一个新的state，来保证某些场景的性能问题(因为状态的改变会导致页面重新渲染，如果返回的state引用相同，则不会存在如此问题)。可以使用Immutable.js来保证state的纯洁性。

#### ****src/redux/stores/helloStore.js****

import { createStore, applyMiddleware } from 'redux';

import helloReducers from '../reducers/helloReducers';

import logger from '../middlewares/loggerMiddleWare';

import thunkMiddleware from 'redux-thunk';

*// middleware可以自己定义，例如下面的logger// 写一个自定义的middleware遵循下面的格式：// const logger = store => next => action => {// // what you do before action, // // example: logger. console.log("dispatching", action);// let result = next(action);// // what you can do after action // //console.log('next state', store.getState());// return result;//}*

let createStoreWithMiddleware = applyMiddleware(*/\*logger, \*/*thunkMiddleware)(createStore);

export default createStoreWithMiddleware(helloReducers);

store中主要绑定从reducer返回的状态

#### ****src/views/HelloPage.jsx****

import React from 'react';

import { connect } from 'react-redux';

import { addCount, addPerson, deletePerson, refresh } from '../redux/actions/helloActions';

import HelloWorld from '../components/hello/HelloWorld';

import Button from 'react-bootstrap-myui/lib/Button';

import Table from 'react-bootstrap-myui/lib/Table';

const HelloPage = React.createClass({

render () {

let { count, list, loaded } = this.props;

let personTbody = list.map((l, i) => (<tr key={"tr"+i}><td>{i}</td><td>{l.name}</td>

<td>{l.age}</td><td><a href="javascript:;" onClick={this.deletePerson.bind(this, i)}>delete</a></td></tr>));

return (

<div id="reactPage">

<HelloWorld />

<Button onClick={this.addperson}>add person</Button>

<Button onClick={this.refresh}>refresh</Button>

{loaded ? (<span>加载完成</span>) : (<span>正在加载数据...</span>)}

<Table striped hover>

<thead>

<tr>

<th>#</th>

<th>name</th>

<th>age</th>

<th>action</th>

</tr>

</thead>

<tbody>

{personTbody}

</tbody>

</Table>

<span>show me the current count : {count || 0}</span>

<div><button onClick={this.addCount}>Add</button></div>

</div>

);

},

addCount() {

let { dispatch, count } = this.props;

dispatch(addCount());

},

addperson() {

let { dispatch } = this.props;

dispatch(addPerson({"name" : "tome", age : 25}));

},

deletePerson(index) {

let { dispatch } = this.props;

dispatch(deletePerson(index));

},

refresh() {

let { dispatch } = this.props;

dispatch(refresh());

}

});

function select(state) {

return {

count : state.count,

list : state.personList.personList,

loaded : state.personList.loaded

}

}

export default connect(select)(HelloPage);

详细的页面，在页面中默认会带有从store中返回的状态，会以props的形式传递到页面中，默认会带有一个dispatch函数，用于发起action，因此这个action可以绑定页面的一些业务逻辑操作，比如：“新增”、“删除”、“修改”……