五、服务端渲染

1、什么是服务端渲染

- 前端渲染：html页面作为静态文件存在，前端请求时后端不对该文件做任何内容上的修改，直接以资源的方式返回给前端，前端拿到页面后，根据写在html页面上的js代码，对该html的内容进行修改。（在浏览器端查看源代码看不到内容）
- 服务端渲染：前端发出请求后，后端在将HTML页面返回给前端之前，先把HTML页面中的特定区域，用数据填充好，再将完整的HTML返回给前端。（在浏览器端查看源代码可以看到文本内容）在SPA场景下，服务端渲染都是针对第一次get请求，它会完整的html给浏览器，浏览器直接渲染出首屏，用不着浏览器端多一个AJAX请求去获取数据再渲染。

2、前端渲染优势与劣势

优势： 前后端分离，前端专注UI，后端专注api开发 。

劣势: 通过前端渲染的流程，就可以明白其对应的劣势。下面说一下对应的执行流程：当我们向服务器发送一个请求，服务器接收到该请求后，会将一个html页面返回给浏览器，浏览器在渲染这个html页面的时候，发现有js文件，会向服务器请求该js文件，服务器将该js文件返回给浏览器，浏览器开始运行js文件，当然js文件中的内容都是我们写的react代码，当整个react代码执行完毕后，react会向页面渲染指定的元素。

通过这个过程的描述，可以发现，如果在访问一个网页的时候是前端渲染，需要经过的步骤比较多，所以前端渲染首屏（第一次看到页面上内容的时间）渲染慢，因为需要下载一些相应的js文件和css文件。

前端渲染的另外一个劣势是不利于SEO（搜索引擎优化）的优化。什么是SEO呢？例如，我们做了一个电商的网站，我们希望用户在使用百度等搜索引擎，在搜索有关电商方面的信息的时候能够搜索到我们做的电商网站，并且希望我们的网站在整个搜索结果的排名靠前，那么为了达到这个目录，我们可以通过一些技术的手段，来对网站进行优化，那么这个过程就叫做SEO的优化。那么，在对网站进行SEO优化的时候，有一个很重要的点就是，像百度等这些搜索引擎，它们只认识服务端渲染返回的内容。因为通过服务端渲染，具体的内容会写到html中，而百度这些搜索引擎的爬虫就可以获取这些html标签中的内容，而客户端渲染，具体的内容都是通过js来完成渲染的，而搜索引擎的爬虫是不认识js渲染的内容的。

所以说，为了让我们使用react编写的网页更容易被搜索引擎搜索到，并且有一个好的排名，就要用到服务端渲染。

3、服务端渲染优势与劣势

优势：

在前面我们讲解过，服务端渲染是指：服务端将整个html结构和相关的内容返回给浏览器，那么对应的内容都是写在html标签中的，所以非常有利于SEO . 同时由于所有的内容都返回给浏览器了，那么首屏加载的时间也会非常的快。

劣势：

更多的服务器端负载。

用户体验较差。

不容易维护，通常前端修改了css或者是html的内容，后端也需要修改

4、如何选择

建议：如果注重SEO的新闻站点，非强交互的页面，建议用SSR（server side render）；像后台管理页面这类强交互的应用，建议使用前端渲染。

5、实现React服务端渲染

5.1 在服务端构建React应用

在服务端构建React应用之前，先说一下React服务端渲染的流程：浏览器发送请求，然后服务器运行React的代码并生成对应的HTML页面，最后服务器将生成好的HTML页面返回给浏览器。这就是React服务端渲染的流程，通过这个流程的描述，我们需要在服务器端编写React代码，注意：以前都是在客户端编写，现在需要在服务端进行编写。关于服务端的实现，我们这里使用Node来完成。

node.js 官方网站： https://nodejs.org/en/

安装好node以后，可以新建一个目录，在该目录下执行：

```
npm init
```

进行项目的初始化，这时会在目录下面创建一个package.json文件。

同时执行如下命令，安装express框架，express框架是基于Node.js的Web开发框架。

```
npm install express --save
```

使用vscode打开项目后，在项目中新建一个src目录，在src目录下新建index.js文件，完成服务端代码编写

下面通过node编写一个简单的服务端程序。

```
var express=require('express')
var app=express();
app.get('/',function(req,res){
    res.send(`
    <html>
        <head>
            <title>Hello World</title>
            <body>Hello World</body>
        </head>
    </html>
    `)
})
var server=app.listen(3000);
```

服务端已经编写好了，并且经过测试没哟问题，下面开始在服务端编写React代码

react的安装：

```
npm install react --save
```

react-dom的安装

```
npm install react-dom
npm install --save-dev @babel/preset-react
```

在服务端编写React代码，并且将编写好的React代码在服务端进行渲染，在这里主要用到了renderToString方法，该方法的作用会将React元素生成一个html字符串返回给浏览器。具体的实现代码如下

src/index.js

```
var express=require('express')
var app=express();
const React=require('react');// 在node中导入react
const {renderToString} = require('react-dom/server');
const App=class extends React.Component{
    render(){
        return React.createElement('h3',null,'Hello World');
    }
}
app.get('/',function(req,res){
     // 然后调用createElement这个方法去生成一个React元素
    // 最后调用renderToString去根据这个React元素去生成一个html字符串并返回给了浏览器
    const content=renderToString(React.createElement(App));
    res.send(content);
    // res.send(`
    // <html>
    //     <head>
    //         <title>Hello World</title>
    //         <body>Hello World</body>
    //     </head>
    // </html>
    // `)
})
var server=app.listen(3000);
```

在上面的代码中，要注意杂node中导入react和react/dom需要用到的是require.

现在，已经完成了最基本的React服务端渲染的功能，但是目前还是有很多的问题，例如无法识别jsx语法，

只能commonjs这个模块化规范（Node中采用的规范），不能用esmodule （React中采用的规范）

下面对是否识别jsx语法来进行验证。

在src目录下创建components目录，在该目录下创建Home目录，然后在创建index.js文件，该文件中的代码

```
const React=require('react');
const Home=()=>{
    return <div>Home</div>
}
module.exports={default:Home}
```

然后在src目录下的index.js文件中，进行如下的修改

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

```
const React=require('react');// 在node中导入react
const {renderToString} = require('react-dom/server');
const Home =require('./components/Home/index')// 导入Home组件
const App=class extends React.Component{
    render(){
        return React.createElement('h3',null,'Hello World');
    }
}
app.get('/',function(req,res){

    // const content=renderToString(React.createElement(App));
    const content=renderToString(<Home/>)// 渲染Home组件
    res.send(content);
    // res.send(`
    // <html>
    //     <head>
    //         <title>Hello World</title>
    //         <body>Hello World</body>
    //     </head>
    // </html>
    // `)
})
var server=app.listen(3000);
```

运行上面的程序，发现出错了（node无法识别jsx语法）。那么应该怎样进行解决呢？需要配置webpack.通过webpack进行打包处理。

5.2 webpack配置

安装webpack

```
npm install --save-dev webpack
npm install --save-dev webpack-cli
```

安装完成后，在项目的根目录下(server-react)，创建webpack.server.js文件，完成相应的配置

webpack.server.js文件的配置信息如下

```
const path=require('path');
const nodeExternals=require('webpack-node-externals');
module.exports={
    // 表示打包的是服务端的代码，需要安装(npm install webpack-node-externals --save-dev)
    target:'node',
    // 表示编译开发环境中的代码
    mode: 'development',
    // 入口文件
    entry: './src/index.js',
    // 输出文件叫bundle.js
    // 并且指定输出的具体目录
    output: {
        filename:'bundle.js',
        // 指定存储的目录是根目录下的build目录
        path:path.resolve(__dirname,'build')
    },
    // 在node中通过require引用node_modules中的内容时
```

```
    //不将node_modules里面的包打进去
    externals: [nodeExternals()],
    // 配置相应的 规则
    module:{
        rules: [{
            test: /\.(js|jsx)?$/,
            // 需要安装babel-loader(npm install -D babel-loader @babel/core
@babel/preset-env webpack)
            // 如果是js文件需要babel-loader进行编译
            loader: 'babel-loader',
            // 如果js文件在node_modules目录下是不需要进行编译的。
            exclude: /node_modules/,
            options: {
                // 指定编译规则，需要对react进行编译
                // 需要安装babel-presets(npm install --save-dev @babel/preset-
react)
                presets: ["@babel/preset-env", "@babel/preset-react"]


            }

        }

        ]
    }

}
```

然后，在package.json文件，配置项目的启动与编译（为了方便webpack的打包与项目的启动）

```
  "name": "server-react",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
    // 配置项目的启动与编译
"scripts": {
    "start": "node ./build/bundle.js",
    "build":"webpack --config webpack.server.js"
  },
```

```
import React from 'react';
const Home=()=>{
    return <div>home</div>
}
export default Home;
```

修成成es6的规范（esmodule），通过import来导入组件。

修改src下的index.js文件代码如下

```
import express from 'express'
import Home from './components/Home/index'
import React from 'react'
import {renderToString} from 'react-dom/server'
var app=express();
app.get('/',function(req,res){
  // 渲染Home组件内容
    const content=renderToString(<Home/>)
    res.send(content);

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

首先输入：npm run build命令来进行编译

然后：npm run start 命令来进行运行（运行的是编译后的 build目录下的bundle.js文件）

在上面的代码中，可以修改一下渲染的内容，渲染出一个完整的html文档的结构

```
import express from 'express'
import Home from './components/Home/index'
import React from 'react'
import {renderToString} from 'react-dom/server'
var app=express();
app.get('/',function(req,res){
      const content=renderToString(<Home/>)
    res.send(`
        <html>
            <head>
                <title>服务端渲染</title>
            </head>
            <body>
             ${content}
            </body>
        </html>
    `);

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

修改完成后，重新输入npm run build命令来进行编译，然后再输入npm run start 命令来进行运行

通过上面的代码修改发现了一个问题，就是每次修改完代码后，需要重新编译打包然后在重新启动，这时才能看到修改后的代码，这样操作起来就非常的麻烦了。

5.3 自动打包与自动重启

下面配置一下webpack的自动启动与自动打包功能。

首先配置自动打包

在package.json文件中，进行如下的配置

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

在原有的 build后面加上了 --watch, 监视文件的变化，如果发现文件改变了，会自动重新打包编译。

下面测试一下，修改后的配置是否起作用：在命令行工具中:首先先输入一次：npm run build 进行打包

这时发现打完包后，并没有退出，而是一直在等待，这时我们修改了Home组件中内容，然后按下ctrl+s键进行保存，这时发现在命令行工具中，打包会自动进行。

那么下面要做的就是，打包自动完成后，需要启动服务器后 才能看到更新后的内容，但是问题是每次自己启动服务器，非常的麻烦，下面看一下怎样配置自动启动服务器。

这时需要安装 nodemon这个工具

```
npm install nodemon -g
```

安装完成后，可以再次修改package.json文件

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

在上面的配置中，完成了对start项的配置，通过nodemon检测build目录下的文件是否发生变化，如果发生了变化，通过node重新启动bundle.js文件。

所以整个配置可以这样理解：webpack检测到项目中的代码发生了变化后，重新进行打包，这时生成新的bundle.js文件，那么nodemon检测到了build这个目录下的bundle.js文件发生了变化，这时会重新启动服务器。

在进行运行时，需要开启两个命令窗口，一个输入npm run build检测编译的情况

另外一个输入npm run start

当检测到文件发生变化后，会自动重新编译，而重新编译完成后，服务器也会重新启动。

但是，我们发现每次这样启动另个窗口还是很麻烦，最好能够开始一个窗口，执行一个命令。这样更加的方便，

首先安装： npm-run-all

```
npm install npm-run-all --save-dev
```

然后修改package.json文件中的配置

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

这时在命令行中输入： npm run dev

表示并行执行以dev开头的命令，这时“dev:start"和"dev:build"都会执行。

这样既运行了打包编译的命令，有运行了服务器重新启动的命令。

5.4 同构处理

5.4.1 同构的概念

在讲解什么是同构的时候，我们先来看一个问题，将Home组件的内容修改成如下的形式：

```
import React from 'react';

const Home=()=>{

    return (<div>
        home, Hello World
        <button onClick={()=>{alert('Hello World')}}>单击</button>
    </div>)
}
export default Home;
```

我们在原来的内容基础上增加了一个按钮，并且增加添加了一个单击的事件，现在刷新页面，发现按钮能够正常的展示出来，但是当单击按钮的时候，发现对应的事件并没有被触发。

原因是： 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， 所以单击按钮没有任何的反映。

如何解决这个问题呢？再说解决方法之前，我们先讲一下“**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**”。 简单的理解就是： 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。

5.4.2 同构实现

静态资源文件处理

在具体的实现同构之前，先来看一下，对服务器对静态资源的一个处理。

回到src目录下的index.js 文件，对该文件中的代码做如下的处理：

```
import express from 'express'
import Home from './components/Home/index'
import React from 'react'
import {renderToString} from 'react-dom/server'
var app=express();
app.get('/',function(req,res){
    const content=renderToString(<Home/>)
    res.send(`
        <html>
            <head>
                <title>服务端渲染</title>
            </head>
            <body>
             ${content}
            </body>
            <script src="./test.js"></script>
        </html>
    `);

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

在这段代码中直接加入了一个