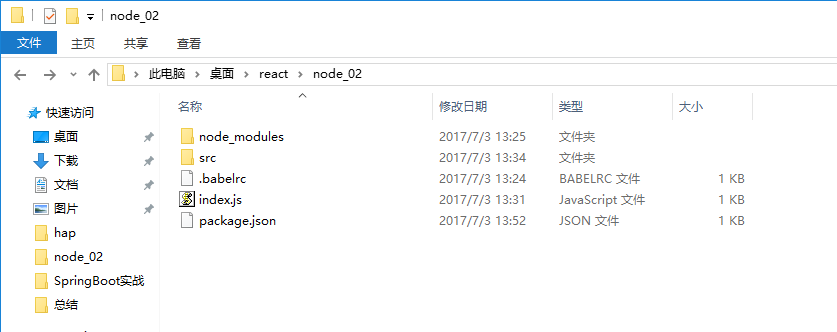
# React和Redux

## 在Node.js中运行React



### 编写React组件

简介的无状态函数 该React组件中是你需要返回的视图

export default表示默认导出了这个组件

default表示可以在别的文件中用import App from './App'导入

没有default 则需要import {App} from './App'导入

src/App.js

import React from 'react';

export default function App(){

return <h1>hello world</h1>;

}

### 在Node.js中渲染组件

导入App.js组件

然后通过renderToString将其渲染成一个HTML字符串打印出来

src/server.js

import React from 'react';

import { renderToString } from 'react-dom/server';

import App from './App';

const appHTML = renderToString(<App/>);

console.log(appHTML);

### 使用Babel编译运行Node.js程序

①安装babel-register

npm install babel-register --save-dev

②入口文件index.js

require('babel-register');

require('./src/server.js');

③安装ES2015和React的预设

npm install --save-dev

babel-preset-es2015 babel-preset-react

④添加配置文件.babelrc,激活ES2015和React的预设

{ "presets": ["react", "es2015"]}

⑤package.json

配置文件package.json

{

"name": "node\_02",

"version": "1.0.0",

"description": "React Redux example",

"scripts": {

"start": "node index"

},

"license": "MIT",

"devDependencies": {

"babel-preset-es2015": "^6.6.0",

"babel-preset-react": "^6.5.0",

"babel-register": "^6.8.0"

},

"dependencies": {

"react": "^15.3.1",

"react-dom": "^15.3.1"

},

"repository": {

"type": "git",

"url": "git+https://github.com/lewis617/react-redux-book.git"

},

"bugs": {

"url": "https://github.com/lewis617/react-redux-book/issues"

},

"homepage": "https://github.com/lewis617/react-redux-book#readme"

}

### 执行

$ npm install $ npm start

## 在浏览器中运行React

### **说明**

在浏览器中渲染React组件

先安装依赖 后使用Webpack打包编译，最后运行

### 编写React组件

src/App.js

import React from 'react';

export default function App(){

return <h1>hello world</h1>;

}

### 在浏览器中渲染React组件

在浏览器中渲染组件使用react-dom的render()

将组件App渲染到id为app的标签中

src/client.js

import React from 'react';

import { render } from 'react-dom';

import App from './App';

render((<App/>),document.querySelector('#app'));

### 添加Webpack.config.js文件来指定打包编译的配置信息

entry表示入口文件

output中的path表示输出目录

filename表示输出的文件名称

module.exports = {

entry: './src/client',

output: {

path: \_\_dirname + '/static/dist',

filename: 'main.js'

},

module: {

loaders: [{ test: /\.js$/, exclude: /node\_modules/, loaders: ['babel'] }]

}

};

### 添加配置文件.babelrc

{ "presets": ["react", "es2015"] }

### 在浏览器中运行

将生成的脚本文件添加到index.Html中

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title></title>

</head>

<body>

<div id="app"></div>

<script src="static/dist/main.js"></script>

</body>

</html>

### package.json

{

"name": "react\_02",

"version": "1.0.0",

"description": "React Redux example",

"scripts": {

"build": "webpack",

"start": "webpack && http-server -p 3000"

},

"license": "MIT",

"devDependencies": {

"babel-core": "^6.8.0",

"babel-loader": "^6.2.4",

"babel-preset-es2015": "^6.6.0",

"babel-preset-react": "^6.5.0",

"webpack": "^1.13.0"

},

"dependencies": {

"http-server": "^0.9.0",

"react": "^15.3.1",

"react-dom": "^15.3.1"

},

"repository": {

"type": "git",

"url": "git+https://github.com/lewis617/react-redux-book.git"

},

"bugs": {

"url": "https://github.com/lewis617/react-redux-book/issues"

},

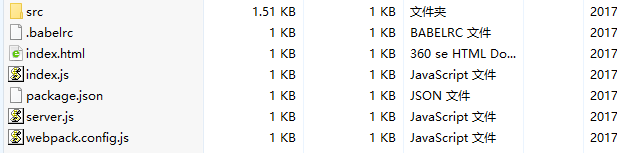
"homepage": "https://github.com/lewis617/react-redux-book#readme"

}

### 运行

npm install npm run build npm start

## JSX



### 组件渲染成视图

App.js

import React, { Component } from 'react';

加上data-前缀自定义属性

function Demo1() {

return (

<li>

<h3>类似HTML</h3>

<p data-attribute="demo1">可以嵌套，可以自定义属性</p>

</li>

);

}

表达式中必须要有返回值，因此在return（）中不能直接使用if else

需要放在带有返回值的function中 然后在{}中调用该函数

function Demo2() {

const name = 'JSX';

const func = () => {

let result = 'hello ';

if (name) {

result += name;

} else {

result += 'world';

}

return result;

};

return (

<li>

<h3>JavaScript表达式</h3>

<p>hello {name || 'world'}</p>

<p className={name ? 'class-a' : 'class-b'}>

hello {name && 'world'}

</p>

<p>

{func()}

</p>

</li>

);

}

JSX的内联样式中 属性值不能是字符串必须是对象

属性名命名必须为驼峰命名法

function Demo3() {

return (

<li>

<h3>样式</h3>

<p style={{ color: 'red', fontSize: '14px' }}>内联样式不是字符串，而是对象</p>

</li>

);

}

注释如下

function Demo4() {

return (

<li>

<h3>注释</h3>

{/\* 注释... \*/}

<p>标签子节点内的注释应该写在大括号中</p>

</li>

);

}

JSX中的数组会自动展开所有成员，如果数组中的每一项都是标签或者组件，则他们都要拥有唯一的key属性值

function Demo5() {

const arr = [

<h3 key={0}>数组</h3>,

<p key={1}>数组会自动展开。注意，数组中每一项元素需要添加key属性</p>,

];

return (<li>{arr}</li>);

}

将组件渲染成视图

export default class App extends Component {

render() {

return (

<div>

<h2>JSX语法</h2>

<ul>

<Demo1 />

<Demo2 />

<Demo3 />

<Demo4 />

<Demo5 />

</ul>

</div>

);

}

}

### 将渲染好的组件添加到指定div中

index.js

import React from 'react';

import { render } from 'react-dom';

import App from './src/App';

render(<App />, document.querySelector('#root'));

### React-hmre预设

热替换 是指当修改部分代码后，程序会自动编译、替换被修改的代码，不需要重新编译所有代码，也不需要刷新浏览器

.babelrc

{

"presets": ["es2015", "react"],

"env": {

"development": {

"presets": ["react-hmre"]

}

}

}

### 配置Webpack

**devtool：**

生成源代码映射 方便调试

源代码映射记录了打包编译后的代码和代码之间的位置对应关系

**entry:**

入口文件

程序的入口文件

用于热替换功能的webpack-hot-middleware/client

工作原理：先连接上服务器，等待重新编译代码的更新通知，随时同步跟新 客户端

**output:**

path表示输出的硬盘路径，本例中编译好的代码放在内存中，故无用

filename表示输出的文件名称

publicPath表示公共路径 及所有资源的URL前缀

**plugins:**

插件列表

webpack.optimize.OccurrenceOrderPlugin()

给经常使用的模块分配最小长度的ID

webpack.HotModuleReplacementPlugin()

用于热替换

webpack.config.js

var path = require('path');

var webpack = require('webpack');

module.exports = {

devtool: 'cheap-module-eval-source-map',

entry: [

'webpack-hot-middleware/client',

'./index.js'

],

output: {

path: path.join(\_\_dirname, 'dist'),

filename: 'bundle.js',

publicPath: '/static/'

},

plugins: [

new webpack.optimize.OccurrenceOrderPlugin(),

new webpack.HotModuleReplacementPlugin()

],

module: {

loaders: [

{

test: /\.js$/,

loaders: ['babel'],

exclude: /node\_modules/,

include: \_\_dirname

}

]

}

};

### 配置Express服务器

配置完Webpack后，只需要在在Express服务器中添加

webpackDevMiddleware webpackHotMiddleware 这两个中间件即可完成开发服务器和热替换的配置工作

server.js

var webpack = require('webpack');

Var webpackDevMiddleware = require('webpack-dev-middleware');

var webpackHotMiddleware = require('webpack-hot-middleware');

var config = require('./webpack.config');

var app = new (require('express'))();

var port = 3000;

var compiler = webpack(config);

app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPath: config.output.publicPath }));

app.use(webpackHotMiddleware(compiler));

app.get("/", function(req, res) {

res.sendFile(\_\_dirname + '/index.html')

});

app.listen(port, function(error) {

if (error) {

console.error(error)

} else {

console.info("==> 馃寧 Listening on port %s. Open up http://localhost:%s/ in your browser.", port, port)

}

});

### package.json

{

"name": "05-jsx",

"version": "0.0.0",

"description": "React Redux example",

"scripts": {

"start": "node server.js"

},

"dependencies": {

"react": "^15.3.1",

"react-dom": "^15.3.1"

},

"devDependencies": {

"babel-core": "^6.3.15",

"babel-loader": "^6.2.0",

"babel-preset-es2015": "^6.3.13",

"babel-preset-react": "^6.3.13",

"babel-preset-react-hmre": "^1.1.1",

"express": "^4.13.3",

"webpack": "^1.9.11",

"webpack-dev-middleware": "^1.2.0",

"webpack-hot-middleware": "^2.9.1"

},

"repository": {

"type": "git",

"url": "git+https://github.com/lewis617/react-redux-book.git"

},

"license": "MIT",

"bugs": {

"url": "https://github.com/lewis617/react-redux-book/issues"

},

"homepage": "https://github.com/lewis617/react-redux-book#readme"

}

### 在浏览器中运行

<!DOCTYPE html>

<html>

<head>

<title>Example</title>

</head>

<body>

<div id="root">

</div>

<script src="/static/bundle.js"></script>

</body>

</html>

### 运行

npm install npm start

## state、props、context

### 父子组件

**① 组合使用state和props**

Counter.js

import React, { Component, PropTypes } from 'react';

**子组件**

function Content(props) {

return <p>Content组件的props.value：{props.value}</p>;

}

验证props

Content.propTypes = {

value: PropTypes.number.isRequired

};

**父组件**

export default class Counter extends Component {

在构造函数中初始化内部状态

constructor() {

super();

this.state = { value: 0 };

}

render() {

return (

<div>

<button onClick={() => this.setState({ value: this.state.value + 1 })}>

INCREMENT

</button>

<br/><br/>

Counter组件的内部状态：

<pre>{JSON.stringify(this.state, null, 2)}</pre>

将数据写在组件标签的属性中

<Content value={this.state.value}/>

</div>

);

}

}

**② 使用props传递数据**

Message1list1.js

import React, { PropTypes } from 'react';

**定义子组件Button**

function Button(props) {

return (

<button style={{ background: props.color }}>

{props.children}

</button>

);

}

验证Button组件的props

Button.propTypes = {

color: PropTypes.string.isRequired,

children: PropTypes.string.isRequired

};

**定义子组件Message**

function Message(props) {

return (

<li>

{props.text} <Button color={props.color}>Delete</Button>

</li>

);

}

验证Message组件的props

Message.propTypes = {

text: PropTypes.string.isRequired,

color: PropTypes.string.isRequired

};

**定义父组件**

function MessageList() {

const color = 'gray';

const messages = [

{ text: 'Hello React' },

{ text: 'Hello Redux' }

];

const children = messages.map((message, key) =>

<Message key={key} text={message.text} color={color}/>

);

return (

<div>

<p>通过props将color逐层传递给里面的Button组件</p>

<ul>

{children}

</ul>

</div>);

}

export default MessageList;

**③ 使用context传递数据**

Message1list2.js

import React, { Component, PropTypes } from 'react';

**子组件**

function Button(props, context) {

return (

<button style={{ background: context.color }}>

{props.children}

</button>

);

}

Button.propTypes = {

children: PropTypes.string.isRequired

};

Button.contextTypes = {

color: PropTypes.string.isRequired

};

**子组件**

function Message(props) {

return (

<li>

{props.text} <Button>Delete</Button>

</li>

);

}

Message.propTypes = {

text: PropTypes.string.isRequired

};

**父组件**

class MessageList extends Component {

getChildContext() {

return { color: 'gray' };

}

render() {

const messages = [

{ text: 'Hello React' },

{ text: 'Hello Redux' }

];

const children = messages.map((message, key) =>

<Message key={key} text={message.text}/>

);

return (

<div>

<p>通过context将color跨级传递给里面的Button组件</p>

<ul>

{children}

</ul>

</div>

);

}

}

MessageList.childContextTypes = {

color: PropTypes.string.isRequired

};

export default MessageList;

### 整合父组件

App.js

import React, { Component } from 'react';

import Counter from './Counter';

import Messagelist1 from './Messagelist1';

import Messagelist2 from './Messagelist2';

export default class App extends Component {

render() {

return (

<div>

<h2>State与props</h2>

<Counter />

<br/>

<h2>Props与context</h2>

<Messagelist1 />

<Messagelist2 />

</div>

);

}

}

### 放入指定div

index.js

import React from 'react';

import { render } from 'react-dom';

import App from './src/App';

render((<App />), document.querySelector('#root'));

## ReactElement与组件实例

### 父组件（该例中未使用子组件,因此没有父组件整合）

App.js

import React from 'react';

const suffix = '被调用，this指向：';

export default class App extends React.Component {

constructor(props) {

super(props);

// this.handler = this.handler.bind(this)

}

在第一次渲染后调用

componentDidMount() {

console.log(`componentDidMount${suffix}`, this);

}

在组件接收到一个新的props是被调用，第一次渲染时不会被调用

componentWillReceiveProps() {

console.log(`componentWillReceiveProps${suffix}`, this);

}

在组件接收到新的props或者state时被调用

在初始化时或者使用forceUpdate时不被调用，可以在你确认不需要组件更新时使用

shouldComponentUpdate() {

console.log(`shouldComponentUpdate${suffix}`, this);

return true;

}

在组件完成更新后立即调用，在初始化时不会被调用

componentDidUpdate() {

console.log(`componentDidUpdate${suffix}`, this);

}

在组件从DOM中移除的时候立刻被调用

componentWillUnmount() {

console.log(`componentWillUnmount${suffix}`, this);

}

handler() {

console.log(`handler${suffix}`, this);

}

render() {

console.log(`render${suffix}`, this);

this.handler();

window.handler = this.handler;

window.handler();

return (

<div>

<h1 onClick={this.handler}>Hello world</h1>

<p>不清楚组件、ReactElement、组件实例以及组件中的this是什么？打开控制台看看就明白了！ </p>

</div>

);

}

}

### 放入指定div

Index.js

import React from 'react';

import { render, unmountComponentAtNode } from 'react-dom';

import App from './src/App';

console.log('首次挂载');

let instance = render(<App />, document.getElementById('root'));

window.renderComponent = () => {

console.log('挂载');

instance = render(<App />, document.getElementById('root'));

};

window.setState = () => {

console.log('更新');

instance.setState({ foo: 'bar' });

};

window.unmountComponentAtNode = () => {

console.log('卸载');

unmountComponentAtNode(document.getElementById('root'));

};

console.log('JSX中的闭合标签是ReactElement');

console.log(<h1>hello world</h1>);

console.log(<App />);

console.log('组件、ReactElement与组件实例');

console.log(App);

console.log(<App />);

console.log(instance);

### HTML

使用三个按钮出发组件的挂载 更新 和卸载

<!DOCTYPE html>

<html>

<head>

<title>Example</title>

</head>

<body>

<button onclick="window.renderComponent()">挂载</button>

<button onclick="window.setState()">更新</button>

<button onclick="window.unmountComponentAtNode()">卸载</button>

<div id="root"></div>

<script src="/static/bundle.js"></script>

</body>

</html>

## 初识Redux

Action: 本质上是JavaScript普通对象。action内使用一个字符串类型的type字段来表示将要执行的动作。多数情况下，type会被定义成字符串常量。除了type字段外，action对象的结构完全由你自己决定。他是store数据的唯一来源

Reducer：是一个形式为(state,action) => state的纯函数，描述了action如何把state转变成下一个state。

纯函数：输入输出数据流全是显示的，函数与外界交换数据只有一个数据渠道--参数和返回值。

永远不要在reducer里做这些操作：

修改传入的参数。

执行有副作用的操作，如API请求和路由跳转

调用非纯函数，如Date.now()或Math.random()

App.js

创建store需要从redux包中导入createStore 方法

import { createStore } from 'redux';

### Reducer

不能修改参数state

function counter(state=0,action){

switch(action.type){

case 'INCREMENT':

return state+1;

case 'DECREMENT':

return state-1;

default:

return state;

}

}

### Sotre

store是个全局的对象，是唯一的，将action和reducer以及state联系在一起。

Store的职能：

维持应用的state

提供getState()方法获取state

提供dispatch(action)方法更新state

通过subscribe(listener)注册监听器

使用reducer纯函数作为第一个参数创建store

可以将初始化state作为第二个参数传入createStore(counter,100)

const store = createStore(counter);

使用getState获取数据 并监听变化

获取初始话的state到currentValue

let currentValue = store.getState();

const listener = ()=> {

const previousValue = currentValue;

currentValue = store.getState();

console.log('pre state:',previousValue,'next state',currentValue);

};

使用subscribe方法监听变化

在subscribe的回调函数listener中将currentValue传给previousValue 作为先前的state

并将更新后的state传给currentValue 作为当前的state

store.subscribe(listener);

### Action

store使用dispatch方法发起action,更新state

store.dispatch({type:'INCREMENT'});

store.dispatch({type:'INCREMENT'});

store.dispatch({type:'DECREMENT'});

## Action创建函数与Redux Thunk中间组件

以上例为基础修改

### Action创建函数

function increment() {

return { type: 'INCREMENT' };

}

function decrement() {

return { type: 'DECREMENT' };

}

### Redux Thunk中间件

中间件可以让action创建函数先不返回action对象，而是返回一个函数

返回函数条件判断

function incrementIfOdd() {

return (dispatch, getState) => {

const value = getState();

if (value % 2 === 0) {

return;

}

dispatch(increment());

};

}

返回函数 延迟dispatch

function incrementAsync(delay = 1000) {

return dispatch => {

setTimeout(() => {

dispatch(increment());

}, delay);

};

}

### 安装激活中间件

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

const store = createStore(counter, applyMiddleware(thunk));

### 执行Action

加一

store.dispatch(increment());

减一

store.dispatch(incrementIfOdd());

奇数加一

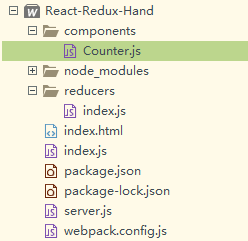
store.dispatch(incrementAsync());

异步加一

store.dispatch(decrement());

## React与Redux的连接：手动连接

### 目录



### 编写React组件

components/Counter.js

import React,{Component,PropTypes} from 'react';

//React组件编写

class Counter extends Component{

/\*3

\*编写奇数加一和异步加一的组件方法 并将方法中的this绑定打组件实例上

\*/

constructor(props){

super(props);

this.incrementIfOdd = this.incrementIfOdd.bind(this);

this.incrementAsync = this.incrementAsync.bind(this);

}

incrementIfOdd(){

if(this.props.value%2 !==0){

this.props.onIncrement();

}

}

incrementAsync(){

setTimeout(this.props.onIncrement,1000);

}

/\*2

\*在render中获取props输入值 然后渲染及界面 包括一个数组和四个按钮

\*/

render(){

const {value,onIncrement,onDecrement} = this.props;

return(

<p>

Clicked:{value}times

{''}

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

{''}

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

{''}

<button onClick={this.incrementIfOdd}>incrementIfOdd</button>

{''}

<button onClick={this.incrementAsync}>incrementAsync</button>

</p>

);

}

}

/\*1

\*设计组件的props输入为 value onIncrement onDecrement

\*分别代表数值，加一方法，减一方法，并限制他们的类型

\*/

Counter.propTypes = {

//value必须为数字 且必须存在

value: PropTypes.number.isRequired,

//必须为function 且必须存在

onIncrement: PropTypes.func.isRequired,

onDecrement: PropTypes.func.isRequired

};

export default Counter;

### 编写Reducer

reducers/index.js

/\*4

\*编写reducer 使其可以接受加一减一的action，并据此更新state

\*reducer的触发 是通过 const store = createStore(counter)创建

\*并由store.dispatch({type:'DECREMENT'})发起的

\*/

export default function counter(state=0,action){

switch(action.type){

case 'INCREMENT':

return state+1;

case 'DECREMENT':

return state-1;

default:

return state;

}

}

### 手动连接

index.js

import React from 'react';

import ReactDOM from 'react-dom';

import {createStore} from 'redux';

import Counter from './components/Counter';

import counter from './reducers';

/\*5

\*将state和发起action的方法连接到组件然后渲染组件并监听变化

\*后面只需要点击按钮就会发起action 再次渲染

\*设置初始state为100 若未设置 则默认为counter reducer中的0

\*/

const store = createStore(counter,66);

function render(){

ReactDOM.render(

<Counter value={store.getState()}

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

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

document.getElementById('root')

);

}

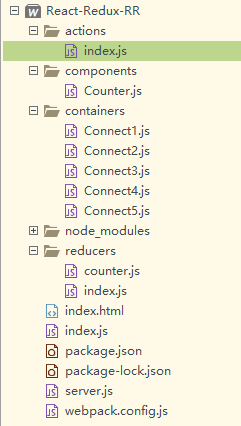
render();

//监听render的变化

store.subscribe(render);

## React与Redux的连接：RR连接

### 目录



### 编写React组件

components/Counter.js

与手动连接不同的是 此处将在组件类利用props编写的方法抽离出来放在actions中进行编写 并通过containers将state和action创建函数绑定到组件中

import React, { PropTypes } from 'react';

function Counter({ counter, increment, decrement, incrementIfOdd, incrementAsync }) {

return (

<p>

Clicked: {counter} times

{' '}

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

{' '}

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

{' '}

<button onClick={incrementIfOdd}>Increment if odd</button>

{' '}

<button onClick={() => incrementAsync()}>Increment async</button>

</p>

);

}

Counter.propTypes = {

counter: PropTypes.number.isRequired,

increment: PropTypes.func.isRequired,

incrementIfOdd: PropTypes.func.isRequired,

incrementAsync: PropTypes.func.isRequired,

decrement: PropTypes.func.isRequired

};

export default Counter;

### 编写Reducer(编写action)

actions/index.js

这里将action对象的type属性值写成了常量，便于reducer引用，减少了出错的概率。

export const INCREMENT\_COUNTER = 'INCREMENT\_COUNTER';

export const DECREMENT\_COUNTER = 'DECREMENT\_COUNTER';

export function increment() {

return {

type: INCREMENT\_COUNTER,

};

}

export function decrement() {

return {

type: DECREMENT\_COUNTER,

};

}

export function incrementIfOdd() {

return (dispatch, getState) => {

const { counter } = getState();

if (counter % 2 === 0) {

return;

}

dispatch(increment());

};

}

export function incrementAsync(delay = 1000) {

return dispatch => {

setTimeout(() => {

dispatch(increment());

}, delay);

};

}

### 编写Reducer(融合action)

reducers/counter.js

import { INCREMENT\_COUNTER, DECREMENT\_COUNTER } from '../actions';

export default function counter(state = 0, action) {

switch (action.type) {

case INCREMENT\_COUNTER:

return state + 1;

case DECREMENT\_COUNTER:

return state - 1;

default:

return state;

}

}

### 编写Reducer(合并Reducer)

reducers/index.js

import { combineReducers } from 'redux';

import counter from './counter';

虽然本例中只有一个reducer，但还是使用了`combineReducers`来进行合并，便于后期的拓展。

在进行合并后，计数器的数值将被转移到`state.counter`中。

const rootReducer = combineReducers({

counter,

});

export default rootReducer;

### 为组件绑定state和action

import Counter from '../components/Counter';

import { connect } from 'react-redux';

import \* as ActionCreators from '../actions';

connect 第一个参数是一个函数他将state.counter传递给组件的counter属性

第二个参数是对象,是所有action创建函数的集合,将所有action创建的函数传到了组件的同名属性中,Counter.js中直接注入

同时为action创建函数隐式绑定了dispatch方法

export default connect(

state => ({ counter: state.counter }),

ActionCreators

)(Counter);

### 使用Provider给整个程序提供store

Provider只是一个React组件，它的职能是通过context将store传递给子组件

因为Provider组件是通过context传递store的，所以里面的组件不管跨多少级 都可以通过 connect()方法获取store并进行连接

connect()组件负责给程序提供store 而connect()则负责生成新的名为Connect的组件

Connect组件在context中拿到store后，从store中获取state和dispatch，最后将state和dispatch加工的action创建函数连接到组件上

import React from 'react';

import ReactDOM from 'react-dom';

import { createStore, applyMiddleware } from 'redux';

import { Provider } from 'react-redux';

import thunk from 'redux-thunk';

import counter from './reducers';

import Connect1 from './containers/Connect1';

const store = createStore(counter, applyMiddleware(thunk));

const rootEl = document.getElementById('root');

ReactDOM.render(

<Provider store={store}>

<div>

<h2>使用react-redux连接</h2>

<ul>

<li>

connect()的前两个参数分别为函数和对象：

<Connect1 />

</li>

</ul>

</div>

</Provider>, rootEl

);

## 实现撤销/重做

# 极客学院React

## 文件说明

### 引入js

<script type="text/javascript" src="js/react.js" ></script>

<script type="text/javascript" src="js/JSXTransformer.js" >

</script>

### 映入jsx格式的js

若引入的JSXTransformer.js 在渲染时使用React.render();

若引入的是babel.min.js 在渲染时使用ReactDOM.render();

<script type="text/jsx" src="ifelse.js"></script>

## 四种判断表达式

在render内调用外部变量或函数时 要加上{ }

### 1. if else

var HtmlMessage = React.createClass({

getName:function(){

if(this.props.name){

return this.props.name;

}else{

return 'world';

}

},

render:function(){

return (

<h1>hello , {this.getName()}</h1>

);

}

});

React.render(

<HtmlMessage name='Mr.Hy'/>,

document.getElementById('example')

);

### 2.三元表达式

{}尽量使用在最终返回函数render中

var HtmlMessage = React.createClass({

render:function(){

return (

<h1>hello , {this.props.name ? this.props.name :'world1'}</h1>

);

}

});

React.render(

<HtmlMessage name='Mr.Hy'/>,

document.getElementById('example')

);

### 3.普通判断

var HtmlMessage = React.createClass({

getName:function(){

return this.props.name ? this.props.name:'world2';

},

render:function(){

var returnName = this.getName();

return (

<h1>hello , {returnName}</h1>

);

}

});

React.render(

<HtmlMessage/>,

document.getElementById('example')

);

### 4.||判断

若前值为真则返回前值，不判断后值 否则直接返回后值

var HtmlMessage = React.createClass({

render:function(){

return (

<h1>hello , {this.props.name || 'world3'}</h1>

);

}

});

React.render(

<HtmlMessage name='Mr.Hy'/>,

document.getElementById('example')

);

## 函数表达式

### 将函数放入render中 强制求值

var name = 'liubei';

var HtmlMessage = React.createClass({

render:function(){

return (

<h1>hello ,

{

(function(obj){

if(obj.props.name){

return obj.props.name;

}else{

return 'world';

}

})(this)

}

</h1>

);

}

});

React.render(

<HtmlMessage/>,

document.getElementById('example')

);

## 非DOM属性

### dangerouslySetInnerHTML

var innerHtml = {

\_\_html: "<h1>I'm inner HTML!!!</h1>"

};

React.render(

<div dangerouslySetInnerHTML={innerHtml}></div>,

document.getElementById('con')

);

## 生命周期

### 初始化阶段

可以使用的函数：

getDefaultProps: 只调用一次，实例之间共享引用

getInitialState: 初始化每个实例特有的状态

componentWillMount: render之前最后一次修改状态的机会

render: 只能访问this.props和this.state,只有一个顶层组件，不允许修改状态和DOM输出

componentDidMount: 成功render并渲染完成真实DOM之后触发，可以修改DOM

实例如下：

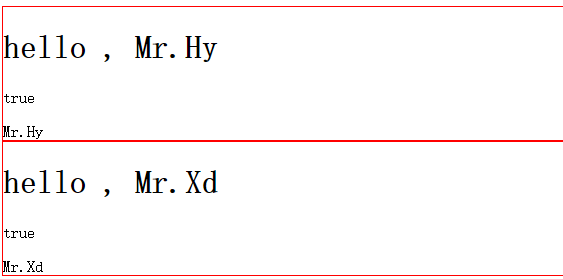
以下实例触发顺序为1234 2345

在componentDidMount中

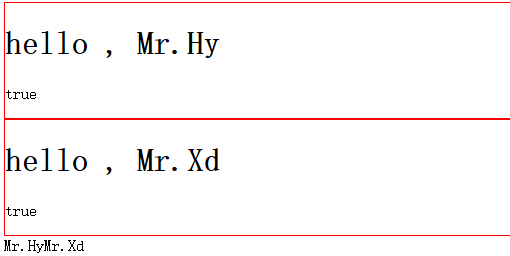
$(React.findDOMNode(this))表示获取当前组件的节点

当前操作相当于一个预操作 只是在所有组件渲染完成之后才触发 如下例:

$(React.findDOMNode(this)).append(""+this.props.name+"");



$("#example").append(""+this.props.name+"");



$(function(){

var count=0;

var myStyle={

border:"1px solid red",

};

var HtmlMessage = React.createClass({

getDefaultProps:function(){

console.log("getDefaultProps",1);

return {

name:'Tom'

};

},

getInitialState:function(){

console.log("getDefaultProps",2);

return {

myCount:count++,

ready:false

};

},

componentWillMount:function(){

console.log("getDefaultProps",3);

this.setState({ready:true});

},

render:function(){

console.log("getDefaultProps",4);

return (

<div ref="childp" style={myStyle}>

<h1>hello , {this.props.name ? this.props.name:"world"}</h1>

<p>{"" + this.state.ready}</p></div>

);

},

componentDidMount:function(){

console.log("getDefaultProps",5);

$(React.findDOMNode(this)).append(""+this.props.name+""); }

});

React.render(

<div>

<HtmlMessage name='Mr.Hy'/>

<HtmlMessage name='Mr.Xd'/>

</div>,

document.getElementById('example')

);

});

### 运行中阶段

运行中阶段可以使用的函数：

componentWillReceiveProps:父组件修改属性触发，可以修改新属性和状态

shouldComponentUpdate:返回false会阻止render调用

componentWillUpdate:不能修改属性和状态

render:只能访问this.props和this.state,只有一个顶层组件，不允许修改状态和DOM输出

componentDidUpdate:可以修改DOM

实例如下：

输入前：4

输入后：12345

$(function(){

var HelloMessage = React.createClass({

componentWillReceiveProps:function(newProps){

console.log("componentWillReceiveProps 1");

console.log(newProps);

},

shouldComponentUpdate:function(){

console.log("shouldComponentUpdate 2");

return true;

},

componentWillUpdate:function(){

console.log("componentWillUpdate 3");

},

render:function(){

console.log("render 4");

return (

<div>

<span>hello,{this.props.name? this.props.name:"world"}</span>

</div>

);

},

componentDidUpdate:function(){

console.log("componentDidUpdate 5");

// $(React.findDOMNode(this)).append(""+this.props.name+"");

}

});

var HelloUniverse = React.createClass({

getInitialState:function(){

return {

name:''

};

},

handleChange:function(event){

this.setState({name:event.target.value});

},

render:function(){

return (

<div>

<HelloMessage name={this.state.name}/>

<br/>

<input type="text" onChange={this.handleChange}/>

</div>

);

}

});

React.render(

<HelloUniverse/>,

document.getElementById('example')

);

});

### 销毁阶段

componentWillUnmount:在删除组件之前进行清理操作，比如计时器和时间监听器

$(function(){

var HelloMessage = React.createClass({

render:function(){

console.log("render 4");

return (

<div>

<span>hello,{this.props.name? this.props.name:"world"}</span>

</div>

);

},

componentWillUnmount:function(){

console.log("BOOOOOOOOOOOOOOOOOOOOOOM!!!");

}

});

var HelloUniverse = React.createClass({

getInitialState:function(){

return {

name:''

};

},

handleChange:function(event){

this.setState({name:event.target.value});

},

render:function(){

if(this.state.name == "123"){

return (

<div>之前组件已销毁</div>

);

}else{

return (

<div>

<HelloMessage name={this.state.name}/>

<br/>

<input type="text" onChange={this.handleChange}/>

</div>

);

}

}

});

React.render(

<HelloUniverse/>,

document.getElementById('example')

);

});

## 属性和状态

### 属性

①将多个属性展开传入组件

var HelloMessage = React.createClass({

render:function(){

return (

<div>

<span>{this.props.name1+" "+this.props.name2}</span>

</div>

);

},

});

var HelloUniverse = React.createClass({

getInitialState:function(){

return {

name1:'张三',

name2:'李四'

};

},

render:function(){

return (

<div>

<HelloMessage {...this.state}/>

</div>

);

}

});

React.render(

<HelloUniverse/>,

document.getElementById('example')

);

①通过渲染好的组件向组件传入属性

var HelloMessage = React.createClass({

render:function(){

return (

<div>

<span>{this.props.name}</span>

</div>

);

},

});

var instance = React.render(

<HelloMessage/>,

document.getElementById('example')

);

instance.setProps({name:'张三'});

### 属性和状态的区别

组件不能对属性进行修改 能修改的数据是状态

实时更改的数据 不要在getInitialState中进行接收

子组件：

var Content = React.createClass({

getInitialState:function(){

return {

inputText:""

};

},

handleChange:function(event){

this.setState({inputText:event.target.value});

},

handleSubmit:function(){

console.log("reply to:"+this.props.selectedName+"\n"+this.state.inputText);

},

render:function(){

return (

<div>

<textarea placeholder="please enter..." onChange={this.handleChange}>

</textarea>

<button onClick={this.handleSubmit}>提交</button>

</div>

);

}

});

父组件：

var Comment = React.createClass({

getInitialState:function(){

return {

names:["张三","李四","王五"],

selectedName:''

};

},

handleSelect:function(event){

this.setState({selectedName:event.target.value});

},

render:function(){

var options = [];

for(var option in this.state.names){

options.push(

<option value={this.state.names[option]}>

{this.state.names[option]}

</option>

)

};

return (

<div>

<select onChange={this.handleSelect}>

{options}

</select>

<Content selectedName={this.state.selectedName}/>

</div>

);

}

});

React.render(

<Comment/>,

document.getElementById("example")

);

## Event事件

### 鼠标事件

var ScrollWheel = React.createClass({

getInitialState:function(){

return {

backgroundColor: '#FFFFFF',

border:'1px solid black',

width: '200px',

height:'100px'

};

},

handleWheel:function(event){

var newColor = (parseInt(this.state.backgroundColor.substr(1),16)+

event.deltaY\*997).toString(16);

newColor = "#"+newColor.substr(newColor.length-6).toUpperCase();

this.setState({backgroundColor:newColor});

},

render:function(){

console.log(this.state);

return (

<div onWheel={this.handleWheel} style={this.state}>

在此div中鼠标滚动

</div>

);

}

});

React.render(

<ScrollWheel/>,

document.getElementById('example')

);

### 键盘事件

var ScrollWheel = React.createClass({

getInitialState:function(){

return {

password:''

};

},

handleKeyPress:function(event){

this.setState({

password:this.state.password+event.which

});

console.log(this.state);

},

handleChange:function(event){

},

render:function(){

return (

<div>

<input onKeyPress={this.handleKeyPress} onChange={this.handleChange}/>

<p style={{'display': this.state.password.indexOf('495051') >= 0 ? 'inline' : 'none'}}>You got it!</p>

</div>

);

}

});

React.render(

<ScrollWheel/>,

document.getElementById('example')

);

### 鼠标坐标显示

var ScrollWheel = React.createClass({

getInitialState:function(){

return {

x:'',

y:''

};

},

handleMouseMove:function(event){

this.setState({

x:event.clientX,

y:event.clientY

});

},

render:function(){

return (

<div onMouseMove={this.handleMouseMove} style={{

border:'1px solid black',

width:'200px',

height:'100px'

}}>

{this.state.x+","+this.state.y}

</div>

);

}

});

React.render(

<ScrollWheel/>,

document.getElementById('example')

);

## 组件的协同使用

### 父子组件

父组件通过属性与子组件进行交互

下例中 子组件中的Onchange方法 调用的是父组件的handleSelect方法

子组件

var GenderSelect = React.createClass({

render:function(){

return (

<select onChange={this.props.handleSelect}>

<option value='1'>男</option>

<option value='2'>女</option>

</select>

);

}

});

父组件

var SignupForm = React.createClass({

getInitialState:function(){

return {

name:'',

password:'',

gender:''

};

},

handleChange:function(dataName,event){

var newState = {};

newState[dataName] = event.target.value;

this.setState(newState);

},

handleSelect:function(event){

this.setState({

gender:event.target.value

});

},

render:function(){

console.log(this.state);

return (

<form>

<input type='text' placeholder='请输入用户名' onChange={this.handleChange.bind(this,'name')}/>

<input type='password' placeholder='请输入密码' onChange={this.handleChange.bind(this,'password')}/>

<GenderSelect handleSelect={this.handleSelect}/>

</form>

);

}

});

React.render(

<SignupForm/>,

document.getElementById('example')

);

### Mixin

抽取函数 实现切面 完成数据的双向绑定

var BindingMixin = {

handleChange:function(dataName){

var that = this;

return function(event){

var newState = {};

newState[dataName] = event.target.value;

that.setState(newState);

}

}

};

var BindingExample = React.createClass({

mixins:[BindingMixin],

getInitialState:function(){

return {

username:'',

password:''

};

},

render:function(){

return (

<div>

<input type='text' onChange={this.handleChange('username')}/>

<input type='password' onChange={this.handleChange('password')}/>

<br/>

<span>用户名：{this.state.username}</span>

<br/>

<span>密码：{this.state.password}</span>

</div>

);

}

});

React.render(

<BindingExample/>,

document.getElementById('example')

);

## 表单详解

### 可控组件

event.preventDefault()防止页面刷新

var MyForm = React.createClass({

getInitialState:function(){

return {

content:'hello world'

};

},

handleChange:function(event){

this.setState({

content:event.target.value

});

},

handleSubmit:function(event){

event.preventDefault();

alert(this.state.content);

},

render:function(){

return (

<form onSubmit={this.handleSubmit}>

<input type='text' value={this.state.content} onChange={this.handleChange}/>

<button type='submit'>submit</button>

</form>

);

}

});

React.render(

<MyForm/>,

document.getElementById('example')

);

### 处理函数复用

var BindingMixin = {

handleChange:function(dataName){

var that = this;

return function(event){

var newState = {};

newState[dataName] = dataName=='checked' ? event.target.checked : event.target.value;

that.setState(newState);

}

}

};

var BindingExample = React.createClass({

mixins:[BindingMixin],

getInitialState:function(){

return {

username:'',

password:'',

checked:false

};

},

render:function(){

console.log(this.state);

return (

<div>

<input type='text' onChange={this.handleChange('username')}/>

<input type='password' onChange={this.handleChange('password')}/>

<input type='checkbox' value='确定用户协议' onChange={this.handleChange('checked')}/>

<br/>

<span>用户名：{this.state.username}</span>

<br/>

<span>密码：{this.state.password}</span>

</div>

);

}

});

React.render(

<BindingExample/>,

document.getElementById('example')

);

# 腾讯课堂React

## 基本例子

### 与极客学院React的不同

<script type="text/jsx">

定义一个组件

class Comp extends React.Component{

必须要有构造

constructor(...args){

super(...args);

this.state={

name:''

};

}

handleChange(event){

this.setState({name:event.target.value})

}

render(){

return (

<div>

调用方法时需要绑定 .bind(this)

<input type="text" onChange={this.handleChange.bind(this)} />

<span>{this.state.name}</span>

</div>

);

}

}

React.render(

<Comp/>,

document.getElementById('example')

);

</script>

# React-Redux

## 介绍

### Action



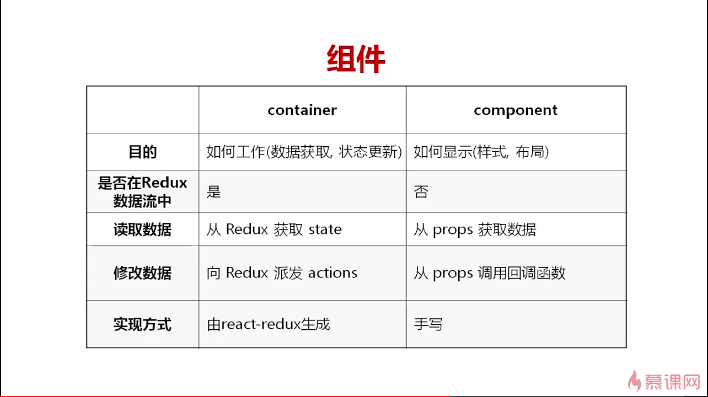
### reducer



### Store



### 组件



### 