**React服务器端渲染**

**1. renderToString**

如果是写过React的小伙伴们，相信对react-dom包的render的方法再熟悉不过，一般我们都是这么写：

**import** {render} **from** 'react-dom';

**import** App **from** './app';

render(<App/>,document.getElementById("root"));

但是我们发现render只是react-dom里的一部分，其他的方法不知道小伙伴们有没有去研究过，比如renderToString这个方法。React可以将React元素渲染成它的初始化Html，并且返回html字符串。然后在服务端生成html。

**const** express = require('express');

**const** app = express();

**const** React = require('react');

**const** {renderToString} = require('react-dom/server');

**const** App = **class** **extends** **React**.**PureComponent**{

render(){

**return** React.createElement("h1",null,"Hello World");;

}

};

app.get('/',**function**(req,res){

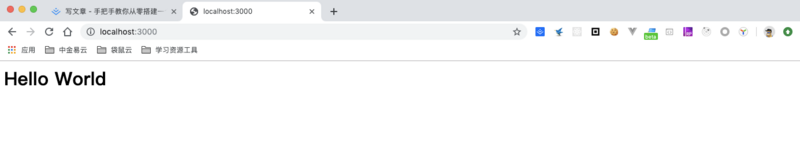
**const** content = renderToString(React.createElement(App));

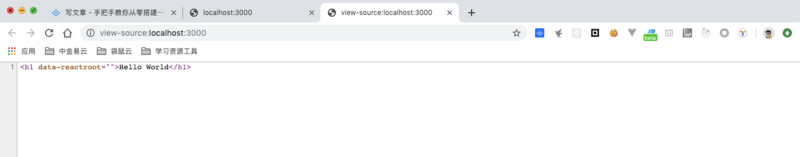
res.send(content);

});

app.listen(3000);

我们接下来验证一下。

  
再看一下网页的源代码。

  
源代码里也有“Hello World”，如果是客户端渲染的话，是无法看到了Hello World的。那么什么叫React的服务端渲染？不就是将react组件的内容可以在网页源代码里看到吗？刚刚这个例子就给我们展现了通过renderToString这个方法去实现服务端渲染。

**2. webpack**

通过上面这个例子我们发现了这么几个问题：

1. 不能jsx的语法
2. 只能commonjs这个模块化规范，不能用esmodule

因此，我们需要对我们的服务端代码进行一个webpack打包的操作，服务端的webpack配置需要注意这几点：

* 一定要有 **target:"node"** 这个配置项
* 一定要有webpack-node-externals这个库

所以你的webpack配置一定是这样子的：

const nodeExternals = require('webpack-node-externals');

**module**.exports = {

target: 'node', *//不将node自带的诸如path、fs这类的包打进去*

externals: [nodeExternals()],*//不将node\_modules里面的包打进去*

};

**3. 同构**

我们将前面的例子稍微做一下调整，然后变成如下这个样子：

**const** express = require('express');

**const** app = express();

**const** React = require('react');

**const** {renderToString} = require('react-dom/server');

**const** App = **class** **extends** **React**.**PureComponent**{

handleClick=(e)=>{

alert(e.target.innerHTML);

}

render(){

**return** <h1 onClick={this.handleClick}>Hello World!</h1>;

}

};

app.get('/',**function**(req,res){

**const** content = renderToString(<App/>);

console.log(content);

res.send(content);

});

app.listen(3000);

给h1这个标签绑定一个click事件，弹出h1标签里的内容。执行之后不会弹出“Hello World!”。因为renderToString只是返回html字符串，元素对应的js交互逻辑并没有返回给浏览器。先讲一下“**同构**”这个概念。何为“**同构**”，简单来说就是“**同种结构的不同表现形态**”。在这里我们用更通俗的大白话来解释react同构就是：同一份react代码在服务端执行一遍，再在客户端执行一遍。

同一份react代码，在服务端执行一遍之后，我们就可以生成相应的html。在客户端执行一遍之后就可以正常响应用户的操作。这样就组成了一个完整的页面。所以我们需要额外的入口文件去打包客户端需要的js交互代码。

**import** React **from** 'react';

**import** {render} **from** 'react-dom';

**import** App **from** './app';

render(<App/>,document.getElementById("root"));

这里就跟我们写客户端渲染代码无差，接着用webpack打包一下，然后再在渲染的html字符串中加入对打包后的js的引用代码。

**import** express **from** 'express';

**import** React **from** 'react';

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

**import** App **from** './src/app';

const app = express();

app.use(express.static("dist"))

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

const content = renderToString(<App/>);

res.send(`

<!doctype html>

<html>

<title>ssr</title>

<body>

<div id="root">${content}</div>

<script src="/client/index.js"></script>

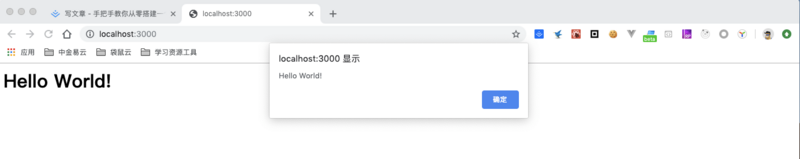
</body>

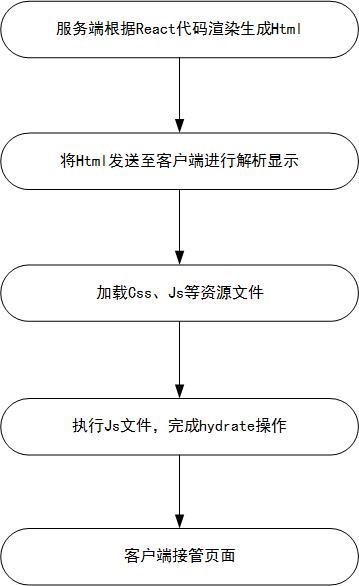
</html>

`);

});

app.listen(3000);

这里“/client/index.js”就是我们打包出来的用于客户端执行的js文件，然后我们再来看一下效果，此时页面就可以正常响应我们的操作了。  
**4. 流程图**



**5. StaticRouter**

在客户端渲染的项目中，react-router提供了BrowserRouter和HashRouter可供我们选择，而这两个router都有一个共同点就是需要读取地址栏的url，但是在服务端的环境中是没有地址栏的，也就是说没有window.location这个对象，那么BrowserRouter和HashRouter就不能在服务端的环境中去使用。react-router给我们提供了另外一种router叫做StaticRouter。  
现在我们有两个页面Login和User，代码如下：

**Login：**

**import** **React** from 'react';

export **default** **class** **Login** **extends** **React**.**PureComponent**{

render(){

**return** <div>登陆</div>

}

}

**User：**

**import** **React** from 'react';

export **default** **class** **User** **extends** **React**.**PureComponent**{

render(){

**return** <div>用户</div>

}

}

**服务端代码：**

**import** express **from** 'express';

**import** React **from** 'react';

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

**import** {StaticRouter,Route} **from** 'react-router';

**import** Login **from** '@/pages/login';

**import** User **from** '@/pages/user';

**const** app = express();

app.use(express.static("dist"))

app.get('\*',**function**(req,res){

**const** content = renderToString(<div>

<StaticRouter location={req.url}>

<Route exact path="/user" component={User}></Route>

<Route exact path="/login" component={Login}></Route>

</StaticRouter>

</div>);

res.send(`

<!doctype html>

<html>

<title>ssr</title>

<body>

<div id="root">${content}</div>

<script src="/client/index.js"></script>

</body>

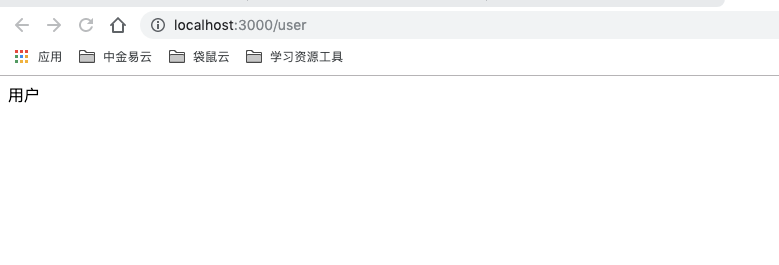
</html>

`);

});

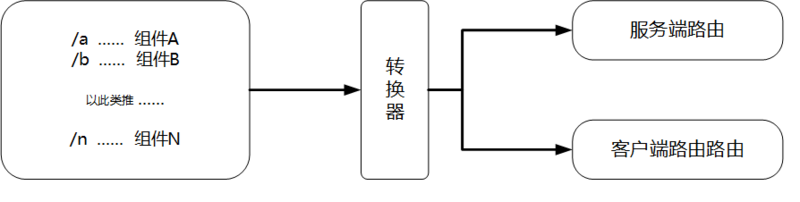
app.listen(3000);

最后的效果：

**/user：**  
  
**/login：**

**前后端路由同构**

不管是服务端路由还是客户端路由，路径和组件之间的关系是不变的，所以提供一个转换器，可以根据我们的需要去转换成服务端或者客户端路由。

**代码**

**routeConf.js**

**import** Login **from** '@/pages/login';

**import** User **from** '@/pages/user';

**import** NotFound **from** '@/pages/notFound';

**export** **default** [{

**type**:'redirect',

exact:true,

**from**:'/',

to:'/user'

},{

**type**:'route',

path:'/user',

exact:true,

component:User

},{

**type**:'route',

path:'/login',

exact:true,

component:Login

},{

**type**:'route',

path:'\*',

component:NotFound

}]

**router生成器**

**import** React **from** 'react';

**import** { createBrowserHistory } **from** "history";

**import** {Route,Router,StaticRouter,Redirect,Switch} **from** 'react-router';

**import** routeConf **from** './routeConf';

const routes = routeConf.map((conf,index)=>{

const {type,...otherConf} = conf;

**if**(type==='redirect'){

**return** <Redirect key={index} {...otherConf}/>;

}**else** **if**(type ==='route'){

**return** <Route key={index} {...otherConf}></Route>;

}

});

**export** const createRouter = (type)=>(params)=>{

**if**(type==='client'){

const history = createBrowserHistory();

**return** <Router history={history}>

<Switch>

{routes}

</Switch>

</Router>

}**else** **if**(type==='server'){

const {location} = params;

**return** <StaticRouter {...params}>

<Switch>

{routes}

</Switch>

</StaticRouter>

}

}

**客户端**

**createRouter**('client')()

**服务端**

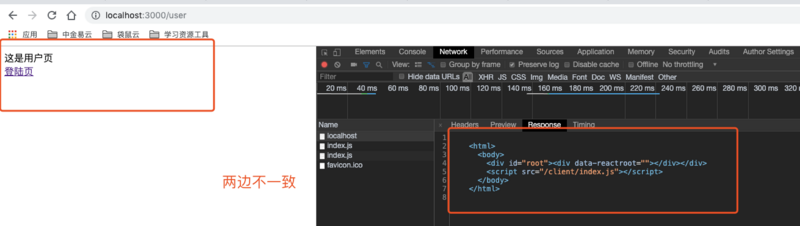
const context = {};

createRouter('server')({location:req.url,context}) //req.url来自node服务

**重定向问题**

**问题描述**

上面讲完前后端路由同构之后，我们发现一个小问题，虽然当url为“**/**”时，路由重定向到了“**/user**”，但是我们打开控制台会发现，返回的内容跟浏览器显示的内容是不一致的。从而我们可以得出这个重定向应该是客户端的路由帮我们做的重定向，但是这样是有问题的，我们想要的应该是要有两个请求，一个请求响应的状态码为302，告诉浏览器重定向到“**/user**”，另一个是浏览器请求“**/user**”下的资源。因此我们在服务端我们需要做个改造。



**改造代码**

**import** express **from** 'express';

**import** React **from** 'react';

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

**import** {createRouter} **from** '@/router';

**const** app = express();

app.use(express.static("dist"))

app.get('\*',**function**(req,res){

**const** context = {};

**const** content = renderToString(<div>

{createRouter('server')({

location:req.url,

context

})}

</div>);

*/\*\**

*\* ------重点开始*

*\*/*

*//当Redirect被使用时，context.url将包含重新向的地址*

**if**(context.url){

*//302*

res.redirect(context.url);

}**else**{

*//200*

res.send(`

<!doctype html>

<html>

<title>ssr</title>

<body>

<div id="root">${content}</div>

<script src="/client/index.js"></script>

</body>

</html>

`);

}

*/\*\**

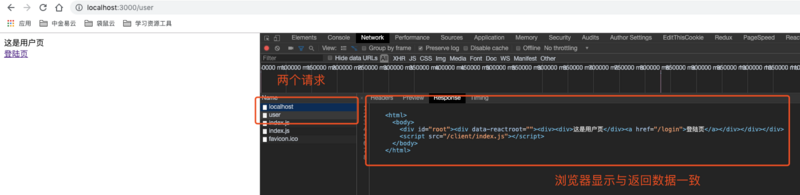
*\* ------重点结束*

*\*/*

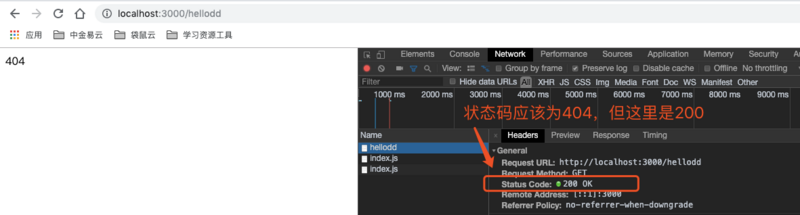
});

app.listen(3000);

这里我们只加了一层判断，检查context.url是否存在，存在则重定向到context.url，反之则正常渲染。因为这是react-router官方文档提供的判断是否有重定向的方式。



**404问题**

虽然我在routeConf.js中配置了404页面，但是会有一问题。  
  
虽然页面正常显示404，但是状态码却是200，这显然是不符合我们的要求的。这里借助StaticRouter的context,context里面我会放一个常量NOT\_FOUND来代表是否需要设置状态码404。

**服务端**

**import** express **from** 'express';

**import** React **from** 'react';

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

**import** {createRouter} **from** '@/router';

const app = express();

app.use(express.static("dist"))

app.get('\*',function(req,res){

const context = {};

const content = renderToString(<div>

{createRouter('server')({

location:req.url,

context

})}

</div>);

//当Redirect被使用时，context.url将包含重新向的地址

**if**(context.url){

//302

res.redirect(context.url);

}**else**{

**if**(context.NOT\_FOUND) res.status(404);//判断是否设置状态码为404

res.send(`

<!doctype html>

<html>

<title>ssr</title>

<body>

<div id="root">${content}</div>

<script src="/client/index.js"></script>

</body>

</html>

`);

}

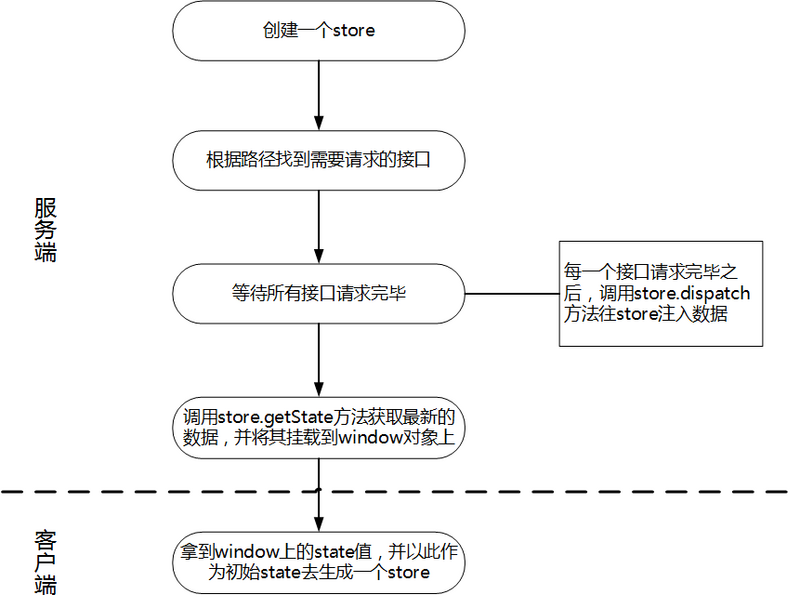
});

app.listen(3000);

主要就是加了这行代码，通过context.NOT\_FOUND是否为true来选择是否设置状态码为404。

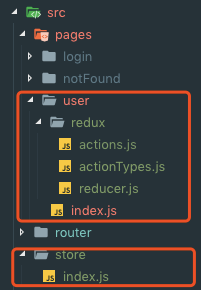
**if***(context.NOT\_FOUND)* res.status*(404)*;*//判断是否设置状态码为404*

**加入redux**



**基本框架**

首先先展示一下基本目录结构：



页面user下面一个redux文件夹，里面有这三个文件：

1. actions.js （集合redux-thunk将dispatch封装到一个个函数里面，解决action类型的记忆问题）
2. actionTypes.js （reducer所需用的action的类型）
3. reducer.js （常规reducer文件）

**actions.js**

**import** {CHANGE\_USERS} **from** './actionTypes';

**export** const changeUsers = (users)=>(dispatch,getState)=>{

dispatch({

type:CHANGE\_USERS,

payload:users

});

}

**actionTypes.js**

**export** **const** CHANGE\_USERS = 'CHANGE\_USERS';

**reducer.js**

import {CHANGE\_USERS} **from** './actionTypes';

const initialState = {

users:[]

}

export **default** (**state** = initialState, action)=>{

const {type,payload} = action;

switch (type) {

case CHANGE\_USERS:

return {

...**state**,

users:payload

}

**default**:

return **state**;

}

}

**/store/index.js**  
这个文件的作用就是对多有reducer做一个整合，向外暴露创建store的方法。

**import** { createStore, applyMiddleware,combineReducers } **from** 'redux';

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

**import** user **from** '@/pages/user/redux/reducer';

**const** rootReducer = combineReducers({

user

});

**export** **default** () => {

**return** createStore(rootReducer,applyMiddleware(thunk))

};

**至于为啥不直接暴露store，原因很简单。主要原因是由于这是单例模式。**

**数据的获取以及注入**

**路由改造**

**routeConf.js**

export default [{

type:'redirect',

exact:true,

from:'/',

to:'/user'

},{

type:'route',

path:'/user',

exact:true,

component:User,

loadData:User.loadData //服务端获取数据的函数

},{

type:'route',

path:'/login',

exact:true,

component:Login

},{

type:'route',

path:'\*',

component:NotFound

}]

首先我们要对前面提到的routeConf.js进行改造。

**页面改造**

在路由改造中，我们提到了需要加一个loadData方法。

**import** React **from** 'react';

**import** {Link} **from** 'react-router-dom';

**import** {bindActionCreators} **from** 'redux';

**import** {connect} **from** 'react-redux';

**import** axios **from** 'axios';

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

@connect(

state=>state.user,

dispatch=>bindActionCreators(actions,dispatch)

)

**class** **User** **extends** **React**.**PureComponent**{

**static** loadData=(store)=>{

*//axios本身就是基于Promise封装的，因此axios.get()返回的就是一个Promise对象*

**return**

axios.get('http://localhost:3000/api/users').then((response)=>{

**const** {data} = response;

**const** {changeUsers} =bindActionCreators(actions,store.dispatch);

changeUsers(data);

});

}

render(){

**const** {users} = **this**.props;

**return** <div>

<table>

<thead>

<tr>

<th>姓名</th>

<th>身高</th>

<th>生日</th>

</tr>

</thead>

<tbody>

{

users.map((user)=>{

const {name,birthday,height} = user;

return <tr key={name}>

<td>{name}</td>

<td>{birthday}</td>

<td>{height}</td>

</tr>

})

}

</tbody>

</table>

</div>

}

}

**export** **default** User;

1. **loadData必须有一个参数接受store，返回必须是一个Promise对象**
2. **必须有store.dispatch这个步骤**

**服务端改造**

**import** express **from** 'express';

**import** React **from** 'react';

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

**import** {Provider} **from** 'react-redux';

**import** {createRouter,routeConfs} **from** '@/router';

**import** { matchPath } **from** "react-router-dom";

**import** getStore **from** '@/store';

const app = express();

app.use(express.static("dist"))

app.get('/api/users',function(req,res){

res.send([{

"name":"小明",

"birthday":"1984-03-03",

"height":"161"

},{

"name":"小红",

"birthday":"1987-12-24",

"height":"158"

},{

"name":"小兰",

"birthday":"1988-08-04",

"height":"158"

},{

"name":"小黑",

"birthday":"1996-02-22",

"height":"165"

}]);

});

app.get('\*',function(req,res){

const context = {};

const store = getStore();

const promises = [];

routeConfs.forEach((route)=> {

const match = matchPath(req.path, route);

**if**(match&&route.loadData){

promises.push(route.loadData(store));

};

});

Promise.all(promises).**then**(()=>{

const content = renderToString(<Provider store={store}>

{createRouter('server')({

location:req.url,

context

})}

</Provider>);

**if**(context.url){

res.redirect(context.url);

}**else**{

res.send(`

<html>

<head>

<title>ssr</title>

<script>

window.INITIAL\_STATE = ${JSON.stringify(store.getState())}

</script>

</head>

<body>

<div id="root">${content}</div>

<script src="/client/index.js"></script>

</body>

</html>

`);

}

});

});

app.listen(3000);

集合上面的代码，我们来说一下需要改动哪几点：

1. **根据路径找到所有需要调用的loadData方法，接着传人store调用去获取Promise对象，然后加入到promises数组中。**
2. **加入Promise.all，将渲染react组件生成html等操作放入其then中。**
3. **在html中加入一个script，使新的state挂载到window对象上。**

**客户端改造**

**const** getStore = (initialState) => {

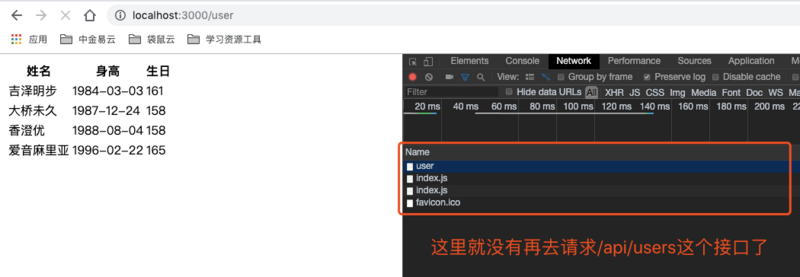
**return** createStore(rootReducer,initialState,applyMiddleware(thunk))

};

**const** store = getStore(window.INITIAL\_STATE);

客户端的改造相对简单了很多，主要就两点：1) getStore支持传入一个初始state。2) 调用getStore，并传入window.INITIAL\_STATE，从而获得一个注入数据的store。

**最终效果图**





**处理css样式**

**解决报错**

module:{

rules:[{

test:/\.css$/,

use: [

'style-loader',

{

loader: 'css-loader',

options: {

modules: true

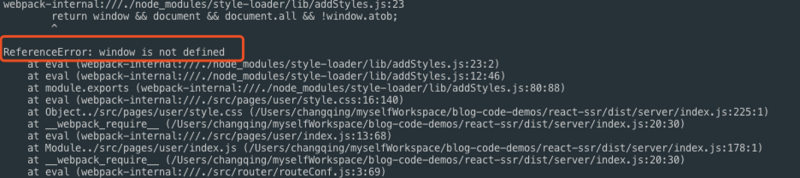
}

}]

}]

}

上面这段webpack配置相信大家并不陌生，是用来处理css样式的，但是这段配置要是放在服务端的webpack配置便出现下面这个问题。

  
从上图的报错来看，提示window未定义，因为是服务端是node环境，因此也就没有window对象。只要style-loader替换掉即可，替换成isomorphic-style-loader即可。

module:{

rules:[{

test:/\.css$/,

use: [

'isomorphic-style-loader',

{

loader: 'css-loader',

options: {

modules: true

}

}]

}]

}