【讲义】React服务端渲染&同构

#2021#

React服务端渲染

背景

- 第一阶段

很久以前, 一个网站的开发还是前端和服务端在一个项目来维护, 可能是用php+jquery.
那时候的页面渲染是放在服务端的, 也就是用户访问一个页面a的时候, 会直接访问服务端路由, 由服务端来渲染页面然后返回给浏览器。

也就是说网页的所有内容都会一次性被写在html里, 一起送给浏览器。
这时候你右键点击查看网页源代码, 可以看到所有的代码; 或者你去查看html请求, 查看"预览", 会发现他就是一个完整的网页。

- 第二阶段

但是慢慢的人们觉得上面这种方式前后端协同太麻烦, 耦合太严重, 严重影响开发效率和体验。
于是随着vue/react的横空出世, 人们开始习惯了纯客户端渲染的spa.

这时候的html中只会写入一些主脚本文件, 没有什么实质性的内容. 等到html在浏览器端解析后, 执行js文件, 才逐步把元素创建在dom上。
所以你去查看网页源代码的时候, 发现根本没什么内容, 只有各种脚本的链接。

- 第三阶段

后来人们又慢慢的觉得, 纯spa对SEO非常不友好, 并且白屏时间很长。
对于一些活动页, 白屏时间长代表了什么? 代表了用户根本没有耐心去等待页面加载完成.

所以人们又想回到服务端渲染, 提高SEO的效果, 尽量缩短白屏时间.

那难道我们又要回到阶段一那种人神共愤的开发模式吗? 不, 我们现在有了新的方案, 新的模式, 叫做同构。

所谓的同构理解为：同种结构的不同表现形态, 同一份react代码, 分别在两端各执行一遍。

创建一个服务端渲染的应用

1. renderToString

spa里一定见过react-dom的render方法, 其实react-dom里还有一个renderToString方法. 咱们来查看一下他的注释.

```
Render a React element to its initial HTML. This should only be used on the server. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.
```

这个注释, 岂不是明牌说这个方法是用来做服务端渲染的? 我们来试一下

2. Hello World

- server.js

```
const express = require('express');
const app = express();
const React = require('react');
const {renderToString} = require('react-dom/server');
const PORT = 3000;

const App = class extends React.PureComponent{
  render(){
    return React.createElement("h1",null,"Hello World111");;
  }
};
app.get('/',function(req,res){
  const content = renderToString(React.createElement(App));
  res.send(content);
});
```

```
app.listen(PORT, () => {
    console.log(`sever listen on ${PORT}`)
});
```

`node server.js`

3. webpack配置

上面咱们看起来就是服务端渲染了对吗? 虽然只是看起来实现了, 但是存在很多问题.

- 可以用jsx语法吗

明显咱们现在是不支持jsx语法的, 根本都没有react的编译环境

- 可以用esm吗

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

所以我们要针对服务端代码, 做一个webpack的打包工作.

- build/webpack-server.config.js

```
const path = require('path');
const nodeExternals = require('webpack-node-externals');
const CopyWebpackPlugin = require('copy-webpack-plugin');
module.exports = {
  entry:{
    index:path.resolve(__dirname,'../server.js')
  },
  mode:'development',
  target:'node',
  devtool: 'cheap-module-eval-source-map',
  output:{
    filename:'[name].js',
    path:path.resolve(__dirname,'../dist/server')
  },
  externals:[nodeExternals()],
```

```
  resolve:{
    alias:{
      '@':path.resolve(__dirname,'../src')
    },
    extensions:['.js']
  },
  module:{
    rules:[{
      test:/\.js$/,
      use:'babel-loader',
      exclude:/node_modules/
    }]
  },
  plugins: [
    new CopyWebpackPlugin([{
      from:path.resolve(__dirname,'../public'),
      to:path.resolve(__dirname,'../dist')
    }])
  ]
}
```

- package.json添加打包的脚本

```
"build:server": "webpack --config build/webpack-server.config.js --watch"
```

- package.json添加运行server.js的脚本

```
"server": "nodemon dist/server/index.js"
```

运行一下

```
npm run build:server
npm run server
```

- esm的模块引入方式已经支持了

```
import React from 'react';
```

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

4. 给h1标签绑定一个click事件

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

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

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);
```

然后咱们看一下效果, 发现怎么点击都没有反应?????

稍微想一下大家就可以理解, renderToString是把元素转成字符串而已, 事件什么的根本没有绑定。

5. 同构

这时候就用到了我们这个同构的概念

同一份代码, 在服务端跑一遍, 就生成了html

同一份代码, 在客户端跑一遍, 就能响应各种用户操作

所以需要将App单独提取出来

- src/app.js

```
import React from 'react';

class App extends React.PureComponent{
    handleClick=(e)=>{
        alert(e.target.innerHTML);
    }
    render(){
        return <h1 onClick={this.handleClick}>Hello World!</h1>;
    }
};

export default App;
```

- src/index.js

就跟正常spa应用一样的写法

```
import React from 'react';
import {render} from 'react-dom';
import App from './app';
render(<App/>,document.getElementById("root"));
```

- build/webpack-client.config.js

处理客户端代码的打包逻辑

```
const path = require('path');
module.exports = {
  entry:{
    index:path.resolve(__dirname,'../src/index.js')
  },
```

```
  mode:'development',
  devtool: 'cheap-module-eval-source-map',
  output:{
    filename:'[name].js',
    path:path.resolve(__dirname,'../dist/client')
  },
  resolve:{
    alias:{
      '@':path.resolve(__dirname,'../src')
    },
    extensions:['.js']
  },
  module:{
    rules:[{
      test:/\.js$/,
      use:'babel-loader',
      exclude:/node_modules/
    }]
  }
}
```

- 添加客户端代码的打包脚本

```
"build:client": "webpack --config build/webpack-client.config.js --watch"
```

运行一下

```
npm run build:client
```

- server引用打包好的客户端资源

```
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);
```

- 再来测试一下

这时候发现我们的页面渲染没问题, 并且也能响应用户操作, 比如点击事件了.

6. hydrate

经过上面的5步, 看起来没问题了, 但是我们的控制台会输出一些warnning

```
Warning: render(): Calling ReactDOM.render() to hydrate server-rendered markup
will stop working in React v18. Replace the ReactDOM.render() call with
ReactDOM.hydrate() if you want React to attach to the server HTML.
```

ReactDOM.hydrate()和ReactDOM.render()的区别就是：

- ReactDOM.render()会将挂载dom节点的所有子节点全部清空掉，再重新生成子节点。
- ReactDOM.hydrate()则会复用挂载dom节点的子节点，并将其与react的virtualDom关联上。

也就是说ReactDOM.render()会将服务端做的工作全部推翻重做，而ReactDOM.hydrate()在服务端做的工作基础上再进行深入的操作.

所以我们修改一下客户端的入口文件src/index.js, 将render修改为hydrate

```
import React from 'react';
import { hydrate } from 'react-dom';
import App from './app';
hydrate(<App/>,document.getElementById("root"));
```

同构流程总结

1. 服务端根据React代码生成html
2. 客户端发起请求, 收到服务端发送的html, 进行解析和展示
3. 客户端加载js等资源文件
4. 客户端执行js文件, 完成hydrate操作
5. 客户端接管整体应用

路由

在客户端渲染时, React提供了BrowserRouter和HashRouter来供我们处理路由, 但是他们都依赖window对象, 而在服务端是没有window的。

但是react-router提供了StaticRouter, 为我们的服务端渲染做服务。

接下来我们模拟添加几个页面, 实现一下路由的功能。

1. 构造Login和User两个页面

- src/pages/login/index.js

```
import React from 'react';
export default class Login extends React.PureComponent{
  render(){
    return <div>登陆</div>
  }
}
```

- src/pages/user/index.js

```
import React from 'react';
export default class User extends React.PureComponent{
  render(){
    return <div>用户</div>
  }
}
```

2. 添加服务端路由

server.js

```
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);
```

3. 添加客户端路由

   src/index.js

```
import React from 'react';
import { hydrate } from 'react-dom';
import App from './app';
import { BrowserRouter as Router, Route } from 'react-router-dom';
import User from './pages/user';
import Login from './pages/login';

hydrate(
    <Router>
        <Route path="/" component={App}>
        <Route exact path="/user" component={User}></Route>
        <Route exact path="/login" component={Login}></Route>
    </Route>
  </Router>,
  document.getElementById("root")
);
```

- 分别访问一下/user和/login

  发现已经可以正常渲染了

路由同构

但是同学们应该已经发现了一个问题, 就是我们这种写法非常的费劲。

既要在客户端写一遍路由, 也要在服务端写一遍路由, 有没有什么方法能只写一遍? 就像app.js一样?

所以我们先找一下两端路由的异同：

- 共同点：路径和组件的映射关系是相同的
- 不同点：路由引用的组件不一样, 或者说实现的方式不一样

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。

- 新建src/pages/notFound/index.js

```
import React from 'react';

export default ()=> <div>404</div>
```

- 路由配置文件

src/router/routeConfig.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转换器

```
import React from 'react';
import { createBrowserHistory } from "history";
import {Route,Router,StaticRouter,Redirect,Switch} from 'react-router';
import routeConfig from  './routeConfig';

const routes = routeConfig.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>
  }
}
```

- 客户端入口

src/index.js

```
import React from 'react';
import { hydrate } from 'react-dom';
import App from './app';

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

- 客户端 app.js

src/app.js

```
import React from 'react';
import { createRouter } from './router'

class App extends React.PureComponent{
    render(){
        return createRouter('client')();
    }
};

export default App;
```

- 服务端入口

server.js

```
import express from 'express';
import React from 'react';
import {renderToString} from 'react-dom/server';
import { createRouter } from './src/router'

const app = express();
app.use(express.static("dist"))
app.get('*',function(req,res){
  const content = renderToString(createRouter('server')({location:req.url}) );
  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);
```

- 重定向问题

这里我们从/重定向到/user的时候, 可以看到html返回的内容和实现页面渲染的内容是不一样的。
这代表重定向操作是客户端来完成的, 而我们期望的是先访问index.html请求, 返回302, 然后出现一个新的user.html请求

https://v5.reactrouter.com/web/api/StaticRouter react提供了一种重定向的处理方式

```
import express from 'express';
import React from 'react';
```

```
import {renderToString} from 'react-dom/server';
import { createRouter } from './src/router'

const app = express();
app.use(express.static("dist"))
app.get('*',function(req,res){
  const context = {};
  const content = renderToString(createRouter('server')({location:req.url,
context}) );
  //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{
    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);
```

这时候我们再测试一下, 就会发现符合预期, 出现了两个请求, 一个302, 一个user.html

- 404问题

我们随便输入一个不存在的路由, 发现内容是如期返回了404, 但是请求确实200的, 这是不对的.

server.js

```
import express from 'express';
import React from 'react';
```

```
import {renderToString} from 'react-dom/server';
import { createRouter } from './src/router'

const app = express();
app.use(express.static("dist"))
app.get('*',function(req,res){
  const context = {};
  const content = renderToString(createRouter('server')({location:req.url,
context}) );
  //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);
```

routeConfig.js

```
import React from 'react';

//改造前
component:NotFound
//改造后
render:({staticContext})=>{
```

```
    if (staticContext) staticContext.NOT_FOUND = true;
    return <NotFound/>
}
```