**本文档分成3个过程，最简结构搭建，融合vuex状态管理，完整的项目实现**

**1.最简结构搭建**

先使用 vue-cli初始化一个项目:

vue init webpack vue-ssr-dem，不要安装test模块，可能会报错

然后，

cd vue-ssr-demo

npm installnpm run dev

请确保创建的vue-cli单页应用可正常运行，接下来开始慢慢折腾吧~~

### **开始折腾**

#### **1. 首先安装 ssr 支持!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!**

npm install vue vue-server-renderer --save

 vue-server-renderer 与 vue 版本必须一致匹配，使用上面的语句安装vue和vue-server-renderer，参考官方文档

#### **在components目录下，修改HellowWorld页面，并新建test页面**

HelloWorld.vue页面

<template>

<div class="hello">

<h1>{{ msg }}</h1>

<h2>Essential Links</h2>

<ul>

<li>

<a href="https://vuejs.org" target="\_blank">

</a>

</li>

<router-link to="/test">test页面</router-link>

</ul>

</div>

</template>

<script>

export default {

name: 'HelloWorld',

data() {

return {

msg: 'Welcome to Your Vue.js App'

}

}

}

</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->

<style scoped>

h1,

h2 {

font-weight: normal;

}

ul {

list-style-type: none;

padding: 0;

}

li {

display: inline-block;

margin: 0 10px;

}

a {

color: #42b983;

}

</style>

test.vue页面

<template>

<div>

<div>

<router-link to="/">Home</router-link>

</div>

<div>创始人：{{founder}}</div>

</div>

</template>

<script>

export default {

data() {

return {

founder:"superzay"

}

},

}

</script>

#### **3. 在src目录下创建两个js文件:**

src

├── entry-client.js # 仅运行于浏览器

└── entry-server.js # 仅运行于服务器

#### **4. 修改router配置。**

无论什么系统路由总是最重要的，服务器端渲染自然也要公用一套路由系统，并且为了避免产生单例的影响，这里主要只为每一个请求都导出一个新的router实例:

import Vue from 'vue'

import Router from 'vue-router'

import HelloWorld from '@/components/HelloWorld'

Vue.use(Router)

export function createRouter () {

return new Router({

mode: 'history', *// 注意这里也是为history模式*

routes: [

{

path: '/',

name: 'Hello',

component: HelloWorld

}, {

path: '/test',

name: 'Test',

component: () => import('@/components/Test') *// 异步组件*

}

]

})

}

#### **5. 改造main.js（我们一般会定义成app.js）**

main.js初始化的只适合在浏览器的运行，所以要改造两端都可以使用的文件，同样为了避免产生单例的影响，这里将导出一个createApp的工厂函数:

import Vue from 'vue'

import App from './App.vue'

import { createRouter } from './router'

export function createApp () {

*// 创建 router 实例*

const router = new createRouter()

const app = new Vue({

*// 注入 router 到根 Vue 实例*

router,

render: h => h(App)

})

*// 返回 app 和 router*

return { app, router }

}

#### **6. entry-client.js加入以下内容：**

//entry-client.js

import { createApp } from './app.js'

//定义服务器/客户端环境

window.TERMINAL = 'client';

const {

app,

router,

} = createApp();

// 因为可能存在异步组件，所以等待router将所有异步组件加载完毕，服务器端配置也需要此操作

router.onReady(() => {

app.$mount('#app');

#### })

#### **7. entry-server.js**

*// entry-server.js*

import { createApp } from './app.js'

export default context => {

*// 因为有可能会是异步路由钩子函数或组件，所以我们将返回一个 Promise，*

*// 以便服务器能够等待所有的内容在渲染前，*

*// 就已经准备就绪。*

return new Promise((resolve, reject) => {

const { app, router } = createApp()

*// 设置服务器端 router 的位置*

router.push(context.url)

*// 等到 router 将可能的异步组件和钩子函数解析完*

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

*// 匹配不到的路由，执行 reject 函数，并返回 404*

if (!matchedComponents.length) {

*// eslint-disable-next-line*

return reject({ code: 404 })

}

*// Promise 应该 resolve 应用程序实例，以便它可以渲染*

resolve(app)

}, reject)

})

}

#### **8. webpack配置**

vue相关代码已处理完毕，接下来就需要对webpack打包配置进行修改了。

官网推荐下面这种配置：

build

├── webpack.base.conf.js # 基础通用配置

├── webpack.client.conf.js # 客户端打包配置

└── webpack.server.conf.js # 服务器端打包配置

但vue-cli初始化的配置文件也有三个：base、dev、prod ，我们依然保留这三个配置文件，只需要增加webpack.server.conf.js即可。

#### **9. webpack客户端的配置**

**webpack.base.conf.js配置**

修改entry入口配置为: ./src/entry-client.js。这样原 dev 配置与 prod 配置都不会受到影响。output输出选项的publicPath属性配置为：process.env.NODE\_ENV === 'production' ? config.build.assetsPublicPath : 'http://localhost:8080/'

服务器端的配置也会引用base配置，但会将entry通过merge覆盖为 server-entry.js。

我们会额外的在做一些处理，生成客户端构建清单client manifest

****好处:****

1. 在生成的文件名中有哈希时，可以取代 html-webpack-plugin 来注入正确的资源 URL。
2. 在通过 webpack 的按需代码分割特性渲染 bundle 时，我们可以确保对 chunk 进行最优化的资源预加载/数据预取，并且还可以将所需的异步 chunk 智能地注入为 <script> 标签，以避免客户端的瀑布式请求(waterfall request)，以及改善可交互时间(TTI - time-to-interactive)。

**webpack.prod.conf.js， webpack.dev.conf.js配置**

2个配置文件做一样的修改，引入一个插件，并配置到plugin中即可（）:

const VueSSRClientPlugin = require('vue-server-renderer/client-plugin')

*// ...*

*// ...*

plugins: [

*//先找到这条插件，增加第二条属性*

new webpack.DefinePlugin({

'process.env': env,

'process.env.VUE\_ENV': '"client"' *// 增加process.env.VUE\_ENV*

}),

*//...*

*// 另外需要将 prod 的HtmlWebpackPlugin 去除，因为我们有了vue-ssr-client-manifest.json之后，服务器端会帮我们做好这个工作。*

*// new HtmlWebpackPlugin({*

*// filename: config.build.index,*

*// template: 'index.html',*

*// inject: true,*

*// minify: {*

*// removeComments: true,*

*// collapseWhitespace: true,*

*// removeAttributeQuotes: true*

*// // more options:*

*// // https://github.com/kangax/html-minifier#options-quick-reference*

*// },*

*// // necessary to consistently work with multiple chunks via CommonsChunkPlugin*

*// chunksSortMode: 'dependency'*

*// }),*

*// 最后插入下面的插件，此插件在输出目录中， 生成 `vue-ssr-client-manifest.json`。*

new VueSSRClientPlugin()

]

*// ...*

#### **10. webpack服务器端**webpack.server.conf.js配置（可参考官方文档）

在build目录下创建webpack.server.conf.js文件

server的配置有用到新插件运行安装: npm i -D webpack-node-externals

webpack.server.conf.js配置如下：

const webpack = require('webpack')

const merge = require('webpack-merge')

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

const baseConfig = require('./webpack.base.conf.js')

const VueSSRServerPlugin = require('vue-server-renderer/server-plugin')

***//下面位置特别容易出错，特别注意，请根据自己项目情况处理***

*// 去除打包css的配置，此处去除的是同目录下的utils.js文件中cssLoader在loaders数组中的位置，本例在0位置*

baseConfig.module.rules[0].options = ''

module.exports = merge(baseConfig, {

*// 将 entry 指向应用程序的 server entry 文件*

entry: './src/entry-server.js',

*// 这允许 webpack 以 Node 适用方式(Node-appropriate fashion)处理动态导入(dynamic import)，*

*// 并且还会在编译 Vue 组件时，*

*// 告知 `vue-loader` 输送面向服务器代码(server-oriented code)。*

target: 'node',

*// 对 bundle renderer 提供 source map 支持*

devtool: 'source-map',

*// 此处告知 server bundle 使用 Node 风格导出模块(Node-style exports)*

output: {

libraryTarget: 'commonjs2'

},

*// https://webpack.js.org/configuration/externals/#function*

*// https://github.com/liady/webpack-node-externals*

*// 外置化应用程序依赖模块。可以使服务器构建速度更快，*

*// 并生成较小的 bundle 文件。*

externals: nodeExternals({

*// 不要外置化 webpack 需要处理的依赖模块。*

*// 你可以在这里添加更多的文件类型。例如，未处理 \*.vue 原始文件，*

*// 你还应该将修改 `global`（例如 polyfill）的依赖模块列入白名单*

whitelist: /\.css$/

}),

plugins: [

new webpack.DefinePlugin({

'process.env.NODE\_ENV': JSON.stringify(process.env.NODE\_ENV || 'development'),

'process.env.VUE\_ENV': '"server"'

}),

*// 这是将服务器的整个输出*

*// 构建为单个 JSON 文件的插件。*

*// 默认文件名为 `vue-ssr-server-bundle.json`*

new VueSSRServerPlugin()

]

})

****注意此处对baseConfig删除了一个属性****

baseConfig.module.rules[0].options = '' *// 去除分离css打包的插件*

#### **11. package.json配置，增加打包服务器端构建命令并修改原打包命令，去掉原来的“build”和‘dev’两条指令，加上下面的指令**

"scripts": {

"dev:client": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",

"dev:server":"node dev-server.js",

"build:client": "node build/build.js",

"build:server": "cross-env NODE\_ENV=production webpack --config build/webpack.server.conf.js --progress --hide-modules",

"build": "rimraf dist && npm run build:client && npm run build:server"

}

如果出现cross-env找不到，请安装npm i -D cross-env

#### **在根目录下创建dev-server.js文件**

先安装axios模块 npm install --save axios

//node服务器所需模块

const express = require('express');

const app = express();

const vueRenderer = require('vue-server-renderer')

const path = require('path');

//定义服务器/客户端环境

global.TERMINAL='server';

//weback.compiler所需模块

const serverConf = require('./build/webpack.server.conf');

const webpack = require('webpack')

const fs = require('fs')

const Mfs = require('memory-fs')

const axios = require('axios')

function compilerServer(cb) {

//使用webpack方法，传递webpack.server.conf配置，生成编译器（或者叫打包器）

const webpackComplier = webpack(serverConf);

//编译器输出目录设置到内存中

var mfs = new Mfs();

webpackComplier.outputFileSystem = mfs;

//启动编译器，开始打包

webpackComplier.watch({}, async(error, stats) => {

if(error) return console.log(error);

stats = stats.toJson();

stats.errors.forEach(err => console.log(err))

stats.warnings.forEach(err => console.log(err))

// server Bundle 对应的json文件的路径

let serverBundlePath = path.join(

serverConf.output.path,

'vue-ssr-server-bundle.json'

)

//读取server Bundle 对应的json文件

let serverBundle = JSON.parse(mfs.readFileSync(serverBundlePath, "utf-8"))

//读取 client Bundle 对应的 json文件

let clientBundle = await axios.get('http://localhost:8080/vue-ssr-client-manifest.json')

// 读取模板

let template = fs.readFileSync(path.join(\_\_dirname, './src/index-template.html'), 'utf-8');

//传递给回调

cb(serverBundle, clientBundle, template)

})

}

app.get('\*', async(req, res) => {

res.status(200);

res.setHeader('Content-Type', 'text/html;charset=utf-8;')

//调用打包服务函数，并传递进回调函数，处理打包结果

compilerServer(function(serverBundle, clientBundle, template) {

//创建bundleRenderer渲染器，传递进serverBundle, clientBundle, template

let bundleRenderer = vueRenderer.createBundleRenderer(serverBundle, {

template,

clientManifest: clientBundle.data,

runInNewContext: false

})

//开始渲染，

//renderToString方法可以接受一个回调函数为第二个参数（参考server.js）

//或者返回一个promise对象，使用then回调

/\*bundleRenderer.renderToString({

url: req.url

}).then((html) => {

res.end(html)

}).catch(err => console.log(err))\*/

//serverBundle可以等价理解为从entry-server.js开始的一个很大的脚本，

//他会在bundleRenderer.renderToString函数中执行，执行抛出的错误会传递给下面回调函数中的err

bundleRenderer.renderToString({

url: req.url

}, (err, html) => {

if(err) {

//定义错误处理逻辑

}

res.end(html)

})

})

})

//启动服务器，监听5000端口

app.listen(5000, () => {

console.log('启动成功')

})

#### **13. 在根目录下创建server.js文件**

const express = require('express')

const app = new express()

const fs = require('fs')

const path = require('path')

const vueRenderer = require('vue-server-renderer')

//定义服务器/客户端环境

global.TERMINAL='server';

//获取客户端和服务端打包文件经过处理生成的2个json文件

//此处我们使用vueRenderer.createBundleRenderer()方法生成的bundleRenderer渲染器

//正常如果使用vueRenderer.createRenderer()方法生成的render渲染器，则只需要服务端原始打包文件

//然后构造出的只是路由对应的html结构，而非整个页面。完整的html页面还需要自己手动构造，比较麻烦，参考官方文档和相关视频

const serverBundle = require('./dist/vue-ssr-server-bundle.json')

const clientManifest = require('./dist/vue-ssr-client-manifest.json')

const template = fs.readFileSync(path.resolve(\_\_dirname, './src/index-template.html'), 'utf-8')

// 创建渲染器

const bundleRenderer = vueRenderer.createBundleRenderer(serverBundle, {

runInNewContext: false, // 推荐

template: template, // 模板html文件

clientManifest: clientManifest, // client manifest

})

//定义静态资源目录,实际部署时，进行相应调整

app.use('/static', express.static(\_\_dirname + '/dist/static'));

//定义路由响应

app.get('\*', async(req, res) => {

let context={url:req.url}

//bundleRenderer.renderToString（）方法调用后，会执行severBundle的逻辑

//serverBundle抛出的错误会传递给下面回调函数中的err

//传递进路由信息，使用渲染器生成最终html文件，

//renderToString函数也可以返回一个promise对象，使用then回调的写法

//serverBundle可以等价理解为从entry-server.js开始的一个很大的脚本，

//他会在bundleRenderer.renderToString函数中执行，执行抛出的错误会传递给下面回调函数中的err

bundleRenderer.renderToString(context,(err,html)=>{

if(err){

//根据实际需求定义

res.end('出错了')

}else{

res.end(html)

}

})

});

app.listen(80, () => {

console.log('启动成功,监听端口80')

})

#### **14.修改index.html**

#### 这里我们一般重命名为index-template.html，然后修改内容为如下，然后将文件移动到src目录下

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<title>vue-cli-ssr</title>

</head>

<body>

*<!--vue-ssr-outlet-->*

</body>

</html>

原来的<div id="app">删掉，只在 body 中保留一个标记即可:<!--vue-ssr-outlet-->。 服务器端会在这个标记的位置自动生成一个<div id="app" data-server-rendered="true">，客户端会通过app.$mount('#app')挂载到服务端生成的元素上，并变为响应式的。

注意一下，此处将模板 html 修改为服务端渲染适用的模板了，但项目中的 dev 模式也适用的这个模板，但会因为找不到#app到报错，可以这样处理一下:

1. 最简单的办法，为dev模式单独建立一个 html 模板。。。
2. 为dev模式也集成服务端渲染模式（就是把webpack.dev.conf.js做和webpack.prod.conf.js一样的调整），这样无论生产环境与开发环境共同处于服务端渲染模式下也是相当靠谱的一件事。(官方例子是这样操作的)

#### **15.运行服务**

生产环境运行服务，先执行npm run build 构建生产环境文件

开发环境运行服务，直接在项目跟目录，执行npm run dev:client，然后在开启一个命令行执行npm run dev:server启动服务，访问localhost:5000

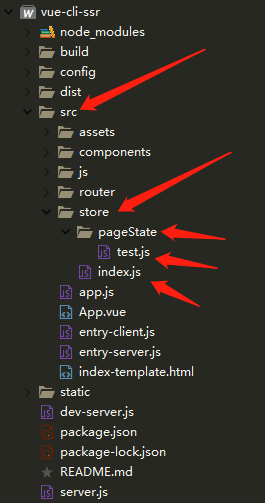
启动服务前，先修改node\_modules\vue-loader\lib\template-compiler\index.js

//将以下代码  
if (!isProduction) {  
  code = prettier.format(code, { semi: false, parser: 'babylon' })  
}  
//修改为：  
if (!isProduction) {  
  code = prettier.format(code, { semi: false, parser: 'babel' })

}

**2.融合 vuex状态管理**

在根目录下的src目录中定义store目录，在store目录中定义，pageState目录和index.js文件，pageState目录中定义test.js文件



安装vuex模块，npm install vuex --save

store/index.js，输出的是一个工厂函数，而不是对象

// index.js

import Vue from 'vue'

import Vuex from 'vuex'

import test from './pageState/test.js'

Vue.use(Vuex)

export function createStore () {

return new Vuex.Store({

modules:{

test,

}

})

}

store/pageState/test.js，注册模块state时，我们采用类似组件data的形式，使用一个函数返回一个对象（但经测试好像直接定义对象也没大问题，了解即可），actions下的getInitData函数必须要返回一个promise实例，我们的所有并发请求和可能的顺序请求应该全部在 new Promise()的参数函数中进行

//test.js

//正常我们使用对axios二次封装的插件，这里直接使用axios

//import { apiPost } from '../../js/http.js'

import axios from 'axios'

export default {

namespaced: true,

state() {

return {

founder: '010'

}

},

getters: {

},

mutations: {

updateFounder(state, arg, rootState) {

state.founder = arg;

}

},

actions: {

getInitData(store, arg) {

return new Promise((resolve, reject) => {

/\*axios.post('http://47.110.129.207/1.php', {

data: {}

}).then((rs) => {

store.commit('updateFounder', rs.data + '-' + new Date().getTime());

resolve()

}, (err) => {

reject();

})\*/

//如果你没有准备好接口，可以用一个定时器模拟上面的请求

setTimeout(() => {

store.commit('updateFounder', 'superzay' + '-' + new Date().getTime());

resolve()

}, 100)

})

},

},

}

1. **改造components目录下的test.vue文件**

加入asyncData函数，函数接收一个对象，对象要含有vuex根实例store和当前路由2条数据（当然你可以传递更多的信息进来），执行store.dispache()函数，触发相应页面的getInitData方法，这里要注意store.dispatche()方法返回值是一个promise对象（且该promise对象状态的改变，与触发的acitions下的方法有一定关系，参考vue--vuex相关笔记，和vuex官方文档）

<template>

<div>

<div>

<router-link to="/">Home</router-link>

</div>

<div>创始人：{{founder}}</div>

</div>

</template>

<script>

export default {

asyncData(obj) {

let store = obj.store;

let router = obj.router;

let rootVm= obj.rootVm;

let arg= {

rootVm

};

// 触发 action 后，会返回 Promise

return store.dispatch('test/getInitData', arg)

},

data() {

return {}

},

computed:{

founder(){

return this.$store.state.test.founder

}

}

}

</script>

1. **改造src目录下的app.js，entry-client.js，entry-server.js三个文件**

先安装模块 npm install --save vuex-router-sync

app.js文件会使用上面的模块，同时还会添加一个全局混合，为每个页面组件添加mounted钩子，在钩子中判断当前页面渲染是否是前端接管后的第一次渲染（如果是，直接取服务端返回的并且已经注入到store中的数据，注入在entry-client.js页面执行），以及当前页面是否需要获取初始数据（即是否有asyncData函数），并且还要在创建vue实例的工厂函数中注册store,且给sotre和router注册当前路由（同步处理）

//app.js

import Vue from 'vue'

import App from './App.vue'

import { createRouter } from './router'

import { createStore } from './store/index.js'

import { sync } from 'vuex-router-sync'

//全局混合

Vue.mixin({

mounted() {

if(!window.\_\_INITIAL\_STATE\_\_ && this.$options.asyncData) {

this.$options.asyncData({

store: this.$store,

router: this.$router,

rootVm: this.$root,

})

}

}

})

export function createApp() {

// 创建 router 实例

const router = createRouter()

const store = createStore();

sync(store, router)

const app = new Vue({

// 注入 router 到根 Vue 实例

router,

store,

render: h => h(App)

})

// 返回 app 和 router

return {

app,

router,

store

}

}

entry-client.js中会注入服务端返会的state（如果有的话），同时在前端接管完成后，清除window.\_\_INITIAL\_STATE\_\_

//entry-client.js

import { createApp } from './app.js'

//定义服务器/客户端环境

window.TERMINAL = 'client';

const {

app,

router,

store

} = createApp();

if(window.\_\_INITIAL\_STATE\_\_) {

store.replaceState(window.\_\_INITIAL\_STATE\_\_)

}

// 因为可能存在异步组件，所以等待router将所有异步组件加载完毕，服务器端配置也需要此操作

router.onReady(() => {

app.$mount('#app');

window.\_\_INITIAL\_STATE\_\_ = undefined;

})

entry-server.js主要是需要在路由对应的组件全部加载完成，且组件对应的初始数据全部获取完成后，再进行渲染

// entry-server.js

import { createApp } from './app.js'

export default context => {

// 因为有可能会是异步路由钩子函数或组件，所以我们将返回一个 Promise，

// 以便服务器能够等待所有的内容在渲染前，

// 就已经准备就绪。

return new Promise((resolve, reject) => {

const {

app,

router,

store

} = createApp()

// 设置服务器端 router 的位置

router.push(context.url)

// 等到 router 将可能的异步组件和钩子函数解析完

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

// 匹配不到的路由，执行 reject 函数，并返回 404

if(!matchedComponents.length) {

// eslint-disable-next-line

return reject({

code: 404

})

}

Promise.all(matchedComponents.map((Component) => {

if(Component.asyncData) {

return Component.asyncData({

store,

router,

rootVm: app,

})

}

})).then(() => {

// 在所有预取钩子(preFetch hook) resolve 后，

// 我们的 store 现在已经填充入渲染应用程序所需的状态。

// 当我们将状态附加到上下文，

// 并且 `template` 选项用于 renderer 时，

// 状态将自动序列化为 `window.\_\_INITIAL\_STATE\_\_`，并注入 HTML。

context.state = store.state

resolve(app)

},()=>{

reject({

code:999,

message:'路径组件，数据拉取失败'

});

})

}, reject)

})

}

重新运行项目查看结果

1. **完整的项目实现**

**这里以cookie/session模式为基础，实现一个完整项目**

先在src目录下添加，js目录，并创建conifg.js和http.js文件，并在app.js中引用

Config.js文件，开发环境使用域名spzay.51vip.biz，开发环境下，自行为本地ssr服务器的主机配置一个域名（修改本地电脑的hosts文件，给ip：127.0.0.1配置一个域名，这里采用dev.51vip.biz，hosts文件的修改不懂的请百度），注意线上演示项目没有采用域名形式

//基础配置文件config.js

let ENV = process.env.NODE\_ENV;

let \_BASE\_URL = '';

let BASE\_URL = '';

//本例开发环境和生产环境采用相同的地址

if(ENV === 'dev') {

\_BASE\_URL = 'http://spzay.51vip.biz:80';

} else {

\_BASE\_URL = 'http://spzay.51vip.biz:80';

}

BASE\_URL = \_BASE\_URL;

export {

BASE\_URL

}

http.js文件，

1,封装的接口挂载在vue原型下，所有公共的配置等等也会挂载到vue原型下，接口中使用的基础配置路径会通过this.BASE\_URL来引用，所以调用接口需要以vue实例的形式（子实例或者根实例），比如rootVm.apiPost()

2,本封装是为cookie/session验证模式做的，发起请求时cookie会自动携带（首先要保证ssr服务器和api服务器使用域名形式访问，且二者不同域但共父域），浏览器端发起http请求，如果出现跨域，ajax请求需要配置withCredentials为true，同时后台小伙伴也要做有关配置。服务器端没有跨域一说，但是要自己手动添加cookie到请求头中，在app.js中会判断终端环境，如果是服务器端，会把携带过来的cookie挂载到vue原型下

3，案例使用的本人的阿里云比较简陋，页面可能要多刷新几次才能正常，有条件的同学可以使用自己的资源

//http.js文件，自定义封装请求接口，

//此处只是demo，根据自己项目实际情况修改

import Axios from 'axios'

//如果网络是正常的，一律走promise实例的成功回调，具体的业务上的问题(失败或者成功)，在一个对象中指明,成功回调只接收对象和undefined两种值，undefined表示不去执行逻辑

//如果网络故障，则走promise实例的失败回调

//post请求方法

const apiPost = function(url, config) {

if(config === undefined) {

config = {}

} else if(typeof config !== 'object') {

throw '配置对象类型错误，请输入object'

}

config.url = this.BASE\_URL + url; //定义路径

config.method = 'post'; //定义方法类型

config.timeout = config.timeout || 10000; //定义超时

config.withCredentials = true; //携带cookie

//设置content-type

config.headers = {

'Content-Type': 'application/json'

}

//数据格式处理

config.transformRequest = [function(data) {

return JSON.stringify(data);

}];

//为服务器端添加cookie

if(TERMINAL === 'server') {

config.headers['Cookie'] = this.cookie===undefined?'':this.cookie;

}

//此处返回的是then方法创建的promise实例

return Axios(config).then((rs) => {

//这里，rs.status状态码在2xx范围，走成功回调，其他走失败回到

//rs.data为接口返回的业务数据，我们一般只返回出去业务数据（即rs.data），不会把整个rs返回

if(rs.data.code === 401) {

//如果token过期，重新登录

if(TERMINAL === 'server') {

return Promise.reject({

code: 401,

message: '未登录或登录失效，请重新登录'

});

} else {

//当前终端为客户端

// xxxxxx 重新登录逻辑

//向外返回成功promise,参数undefined

return Promise.resolve(undefined)

}

} else {

return Promise.resolve(rs.data)

}

}, (err) => {

//返回失败状态和数据

console.log('ajax出错啦');

return Promise.reject(err);

})

}

export {

apiPost

}

App.js改造

主要是引用一些js文件，并将相关变量挂载到vue原型下，如果终端是服务器端，再把页面请求时带过来的cookie也挂载到原型下

//app.js

import Vue from 'vue'

import App from './App.vue'

import { createRouter } from './router'

import { createStore } from './store/index.js'

import { sync } from 'vuex-router-sync'

import {BASE\_URL} from './js/config.js'

import {apiPost} from './js/http.js'

//vue原型挂载

Vue.prototype.BASE\_URL=BASE\_URL;

Vue.prototype.apiPost=apiPost;

//全局混合

Vue.mixin({

mounted() {

if(!window.\_\_INITIAL\_STATE\_\_ && this.$options.asyncData) {

this.$options.asyncData({

store: this.$store,

router: this.$router,

rootVm:this.$root,

})

}

}

})

export function createApp(cookie) {

//如果是服务器端，挂载cookie

if(TERMINAL==='server'){

Vue.prototype.cookie=cookie

}

// 创建 router 实例

const router = createRouter()

const store = createStore();

sync(store, router)

const app = new Vue({

// 注入 router 到根 Vue 实例

router,

store,

render: h => h(App)

})

// 返回 app 和 router

return {

app,

router,

store

}

}

test.js改造

//test.js

export default {

namespaced: true,

state() {

return {

founder: '010'

}

},

getters: {

},

mutations: {

updateFounder(state, arg, rootState) {

state.founder = arg;

}

},

actions: {

getInitData(store, arg) {

return new Promise((resolve, reject) => {

arg.rootVm.apiPost('/1-withCredentials.php', {

data: {}

}).then((rs) => {

let token = rs.data.token === "" ? "无" : rs.data.token;

store.commit('updateFounder', rs.data.founder + '--------cookie\_token:' + token + '--------' + new Date().getTime());

resolve()

}, (err) => {

reject();

})

})

},

},

}

修改dev-server.js，server.js文件和entry-server.js文件，

2个server文件主要是添加cookie模块，然后解析cookie，并向下传递，entery-server.js文件在接收到cookie后在传递给app.js文件

先安装cookie-parser模块：npm install --save cookie-parser

dev-server.js

//node服务器所需模块

const express = require('express');

var cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());

const vueRenderer = require('vue-server-renderer')

const path = require('path');

//定义服务器/客户端环境

global.TERMINAL = 'server';

//weback.compiler所需模块

const serverConf = require('./build/webpack.server.conf');

const webpack = require('webpack')

const fs = require('fs')

const Mfs = require('memory-fs')

const axios = require('axios')

function compilerServer(cb) {

//使用webpack方法，传递webpack.server.conf配置，生成编译器（或者叫打包器）

const webpackComplier = webpack(serverConf);

//编译器输出目录设置到内存中

var mfs = new Mfs();

webpackComplier.outputFileSystem = mfs;

//启动编译器，开始打包

webpackComplier.watch({}, async(error, stats) => {

if(error) return console.log(error);

stats = stats.toJson();

stats.errors.forEach(err => console.log(err))

stats.warnings.forEach(err => console.log(err))

// server Bundle 对应的json文件的路径

let serverBundlePath = path.join(

serverConf.output.path,

'vue-ssr-server-bundle.json'

)

//读取server Bundle 对应的json文件

let serverBundle = JSON.parse(mfs.readFileSync(serverBundlePath, "utf-8"))

//读取 client Bundle 对应的 json文件

let clientBundle = await axios.get('http://localhost:8080/vue-ssr-client-manifest.json')

// 读取模板

let template = fs.readFileSync(path.join(\_\_dirname, './src/index-template.html'), 'utf-8');

//传递给回调

cb(serverBundle, clientBundle, template)

})

}

app.get('\*', async(req, res) => {

res.status(200);

res.setHeader('Content-Type', 'text/html;charset=utf-8;')

//调用打包服务函数，并传递进回调函数，处理打包结果

compilerServer(function(serverBundle, clientBundle, template) {

//创建bundleRenderer渲染器，传递进serverBundle, clientBundle, template

let bundleRenderer = vueRenderer.createBundleRenderer(serverBundle, {

template,

clientManifest: clientBundle.data,

runInNewContext: false

})

//开始渲染，

//renderToString方法可以接受一个回调函数为第二个参数（参考server.js）

//或者返回一个promise对象，使用then回调

/\*bundleRenderer.renderToString({

url: req.url

}).then((html) => {

res.end(html)

}).catch(err => console.log(err))\*/

//serverBundle可以等价理解为从entry-server.js开始的一个很大的脚本，

//他会在bundleRenderer.renderToString函数中执行，执行抛出的错误会传递给下面回调函数中的err

let cookie='';

Object.keys(req.cookies).forEach((item,index)=>{

cookie=cookie+item+'='+req.cookies[item]+';';

});

bundleRenderer.renderToString({

url: req.url,

cookie:cookie,//向entry-server.js模块传递cookie,最终会传递给app.js模块，并挂载到vue原型下

}, (err, html) => {

if(err) {

//定义错误处理逻辑

}

res.end(html)

})

})

})

//启动服务器，监听5000端口

app.listen(5000, () => {

console.log('启动成功')

})

server.js

const express = require('express');

var cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());

const fs = require('fs')

const path = require('path')

const vueRenderer = require('vue-server-renderer')

//定义服务器/客户端环境

global.TERMINAL = 'server';

//获取客户端和服务端打包文件经过处理生成的2个json文件

//此处我们使用vueRenderer.createBundleRenderer()方法生成的bundleRenderer渲染器

//正常如果使用vueRenderer.createRenderer()方法生成的render渲染器，则只需要服务端原始打包文件

//然后构造出的只是路由对应的html结构，而非整个页面。完整的html页面还需要自己手动构造，比较麻烦，参考官方文档和相关视频

const serverBundle = require('./dist/vue-ssr-server-bundle.json')

const clientManifest = require('./dist/vue-ssr-client-manifest.json')

const template = fs.readFileSync(path.resolve(\_\_dirname, './src/index-template.html'), 'utf-8')

// 创建渲染器

const bundleRenderer = vueRenderer.createBundleRenderer(serverBundle, {

runInNewContext: false, // 推荐

template: template, // 模板html文件

clientManifest: clientManifest, // client manifest

})

//定义静态资源目录,实际部署时，进行相应调整

app.use('/static', express.static(\_\_dirname + '/dist/static'));

//定义路由响应

app.get('\*', async(req, res) => {

let cookie = '';

Object.keys(req.cookies).forEach((item, index) => {

cookie = cookie + item + '=' + req.cookies[item] + ';';

});

let context = {

url: req.url,

cookie: cookie //向entry-server.js模块传递cookie,最终会传递给app.js模块，并挂载到vue原型下

}

//bundleRenderer.renderToString（）方法调用后，会执行severBundle的逻辑

//serverBundle抛出的错误会传递给下面回调函数中的err

//传递进路由信息，使用渲染器生成最终html文件，

//renderToString函数也可以返回一个promise对象，使用then回调的写法

//serverBundle可以等价理解为从entry-server.js开始的一个很大的脚本，

//他会在bundleRenderer.renderToString函数中执行，执行抛出的错误会传递给下面回调函数中的err

bundleRenderer.renderToString(context, (err, html) => {

if(err) {

//根据实际需求定义

res.end('出错了')

} else {

res.end(html)

}

})

});

app.listen(80, () => {

console.log('启动成功,监听端口80')

})

entry-server.js

// entry-server.js

import { createApp } from './app.js'

export default context => {

// 因为有可能会是异步路由钩子函数或组件，所以我们将返回一个 Promise，

// 以便服务器能够等待所有的内容在渲染前，

// 就已经准备就绪。

return new Promise((resolve, reject) => {

const {

app,

router,

store

} = createApp(context.cookie)

// 设置服务器端 router 的位置

router.push(context.url)

// 等到 router 将可能的异步组件和钩子函数解析完

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

// 匹配不到的路由，执行 reject 函数，并返回 404

if(!matchedComponents.length) {

// eslint-disable-next-line

return reject({

code: 404

})

}

Promise.all(matchedComponents.map((Component) => {

if(Component.asyncData) {

return Component.asyncData({

store,

router,

rootVm:app,

})

}

})).then(() => {

// 在所有预取钩子(preFetch hook) resolve 后，

// 我们的 store 现在已经填充入渲染应用程序所需的状态。

// 当我们将状态附加到上下文，

// 并且 `template` 选项用于 renderer 时，

// 状态将自动序列化为 `window.\_\_INITIAL\_STATE\_\_`，并注入 HTML。

context.state = store.state

resolve(app)

}, () => {

reject({

code: 999,

message: '路径组件，数据拉取失败'

});

})

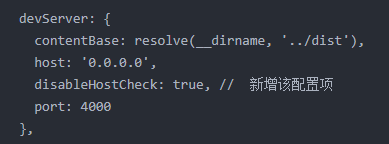
}, reject)

})

}

webpack.dev.conf.js修改

在devServer属性下，添加disableHostCheck: true,如下图所示



默认为false,这是webpack本身出于安全考虑，因为不检查主机的应用程序容易受到DNS重新绑定攻击。但是，在我们的开发环境下，可以禁用掉disableHostCheck这一配置项。

Ok至此一个完整的例子结束，重新运行项目

注意：请求页面时可以多刷新几次，api服务器不太稳定，访问ssr服务器请使用http://dev.51vip.biz:5000路径，当页面请求完成后，打开浏览器开发者工具，在application下，找到cookies部分，然后创建一条cooke,键名取token,值取sdf324sdfe32（可以自定义），然后域名取.51vip.biz（这里不能错），其他默认即可，然后再刷新页面，或者切换几次页面，就可以看到cookie已经携带上了