**React服务器端渲染**

**一、**简易的React服务端渲染

**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){

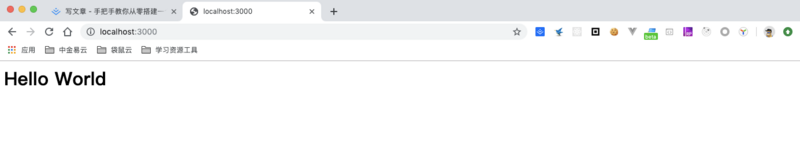
**const** content = renderToString(React.createElement(App));

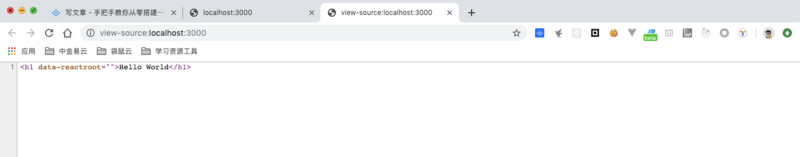
res.send(content);

});

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>

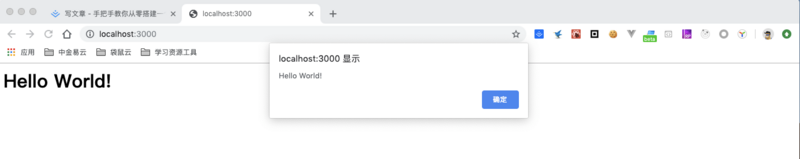
</body>

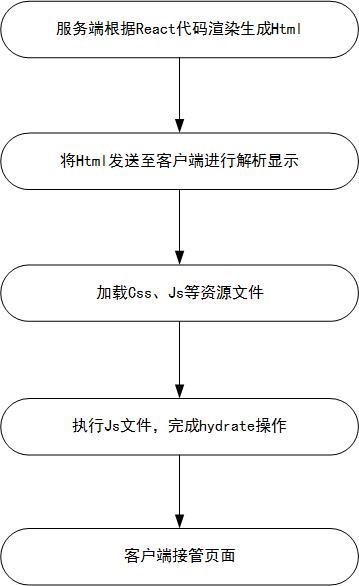
</html>

`);

});

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，那么这就无解了吗，当然不是！react-router给我们提供了另外一种router叫做StaticRouter，react-router官网给它的描述中有这么一句话。

This can be useful **in** server-side rendering scenarios when the **user** **isn**’t actually clicking around, so the **location** **never** actually changes.

我们画一下重点“**be useful in server-side rendering scenarios**”，意思很明确，就是为了服务端渲染而打造的。接下来我们就结合StaticRouter是去实现一下。  
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>

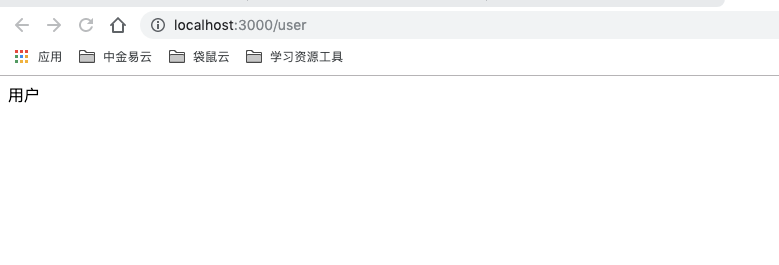
</html>

`);

});

app.listen(3000);

最后的效果：

**/user：**  
  
**/login：**

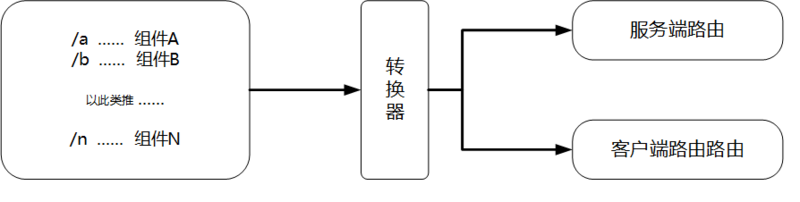


**前后端路由同构**

通过上面的小实验，我们已经掌握了如何在服务端去添加路由，接下来我们需要处理的就是前后端路由同构的问题。由于前后端都需要添加路由，如果两端都各自写一遍的话，费时费力不说，维护还很不方便，所以我们希望一套路由可以在两端都跑通。接下来我们就去实现一下。

**思路**

这里先说下思路，首先我们肯定不会在服务器端写一遍路由，然后在去客户端写一遍路由，这样费时费力不说，而且不易维护。相信大多数人和我一样都希望少写一些代码，少写代码的第一要义就是把通用的部分给抽出来。那么接下来我们就找一下通用的部分，仔细观察不难发现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，a路径对应a组件，b路径对应b组件，所以这里我们希望路径和组件之间的关系可以用抽象化的语言去描述清楚，也就是我们所说路由配置化。最后我们提供一个转换器，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 = {};

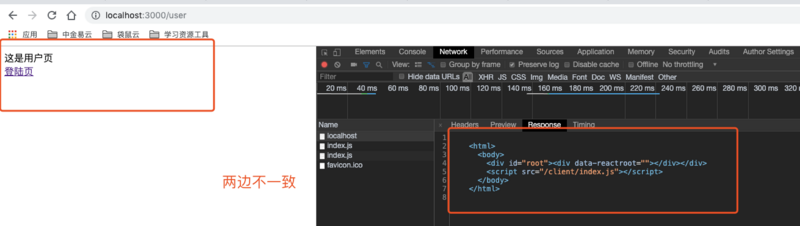
createRouter('server')({location:req.url,context}) //req.url来自node服务

这里给的只是单层路由的实现，在实际项目中我们会更多的使用嵌套路由，但是二者原理是一样的，嵌套路由的话，小伙伴私下可以实现一下哦！

**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**”了，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*

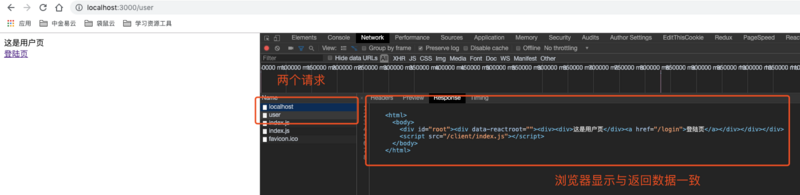
*\*/*

});

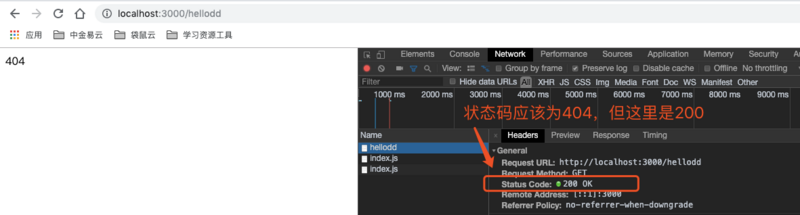
app.listen(3000);

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。至于为什么这么判断，因为这是react-router官方文档提供的判断是否有重定向的方式，有兴趣的小伙伴可以看一下文档以及源码。文档地址如下：

[https://reacttraining.com/rea...](https://reacttraining.com/react-router/web/api/StaticRouter)



**404问题**

虽然我在routeConf.js中配置了404页面，但是会有一问题，我们来看一下这张图。  
  
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，因此我们需要改造一下。这里我的思路是借助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，放置的时机我选择在Route的render方法里面去设置,具体代码如下。

**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*

**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,

loadData:User.loadData

},{

**type**:'route',

path:'/login',

exact:true,

component:Login

},{

**type**:'route',

path:'\*',

*//将component替换成render*

render:({staticContext})=>{

**if** (staticContext) staticContext.NOT\_FOUND = true;

**return** <NotFound/>

}

}]

这里我没有选择直接在NotFound组件constructor生命周期上修改，主要考虑这个404组件日后可能会给其他客户端渲染的项目用，尽量保持组件的通用性。

*//改造前*

component:NotFound

*//改造*

render:({staticContext})=>{

**if** (staticContext) staticContext.NOT\_FOUND = true;

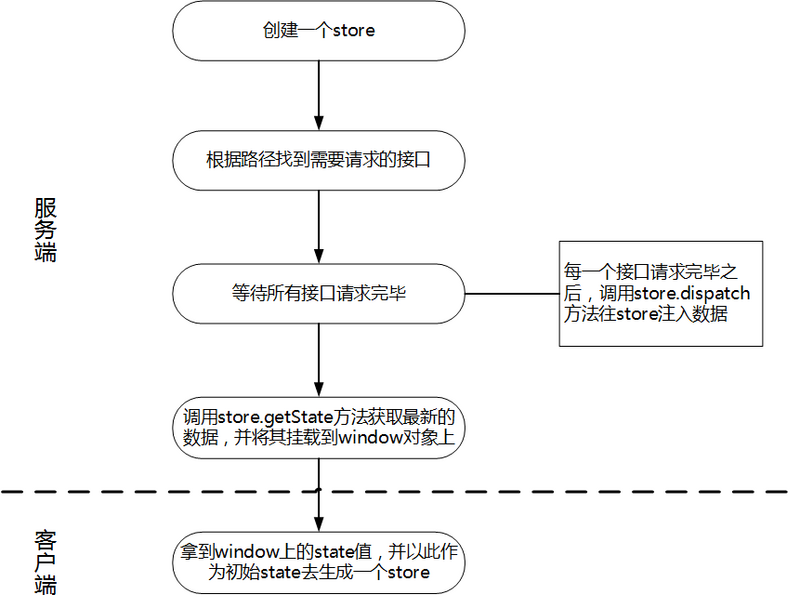
**return** <NotFound/>

}

**加入redux**

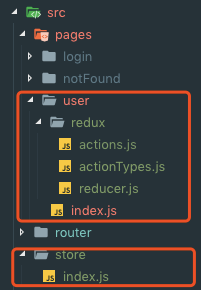
第一次接触服务端渲染的小伙伴们可能会不太理解这里为什么要单独讲一下如何将redux集成到服务端渲染的项目中去。因为前面在讲原理的时候，我们已经很清楚服务端的作用了，那就是根据访问路径生成对应的html，而redux是javaScript的状态容器，服务端渲染生成html也不需要这玩意儿啊，说白了就是客户端的东西，按照客户端的方式集成不就完了，干嘛还非得单独拎出来讲一下？

这里我就要解释一下了，以掘金的文章详情页为例，假设掘金是服务端渲染的项目，那么每一篇文章的网页源代码里面应该是要包含该文章的完整内容的，这也就意味着接口请求是在服务端渲染html之前就去请求的，而不是在客户端接管页面之后再去请求的，所以服务端拿到请求数据之后得找个地方以供后续的html渲染使用。我们先看看客户端拿到请求数据是如何存储的，一般来说无外乎这两种方式，一个就是组件的state，另一个就是redux。再回到服务端渲染，这两种方式我们一个个看一下，首先是放在组件的state里面，这种方式显然不可取的，都还没有renderToString呢，哪来的state，所以只剩下第二条路redux，redux的createStore第二参数就是用于传入初始化state的，我们可以通过这个方法实现数据注入，大致的流程图如下。



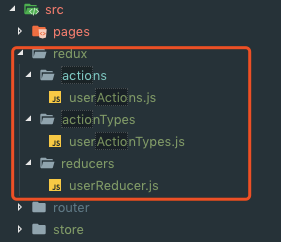
**基本框架**

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：

* 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）
* actionTypes.js （reducer所需用的action的类型）
* reducer.js （常规reducer文件）

有人可能会喜欢以下这种结构，把redux相关的东西放到一个文件夹里，然后分几个大类。但是我个人比较推崇将redux的东西随页面放在一起，这样做的好处就是找起来比较方便，易于维护。可以根据个人喜好来，我只是提供一种我的方式。  


**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，如果服务端对这个store的数据做了修改，那么这个修改将会一直被保留下来。简单来说a用户的访问会对b用户的访问产生影响，所以直接暴露store是不可取的。**

**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，改造的方式很简单，就是加上一个loadData方法，提供给服务端获取数据，至于User.loadData方法的具体实现我们放到后面再讲。看到这里或许有小伙伴会有这样的疑问，通过component就可以拿到User组件，拿到User组件不就可以拿到loadData方法了吗？为啥还要单独配一个loadData呢？针对这个疑问，我在这里做一下说明，就拿User为例，首先你不一定会用component也有可能会用render，其次component对应组件未必就是User，有可能会用类似react-loadable这样的库对User进行一个包装从而形成一个异步加载组件，这样就无法通过component去拿到User.loadData方法了。鉴于component的不确定性，保险起见还是单独配一个loadData更为稳妥一些。

**页面改造**

在路由改造中，我们提到了需要加一个loadData方法，接下来我们就实现一下。

**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;

render部分很简单，单纯地显示一个表格，表格有三列分别为姓名、身高、生日这三列。其中表格的数据来源于props的users，当通过接口获取到数据后通过changeUsers方法来修改props的users的值（这个说法其实不准确，users实际来源于store，然后通过connect方法注入到porps中，为了方便那么理解姑且这么说）。整个页面的主逻辑大致就是这样，接下来我们着重讲一下loadData需要注意的地方。

**loadData必须有一个参数接受store，返回必须是一个Promise对象**

必须要有一个参数接受store，这个比较好理解，根据前面画的流程图，我们是需要修改store里面的state的值的，没有store，修改state值就无从谈起。返回Promise对象主要因为javascript是异步的，但是我们需要等待数据请求完毕之后才能渲染react组件去生成html。当然这里选择callbak也可以实现，但是不推荐，Promise在处理异步上要比callback好太多了，网上有很多文章做了Promise和callback的对比，有兴趣的小伙伴们可以自行查阅，这里就不讨论了。除了Promise比callback在处理异步上更好之外，最主要的一点，同时也是callback的致命伤，就是在嵌套路由的情况下，我们需要调用多个组件的loadData，比如说下面这个例子。

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

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

**import** Header **from** '@/component/header';

**import** Footer **from** '@/component/footer';

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

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

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

static loadData = ()=>{

//请求数据

}

render(){

const {menus,data} = **this**.props;

**return** <div>

<Header menus={menus}/>

<Switch>

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

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

</Switch>

<Footer data={data}/>

</div>

}

}

当路径为/user时，我们不仅要调用User.loadData,还要调用App.loadData。如果是Promise，我们可以利用Promise.all来轻松解决多个异步任务的完成响应问题，相反用callback则变得非常复杂。

**必须有store.dispatch这个步骤**

这点其实也比较好理解，修改store的state的值只能通过调用store.dispatch来完成。但是这里你可以直接调用store.dispatch，或者像我集合第三方的redux中间件来实现，我这里用的redux-thunk，这里我将store.dispatch的操作封装在changeUsers中去了。

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

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

dispatch({

type:CHANGE\_USERS,

payload:users

});

}

**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);

**ps：这边加了一个接口“/api/users”是为了后续演示用的，不在改造范围内**

集合上面的代码，我们来说一下需要改动哪几点：

**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。**

这里使用的是react-router-dom提供的matchPath方法，因为这里是单级路由，matchPath方法足够了。但是如果多级路由的话，可以使用react-router-config这个包，具体使用方法我就不赘述了，官方文档介绍的肯定比我介绍的更加详细全面。另外有些小伙伴们的路由配置规则可能跟官方提供的不一样，就比如我自己在公司项目中的路由配置方式就跟官方的不一样，对于这种情况就需要自己写一个匹配函数了，不过也简单，一个递归的应用而已，相信难不倒大家。

**加入Promise.all，将渲染react组件生成html等操作放入其then中。**

前面我们讲过在多级路由中，会存在需要调用多个loadData的情况，用Promise.all可以非常好地解决多个异步任务完成响应的问题。

**在html中加入一个script，使新的state挂载到window对象上。**

根据流程图，客户端创建store的时候，得传入一个初始state，以达到数据注入的效果，这里挂载到window这个全局对象上也是为了方便客户端去获取这个state。

**客户端改造**

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

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

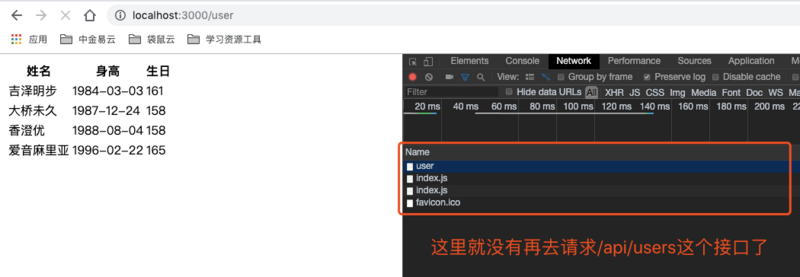
};

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

客户端的改造相对简单了很多，主要就两点：

* getStore支持传入一个初始state。
* 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**



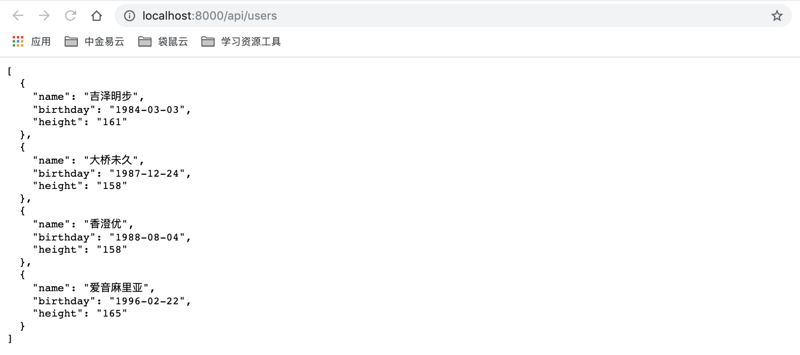


**node层加入接口代理**

在讲“**加入redux**”这个模块的时候，用到了一个“**/api/users**”接口，这个接口在写在当前服务上的，但是在实际项目中，我们更多地会是调用其他服务的接口。此外除了在服务端调用接口，客户端同样也有需要调用接口，而客户端调用接口就要面临着跨域的问题。因此在node层加入接口代理，不光可以实现多个服务的调用，也能解决跨域问题，一举两得。接口代理我选用http-proxy-middleware这个包，它可以作为express的中间件使用，具体的使用可以查看官方文档了，我这里就不赘述了，我就给个示例供大家参考一下。

**准备工作**

在使用http-proxy-middleware之前，先做点准备工作，目前“**/api/users**”是在当前服务上，为了方便演示，我先搭了一个基于json-server的简单mock服务，端口选用了8000，与当前服务的3000端口做一下区分。最后我们访问 “[http://localhost](http://localhost/):8000/api/users” 可以获取我们想要的数据，效果如下。



**开始配置**

操作很简单，就是在我们服务端加入一行代码就可以了。

**import** proxy from 'http-proxy-middleware';

app.use('/api',proxy({target: 'http://localhost:8000', changeOrigin: true }))

如果是多个服务的话,可以这么做。

**import** proxy **from** 'http-proxy-middleware';

**const** findProxyTarget = (path)=>{

console.log(path.split('/'));

**switch**(path.split('/')[1]){

**case** 'a':

**return** 'http://localhost:8000';

**case** 'b':

**return** 'http://localhost:8001';

**default**:

**return** "http://localhost:8002"

}

}

app.use('/api',**function**(req,res,next){

proxy({

target:findProxyTarget(req.path),

pathRewrite:{

"^/api/a":"/api",

"^/api/b":"/api"

},

changeOrigin: true })(req,res,next);

})

* /api/a/users => [http://localhost](http://localhost/):8000/api/users
* /api/b/users => [http://localhost](http://localhost/):8001/api/users
* /api/users => [http://localhost](http://localhost/):8002/api/users

不同的服务可以用不同的前缀来区分，通过这个额外的前缀就可以得出对应的target，另外记得用pathRewrite把额外的前缀给去掉，否则就会出现代理错误。

**处理css样式**

**解决报错**

module:{

rules:[{

test:/\.css$/,

use: [

'style-loader',

{

loader: 'css-loader',

options: {

modules: true

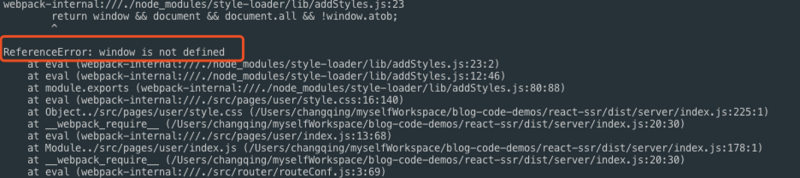
}

}]

}]

}

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。解决问题的办法也很简单，因为从图可知这个报错是来自style-loader的，只要style-loader替换掉即可，替换成isomorphic-style-loader即可。如是你要处理scss、less等等这些文件的话，方法也是一样的，替换style-loader为isomorphic-style-loader即可。

module:{

rules:[{

test:/\.css$/,

use: [

'isomorphic-style-loader',

{

loader: 'css-loader',

options: {

modules: true

}

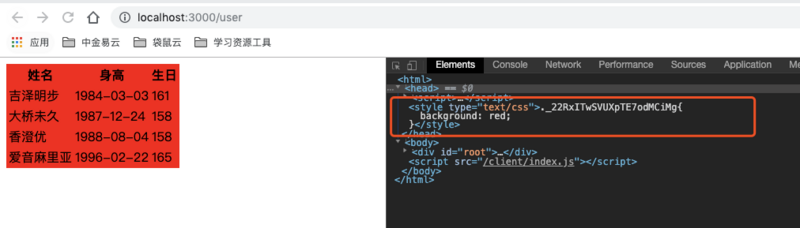
}]

}]

}

**潜在问题**

**问题描述**



  
从上面两张图，我们发现一个问题，控制台我们是可以看到style标签的，但是网页源代码却没有，这也就意味这个style标签是js帮我们生成的。这样不好的一点就是页面会闪一下，用户体验不是很好，希望网页源代码可以有这段css，如何实现呢？下面我们就来讲一下。

**解决办法**

isomorphic-style-loader的官方文档上提供了这个方法，大家可以去isomorphic-style-loader的官方文档查阅一下。这里我说一下需要注意的点。

**webpack**

webpack配置需要注意两点：

1. webpack.client.conf.js（客户端）和webpack.server.conf.js（服务端）style-loader必须替换成isomorphic-style-loader。
2. css-loader必须开启CSS Modules，及其options.modules=true。

**组件**

**import** withStyles **from** 'isomorphic-style-loader/withStyles';

**import** style **from** './style.css';

**export** **default** withStyles(style)(User);

操作步骤：

1. 引入withStyles方法。
2. 引入css文件。
3. 用withStyles包裹需要导出的组件。

**服务端**

**import** StyleContext **from** 'isomorphic-style-loader/StyleContext';

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

**const** css = **new** Set()

**const** insertCss = (...styles) => styles.forEach(style => css.add(style.\_getCss()))

**const** content = renderToString(

<StyleContext.Provider value={{ insertCss }}>

<Provider store={store}>

{createRouter('server')({

location:req.url,

context

})}

</Provider>

</StyleContext.Provider>

);

res.send(`

**<!doctype html>**

<html>

<head>

<title>ssr</title>

<style>${[...css].join('')}</style>

<script>

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

</script>

</head>

<body>

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

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

</body>

</html>

`);

})

操作步骤：

1. 引入StyleContext。
2. 新建Set对象css（为了保证唯一性这里选用Set）
3. 定义一个insertCss方法，内部逻辑很简单，调用每一个style对象的\_getCss方法获取css内容并加到之前定义的Set对象css中去。
4. 加入StyleContext.Provider，其value是一个对象，该对象里面有一个insertCss属性，该属性对应的值就是前面定义的insertCss方法。
5. 在渲染的模板中加入style标签，其内容为[...css].join('')

**客户端**

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

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

**import** StyleContext **from** 'isomorphic-style-loader/StyleContext'

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

**const** insertCss = (...styles) => {

**const** removeCss = styles.map(style => style.\_insertCss())

**return** () => removeCss.forEach(dispose => dispose())

}

hydrate(

<StyleContext.Provider value={{ insertCss }}>

<App />

</StyleContext.Provider>,

document.getElementById("root")

);

操作步骤：

1. 引入StyleContext
2. 定义insertCss方法，内部逻辑为先拿到传入的每一个style对象中\_insertCss执行过后的返回值，最后返回一个函数，该函数作用就是将前面拿到返回值再每一个执行一遍。
3. 加入StyleContext.Provider，其value是一个对象，该对象里面有一个insertCss属性，该属性对应的值就是前面定义的insertCss方法

**最终效果：**



**react-helmet**

说到seo优化，有一点大家一定可以答上来，那就是在head标签里加入title标签以及两个meta标签（keywords、description）。在单页面应用中，title和meta都是固定的，但是在多页应用中，不同页面的title和meta可能是不一样的，因此服务端渲染项目是需要支持title和meta的修改的。react生态圈已经有人做了一个库来帮助我们去实现这个功能，这个库就是react-helmet。这里我们说一下其基本使用，更多使用方法大家可以查看其官方文档。

**组件**

**import** {**Helmet**} from "react-helmet";

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

render(){

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

**return** <div>

<**Helmet**>

<title>用户页</title>

<meta name="keywords" content="user" />

<meta name="description" content={users.map(user=>user.name).join('，')} />

</**Helmet**>

</div>

}

}

操作步骤：

1. 在render方法中入一个React元素Helmet。
2. Helmet内部加入title、meta等数据。

**服务端**

**import** {Helmet} **from** "react-helmet";

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

**const** content = renderToString(

<StyleContext.Provider value={{ insertCss }}>

<Provider store={store}>

{createRouter('server')({

location:req.url,

context

})}

</Provider>

</StyleContext.Provider>

);

const helmet = Helmet.renderStatic();

res.send(`

**<!doctype html>**

<html>

<head>

${helmet.title.toString()}

${helmet.meta.toString()}

<style>${[...css].join('')}</style>

<script>

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

</script>

</head>

<body>

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

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

</body>

</html>

`);

})

操作步骤：

1. 执行Helmet.renderStatic()获取title、meta等数据。
2. 将数据绑定到html模版上。

注意事项:Helmet.renderStatic一定要在renderToString方法之后调用，否则无法获取到数据。