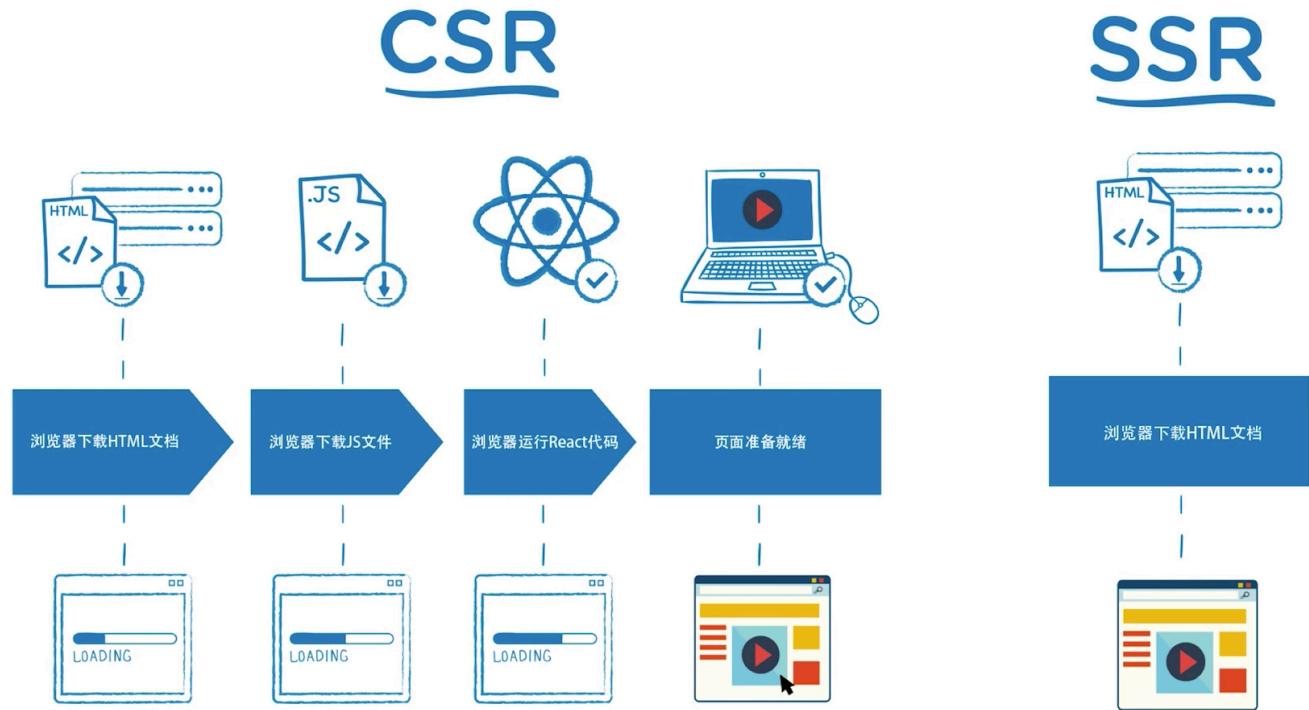


react-ssr

见



单页面应用弊端：

TTFB 事件比较长（首屏时间比较长）。

并不具有seo排名的条件。百度等搜索引擎爬虫并不能从 js文件 中获取数据。

浏览器发送请求

服务器运行 React 代码生成页面

服务器返回页面

2-1 ~ 2-2 在服务端编写react组件 服务端 webpack 的配置

webpack中target 属性的作用：（核心 表明打包后f的代码运行的宿主环境）

<https://www.dazhuanlan.com/andychina/topics/1124872>

类似下面两行代码。

服务器端打包的代码中不需要 path这个模块。

浏览器端打包的代码中需要path这个模块（因为浏览器环境没有path这个内置模块）。

```
1 // 服务器端
2 require('path');
3
4 // 浏览器端（客户端）
5 require('path');
6
7 bundle.js
8
9 module.exports = {
10   target: 'node',
11 }
```

presets 配置项的作用（图片中基础的配置已经完成）

```
module.exports = {
  target: 'node',
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'build')
  },
  module: {
    rules: [
      test: /\.js$/,
      loader: 'babel-loader',
      exclude: /node_modules/,
      options: {
        presets: ['react', 'stage-0', ['env', {
          targets: {
            browsers: ['last 2 versions']
          }
        }]]
      }
    ]
  }
}
```

babel 中 plugin 和 presets 的作用

<http://www.manongjc.com/detail/11-bbjbqsiycbzckcx.html>

Webpack-node-externals 作用

当我们已经制定了target == node 的话 我们引用类似 path fs 这种node 内置核心模块后 并不会将这些代码打包到node_moudle中， 当时我们引用express这种时候还是会被打包到 最终的bundles.js中。

```
target: 'node',
mode: 'development',
entry: './src/index.js',
output: {
  filename: 'bundle.js',
  path: path.resolve(__dirname, 'build')
},
externals: [nodeExternals()],
module: {
  rules: [
    test: /\.js?$/,
    loader: 'babel-loader',
    exclude: /node_modules/,
    options: {
      presets: ['react', 'stage-0', ['env', {
        targets: {
          browsers: ['last 2 versions']
        }
      }]]
    }
  ]
},
```

```
{  
  "name": "server",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "node ./build/bundle.js"  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "babel-core": "^6.26.3",  
    "babel-loader": "^7.1.5",  
    "babel-preset-env": "^1.7.0",  
    "babel-preset-react": "^6.24.1",  
    "babel-preset-stage-0": "^6.24.1",  
    "express": "^4.16.3",  
    "react": "^16.4.1",  
    "webpack": "^4.16.0",  
    "webpack-cli": "^3.0.8",  
    "webpack-node-externals": "^1.7.2"  
  }  
}
```

2-3 ~ 2.4 实现服务端组件渲染，建立在虚拟dom上的服务端渲染

1. ReactDOM.render.jsx(dom). 将我们的react组件挂载到dom上
2. 在server 端需要将react组件使用renderToString 方法将组件转为字符串返回给客户端
(ReactDOM.renderToString 方法可以将虚拟dom转化为字符串返回给浏览器端)

```
import express from 'express';
import Home from './containers/Home';
import React from 'react';
import { renderToString } from 'react-dom/server';

const app = express();
const content = renderToString(<Home />);

app.get('/', function (req, res) {
  res.send(`
    <html>
      <head>
        <title>ssr</title>
      </head>
      <body>
        ${content}
      </body>
    </html>
  `);
});

var server = app.listen(3000);
```

虚拟dom的好处

虚拟dom 是真实dom 的javascript 对象映射

1.提升页面的渲染性能

2.使得服务端渲染更加容易。实际虚拟dom 也可以做native 同一套虚拟dom 在web端可以将虚拟dom转化为真实dom 通过render 挂载到id = root上面，在服务端可以通过renderToString方法返回给客户端。native端可以将虚拟dom 转化为native端的元素展示在页面中

服务端渲染的弊端

对于服务器压力增大，原本浏览器进行的js运算放在了服务器端

```
// 客户端渲染
// React代码在浏览器上执行，消耗的是用户浏览器的性能

// 服务器端渲染
// React代码在服务器上执行，消耗的是服务器端的性能
```

<https://stackoverflow.com> 可以搜索一些报错问题

2.5 webpack 的自动打包和服务器的自动重启

1. watch 参数 可以监听相关文件（入口文件， 相关依赖文件）的变化。只要有文件变化就进行重新的编译 服务器重启 后改变的代码就生效。

```
"scripts": {
  "start": "node ./build/bundle.js",
  "build": "webpack --config webpack.server.js --watch"
},
```

2. nodemon 帮助node实现文件的监听， 上面watch 参数只能重新编译出新的代码，但是这个代码需要服务器重新重启才能生效

nodemon 监听 “1” 也就是build文件夹下面的变化 一旦有变化就执行 “2”的这个指令

superviser 也可以实现相同的效果

```
'scripts':{
  "start": "nodemon --watch build. --exec node './build/bundles.js/'",
  "build": "webpack --config webpack.server.js --watch"
}
```

上面watch 参数 以及nodemon的使用使得

1.只要入口文件/相关依赖文件发生的变化 就重新进行打包编译---->导致build 文件下bundles.js文件发生变化（产物变化） ---->这个build文件夹下面的变化被nodemon监听到 ----> nodemon执行了。 node ./builde/bundles.js 指令（这个文件中包含着服务器代码）， 服务被重启

2-6 npm-run-all 提升开发效率

npm-run-all 这个包可以运行多个指令(上节课需要连续执行npm run build 然后执行。 npm run start)

```
npm install npm-run-all -g
```

```
"scripts": {
  "dev": "npm-run-all --parallel dev:***",
  "dev:start": "nodemon --watch build --exec node './buil
  "dev:build": "webpack --config webpack.server.js --watc
},
```

3-1什么是同构

一套react代码在服务器端执行一次，在客户端执行一次

<https://www.jianshu.com/p/09037f948fab>

```
import React from 'react';

const Home = () => {
  return (
    <div>
      <div>This is Dell Lee!</div>
      <button onClick={()=>{alert('click')}}>
        click
      </button>
    </div>
  )
}

export default Home;
```

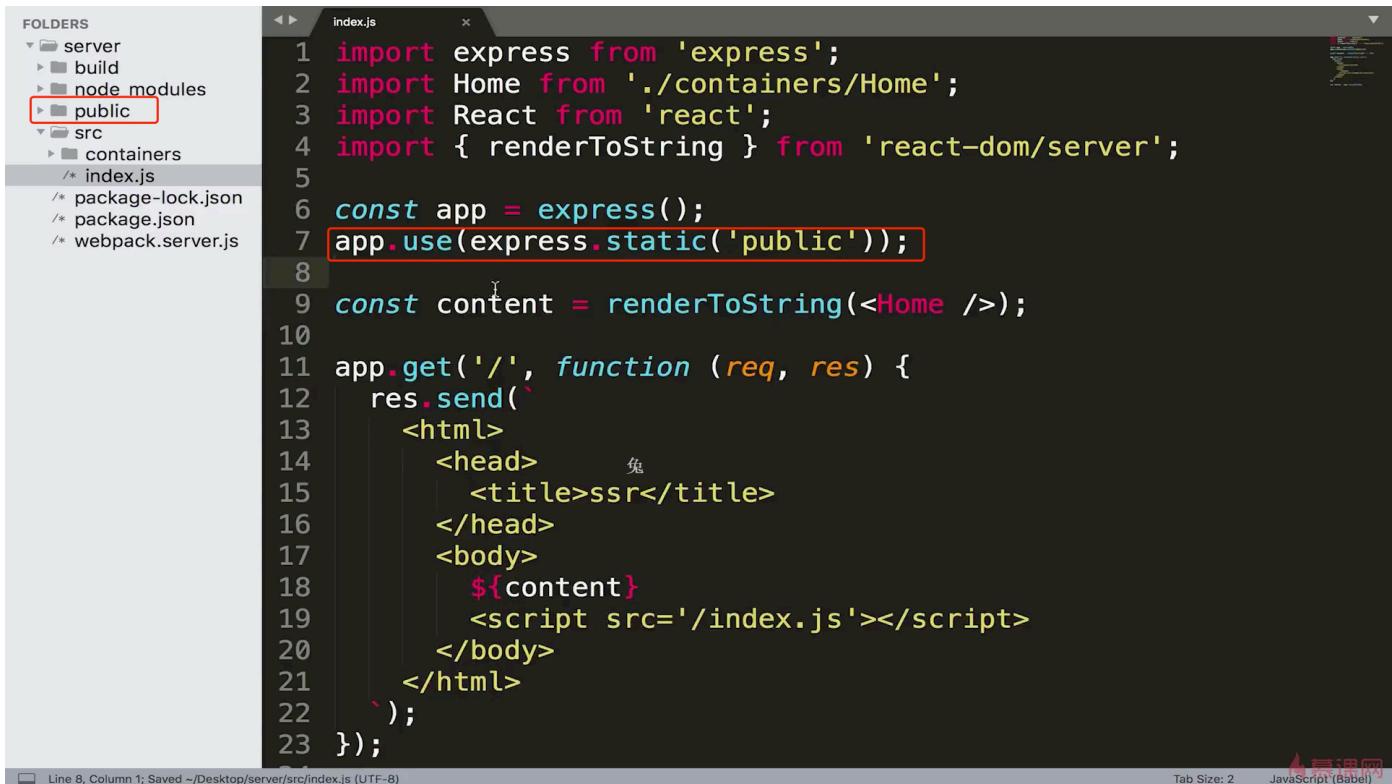
我们给Home组件的元素上点击了click事件但是不会生效，这是因为`renderToString`方法不会处理事件相关的东西，只会把组件ui相关的东西处理好返回给客户端。所以客户端需要再次运行下这个代码。

同构：就是一套React代码在服务器上运行一遍，到达浏览器又运行一遍。服务端渲染完成页面结构，浏览器端渲染完成事件绑定。

3-2 在浏览器上执行一段js代码

1. express 静态资源管理

`app.use(express.static("pathname"))` 只要请求的是静态资源，就会到 pathname 中进行查找



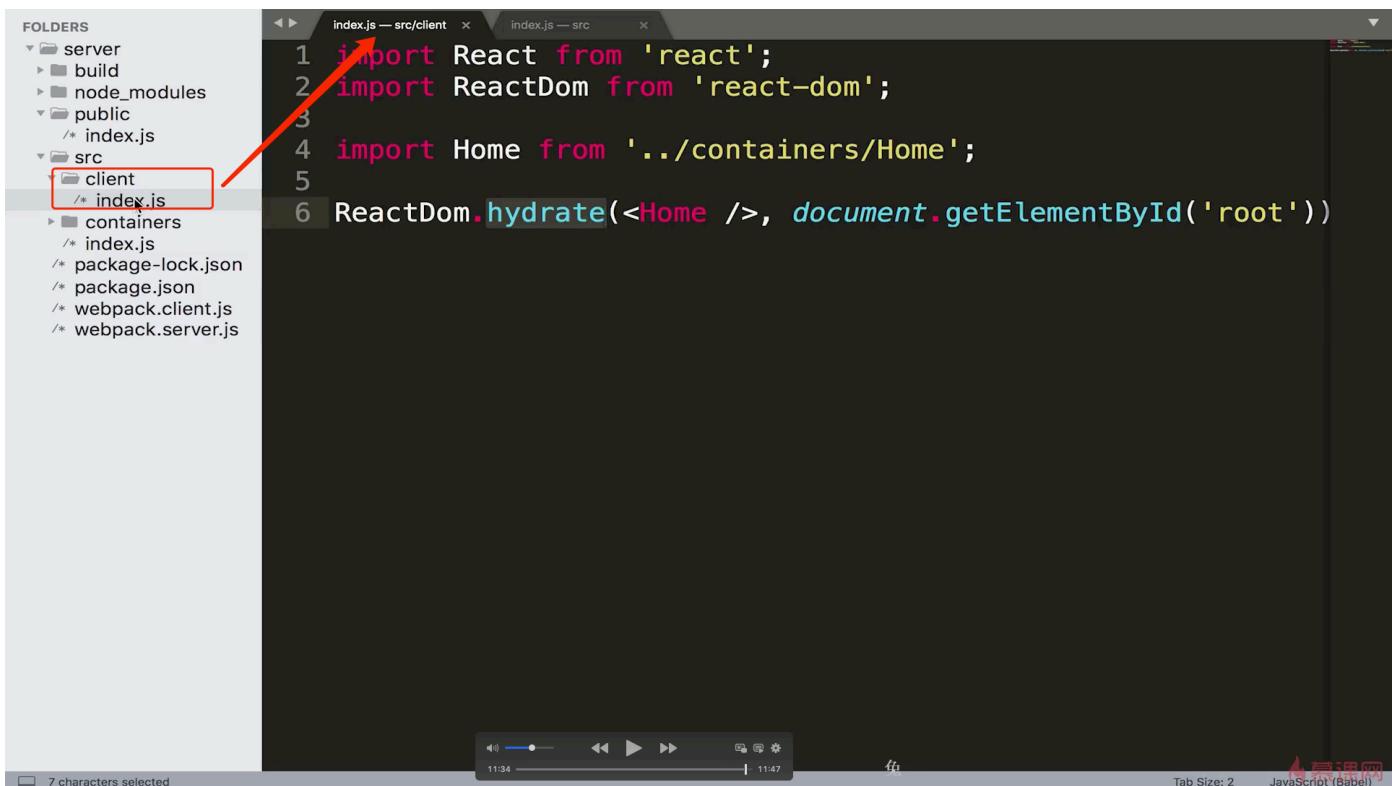
```
index.js
1 import express from 'express';
2 import Home from './containers/Home';
3 import React from 'react';
4 import { renderToString } from 'react-dom/server';
5
6 const app = express();
7 app.use(express.static('public'));

8
9 const content = renderToString(<Home />);
10
11 app.get('/', function (req, res) {
12   res.send(
13     <html>
14       <head>    兔
15         <title>ssr</title>
16       </head>
17       <body>
18         ${content}
19         <script src='/index.js'></script>
20       </body>
21     </html>
22   );
23 });

Line 8, Column 1; Saved ~/Desktop/server/src/index.js (UTF-8)
Tab Size: 2
JavaScript (Babel)
```

3.3 让react代码在浏览器上运行

1. 定义好需要在client执行的代码，这个代码也需要进行babel转译



```
index.js — src/client
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 import Home from '../containers/Home';
5
6 ReactDOM.hydrate(<Home />, document.getElementById('root'))
```

2. 新增 webpack.client.js 对上面client的文件进行打包编译

The screenshot shows a code editor with a file named 'webpack.client.js' open. The code is a module export object with various configurations for a development build. It includes rules for babel-loader, specifying targets and browsers. The code is syntax-highlighted in a dark theme.

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: './src/client/index.js',
  output: {
    filename: 'index.js',
    path: path.resolve(__dirname, 'public')
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        options: {
          presets: ['react', 'stage-0', ['env', {
            targets: {
              browsers: ['last 2 versions']
            }
          }]]
        }
      }
    ]
  }
};
```

3. 新增script中对于客户端代码打包的指令

The screenshot shows a code editor with a file named 'package.json' open. The JSON object contains a 'scripts' field with several build commands. The 'dev:build:server' and 'dev:build:client' entries are highlighted with red boxes. The code is syntax-highlighted in a dark theme.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "dev": "npm-run-all --parallel dev:***",
    "dev:start": "nodemon --watch build --exec node ./buil",
    "dev:build:server": "webpack --config webpack.server.js",
    "dev:build:client": "webpack --config webpack.client.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "babel-core": "^6.26.3",
    "babel-loader": "^7.1.5",
    "babel-preset-env": "^1.7.0",
    "babel-preset-react": "^6.24.1",
    "babel-preset-stage-0": "^6.24.1",
    "express": "^4.16.3",
    "react": "^16.4.1",
    "react-dom": "^16.4.1",
    "web|"
  }
}
```

4. 返回给客户端的代码添加上需要在客户端运行的代码

The screenshot shows a code editor with two tabs: 'index.js — src/client' and 'index.js — src'. The left sidebar displays the project's folder structure:

```
FOLDERS
  - server
  - build
  - node_modules
  - public
    - index.js
  - src
    - client
    - containers
    - index.js
  - package-lock.json
  - package.json
  - webpack.client.js
  - webpack.server.js
```

The 'src/index.js' file contains the following code:

```
5 const app = express();
6 app.use(express.static('public'));
7
8 const content = renderToString(<Home />);
9
10 app.get('/', function (req, res) {
11   res.send(`
12     <html>
13       <head>
14         <title>ssr</title>
15       </head>
16       <body>
17         <div id="root">
18           ${content}
19         </div>
20         <script src="/index.js"></script>
21       </body>
22     </html>
23   `);
24 });
25
26
27 var server = app.listen(3000);
```

Red arrows highlight several parts of the code: 'app.use(express.static('public'))', 'res.send(`...`)', and the script tag 'script src="/index.js">'.

3.4 工程代码优化整理

Webpack-merge 用来合并webpack 配置文件

1. webpack.base.js

The screenshot shows a code editor with three tabs: 'webpack.client.js', 'webpack.server.js', and 'webpack.base.js'. The left sidebar displays the project's folder structure:

```
FOLDERS
  - server
  - build
  - node_modules
  - public
  - src
    - index.js
  - package-lock.json
  - package.json
  - webpack.base.js
  - webpack.client.js
  - webpack.server.js
```

The 'webpack.base.js' file contains the following code:

```
1 module.exports = {
2   module: {
3     rules: [
4       {
5         test: /\.js$/,
6         loader: 'babel-loader',
7         exclude: /node_modules/,
8         options: {
9           presets: ['react', 'stage-0', ['env', {
10             targets: {
11               browsers: ['last 2 versions']
12             }
13           }]]
14         }
15       }
16     }
17   }
18 }
```

2. webpack.server.js

FOLDERS

- server
- build
- node_modules
- public
- src

```
webpack.client.js  webpack.server.js  webpack.base.js
1 const path = require('path');
2 const nodeExternals = require('webpack-node-externals');
3 const merge = require('webpack-merge');
4 const config = require('./webpack.base.js');

5
6 const serverConfig = {
7   target: 'node',
8   mode: 'development',
9   entry: './src/index.js',
10  output: {
11    filename: 'bundle.js',
12    path: path.resolve(__dirname, 'build')
13  },
14  externals: [nodeExternals()]
15};
16
17 module.exports = merge(config, serverConfig);
```

Line 17, Column 46

02:47 04:38

Tab Size: 2 JavaScript (Babel)

3.webpack.client.js

FOLDERS

- server
- build
- node_modules
- public
- src

```
webpack.client.js
1 const path = require('path');
2 const merge = require('webpack-merge');
3 const config = require('./webpack.base.js');

4
5 const clientConfig = {
6   mode: 'development',
7   entry: './src/client/index.js',
8   output: {
9     filename: 'index.js',
10    path: path.resolve(__dirname, 'public')
11  }
12};
13
14 module.exports = merge(config, clientConfig);
```

Line 12, Column 3

02:52 04:38

Tab Size: 2 JavaScript (Babel)

将之前src下面的index.js 嵌套到server目录下，这样目录结构清晰

不要忘记改对应webpack下面的入口

The screenshot shows a code editor with a sidebar labeled 'FOLDERS' containing project files: server, build, node_modules, public, src, client, and containers. Inside 'src', there is a 'server' folder which contains an 'index.js' file. This 'index.js' file is highlighted with a red box. The code in 'index.js' is as follows:

```
index.js
1 import express from 'express';
2 import Home from './containers/Home';
3 import React from 'react';
4 import { renderToString } from 'react-dom/server';
5
6 const app = express();
7 app.use(express.static('public'));
8
9 const content = renderToString(<Home />);
10
11 app.get('/', function (req, res) {
12   res.send(`
13     <html>
14       <head>
15         <title>ssr</title>
16       </head>
17       <body>
18         <div id="root">${content}</div>
19         <script src="/index.js"></script>
20       </body>
21     </html>
22   `);
23 });

Line 25, Column 31
Tab Size: 2
JavaScript (Babel)
易课网
```

4.1 在SSR框架中引入路由机制

只有当js中的react代码接管页面操作之后才会有事件绑定



兔

慕课网

传统pc端 react项目依赖于基础路由 broserRouter
服务器端 react项目依赖于 staticRouter

同构的时候需要在服务端和客户端都跑一遍。

The screenshot shows a code editor with a dark theme. On the left is a file tree with the following structure:

```
server
build
node_modules
public
src
client
containers
Home
index.js
Routes.js
```

The file tree shows that the file `Routes.js` is selected and highlighted with a red box.

The main pane displays the content of the `Routes.js` file:

```
import React from 'react';
import { Route } from 'react-router-dom';
import Home from './containers/Home';

export default (
  <div>
    <Route path='/' exact component={Home}></Route>
  </div>
)
```

At the bottom of the editor, there is a status bar showing "Line 9, Column 2". Below the editor is a video player interface with a play button, a progress bar from 0:01 to 33:26, and a tab size indicator of "Tab Size: 2". The video title is "d-1~4-3,5-1-5-2.mp4".

The screenshot shows a code editor with a dark theme. On the left is a file tree with the following structure:

```
server
build
node_modules
public
src
client
index.js
```

The file tree shows that the file `index.js` is selected and highlighted with a red box.

The main pane displays the content of the `index.js` file:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import Routes from '../Routes';

const App = () => {
  return (
    <BrowserRouter>
      <Routes>
    </BrowserRouter>
  )
}

ReactDOM.hydrate(<Home />, document.getElementById('root'))
```

At the bottom of the editor, there is a status bar showing "Line 9, Column 15; Saved ~/Desktop/server/src/client/index.js (UTF-8)". Below the editor is a video player interface with a play button, a progress bar from 0:08 to 33:26, and a tab size indicator of "Tab Size: 2". The video title is "d-1~4-3,5-1-5-2.mp4".

location: 由于服务器端并不能主动感知客户端地址栏的变化，所以需要location 属性获取到当前的url地址

context: 服务器端用于数据传递给子组件

The screenshot shows a code editor with a dark theme. On the left is a file tree with the following structure:

```
FOLDERS
server
build
node_modules
public
src
client
index.js
containers
server
index.js
Routes.js
package-lock.json
package.json
webpack.base.js
webpack.client.js
webpack.server.js
```

The main pane displays the `index.js` file content:

```
2 import React from 'react';
3 import { renderToString } from 'react-dom/server';
4 import { StaticRouter } from 'react-router-dom';
5 import Routes from '../Routes';
6
7 const app = express();
8 app.use(express.static('public'));
9
10 app.get('/', function (req, res) {
11
12   const content = renderToString(
13     <StaticRouter locaiton={req.path} context={}>
14       {Routes}
15     </StaticRouter>
16   );
17
18   res.send(`<html>
19     <head>    兔
20       <title>ssr</title>
21     </head>
22     <body>
23       <div id="root">${content}</div>
```

Annotations highlight the `location` and `context` props of the `StaticRouter` component.

The screenshot shows a Mac OS X desktop with a browser window titled "ssr". The address bar shows `localhost:3000/`. The page content is a simple HTML structure with the text "兔" in the head section.

Below the browser is the Chrome DevTools Console tab, which shows a warning message:

```
Warning: Expected server HTML to contain a matching <div> in <div>.
```

The message is highlighted with a red border.

The screenshot shows the Chrome DevTools Console tab with the message from the previous screenshot. Below it, the "What's New" tab is active, showing "Highlights from the Chrome 67 update".

hydrate:

<https://www.zhihu.com/question/66068748>

4-2 多页面路由跳转

4-3 使用Link标签串联起整个路由流程

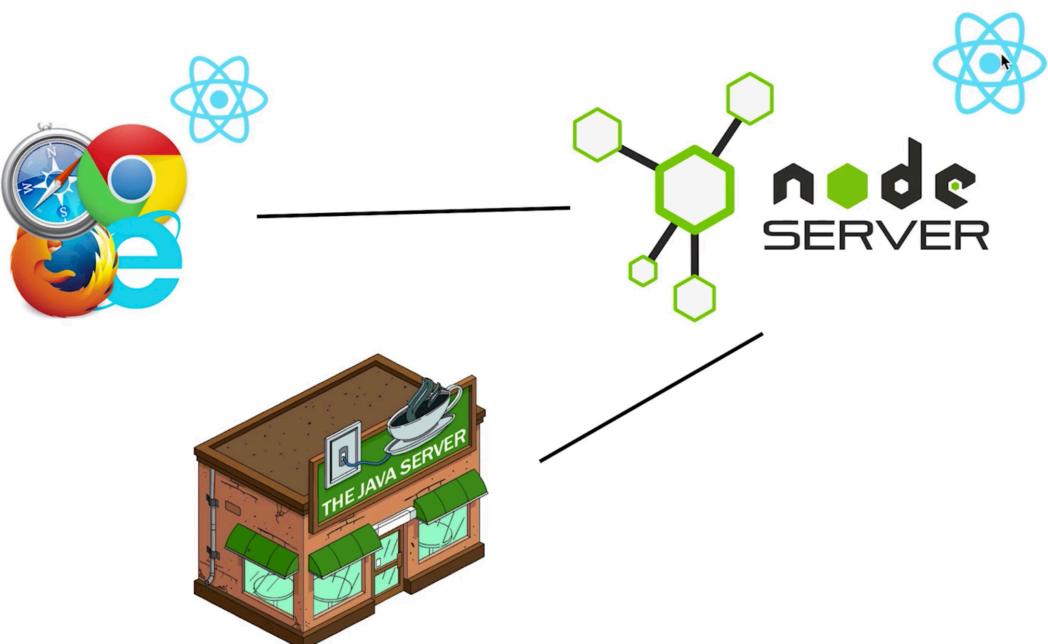
The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a project structure with folders like server, build, node_modules, public, and src. Inside src, there are client, components, containers, and server sub-folders. Header.js is located in the components folder. The code editor window displays the following code for Header.js:

```
1 import React from 'react';
2 import { Link } from 'react-router-dom';
3
4 const Header = () => {
5   return (
6     <div>
7       <Link to='/'>Home</Link>
8       <Link to='/login'>Login</Link>
9     </div>
10  )
11 }
12
13 export default Header; 两个页面中都引入了Header组件
14
```

Two red arrows point from the file tree to the Home and Login components in the code editor, indicating they both import the Header component.

服务端渲染只发生在我们第一次进入页面的时候（第一个页面，也并非是项目的主页），再次发生页面的跳转实际上bundles。js中的代码控制的，进行的还是react-router的路由机制

4-4 中间层



慕课网

5-1 在同构项目中引入redux

在客户端创建使用store

```

暂停
server
27:59 / 33:26
Build
node_modules
public
src
client
components
containers
server
/* Routes.js
/* package-lock.json
/* package.json
/* webpack.base.js
/* webpack.client.js
/* webpack.server.js
index.js
4-1~4-3,5~1~5-2.mp4
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { BrowserRouter } from 'react-router-dom';
4 import Routes from './Routes';
5 import { createStore, applyMiddleware } from 'redux';
6 import { Provider } from 'react-redux';
7 import thunk from 'redux-thunk';
8
9 const reducer = (state = {name: 'dell'}, action) => {
10   return state;
11 }
12
13 const store = createStore(reducer, applyMiddleware(thunk));
14
15 const App = () => {
16   return (
17     <Provider store={store}>
18       <BrowserRouter>
19         {Routes}
20       </BrowserRouter>
21     </Provider>
22   )
23 }

```

Line 7, Column 33 27:59 33:26 Tab Size: 2 JavaScript (Babel)

在server端如何使用store

FOLDERS

```

server
  build
  node_modules
  public
src
  client
    index.js
  components
  containers
    Home
      index.js
    Login
  server
    index.js
    utils.js
  Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

```

3 port { StaticRouter } from 'react-router-dom';
4 port Routes from '../Routes';
5 port { createStore, applyMiddleware } from 'redux';
6 port { Provider } from 'react-redux';
7 port thunk from 'redux-thunk';
8
9 port const render = (req) => {
10
11 const reducer = (state = {name: 'dell'}, action) => {
12   return state;
13 }
14
15 const store = createStore(reducer, applyMiddleware(thunk));
16
17 const content = renderToString((
18   <Provider store={store}>
19     <StaticRouter location={req.path} context={{}}>
20       {Routes}
21     </StaticRouter>
22   </Provider>
23 ));
24
25 return `

```

Line 7, Column 25; Saved ~/Desktop/server/src/server/utils.js (UTF-8) Tab Size: 2 JavaScript (Babel)

在组件中使用store中的数据

FOLDERS

```

暂停
server
  build
  node_modules
  public
src
  client
    index.js
  components
  containers
    Home
      index.js
    Login
  server
    Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

```

1 import React from 'react';
2 import Header from '../../../../../components/Header';
3 import { connect } from 'react-redux';
4
5 const Home = (props) => {
6   return (
7     <div>
8       <Header />
9       <div>This is {props.name}</div>
10      <button onClick={()=>{alert('click1')}}>
11        click
12      </button>
13    </div>
14  )
15 }
16
17 const mapStateToProps = state => ({
18   name: state.name
19 })
20
21 export default connect(mapStateToProps, null)(Home);
22

```

Line 9, Column 31; Saved ~/Desktop/server/src/containers/Home/index.js (UTF-8) Tab Size: 2 JavaScript (Babel)

5-2 创建 Store代码的复用

实际上面的代码是有问题的，会产生用户数据错乱问题，因为都是单例的store

```
1 import { createStore, applyMiddleware } from 'redux';
2 import thunk from 'redux-thunk';
3
4 const reducer = (state = {name: 'dell'}, action) => {
5   return state;
6 }
7
8 const getStore = () => {
9   return createStore(reducer, applyMiddleware(thunk));
10}
11
12 export default getStore;
```

通过函数的方式形成属于每个用户自己的store
如果我们直接返回的是CreateStore的话所有用户使用的是一个相同的store

此时可以保证用户可以有自己独立的store

```
2 import { renderToString } from 'react-dom/server';
3 import { StaticRouter } from 'react-router-dom';
4 import Routes from '../Routes';
5 import { Provider } from 'react-redux';
6 import getStore from '../store';
7
8 export const render = (req) => {
9
10  const content = renderToString((
11    <Provider store={getStore()}>
12      <StaticRouter location={req.path} context={{}}>
13        {Routes}
14      </StaticRouter>
15    </Provider>
16  ));
17
18  return `
19    <html>
20      <head>
21        <title>ssr</title>
22      </head>
23      <body>
24        ${content}</div>
25    </html>
26  `;
27}
```

5-3构建 redux 代码结构

The screenshot shows a code editor with a sidebar containing a project tree. The tree includes 'server', 'build', 'node_modules', 'public', 'src' (which contains 'client', 'components', and 'containers'), and a red-highlighted 'store' folder. Inside 'store', there are files like 'actions.js', 'constants.js', 'index.js', 'reducer.js', and 'index.js'. A red arrow points from the text '这里面定义 Home 属于自己的 reducer' to the line of code 'import homeReducer from '../containers/Home/store/reducer';'. The code editor window displays the following code:

```
1 import { createStore, applyMiddleware, combineReducers } from 'redux';
2 import thunk from 'redux-thunk';
3 import homeReducer from '../containers/Home/store/reducer';
4 |
5 const reducer = combineReducers({
6   home: homeReducer
7 });
8
9
10 const createStore = () => {
11   return createStore(reducer, applyMiddleware(thunk));
12 }
13
14 export default createStore;
```

Below the code, a note in red text says: '实际这个目录结构并不是很好，直接在总的 store 中添加属于各个业务线的 reducer 就可以了'.

This screenshot shows a code editor with a sidebar displaying a project tree. The tree includes 'server', 'build', 'node_modules', 'public', 'src' (containing 'client', 'components', and 'containers'), and a red-highlighted 'store' folder. Inside 'store', there are files like 'actions.js', 'constants.js', 'index.js', 'reducer.js', and 'index.js'. A red arrow points from the text '暂停' (Paused) in the status bar to the 'store' folder in the tree. The code editor window displays the following code:

```
1 import axios from 'axios';
2
3 export const getHomeList = () => {
4   return () => {
5     axios.get('http://47.95.113.63/ssr/api/news.json?secret=abcd')
6       .then((res) => {
7         console.log(res);
8       });
9   };
10 }
```

The status bar at the bottom left shows 'Line 5, Column 35'. The status bar at the bottom right shows 'Tab Size: 2' and 'JavaScript (Babel)'.

index.js

```

9     <div>
10    <Header />
11    <div>This is {this.props.name}</div>
12    <button onClick={()=>{alert('click1')}}>
13      click
14    </button>
15  </div>
16)
17}
18
19 componentDidMount() {
20  this.props.getHomeList();
21}
22}
23
24 const mapStateToProps = state => ({
25  name: state.home.name
26});
27
28 const mapDispatchToProps = dispatch => ({
29  getHomeList() {
30    dispatch(getHomeList());
31  }
32})
33
34 export default connect(mapStateToProps, mapDispatchToProps)(Home);
35

```

actions.js

```

1 import axios from 'axios';
2 actionCreator
3 const changeList = (list) => ({
4   type: 'change_home_list',
5   list
6 })
7
8 export const getHomeList = () => {
9   return (dispatch) => {
10     axios.get('http://47.95.113.63/ssr/api/news.json?secret=abcd')
11       .then((res) => {
12         const list = res.data.data;
13         dispatch(changeList(list));
14       });
15   }
16 }

```

浏览器显示的UI：

- Home
- Login
- This is
- click

开发者工具Elements面板显示的数据：

```

data:
  data: Array[2]
    0:
      id: 12
      title: ""
    1:
      id: 2
      title: ""
    2: {id: 3, title: ""}

```

FOLDERS

```

server
build
node_modules
public
src
client
components
containers
  Home
    store
      actions.js
      contants.js
      index.js
      reducer.js
      index.js
Login
server
store
  index.js
Routes.js
package-lock.json
package.json
webpack.base.js
webpack.client.js
webpack.server.js

```

actions.js

```

1 import axios from 'axios';
2 actionCreator
3 const changeList = (list) => ({
4   type: 'change_home_list',
5   list
6 })
7
8 export const getHomeList = () => {
9   return (dispatch) => {
10     axios.get('http://47.95.113.63/ssr/api/news.json?secret=abcd')
11       .then((res) => {
12         const list = res.data.data;
13         dispatch(changeList(list));
14       });
15   }
16 }

```

因为使用了 thunk中间件：
所以我们返回的这个函数中可以接受dispatch这个参数，经过中间件的处理后 最终dispatch一个action

The screenshot shows a code editor interface with a sidebar on the left displaying a file tree. The tree includes folders like 'server', 'build', 'node_modules', 'public', 'src', 'client', 'components', 'containers', 'Home', 'store', and various files such as 'actions.js', 'contants.js', 'index.js', 'reducer.js', 'Routes.js', 'package-lock.json', 'package.json', 'webpack.base.js', 'webpack.client.js', and 'webpack.server.js'. Two tabs are open in the main area: 'actions.js' and 'reducer.js'. The 'reducer.js' tab contains the following code:

```
const defaultState = {
  name: 'dell lee',
  newsList: []
}

export default (state = defaultState, action) => {
  switch(action.type) {
    case 'change_home_list':
      return {
        ...state,
        newsList: action.list
      }
    default:
      return state;
  }
}
```

The status bar at the bottom indicates 'Line 11, Column 28'.

5-4 问题

根据下图我们发现我们这种在didmount中请求数据的方式依然客户端渲染，因为didmount生命周期并不会在服务端执行

1. 服务器接收到请求，这个时候store是空的
2. 服务器端不会执行componentDidMount，所以列表内容获取不到

3. 客户端代码运行，这个时候store依然是空的
4. 客户端执行componentDidMount，列表数据被获取
5. store中的列表数据被更新
6. 客户端渲染出store中list数据对应的列表内容

react-router中已经考虑好了上面的情况，为了做服务端渲染获取数据

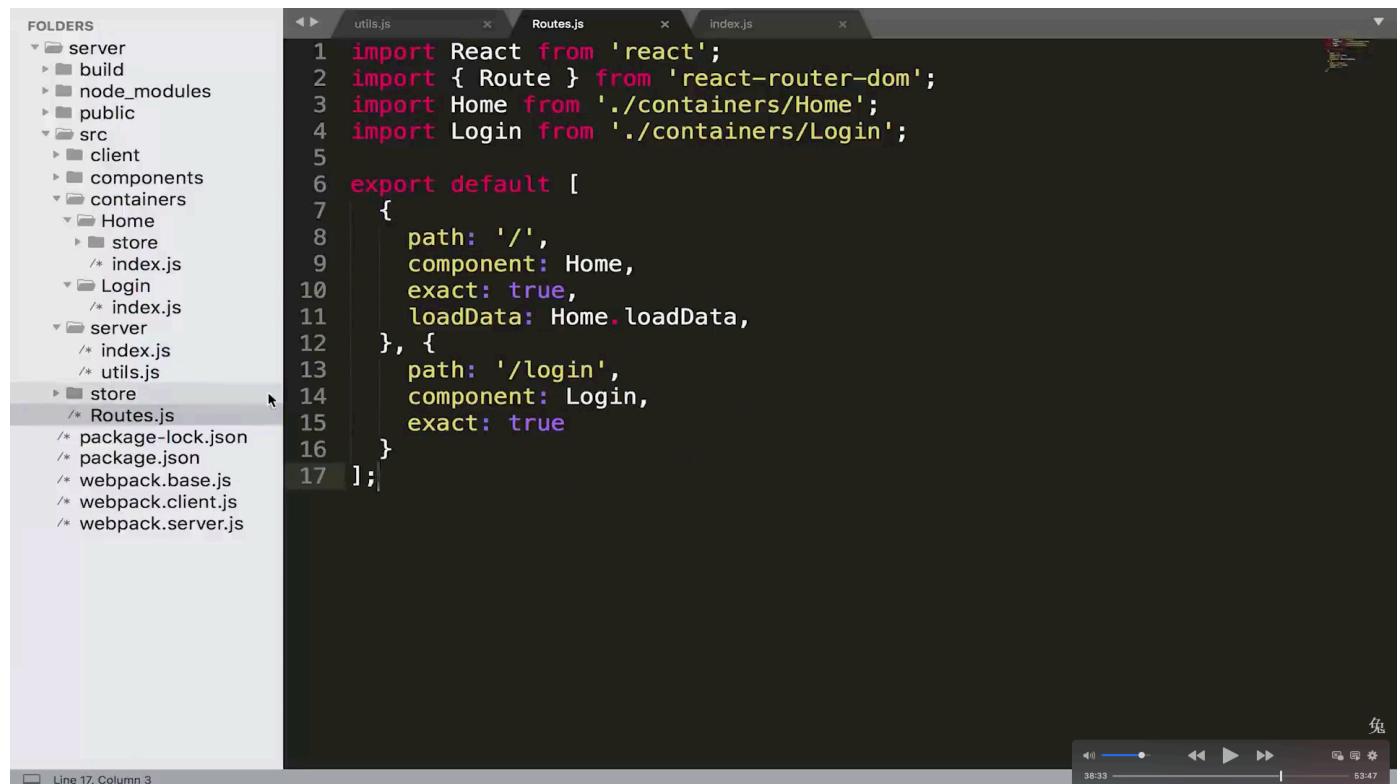
为了让服务端渲染完整的带有数据的页面就需要在进入这个路由之前将当前页面中所需要的数据提前加载好并且更新store中的数据，这样客户端请求的代码中（刷新某个页面/首屏加载）store已经能够满足当前页面的数据要求，客户端不需要请求。

需要的知识点

1.matchRoutes 匹配当前路由所涉及的所有组件，并且调用其定义好的LoadData方法。（注意didmount中的请求数据依然在）

2.Promise.all().then(返回客户端需要的字符串)

3.favicon 也会请求数据需要我们配置一个favicon的文件放在public中



```
1 import React from 'react';
2 import { Route } from 'react-router-dom';
3 import Home from './containers/Home';
4 import Login from './containers/Login';
5
6 export default [
7   {
8     path: '/',
9     component: Home,
10    exact: true,
11    loadData: Home.loadData,
12  },
13  {
14    path: '/login',
15    component: Login,
16    exact: true
17 }];

```

server 相关的代码配置如下

FOLDERS

```

server
build
node_modules
public
src
client
components
containers
Home
store
actions.js
constants.js
index.js
reducer.js
index.js
Login
server
index.js
utils.js
store
Routes.js
package-lock.json
package.json
webpack.base.js
webpack.client.js
webpack.server.js

```

```

1 import express from 'express';
2 import { matchRoutes } from 'react-router-config'
3 import { render } from './utils';
4 import getStore from '../store';
5 import routes from '../Routes';
6
7 const app = express();
8 app.use(express.static('public'));
9
10 app.get('*', function (req, res) {
11   const store = getStore();
12   // 根据路由的路径，来往store里面加数据
13   const matchedRoutes = matchRoutes(routes, req.path);
14   // 让matchRoutes里面所有的组件，对应的loadData方法执行一次
15   const promises = [];
16   matchedRoutes.forEach(item => {
17     if (item.route.loadData) {
18       promises.push(item.route.loadData(store))
19     }
20   })
21   Promise.all(promises).then(() => {
22     res.send(render(store, routes, req));
23   })
24 });
25
26 var server = app.listen(3000);

```

Line 2, Column 50

Tab Size: 2

JavaScript (Babel)

FOLDERS

```

server
build
node_modules
public
src
client
components
containers
Home
store
actions.js
constants.js
index.js
reducer.js
index.js
Login
server
index.js
utils.js
store
Routes.js
package-lock.json
package.json
webpack.base.js
webpack.client.js
webpack.server.js

```

```

1 import { renderToString } from 'react-dom/server',
2 import { StaticRouter, Route } from 'react-router-dom';
3 import { matchRoutes } from 'react-router-config';
4 import { Provider } from 'react-redux';
5
6
7 export const render = (store, routes, req) => {
8
9   const content = renderToString((
10     <Provider store={store}>
11       <StaticRouter location={req.path} context={{}}>
12         <div>
13           {routes.map(route => (
14             <Route {...route}/>
15           ))}
16         </div>
17       </StaticRouter>
18     </Provider>
19   ));
20
21   return `
22     <html>
23       <head>
24         <title>ssr</title>
25       </head>
26       <body>
27         <div id="root">${content}</div>
28         <script src="/index.js"></script>
29   `;

```

Line 8, Column 39; Saved ~/Desktop/server/src/server/utils.js (UTF-8)

Tab Size: 2

JavaScript (Babel)

5-5 页面出现闪屏的原因以及如何解决

产生原因：

用户请求首屏（或者刷新一个页面）服务端的store已经请求好了数据，但是客户端获取的store 依然是一个空的对象（看下图 getStore方法如果没有做过其他操作其在客户端就是一个初始状态），此时没有数据----》白屏----》请求完数据后展示。 所以出现了闪屏的效果

解决方法：数据的脱水，注水

也就是服务端返回给客户端一个store的初始值。然后客户端将这个初始值给store。

The screenshot shows a code editor with a sidebar containing a file tree. The tree includes folders like 'server', 'build', 'node_modules', 'public', 'src', 'client', 'components', 'containers', 'Home', 'Login', 'server', 'store', and files like 'index.js', 'Routes.js', 'utils.js', 'package-lock.json', 'package.json', 'webpack.base.js', 'webpack.client.js', and 'webpack.server.js'. The main editor area displays the following code:

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { BrowserRouter, Route } from 'react-router-dom';
4 import routes from '../Routes';
5 import getStore from '../store';
6 import { Provider } from 'react-redux';
7
8 const App = () => {
9   return (
10     <Provider store={getStore()}>
11       <BrowserRouter>
12         {routes.map(route => (
13           <Route {...route}/>
14         ))}
15       </BrowserRouter>
16     </Provider>
17   )
18 }
19
20 ReactDOM.hydrate(<App />, document.getElementById('root'))
```

Line 3, Column 30; Saved ~/Desktop/server/src/client/index.js (UTF-8) Tab Size: 2 JavaScript (Babel)

数据的脱水注水

FOLDERS

```

server
  build
  node_modules
  public
src
  client
  components
  containers
  server
    index.js
    utils.js
store
  index.js
  Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

utils.js

```

6 export const render = (store, routes, req) => {
7
8   const content = renderToString((
9     <Provider store={store}>
10      <StaticRouter location={req.path} context={{}}
11        <div>
12          {routes.map(route => (
13            <Route {...route}/>
14          ))}
15        </div>
16      </StaticRouter>
17    </Provider>
18  ));
19
20   return `
21     <html>
22       <head>
23         <title>ssr</title>
24       </head>
25       <body>
26         <div id="root">${content}</div>
27         <script>
28           window.context = {
29             state: ${JSON.stringify(store.getState())}
30           }
31         </script>
32         <script src='/index.js'></script>

```

Line 19, Column 1; Saved ~/Desktop/server/src/server/utils.js (UTF-8) Tab Size: 2 JavaScript (Babel) 豪课网

客户端脱水：（实际上就是客户端数据的初始化依赖于服务端‘注水’）

FOLDERS

```

server
  build
  node_modules
  public
src
  client
    index.js
    components
    containers
  server
    index.js
    utils.js
store
  index.js
  Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

index.js — client index.js — store

```

1 import { createStore, applyMiddleware, combineReducers } from 'redux';
2 import thunk from 'redux-thunk';
3 import { reducer as homeReducer } from '../containers/Home/store';
4
5 const reducer = combineReducers({
6   home: homeReducer
7 });
8
9 export const getStore = () => {
10   return createStore(reducer, applyMiddleware(thunk));
11 }
12
13 export const getClientStore = () => {
14   const [defaultState] = window.context.state;
15   return createStore(reducer, defaultState, applyMiddleware(thunk));
16 }

```

12 characters selected Tab Size: 2 JavaScript (Babel) 豪课网

注意服务端渲染：渲染的是客户端请求的第一个页面（首屏/刷新其他页面）

componentDidMount 生命周期对于数据的请求不能完全的抹除（如用户首屏请求的是login页面然后跳转到list页面，此时list页面的数据依然需要自己ajax请求。）所以折中判断下，如果服务端已经给我们请求好了数据（也就是初始状态：首屏/刷新当前页面），我们就不需要重复加载数据。

```
index.js — client index.js — store index.js — containers/Home
13 render() {
14   return (
15     <div>
16       <Header />
17       {this.getList()}
18       <button onClick={()=>{alert('click1')}}>
19         click
20       </button>
21     </div>
22   )
23 }
24
25 componentDidMount() {
26   if (!this.props.list.length) {
27     this.props.getHomeList();
28   }
29 }
30
31
32 Home.loadData = (store) => {
33   // 这个函数，负责在服务器端渲染之前，把这个路由需要的数据提前加载好
34   return store.dispatch(getHomeList())
35 }
36
37 const mapStateToProps = state => ({
38   list: state.home.newsList
39 });
40
```

6-1让中间层承担数据获取职责

问题点：

我们需要我们搭建的node-server 承担起接口转发，初始页面渲染的指责（接口转发可以更好的定位问题）

但是之前的代码是这样：

如果是服务端渲染了这个页面，页面的数据是我们node服务器请求接口获取数据，更新store，渲染到页面上。最终返回给客户端。

如果是客户端渲染这个页面，（比如我们从登陆页面跳转到list页面，此时list页面的数据是我们前端发送ajax请求直接到javaserver端）最终完成store更新以及页面的渲染。

也就是 图片中的箭头，这是不合理的。

解决办法：

express-http-proxy 中间件，可以在express框架中添加一个proxy。

当客户端请求/api/* 的路径时候就会被这个proxy进行代理到其他路径！

但是他并不代理服务端发送的请求！！！ 导致在服务端渲染的情况下页面一直转圈，不能正确返回页面。

The screenshot shows a code editor with a sidebar containing a file tree. A red box highlights the 'server' folder, which contains several subfolders and files. Another red box highlights a line of code in a file named 'actions.js'. This line contains a call to 'axios.get()' with a URL that includes a query parameter 'secret=abcd'. The code is written in JavaScript, and the editor interface includes tabs for 'index.js — server', 'index.js — containers/Home', and 'actions.js'.

```
1 import axios from 'axios';
2 import { CHANGE_LIST } from './constants';
3
4 const changeList = (list) => ({
5   type: CHANGE_LIST,
6   list
7 })
8
9 export const getHomeList = () => {
10   // http://47.95.133.63/ssr/api/news.json?secret=abcd
11   // 浏览器运行
12   // /api/news.json = http://localhost:3000/api/news.json
13   // 服务器运行
14   // /api/news.json = 服务器根目录下/api/news.json
15
16   return (dispatch) => {
17     return axios.get('/api/news.json?secret=abcd')
18       .then((res) => {
19         const list = res.data.data;
20         dispatch(changeList(list));
21       });
22   }
23 }
```

问题点：

在客户端运行的时候：我们最终访问的是 <http://localhost:3000/api/news.json> 这个访问被 express-http-proxy 截获最终访问<http://47.95.133.63...../news.json>

在服务端运行的时候：我们访问的是服务器根目录下面的 api/news.json 这种不被截获！但是服务端没有这个路径/文件所以一直loading

解决办法：6-2

1. getHomeList 判断自己在客户端执行还是在服务端执行
2. 调用getHomeList 时候给他传递参数

6-2 服务端请求和客户端请求的不同处理

```

export const getHomeList = (server) => {
  // http://47.95.113.63/ssr/api/news.json?secret=abcd
  // 浏览器运行
  // /api/news.json = http://localhost:3000/api/news.json
  // 服务器运行
  // /api/news.json = 服务器根目录下/api/news.json
  let url = ''
  if (server) {
    url = 'http://47.95.113.63/ssr/api/news.json?secret=abcd'
  } else {
    url = '/api/news.json?secret=abcd'
  }

  return (dispatch) => {
    return axios.get(url)
      .then((res) => {
        const list = res.data.data;
        dispatch(changeList(list))
      });
  }
}

```

```

12
13   render() {
14     return (
15       <div>
16         <Header />
17         {this.getList()}
18         <button onClick={()=>{alert('click1')}}>
19           click
20         </button>
21       </div>
22     )
23   }
24
25   componentDidMount() {
26     if (!this.props.list.length) {
27       this.props.getHomeList(false);
28     }
29   }
30 }
31
32 Home.loadData = (store) => {
33   // 这个函数，负责在服务器端渲染之前，把这个路由需要的数据提前加载好
34   return store.dispatch(getHomeList(true))
35 }
36
37 const mapStateToProps = state => ({
38   list: state.home.newsList
39 })

```

Line 27, Column 35; Saved ~/Desktop/server/src/containers/Home/index.js (UTF-8)

16:25 1:34:12

Tab Size: 2 JavaScript (Babel)

6-3 axios中instance的使用

本节是对上节代码的优化

6-2 的解决方法不适合全局使用，因为项目中有很多的接口请求，不能每个都这么配置

axios中有两个比较重要的概念：

1.instance

2.interceptors

The screenshot shows a code editor with a dark theme. On the left is a file tree (Folders) with the following structure:

```
server
  build
  node_modules
  public
  src
    client
      index.js
      request.js
    components
    containers
    server
      index.js
      request.js
      utils.js
```

The right pane contains the content of `request.js`:

```
1 import axios from 'axios'; 兔
2
3 const instance = axios.create({
4   baseURL: '/'
5 });
6
7 export default instance;
```

At the bottom of the editor, status bars indicate "Line 7, Column 15; Saved ~/Desktop/server/src/client/request.js (UTF-8)", "Tab Size: 2", and "JavaScript (Babel)".

The screenshot shows a code editor with a dark theme. On the left is a file tree (Folders) with the same structure as the first screenshot.

The right pane contains the content of `request.js — server`:

```
1 import axios from 'axios'; 兔
2
3 const instance = axios.create({
4   baseURL: 'http://47.95.113.63/ssr'
5 });
6
7 export default instance;
```

At the bottom of the editor, status bars indicate "Line 7, Column 16; Saved ~/Desktop/server/src/server/request.js (UTF-8)", "Tab Size: 2", and "JavaScript (Babel)".

The screenshot shows a code editor with a dark theme. On the left is a sidebar with project files: server, build, node_modules, public, and src (containing client, components, containers, server, store, and Routes.js). The main area shows the contents of actions.js:

```
actions.js 6-4-6-10.mp4
1 import axios from 'axios';
2 import { CHANGE_LIST } from './constants';
3 import clientAxios from '../../../../../client/request';
4 import serverAxios from '../../../../../server/request';
5
6 const changeList = (list) => ({
7   type: CHANGE_LIST,
8   list
9 })
10
11 export const getHomeList = (server) => {
12   const request = server ? serverAxios : clientAxios;
13
14   return (dispatch) => {
15     return request.get('/api/news.json?secret=abcd')
16       .then((res) => {
17         const list = res.data.data;
18         dispatch(changeList(list))
19       });
20   };
21 }
22 }
```

Annotations include red arrows pointing from the imports of clientAxios and serverAxios to their respective usage in the code. A blue box highlights the argument 'server' in the export statement, and a red rounded rectangle highlights the assignment of 'request' to either 'serverAxios' or 'clientAxios'. The status bar at the bottom shows 'Line 12, Column 1', 'Tab Size: 2', and 'JavaScript (Babel)'.

6-4 巧用 redux-thunk 中的 withExtraArgument

本节是对上节代码的优化

上节的代码中我们依然需要传递参数的方式进行判断

反思：我们在服务端运行store的时候使用的是服务端的store实例。

在客户端运行的时候使用的是客户端的store。

不同的store 使用不同的请求实例！！！

```
applyMiddleware(thunk.withExtraArgument(serverAxios))
```

FOLDERS

```
1 import { createStore, applyMiddleware, combineReducers } from 'redux'
2 import thunk from 'redux-thunk';
3 import { reducer as homeReducer } from '../containers/Home/store';
4 import clientAxios from '../client/request';
5 import serverAxios from '../server/request';
6
7 const reducer = combineReducers({
8   home: homeReducer
9 });
10
11 export const getStore = () => {
12   // 改变服务器端store的内容，那么就一定要使用serverAxios
13   return createStore(reducer, applyMiddleware(thunk.withExtraArgument(
14     )
15
16 export const getClientStore = () => {
17   const defaultState = window.context.state;
18   // 改变客户端store的内容，一定要使用clientAxios
19   return createStore(reducer, defaultState, applyMiddleware(thunk.withExtraArgument(
20   
```

FOLDERS

```
1 import { CHANGE_LIST } from './constants';
2
3 const changeList = (list) => ({
4   type: CHANGE_LIST,
5   list
6 });
7
8 export const getHomeList = () => {
9   return (dispatch, getState, axiosInstance) => {
10     return axiosInstance.get('/api/news.json?secret=abcd')
11       .then((res) => {
12         const list = res.data.data;
13         dispatch(changeList(list))
14       });
15   }
16 }
```

6-4 使用 renderRoutes 方法实现对多级路由的支持

多级路由

renderRoutes 作用 渲染一级路由，并且把耳机路由的相关信息交给二级路由组件

```
1 import React from 'react';
2 import { Route } from 'react-router-dom';
3 import App from './App';
4 import Home from './containers/Home';
5 import Login from './containers/Login';
6
7 // 当我加载显示HOME组件之前，我希望调用Home.loadData方法，提前获取到必要的异步
8 // 然后再做服务器端渲染，把页面返回给用户
9 export default [
10   {
11     path: '/',
12     component: App,
13     routes: [
14       {
15         path: '/',
16         component: Home,
17         exact: true,
18         loadData: Home.loadData,
19         key: 'home'
20       },
21       {
22         path: '/login',
23         component: Login,
24         exact: true,
25         key: 'login'
26     ]
27 }];
28 
```

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import { BrowserRouter, Route } from 'react-router-dom';
4 import { renderRoutes } from 'react-router-config';
5 import routes from '../Routes';
6 import { getClientStore } from '../store';
7 import { Provider } from 'react-redux';
8
9 const store = getClientStore();
10
11 const App = () => {
12   return (
13     <Provider store={store}>
14       <BrowserRouter>
15         <div>
16           {renderRoutes(routes)}
17         </div>
18       </BrowserRouter>
19     </Provider>
20   )
21 }
22
23 ReactDOM.hydrate(<App />, document.getElementById('root'))
```

```
index.js      App.js      兔*
1 import React from 'react';
2 import Header from './components/Header';
3 import { renderRoutes } from 'react-router-config';
4
5 const App = (props) => {
6   return (
7     <div>
8       <Header />
9       {renderRoutes(props.route.routes)}
10    </div>
11  )
12}
13
14 export default App;
15
```

通过最上层的renderRoutes方法 当前组件 props 中已经添加了 route 属性

6-4 登陆功能的制作

基本业务代码没内容

6-5 登陆状态切换

问题 node服务器 做proxy做接口转发的时候不会转发cookie等信息

```
index.js      actions.js      1. 刚进入页面，处于非登陆状态
1 1. 刚进入页面，处于非登陆状态
2 2. 用户点击登陆按钮，进行登陆操作
3 (1) 浏览器发请求给NodeJS服务器
4 (2) 转发给api服务器，进行登陆
5 (3) api服务器生成cookie
6 (4) 浏览器上存在了cookie，登陆成功
7 3. 当用户重新刷新页面的时候
8 (1) 浏览器去请求html (携带了cookie)
9 (2) NodeJS服务器进行服务器端渲染
10 (3) 进行服务器端渲染，首先要去api服务器取数据 (没有携带cookie)
11
12
```

6-6 解决登陆cookie 传递的问题

通过函数的方式生成axios实例 然后获取到cookie 然后请求后端接口

The screenshot shows a code editor interface with a sidebar labeled "FOLDERS" containing project files like server, build, node_modules, public, src (client, components, containers, server), and various utility and configuration files. Two tabs are open: "request.js" and "index.js". The "request.js" tab contains the following code:

```
1 import axios from 'axios';
2
3 const createInstance = (req) => axios.create({
4   baseURL: 'http://47.95.113.63/ssr',
5   headers: {
6     cookie: req.get('cookie') || ''
7   }
8 });
9
10 export default createInstance;
```

The code defines a function "createInstance" that returns an axios instance with a custom header "cookie" set to the value of "req.get('cookie')". The "index.js" tab is also visible but contains no code.

7-1 secret密钥的统一管理

The screenshot shows a code editor interface with a sidebar labeled "FOLDERS" containing project files like server, build, node_modules, public, src (client, components, containers, server), and various utility and configuration files. Three tabs are open: "request.js — client", "request.js — server", and "config.js". The "request.js — server" tab contains the following code:

```
1 import axios from 'axios';
2 import config from '../config';
3
4 const createInstance = (req) => axios.create({
5   baseURL: 'http://47.95.113.63/ssr',
6   headers: {
7     cookie: req.get('cookie') || ''
8   },
9   params: {
10     secret: config.secret
11   }
12 });
13
14 export default createInstance;
```

A red box highlights the line "secret: config.secret" in the "params" object of the "createInstance" function. The "config.js" tab is also visible but contains no code.

7-2 借助context 实现 404功能

更改200状态为404

componentDidmount 只有在客户端执行，不会在服务端渲染时候执行

The screenshot shows a code editor with two tabs: 'index.js — server' and 'index.js — containers/NotFound'. The 'index.js — server' tab contains the following code:

```
19   }
20 });
21 });
22
23 app.get('*', function (req, res) {
24   const store = getStore(req);
25   // 根据路由的路径, 来往store里面加数据
26   const matchedRoutes = matchRoutes(routes, req.path);
27   // 让matchRoutes里面所有的组件, 对应的loadData方法执行一次
28   const promises = [];
29   matchedRoutes.forEach(item => {
30     if (item.route.loadData) {
31       promises.push(item.route.loadData(store))
32     }
33   })
34   Promise.all(promises).then(() => {
35     const context = {};
36     const html = render(store, routes, req, context);
37
38     if (context.NOT_FOUND) {
39       res.status(404);
40       res.send(html);
41     } else {
42       res.send(html);
43     }
44   })
45 })
46});
```

The 'index.js — containers/NotFound' tab contains the following code:

```
1 import { StaticRouter, Route } from 'react-router-dom';
2 import { renderRoutes } from 'react-router-config';
3 import { Provider } from 'react-redux';
4
5 export const render = (store, routes, req, context) => {
6
7   const content = renderToString(
8     <Provider store={store}>
9       <StaticRouter location={req.path} context={context}>
10         <div>
11           {renderRoutes(routes)}
12         </div>
13       </StaticRouter>
14     </Provider>
15   );
16
17   return `
18     <html>
19       <head>
20         <title>ssr</title>
21       </head>
22       <body>
23         <div id="root">${content}</div>
24       <script>
25         window.context = {
26           state: ${JSON.stringify(store.getState())}
27         }
28       </script>
29     </html>
30   `;
```

A red arrow points from the 'context' variable in the 'index.js — server' code to the 'render' function in the 'index.js — containers/NotFound' code.

The screenshot shows a code editor with two tabs: 'utils.js' and 'index.js — server'. The 'utils.js' tab contains the following code:

```
3 import { StaticRouter, Route } from 'react-router-dom';
4 import { renderRoutes } from 'react-router-config';
5 import { Provider } from 'react-redux';
6
7 export const render = (store, routes, req, context) => {
8
9   const content = renderToString(
10     <Provider store={store}>
11       <StaticRouter location={req.path} context={context}>
12         <div>
13           {renderRoutes(routes)}
14         </div>
15       </StaticRouter>
16     </Provider>
17   );
18
19   return `
20     <html>
21       <head>
22         <title>ssr</title>
23       </head>
24       <body>
25         <div id="root">${content}</div>
26       <script>
27         window.context = {
28           state: ${JSON.stringify(store.getState())}
29         }
30       </script>
31     </html>
32   `;
```

The 'index.js — server' tab contains the following code:

```
19   }
20 });
21 });
22
23 app.get('*', function (req, res) {
24   const store = getStore(req);
25   // 根据路由的路径, 来往store里面加数据
26   const matchedRoutes = matchRoutes(routes, req.path);
27   // 让matchRoutes里面所有的组件, 对应的loadData方法执行一次
28   const promises = [];
29   matchedRoutes.forEach(item => {
30     if (item.route.loadData) {
31       promises.push(item.route.loadData(store))
32     }
33   })
34   Promise.all(promises).then(() => {
35     const context = {};
36     const html = render(store, routes, req, context);
37
38     if (context.NOT_FOUND) {
39       res.status(404);
40       res.send(html);
41     } else {
42       res.send(html);
43     }
44   })
45 })
46});
```

A red arrow points from the 'context' variable in the 'index.js — server' code to the 'render' function in the 'utils.js' code. Another red arrow points from the 'context' variable in the 'utils.js' code to the 'staticContext' variable in the 'index.js — server' code.

The screenshot shows a code editor interface with a sidebar on the left displaying a project's folder structure. The main area contains two tabs: 'index.js — server' and 'index.js — containers/NotFound'. The 'containers/NotFound' tab is active and shows the following code:

```
1 import React, { Component } from 'react';
2
3 class NotFound extends Component {
4
5   componentWillMount() {
6     const { staticContext } = this.props;
7     staticContext && (staticContext.NOT_FOUND = true);
8   } 这里对staticContext中的NOT_FOUND字段进行改变 实际这个指针指向的是我们在404-1图片中设置的content=()中的内容
9
10  render() {
11    return <div>404, sorry, page not found</div>
12  }
13
14 }
15
16 export default NotFound;
17
18
```

The line at index 7 is highlighted with a red box and has a comment: '这里对staticContext中的NOT_FOUND字段进行改变 实际这个指针指向的是我们在404-1图片中设置的content=()中的内容'.

7-3 实现服务器端301重定向功能

客户端的时候我们遇到 <Redirect \> 组件的时候我们访问的话会直接进行重定向操作。

当时服务端渲染的时候只会返回包含 `组件` 的字符串。但是并不会执行重定向的操作。

FOLDERS

```

23 index.js x utils.js x
24 app.get('/:path', function (req, res) {
25   const store = getStore(req);
26   // 根据路由的路径，来往store里面加数据
27   const matchedRoutes = matchRoutes(routes, req.path);
28   // 让matchRoutes里面所有的组件，对应的loadData方法执行一次
29   const promises = [];
30   matchedRoutes.forEach(item => {
31     if (item.route.loadData) {
32       promises.push(item.route.loadData(store))
33     }
34   })
35   Promise.all(promises).then(() => {
36     const context = {};
37     const html = render(store, routes, req, context);
38
39     if (context.action === 'REPLACE') {
40       res.redirect(301, context.url)
41     } else if (context.NOT_FOUND) {
42       res.status(404);
43       res.send(html);
44     } else {
45       res.send(html);
46     }
47   });
48 });
49
50 var server = app.listen(3000);

```

Line 46, Column 5

如果我们访问地址对应的组件中`render{ return(<Redirect />) }`组件的时候 staticRouter 会自动向staticContext 所指向的数据容器中添加一些字段。如下：

FOLDERS

```

1 import React from 'react';
2 import { renderToString } from 'react-dom/server';
3 import { StaticRouter, Route } from 'react-router-dom';
4 import { renderRoutes } from 'react-router-config';
5 import { Provider } from 'react-redux';
6
7 export const render = (store, routes, req, context) => {
8
9   const content = renderToString(
10     <Provider store={store}>
11       <StaticRouter location={req.path} context={context}>
12         <div>
13           {renderRoutes(routes)}
14         </div>
15       </StaticRouter>
16     </Provider>
17   );
18
19   return `
20     <html>
21       <head>
22         <title>ssr</title>
23       </head>
24       <body>
25         <div id="root">${content}</div>
26         <script>
27           window.context = {
28             state: ${JSON.stringify(store.getState())}

```

Line 15, Column 24

当我们访问的一个路径对应的组件中存在`Redirect`组件的时候，`staticRouter`会帮我们向`context`中添加下面的这些字段（自动添加）

7-4 数据请求失败情况下Promise的处理

```
// promises = [ a, b, c, d ]
Promise.all(promises).then(() => {
  const context = {};
  const html = render(store, routes, req, context);
  if (context.action === 'REPLACE') {
    res.redirect(301, context.url)
  }else if (context.NOT_FOUND) {
    res.status(404);
    res.send(html);
  }else {
    res.send(html);
  }
}).catch(()=>{
  const context = {};
  const html = render(store, routes, req, context);
  if (context.action === 'REPLACE') {
    res.redirect(301, context.url)
  }else if (context.NOT_FOUND) {
    res.status(404);
    res.send(html);
  }else {
    res.send(html);
  }
});
```

```

// promises = [ a, b, c, d ]
Promise.all(promises).then(() => {
  const context = {};
  const html = render(store, routes, req, context);
  if (context.action === 'REPLACE') {
    res.redirect(301, context.url)
  } else if (context.NOT_FOUND) {
    res.status(404):
    res.send(html);
  } else {
    res.send(html);
  }
}) .catch((err) => {
  const context = {};
  const html = render(store, routes, req, context);
  if (context.action === 'REPLACE') {
    res.redirect(301, context.url)
  } else if (context.NOT_FOUND) {
    res.status(404);
    res.send(html);
  } else {
    res.send(html);
  }
});

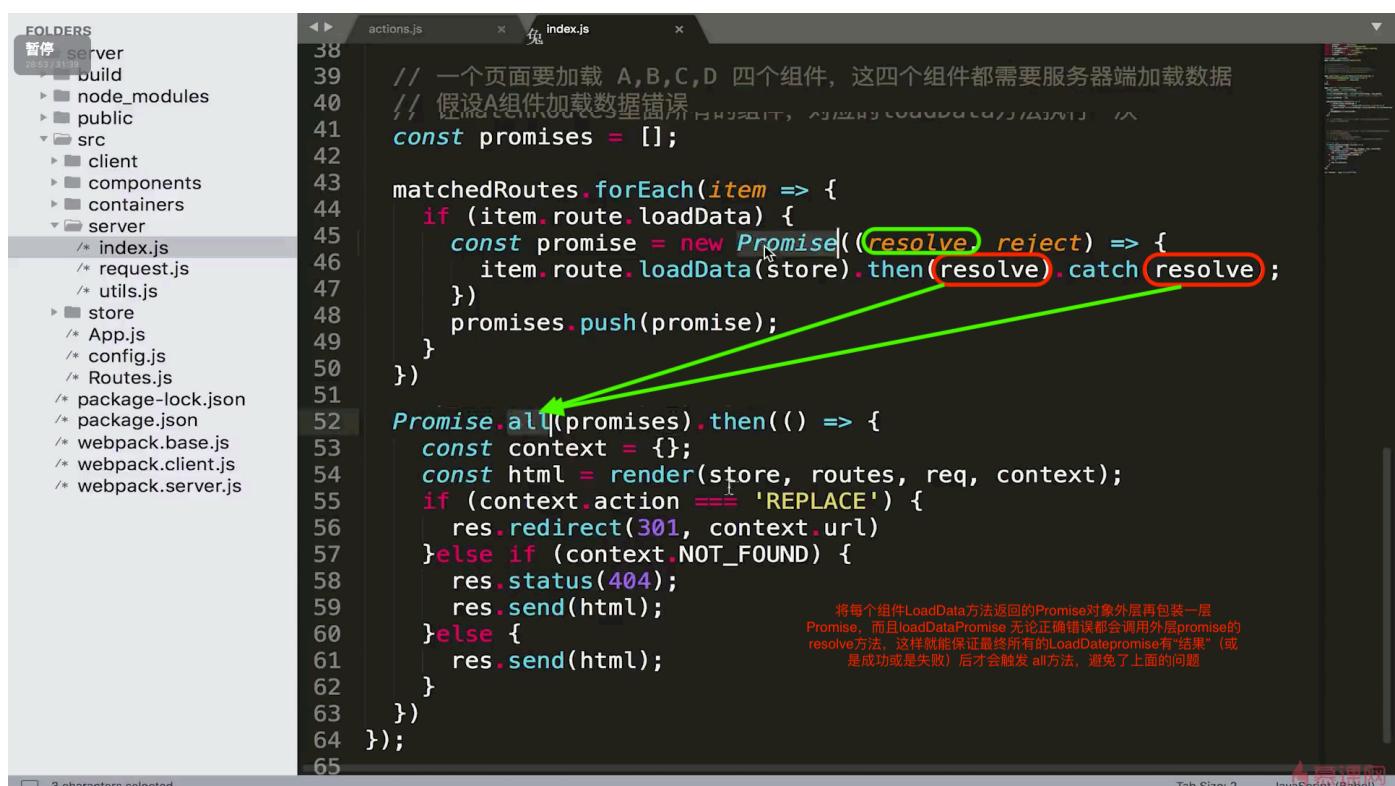
```

// 一个页面要加载 A,B,C,D 四个组件，这四个组件都需要服务器端加载数据
 // 假设A组件加载数据错误
 // B, C, D 组件有几种情况
 // 1. B, C, D 组件数据已经加载完成了
 // 2. 假设B, C, D 接口比较慢，B, C, D 组件数据没有加载完成

兔

可见我们粗鲁的使用.catch 报错后一点没有给 bcd机会
直接走了catch的逻辑，不等bcd 接口请求成功

痛点！等所有loadDataPromise 有结果了我们再返回，结果可好可坏，有结果我们再返回数据！不要被一个接口的报错臭了一锅汤



The screenshot shows a code editor with two tabs: 'actions.js' and 'index.js'. The 'actions.js' tab is active, displaying the following code:

```

38 // 一个页面要加载 A,B,C,D 四个组件，这四个组件都需要服务器端加载数据
39 // 假设A组件加载数据错误
40 // B, C, D 组件有几种情况
41 const promises = [];
42
43 matchedRoutes.forEach(item => {
44   if (item.route.loadData) {
45     const promise = new Promise((resolve, reject) => {
46       item.route.loadData(store).then(resolve).catch(reject);
47     })
48     promises.push(promise);
49   }
50 })
51
52 Promise.all(promises).then(() => {
53   const context = {};
54   const html = render(store, routes, req, context);
55   if (context.action === 'REPLACE') {
56     res.redirect(301, context.url)
57   } else if (context.NOT_FOUND) {
58     res.status(404);
59     res.send(html);
60   } else {
61     res.send(html);
62   }
63 })
64 });
65

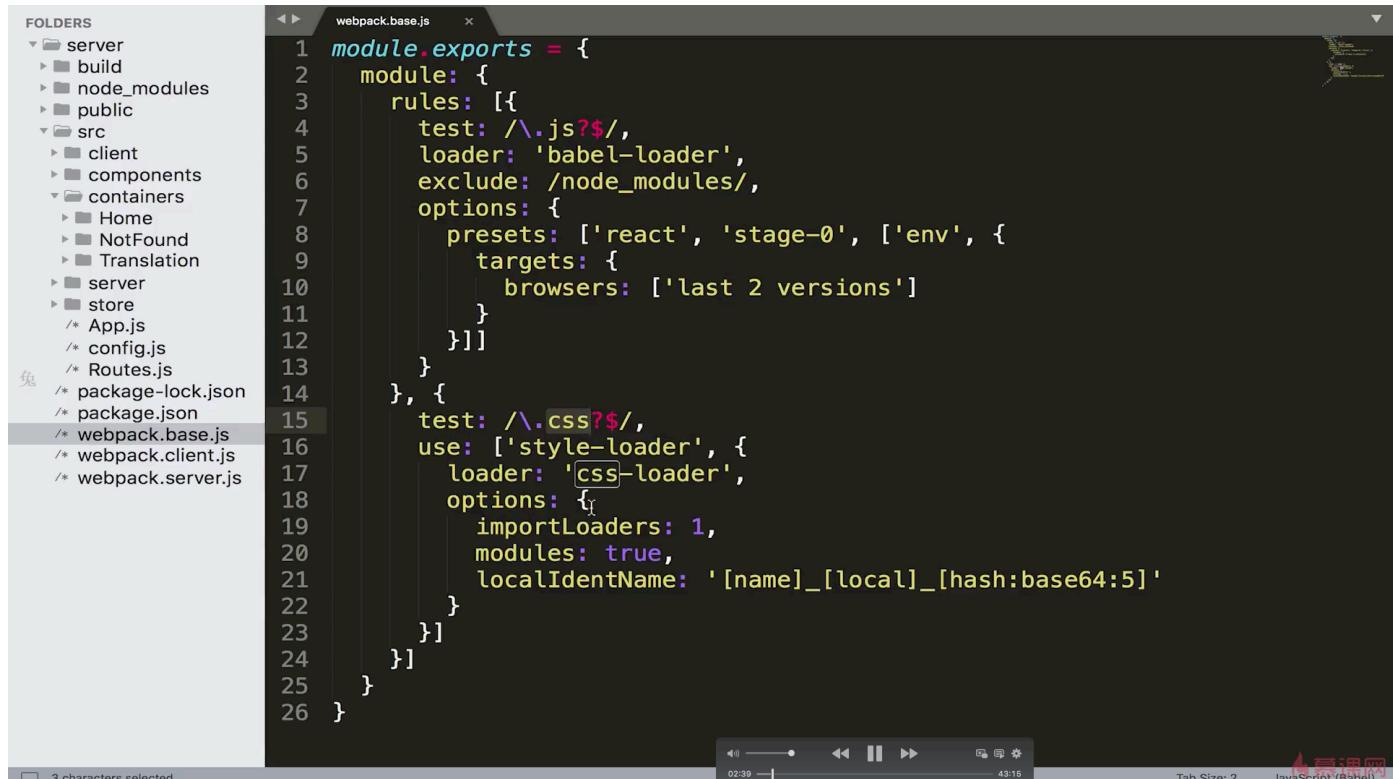
```

A green arrow points from the line 'item.route.loadData(store).then(resolve).catch(reject);' to the line 'Promise.all(promises).then(() => {'. Another green arrow points from the line 'Promise.all(promises).then(() => {' back up to the line 'const promises = [];'.

A red box highlights the 'reject' parameter in the promise constructor and the 'reject' parameter in the 'then' method of the promise returned by 'loadData'. A red circle highlights the 'reject' parameter in the 'catch' block of the promise returned by 'loadData'.

A note at the bottom right of the code editor states: '将每个组件LoadData方法返回的Promise对象外层再包装一层Promise，而且 loadDataPromise 无论正确错误都会调用外层promise的 resolve方法，这样就能保证最终所有的LoadDatepromise有“结果”（或是成功或是失败）后才会触发 all方法，避免了上面的问题'.

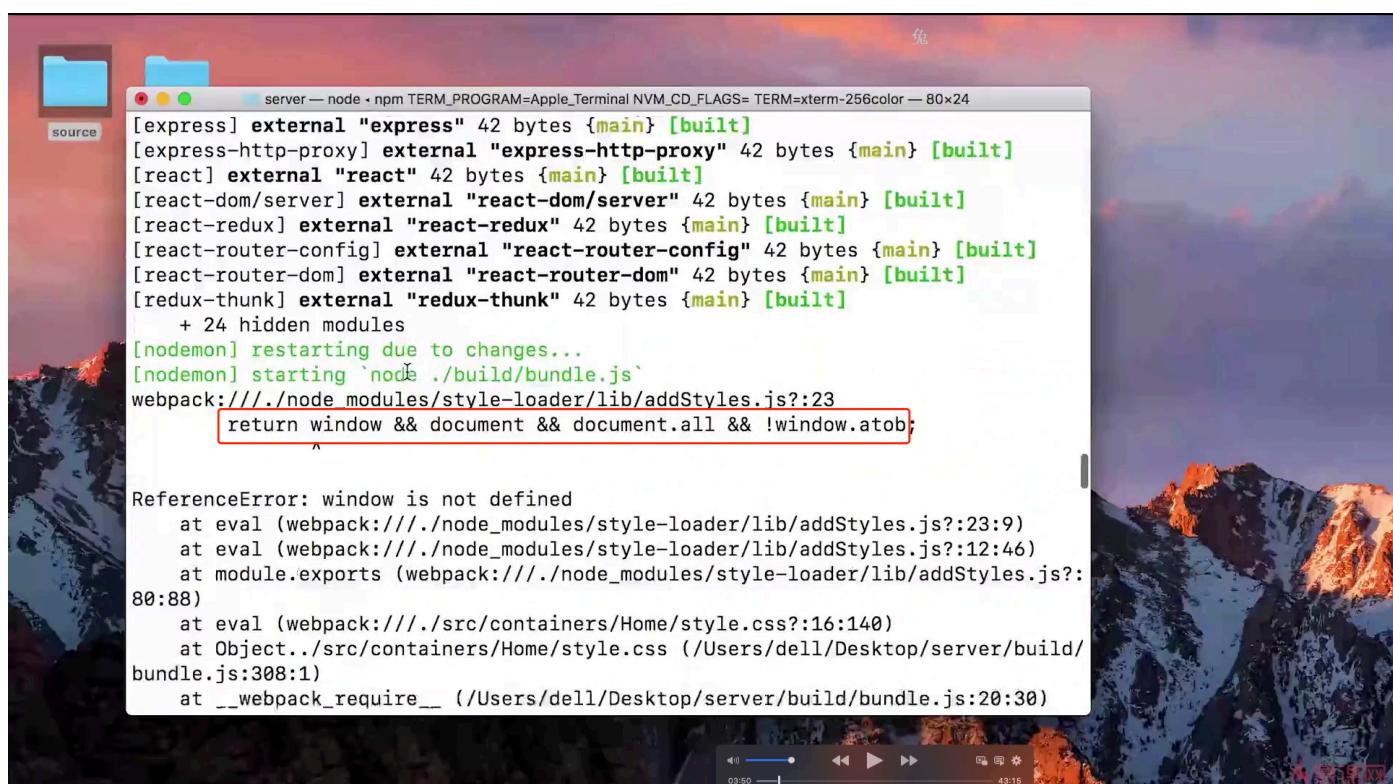
8-1如何支持css样式修饰



```
module.exports = {
  module: {
    rules: [
      { test: /\.js$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        options: {
          presets: ['react', 'stage-0', ['env', {
            targets: {
              browsers: ['last 2 versions']
            }
          }]]
        }
      },
      { test: /\.css$/,
        use: ['style-loader', {
          loader: 'css-loader',
          options: {
            importLoaders: 1,
            modules: true,
            localIdentName: '[name]_[local]_[hash:base64:5]'
          }
        }]
      }
    ]
  }
}

3 characters selected
```

```
npm install style-loader css-loader --save-dev
```



```
[express] external "express" 42 bytes {main} [built]
[express-http-proxy] external "express-http-proxy" 42 bytes {main} [built]
[react] external "react" 42 bytes {main} [built]
[react-dom/server] external "react-dom/server" 42 bytes {main} [built]
[react-redux] external "react-redux" 42 bytes {main} [built]
[react-router-config] external "react-router-config" 42 bytes {main} [built]
[react-router-dom] external "react-router-dom" 42 bytes {main} [built]
[redux-thunk] external "redux-thunk" 42 bytes {main} [built]
+ 24 hidden modules
[nodemon] restarting due to changes...
[nodemon] starting `node ./build/bundle.js`
webpack:///./node modules/style-loader/lib/addStyles.js?:23
    return window && document && document.all && !window.atob;
```

```
ReferenceError: window is not defined
  at eval (webpack:///./node_modules/style-loader/lib/addStyles.js?:23:9)
  at eval (webpack:///./node_modules/style-loader/lib/addStyles.js?:12:46)
  at module.exports (webpack:///./node_modules/style-loader/lib/addStyles.js?:80:88)
  at eval (webpack:///./src/containers/Home/style.css?:16:140)
  at Object../src/containers/Home/style.css (/Users/dell/Desktop/server/build/bundle.js:308:1)
  at __webpack_require__ (/Users/dell/Desktop/server/build/bundle.js:20:30)
```

style-loader需要往window上面挂载一些东西，但是服务端渲染是没有window对象的

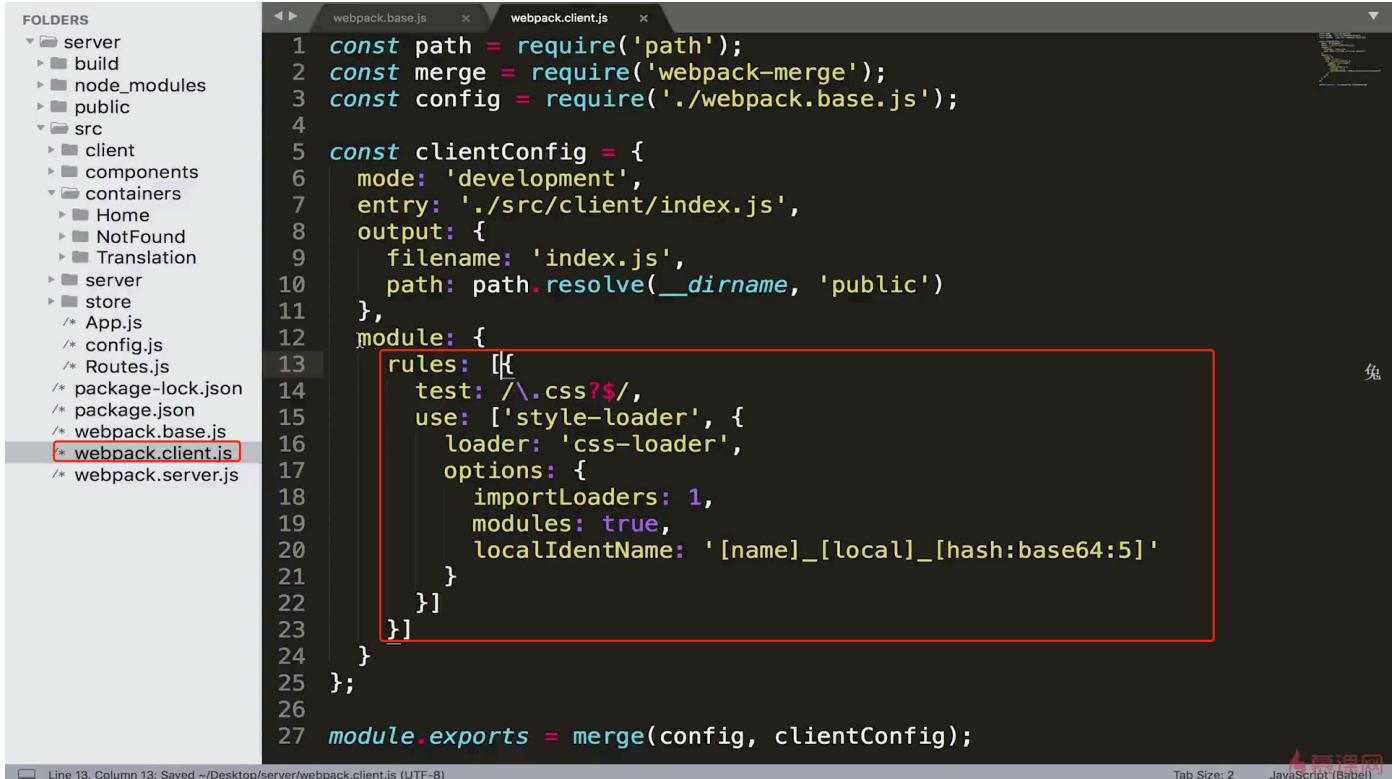
解决办法：

不能再base中添加 rules 也就是将客户端和服务器端的css 解析打包配置拆分开

客户端：style-loader

服务器端：isomorphic

客户端：

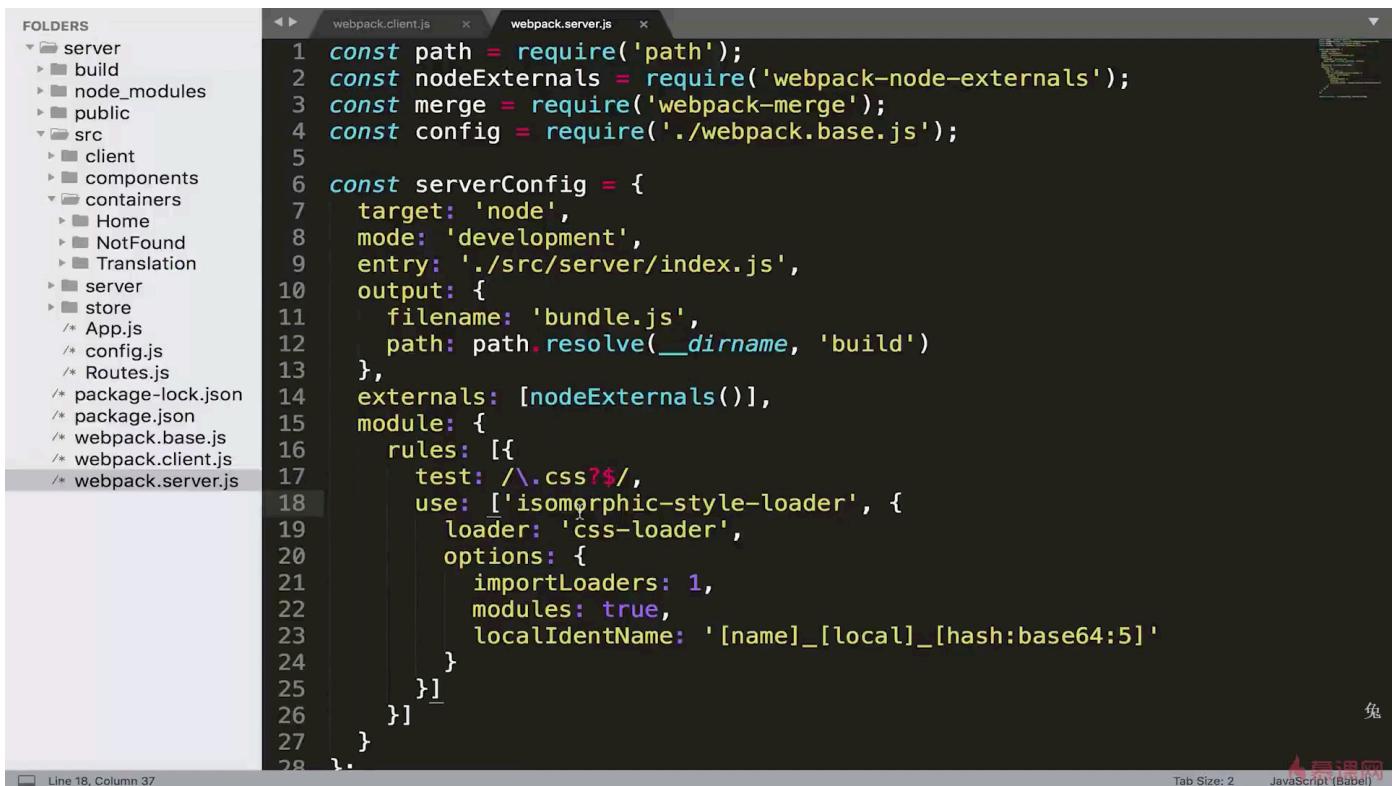


```
FOLDERS
webpack.base.js webpack.client.js
1 const path = require('path');
2 const merge = require('webpack-merge');
3 const config = require('./webpack.base.js');
4
5 const clientConfig = {
6   mode: 'development',
7   entry: './src/client/index.js',
8   output: {
9     filename: 'index.js',
10    path: path.resolve(__dirname, 'public')
11  },
12  module: {
13    rules: [
14      {
15        test: /\.css$/,
16        use: ['style-loader', {
17          loader: 'css-loader',
18          options: {
19            importLoaders: 1,
20            modules: true,
21            localIdentName: '[name]_[local]_[hash:base64:5]'
22          }
23        }]
24      }
25    ],
26  },
27  module.exports = merge(config, clientConfig);

```

Line 13, Column 13; Saved ~/Desktop/server/webpack.client.js (UTF-8) Tab Size: 2 JavaScript (Babel)

服务端：



```
FOLDERS
webpack.client.js webpack.server.js
1 const path = require('path');
2 const nodeExternals = require('webpack-node-externals');
3 const merge = require('webpack-merge');
4 const config = require('./webpack.base.js');
5
6 const serverConfig = {
7   target: 'node',
8   mode: 'development',
9   entry: './src/server/index.js',
10  output: {
11    filename: 'bundle.js',
12    path: path.resolve(__dirname, 'build')
13  },
14  externals: [nodeExternals()],
15  module: {
16    rules: [
17      {
18        test: /\.css$/,
19        use: ['isomorphic-style-loader', {
20          loader: 'css-loader',
21          options: {
22            importLoaders: 1,
23            modules: true,
24            localIdentName: '[name]_[local]_[hash:base64:5]'
25          }
26        }]
27      }
28    ],
29  }
30 }

```

Line 18, Column 37; Saved ~/Desktop/server/webpack.server.js (UTF-8) Tab Size: 2 JavaScript (Babel)

本节遗留问题：isomorphic 虽然能够解决style-loader 在服务端不被支持的问题，但是其又会导致样式闪屏（页面结构先被加载进来，但是样式后被加载进来）而且样式是通过js加载进来的 禁用js后样式无法加载。

8-2如何实现css样式的服务器端渲染

上面的一节只是支持了css。而且最终的遗留问题也是因为样式并没有在服务端渲染好。

isomorphic-style-loader 发现页面引入了样式。div上面有class名字，他就会在渲染页面的时候把这个class名字加到最终返回的字符串中

style-loder：也会干同样的事情，但是额外在html中添加了<style></style>标签，里面包含了这些样式，所以页面能够有样式。

所以我们在服务器端直接把这个样式放在 style标签中不就好了！

The screenshot shows a code editor with a file named 'index.js' open. The code is written in JavaScript and uses React components. A red box highlights a specific section of the code where a CSS class is being used.

```
index.js
1 import React, { Component } from 'react';
2 import { connect } from 'react-redux';
3 import { getHomeList } from './store/actions';
4 import styles from './style.css';

5
6 class Home extends Component {
7
8     componentWillMount() {
9         if (this.props.staticContext) {
10             this.props.staticContext.css = styles._getCss();
11         }
12     }
13
14     getList() {
15         const { list } = this.props;
16         return list.map(item => <div key={item.id}>{item.title}</div>)
17     }
18
19     render() {
20         return (
21             <div className={styles.test}>
22                 {this.getList()}
23                 <button onClick={()=>{alert('click1')}}>
24                     click
25                 </button>
26             </div>
27         )
28     }
}
```

A green arrow points from the text "isomorphic-style-loader 提供的方法" to the line of code where the CSS class is being used: `this.props.staticContext.css = styles._getCss();`.

Annotations in the code:

- isomorphic-style-loader 提供的方法 (Annotation pointing to the line `this.props.staticContext.css = styles._getCss();`)
- 兔 (Annotation pointing to the button element in the render method)

FOLDERS

```

  server
    build
    node_modules
    public
  src
    client
    components
    containers
      Home
        store
          index.js
          style.css
      NotFound
      Translation
  server
    index.js
    request.js
    utils.js
  store
    App.js
    config.js
    Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

webpack.server.js index.js — containers/Home index.js — server

```

29
30 matchedRoutes.forEach(item => {
31   if (item.route.loadData) {
32     const promise = new Promise((resolve, reject) => {
33       item.route.loadData(store).then(resolve).catch(reject);
34     })
35     promises.push(promise);
36   }
37 }
38
39 Promise.all(promises).then(() => {
40   const context = {};
41   const html = render(store, routes, req, context);
42   console.log(context.css, '-----');
43
44
45   if (context.action === 'REPLACE') {
46     res.redirect(301, context.url)
47   } else if (context.NOT_FOUND) {
48     res.status(404);
49     res.send(html);
50   } else {
51     res.send(html);
52   }
53 });
54 });
55
56 var server = app.listen(3000);

```

4 characters selected Tab Size: 2 JavaScript (Babel)

FOLDERS

```

  server
    build
    node_modules
    public
  src
    client
    components
    containers
      Home
        store
          index.js
          style.css
      NotFound
      Translation
  server
    index.js
    request.js
    utils.js
  store
    App.js
    config.js
    Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

```

webpack.server.js index.js — containers/Home index.js — server utils.js

```

10
11   <StaticRouter location={req.path} context={context}>
12     <div>
13       {renderRoutes(routes)}
14     </div>
15   </StaticRouter>
16   <Provider>
17     <div id="root">${content}</div>
18   </Provider>
19
20   const cssStr = context.css ? context.css : '';
21
22   return `
23     <html>
24       <head>
25         <title>ss</title>
26         <style>${cssStr}</style>
27       </head>
28       <body>
29         <div id="root">${content}</div>
30         <script>
31           window.context = {
32             state: ${JSON.stringify(store.getState())}
33           }
34         </script>
35         <script src='/index.js'></script>
36       </body>
37     </html>
38   `;

```

Line 19, Column 51 Tab Size: 2 JavaScript (Babel)

8-3 多组件中的样式如何整合

上节课遗留的问题：

- 注意staticContext 只传递给当前路由匹配的到的组件，并不会传递给这个组件包含的父组件。

2. 每个组件中我们都更改了 staticContext.css 的值，我们获取的最后的值肯定是最后一次更改的值，之前的值都被抹去了

```
componentWillMount() {
  if (this.props.staticContext) {
    this.props.staticContext.css = styles._getCss();
  }
}
```

解决办法

使用数组容器 每个组件push 进去自己的样式就好了

```
FOLDERS
server
  build
  node_modules
  public
src
  client
  components
  containers
    Home
    store
    index.js
    style.css
  NotFound
  Translation
server
  index.js
  request.js
  utils.js
store
  App.js
  config.js
  Routes.js
  package-lock.json
  package.json
  webpack.base.js
  webpack.client.js
  webpack.server.js

index.js
27 // 让matchRoutes里面所有的组件，对应的loadData方法执行一次
28 const promises = [];
29
30 matchedRoutes.forEach(item => {
31   if (item.route.loadData) {
32     const promise = new Promise((resolve, reject) => {
33       item.route.loadData(store).then(resolve).catch(reject);
34     })
35     promises.push(promise);
36   }
37 })
38
39 Promise.all(promises).then(() => {
40   const context = {
41     css: []
42   };
43   const html = render(store, routes, req, context);
44
45   if (context.action === 'REPLACE') {
46     res.redirect(301, context.url)
47   } else if (context.NOT_FOUND) {
48     res.status(404);
49     res.send(html);
50   } else {
51     res.send(html);
52   }
53 })
```

```
componentWillMount() {
  if (this.props.staticContext) {
    this.props.staticContext.css.push(styles._getCss());
  }
}
```

The screenshot shows a code editor with a sidebar containing a file tree. The tree includes folders like 'server', 'build', 'node_modules', 'public', 'src', 'client', 'components', 'containers', 'Home', 'NotFound', 'Translation', and 'server' (which contains 'index.js', 'request.js', and 'utils.js'). The 'utils.js' file is open in the main pane, showing the following code:

```
1 import React from 'react';
2 import { renderToString } from 'react-dom/server';
3 import { StaticRouter, Route } from 'react-router-dom';
4 import { renderRoutes } from 'react-router-config';
5 import { Provider } from 'react-redux';
6
7 export const render = (store, routes, req, context) => {
8
9     const content = renderToString((
10         <Provider store={store}>
11             <StaticRouter location={req.path} context={context}>
12                 <div>
13                     {renderRoutes(routes)}
14                 </div>
15             </StaticRouter>
16         </Provider>
17     )));
18
19     const cssStr = context.css.length ? context.css.join('\n') : '';
20
21     return `
22         <html>
23             <head>
24                 <title>ssr</title>
25                 <style></style>
26             </head>
27             <body>
28                 <div id="root">${content}</div>

```

Line 19, Column 61: Saved ~/Desktop/server/src/server/utils.js (UTF-8) Tab Size: 2 JavaScript (Babel)

8-4 loadData方法潜在问题修正

问题核心：高阶组件有可能丢失原组件的静态方法

当我们通过路由访问页面时候（初始状态/刷新页面），会调用页面所涉及组件的LoadData方法加载好所需要的数据，

但是我们之前在文件导出组件的时候导出的是经过高阶组件生成的一个新的组件。这个新的组件上面可能没有这个loadData方法（虽然connect方法对之前的组件进行了静态方法的拷贝）但是出于严谨，应该将LoadData方法挂载到高阶组件生成的组件上面

FOLDERS

```

    > server
    > build
    > node_modules
    > public
    > src
        > client
        > components
        > containers
            > Home
                > store
                    /* index.js
                     * style.css
                    */
            > NotFound
            > Translation
        > server
        > store
            /* App.js
             * config.js
             * Routes.js
            */
            /* package-lock.json
             * package.json
            */
            /* webpack.base.js
             * webpack.client.js
            */
            /* webpack.server.js
            */

```

Routes.js index.js

```

27 ,
28 }
29
30     componentDidMount() {
31         if (!this.props.list.length) {
32             this.props.getHomeList();
33         }
34     }
35 }
36
37 const mapStateToProps = state => ({
38     list: state.home.newsList
39 });
40
41 const mapDispatchToProps = dispatch => ({
42     getHomeList() {
43         dispatch(getHomeList());
44     }
45 });
46
47 const ExportHome = connect(mapStateToProps, mapDispatchToProps)(Home)
48
49 ExportHome.loadData = (store) => {
50     return store.dispatch(getHomeList())
51 }
52
53 export default ExportHome;
54

```

Line 35, Column 2 99% Tab Size: 2 JavaScript (Babel)

8-5 使用高阶组件精简代码

样式的逻辑很重复啊！直接使用高阶组件进行逻辑服用

FOLDERS

```

    暂停   兔
    > server
        > build
        > node_modules
        > public
    > src
        > client
        > components
            > Header
                > store
                    /* index.js
                     * style.css
                    */
            > containers
                > Home
                    > store
                        /* index.js
                         * style.css
                        */
                > NotFound
                > Translation
            > server
            > store
                /* App.js
                 * config.js
                 * Routes.js
                */
                /* package-lock.json
                 * package.json
                */
                /* webpack.base.js
                 * webpack.client.js
                */
                /* webpack.server.js
                */

```

index.js — containers/Home index.js — components/Header

```

1 import React, { Fragment, Component } from 'react';
2 import { Link } from 'react-router-dom';
3 import { connect } from 'react-redux';
4 import { actions } from './store/';
5 import styles from './style.css';
6
7 class Header extends Component {
8
9     componentWillMount() {
10         if (this.props.staticContext) {
11             this.props.staticContext.css.push(styles._getCss());
12         }
13     }
14
15     render() {
16         const { login, handleLogin, handleLogout } = this.props;
17         return (
18             <div className={styles.test}>
19                 <Link to='/'>首页</Link>
20                 <br />
21                 {
22                     login ? <Fragment>
23                         <Link to='/translation'>翻译列表</Link>
24                         <br />
25                         <div onClick={handleLogout}>退出</div>
26                     </Fragment> : <div onClick={handleLogin}>登陆</div>
27                 }
28             </div>

```

Line 1, Column 1; Detect Indentation: Setting indentation to tabs Tab Size: 2 JavaScript (Babel)

具体代码：

最终在组件中 使用下这个方法就可以了

```
1 import React, { Component } from 'react';
2
3 // 这个函数，是生成高阶组件的函数
4 // 这个函数，返回一个组件
5 export default (DecoratedComponent, styles) => {
6   // 返回的这个组件，叫做高阶组件
7   return class NewComponent extends Component {
8
9     componentWillMount() {
10       if (this.props.staticContext) {
11         this.props.staticContext.css.push(styles._getCss());
12       }
13     }
14
15     render() {
16       return <DecoratedComponent {...this.props} />
17     }
18   }
19 }
20
21 }
```

12 characters selected Tab Size: 2 JavaScript (Babel) 菜谱网

9-1什么是SEO 为什么服务器端渲染对seo更加友好

seo search engine optimazetion

基础版：

<title>

二代代搜索引擎（基于全文索引的搜索引擎），不仅仅根据title判断网页的内容还会根据整个页面全文的内容进行判断

但是我们搜索结果的标题 还是由title决定的

</title>:

<meta name="description" content="我是一个重要的页面" >

展示在搜索结果的介绍中

</meta>

慕课网-程序员的梦工厂



title

慕课网(IMOOC)是IT技能学习平台。慕课网(IMOOC)提供了丰富的移动端开发、php开发、web前端、android开发以及html5等视频教程资源公开课。并且富有交互性及趣味性,你...

<https://www.imooc.com/> - 百度快照 - 187条评论

meta description

慕课

慕课网精品课程,为您提供专业的IT实战开发课程,包含前端开发、后端开发、

前端开发教程

慕课网前端开发精品课程,为您提供专业的IT实战开发课程,web前端开发教程

9-2 如何做好SEO

1.原创性

2.链接的地址 跳转后的页面和之前的页面相关性要强, 搜索引擎认为这个连接的价值大, 外部链接指向这个网站越多, 证明这个网站价值更大

3.多媒体的优化, 比如图片的原创性, 和高清。目前先进的搜索引擎比如google 也能识别这些多媒体。丰富度越高排名会靠前

9-3 使用React-Helmet进行每个页面title和meta的独立定制

客户端: 会自动将Helmet 组件内部的child (title/meta) 放在html中正确的位置

The screenshot shows a code editor with a file named 'index.js' open. The code uses the 'Helmet' library to set the title and meta description for the page. A yellow box highlights the Helmet component usage:

```
import { Helmet } from 'react-helmet';
import { getHomeList } from './store/actions';
import styles from './style.css';
import withStyle from '.../.../withStyle';

class Home extends Component {
  getList() {
    const { list } = this.props;
    return list.map(item => <div className={styles.item} key={item}>)
  }

  render() {
    return (
      <Fragment>
        <Helmet>
          <title>这是DellLee的SSR新闻页面 - 丰富多彩的资讯</title>
          <meta name="description" content="这是DellLee的SSR新闻页面 - 丰富多彩的资讯" />
        </Helmet>
        <div className={styles.container}>
          {this.getList()}
        </div>
      </Fragment>
    )
  }

  componentDidMount() {
    if (!this.props.list)
  }
}
```

Line 20, Column 72 18:10 43:31 Tab Size: 2 JavaScript (Babel)

服务端:

FOLDERS

```

暂停 21:29 / 43:31
  - server
    - build
    - node_modules
    - public
  - src
    - client
    - components
  - containers
    - Home
      - store
        - bg.jpeg
        - index.js
        - paris.jpeg
        - style.css
    - NotFound
    - Translation
  - server
    - index.js
    - request.js
  - utils.js
  - store
    - App.js
    - config.js
    - Routes.js
    - withStyles.js
  - package-lock.json
  - package.json
  - webpack.base.js
  - webpack.client.js
  - webpack.server.js

```

utils.js

```

6 import { Helmet } from "react-helmet";
7
8 export const render = (store, routes, req, context) => {
9
10   const content = renderToString(
11     <Provider store={store}>
12       <StaticRouter location={req.path} context={context}>
13         <div>
14           {renderRoutes(routes)}
15         </div>
16       </StaticRouter>
17     </Provider>
18   );
19   const helmet = Helmet.renderStatic();
20
21   const cssStr = context.css.length ? context.css.join('\n') : 
22     Helmet.renderStatic会将组件中的我们使用的<Helmet></Helmet>组件中的内容进行提取获取
23   return `
24     <html>
25       <head>
26         ${helmet.title.toString()}
27         ${helmet.meta.toString()}
28         <style>${cssStr}</style>
29       </head>
30       <body>
31         <div id="root">${content}</div>
32         <script>
33           window.context =

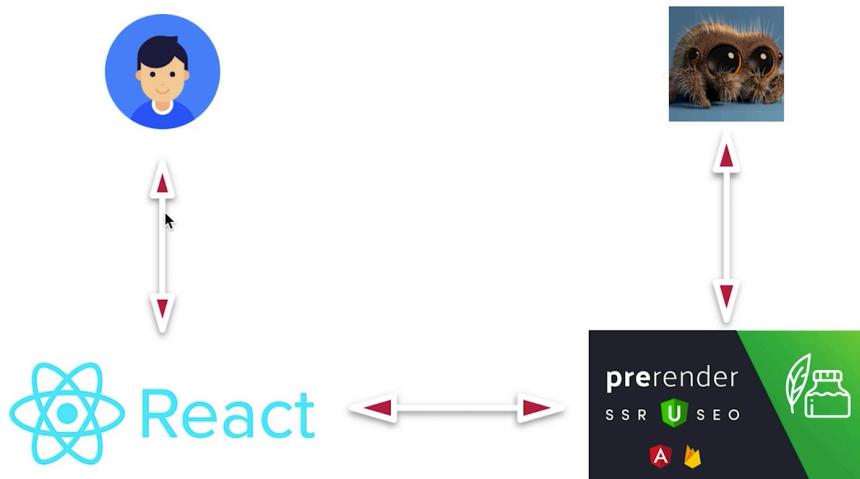
```

Line 26, Column 23

Tab Size: 2 JavaScript (Babel)

预渲染

暂停
31:01 / 43:31

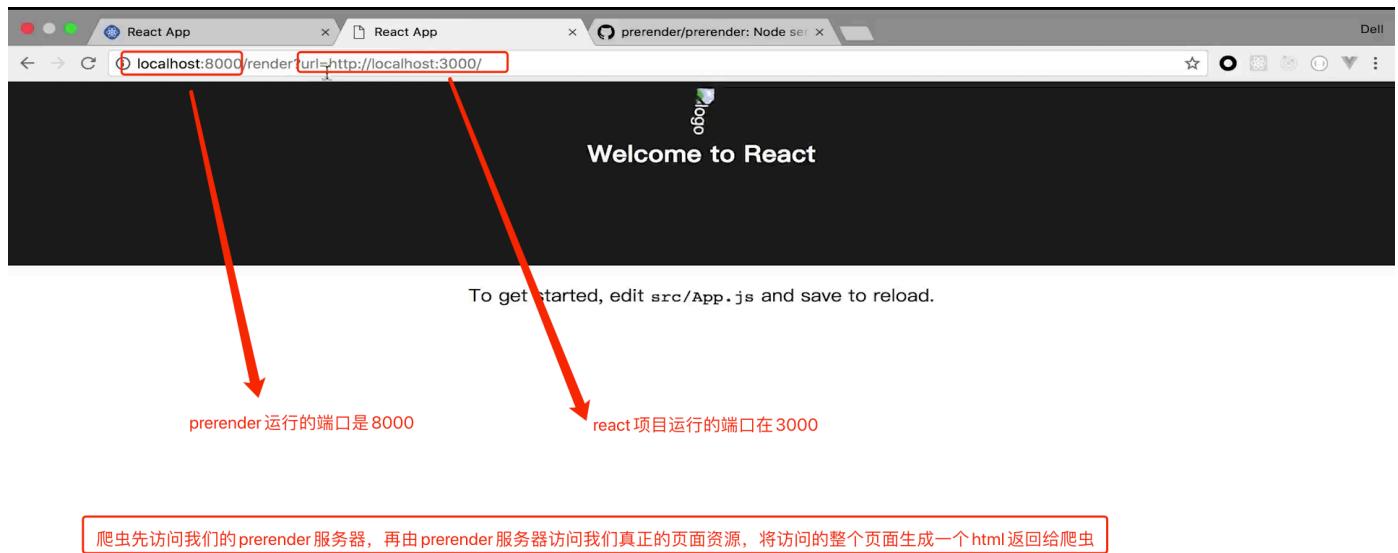


普通用户访问这个页面只是一个普通的客户端渲染的页面

但是搜索引擎爬虫访问这个项目的时候， 经过预渲染已经是"有页面内容"的页面了

实际上预渲染就是访问一下页面， 然后将页面所有的内容生成一个html 然后返回给蜘蛛爬虫

可以搜索下prerender 项目了解下



但是预渲染技术就需要我们识别下访问我们页面的是 爬虫还是用户

所以在项目架构的外层 添加一层Nginx服务器，由nginx服务器通过user-agent/ip等信息 来识别。

prerender 底层有个pentam.js

prerender原理，爬虫访问prerender服务器，prerender服务器会生成一个小浏览器，再由这个小的浏览器去访问目标页面。识别网页中的内容，返回给爬虫，但是这个过程可能耗时。如果仅仅是出于Seo效果的话 prerender的架构更好。

但是只是出于首屏时间的话还是ssr架构

兔

