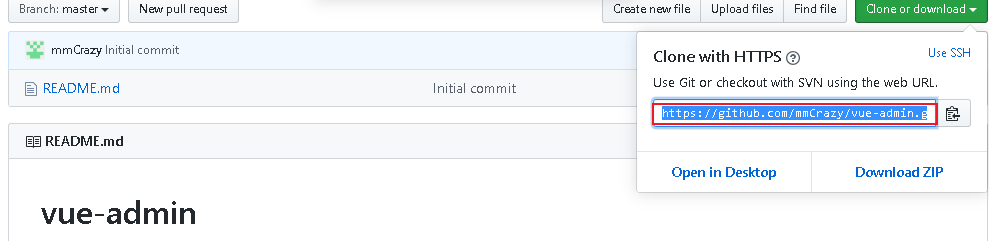
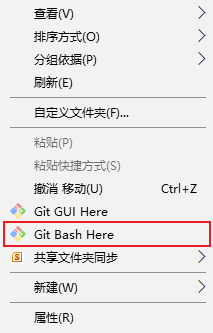
首先安装node和git,这里这两个安装就自行百度吧

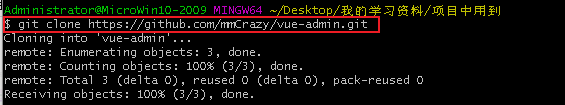
然后在github上创建新项目下面以vue-admin作为项目名称为例，复制项目地址



然后在你所要存放项目的文件夹中空白处右键，就会看到git这个选项并点击



运行git后，然后输入克隆命令git clone +项目地址，在git中粘贴使用shift + insert 键



在编程软件打开项目所在的文件夹，笔者使用的是vscode，打开终端ctrl + ~，使用npm –v查看node版本



然后安装vue cli，安装命令npm install –g @vue/cli，此处安装的是4.0版本，如之前安装了旧版本可以先卸载，使用命令npm uninstall vue-cli –g，如果网速慢可以安装淘宝镜像npm install –g cnpm --registry=https://registry.npm.taobao.org（安装淘宝镜像后命令的npm需改cnpm）



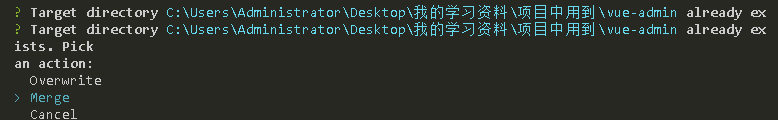
查看vue-cli版本使用命令vue -V



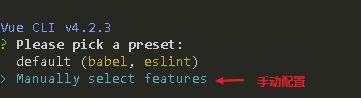
使用命令vue create + 文件夹名 创建文件夹，复用原有文件夹vue-admin，vue create vue-admin（这里可以使用vue ui图形化创建）



Merge复用原有文件夹中创建文件

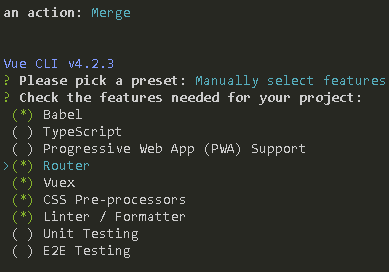


手动配置项目

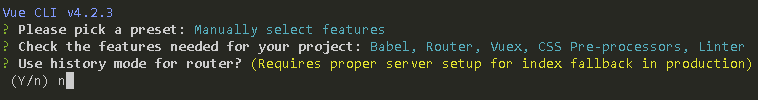


这里手动选择功能，按↑↓，按空格键为选中你需要的

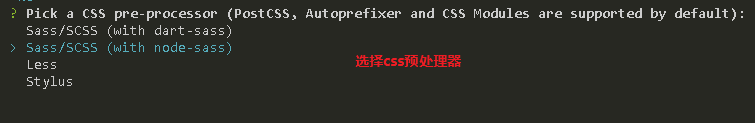




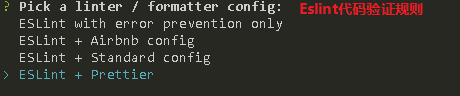
回车后，是否使用history，知道vue的路由有两种模式history和hash，这里不使用，使用hash



这里css预处理器选择Sass/SCSS（with node-sass）



Eslint代码验证规则这里使用Eslint + Prettier



这里使用保存就进行检测

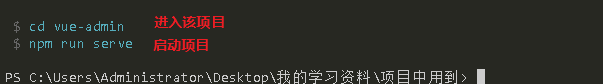


这里选择独立文件放置



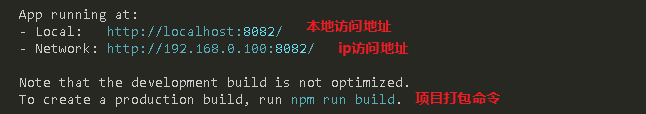
是否保存当前配置，这里不保存N







此处按住ctrl点击访问地址即可访问



git的使用

下面提一下使用git提交代码到github，由于上面进行的一次克隆，就不用再次初始化了（git init），

然后是将所有文件添加到仓库 git add .

将add的文件commit到仓库 git commit -m "注释语句"

将本地的仓库关联到github上 git remote add origin +(仓库地址<https://github.com/mmCrazy/vue-admin.git>)

上传代码到github远程仓库 git push -u origin master

中间可能会让你输入Username和Password，你只要输入github的账号和密码就行了

接下来创建vue.config.js

因为vue.config.js需要我们自己手动配置创建，然后这个文件的配置上网搜一下就有了，下面是目录的简介



然后我们开始login的登录页面的制作，首先我们在views的vue文件添加login.vue文件，其他的文件，然后将router中的路由改掉，使用redirect修改默认页面，然后下面使用箭头函数引入login文件（此处可以在头部直接用import引入，然后在component处的箭头函数修改为Login即可）



然后我们进行一个css初始化样式，在src文件夹下创建一个style文件，将初始化的scss放入style文件夹中，在vue.config.js中引入的scss文件,我们将所有的样式都引入到main.scss文件里，包括我们的初始化css文件引入方式 ： @import "./normalize.scss";

// css预设器配置项

    loaderOptions: {

      scss:{

        prependData: `@import "./src/style/main";`

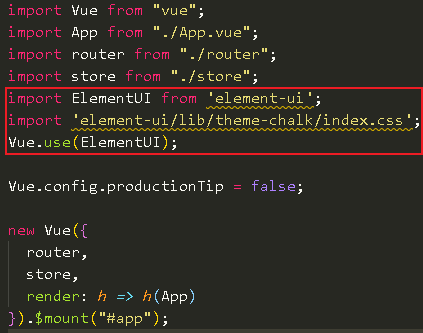
      }

    },

Element UI的使用

然后我们开始制作login组件登录页面了，此项目使用的是Element UI来写，我们要安装Element UI的依赖和在main.js入口文件中进行全局引入（见Element UI官网）

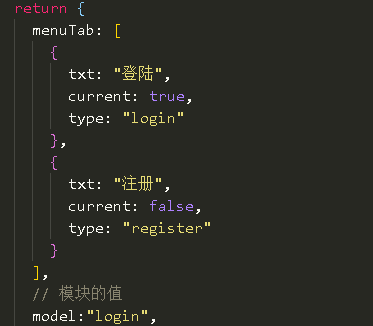
安装Element UI依赖: npm i element-ui -S

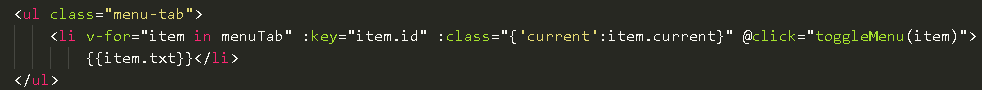


此时的App.vue文件代码如下，对此文件进行一个初始化，修改去掉一些无关代码，剩下我们的路由显示标签



下面进行登录、注册的表单制作与相关校验，首先在data方法里面的return中添加一个menuTab数组包含登录和注册的相关数据和添加一个model的模块值用来切换是否显示注册时候的确认密码的输入框，在输入密码的那个输入框使用v-if控制，下面遍历出两个li标签，其他的数据是添加点击事件





在Element UI中的From表单中的“自定义校验规则”的代码拷贝进login文件在，对Element UI的代码进行修改，也就是以下这几个命名函数，还有再修改一些相关的代码,使用正则进行输入的格式的校验（注意获取验证码的标签如果是绑定了v-model.number的把number去掉否则如果第一个输入的是数字就无法输入字母了）

// 输入用户名 ----验证邮箱正则验证

*var* validateUsername = (*rule*, *value*, *callback*) *=>* {

*let* reg = /\w[-\w.+]\*@([A-Za-z0-9][-A-Za-z0-9]+\.)+[A-Za-z]{2,14}/;

      if (value === "") {

        callback(new Error("请输入用户名"));

      } else if (!reg.test(value)) {

        callback(new Error("请输入正确的用户名"));

      } else {

        callback();

      }

    };

    //输入密码--- 验证密码正则验证

*var* validatePassword = (*rule*, *value*, *callback*) *=>* {

*let* reg = /^(?=.\*[0-9])(?=.\*[a-zA-Z])(.{8,})$/;

      if (value === "") {

        callback(new Error("请输入密码"));

      } else if (!reg.test(value)) {

        callback(new Error("输入6-20位包含数字、字母的密码"));

      } else {

        callback();

      }

    };

    //输入重复密码----验证重复密码

*var* validatePasswords = (*rule*, *value*, *callback*) *=>* {

      if (value === "") {

        callback(new Error("请再次输入密码"));

      } else if (value != this.ruleForm.password) {

        callback(new Error("输入密码不一致"));

      } else {

        callback();

      }

    };

    //输入验证码----验证码正则验证

*var* checkCode = (*rule*, *value*, *callback*) *=>* {

*let* reg = /^[a-z0-9]{6}$/;

      if (value === "") {

        callback(new Error("请输入验证码"));

      } else if (!reg.test(value)) {

        callback(new Error("请输入正确的验证码"));

      } else {

        callback();

      }

};

 // 表单数据  ---开始

      ruleForm: {

        username: "",

        password: "",

        passwords: "",

        code: ""

      },

      rules: {

        username: [

          {

            validator: validateUsername,

            trigger: "blur"

          }

        ],

        password: [

          {

            validator: validatePassword,

            trigger: "blur"

          }

        ],

        passwords: [

          {

            validator: validatePasswords,

            trigger: "blur"

          }

        ],

        code: [

          {

            validator: checkCode,

            trigger: "blur"

          }

        ]

      }

      // 表单数据  ---结束

然后进行布局，样式等编写

git的使用

此时我们把代码进行提交到github上

git add .

git commit –m “登陆注册校验完成”

查看还有没有其他人进行代码提交，避免代码冲突 git pull

git push

有花括号的import的引用和没有的区别

下面使用export default暴露出去的一个文件中只能有一个，此时import可以没有花括号进行引入





而使用export暴露出去的需要花括号引入否则会报错，但一个文件中可以有多个export

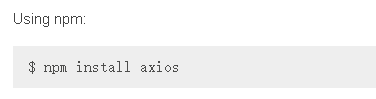
Export function xxx(){}

import { xxx } from “-------”

Axios的使用与proxy解决跨域

制作获取验证码这部分此处我们先在src目录下创建一个文件夹，此处我命名为utils然后创建一个request.js文件这个文件是用来存放我们的拦截器的

然后我们进行axios依赖的安装，见官网



然后我们进行一个axios的使用，首先在request.js文件引入axios，创建axios，然后将拦截器代码粘贴到request.js中,然后将service暴露出去

 import axios from 'axios';

*const* BASEURL = process.env.NODE\_ENV === 'production' ? '' : '/devApi';

//创建axios，然后赋值变量给serve

*const* service = axios.create({

    baseURL: BASEURL,

    timeout: 15000,

  });

//请求接口前，做一些数据处理（请求拦截器）

service.interceptors.request.use(*function* (*config*) {

    // Do something before request is sent

    return *config*;

  }, *function* (*error*) {

    // Do something with request error

    return *Promise*.reject(*error*);

  });

//请求接口后，返回数据进行拦截（响应拦截器）

service.interceptors.response.use(*function* (*response*) {

    // Do something with response data

    return *response*;

  }, *function* (*error*) {

    // Do something with response error

    return *Promise*.reject(*error*);

  });

export default service;

因为此处**存在跨域问题，在vue.config.js文件中配置跨域，该文件下的devServer里面使用proxy，下面的/devApi替换过去的BASEURL，也就是target，而 ^/devApi 就是这里为空是因为target已经附带了文件夹productapi,如果target只是一个url网址不包含文件夹,则此处需要改为请求文件夹的名称**

API服务器的地址 <http://www.web-jshtml.cn/productapi>

 devServer: {

….

….

    proxy: {

      '/devApi': {

          target: "http://www.web-jshtml.cn/productapi", //API服务器的地址  http://www.web-jshtml.cn/api

          changeOrigin: true,

          pathRewrite: {

              '^/devApi': '' //这里为空是因为target已经附带了文件夹productapi,如果target只是一个url网址,则此处需要改为请求文件夹的名称

          }

      }

},

….

….

}

然后在src下创建一个api文件夹,用来存放请求,然后此处做获取验证码的请求，在此文件夹创建login.js文件，然后将service引入将获取验证码的function暴露出去，此处使用request的方式获取是方便更改请求，这里需要return出去

import service from "../utils/request";

/\*\*

 \* 验证码

 \*/

export *function* GetSms(*data*){

 return service.request({

        method:"post",

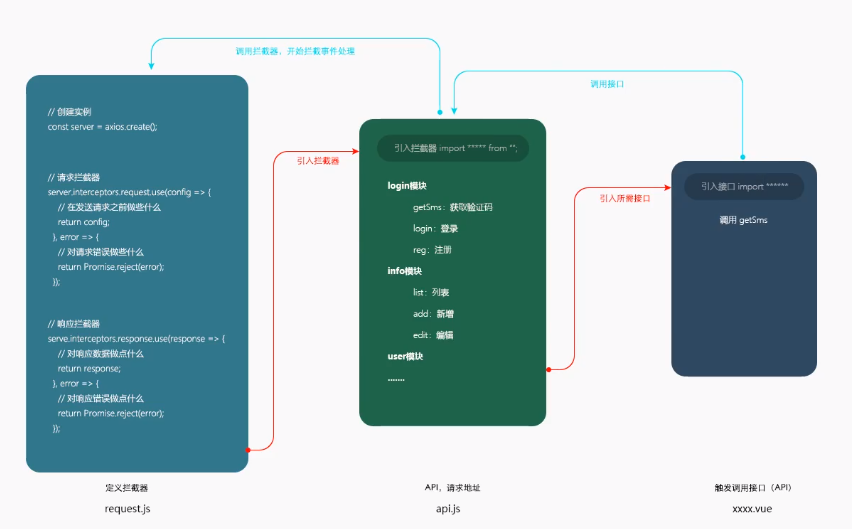
        url:"/getSms/",

        data //data: data

    })

}

然后回到login.vue文件中将login.js的GetSms方法引入，然后在获取验证码的点击事件中执行将输入的邮箱作为参数传入GetSms方法中，



当获取验证码的时候发送请求，下面我们对拦截器的一个编写，也就是数据响应后或者错误应该做的什么，下面是request.js文件的响应拦截器，当获取到数据之后，如果邮箱正确返回的resCode的值为0，所以当邮箱不正确或者为空的时候我们需要把后台捕获到的东西进行一个返回，然后下面使用的Message.error为Element UI的Message 消息提示，这里是需要进行部分引入的，也就是将数据使用UI组件弹出错误信息

//请求接口后，返回数据进行拦截（响应拦截器）

service.interceptors.response.use(*function* (*response*) {

    //对响应数据后做一些事情

*let* data = *response*.data;

    if(data.resCode !== 0){

        Message.error(data.message);

        return *Promise*.reject(data);

    }else{

        return *response*;

    }

  }, *function* (*error*) {

    // 对响应错误后做一些事情

    return *Promise*.reject(*error*);

  });

然后我们前端上也必须是要进行输入邮箱的检测如下代码的 – //提示（这里使用的是Element UI的Message 消息提示组件），然后下面的//请求接口 则是对请求返回的一些处理，GetSms则是对发送请求和返回的一些处理，就是将我们的用户名作为参数传入后台然后返回给我们数据，也就是要看上面的拦截器所返回的值，返回response则调用then，reject则调用catch

 getSms() {

      //提示

      if (this.ruleForm.username == "") {

        this.$message.error("邮箱不能为空");

        return false;

      }

      //请求接口

*let* requestDate = { username: this.ruleForm.username, module: this.model };

      GetSms(requestDate)

        .then(*response* *=>* {

*console*.log(response);

        })

        .catch(*error* *=>* {

*console*.log(error);

        });

    }

下面当切换登陆注册界面的时候我们需要对表单进行一个重置,这里使用的是Element UI的form表单的重置的方法,将该代码粘贴到,登陆注册按钮的点击事件toggleMenu方法里面（这里我们也可以对重置这段代码进行封装，单独写一个方法resetFormData然后在toggleMenu中调用），当点击切换的时候,我们模块的值model也是需要改变的就是改变我们的底部的登陆注册按钮这里使用的是三目运算符。这个按钮使用属性绑定禁用该按钮Element UI的:disabled，等待获取验证码成功的时候再启用

toggleMenu(*data*) {

      this.menuTab.forEach(*element* *=>* {

        element.current = false;

      });

      data.current = true;

      //修改模块值

      this.model = data.type;

      //重置表单

     this.resetFormData();

      //清除获取验证码的定时器和改变text

      this.clearCountDown();

    },

    //重置表单

    resetFormData(){

       this.$refs["ruleForm"].resetFields();

    },

然后我们发送验证码的时候一般都会有一个延时倒计时的状态，所以我们在getSms方法中禁用验证码的按钮和改变按钮的text值（我们把这个功能进行一个封装，减少我们后面代码的重复性），所以这里我们上面的代码的GetSms请求方法放进一个定时器中setTimeout，模拟发送延时效果，然后请求成功我们使用Element UI的Message消息进行一个弹窗，然后下面使底部的登录/注册按钮启用，再使用一个包含setInterval定时器的countDown方法，这个方法是用来改变验证码的状态和按钮的text值的

//修改获取验证码按钮的状态

    updataButtonStatus(*params*){

       this.codeButtonStatus.status = params.status;

      this.codeButtonStatus.text = params.text;

    },

/\*\*

 \* 获取验证码

 \* \*/

    getSms() {

      //提示

      if (this.ruleForm.username == "") {

        this.$message.error("邮箱不能为空");

        return false;

      }

      //此时验证码按钮为禁用和文字为发送中

this.updataButtonStatus({

        status: true,

        text: "发送中"

      });

      // this.codeButtonStatus.status = true;

      // this.codeButtonStatus.text = "发送中";

      //请求接口

*let* requestDate = {

        username: this.ruleForm.username,

        module: this.model

      };

      setTimeout(() *=>* {

        GetSms(requestDate)

          .then(*response* *=>* {

*let* data = response.data;

            this.$message({

              showClose: true,

              message: data.message,

              type: "success"

            });

            //启用登录/注册按钮

            this.loginButtonStatus = false;

            this.countDown(60);

*console*.log(data);

          })

          .catch(*error* *=>* {

*console*.log(error);

          });

      }, 3000);

    },

//获取验证码倒计时

    countDown(*number*) {

      if (this.timer) {

        clearInterval(this.timer);

      }

*let* time = number;

      this.timer = setInterval(() *=>* {

        time--;

*console*.log(time);

        if (time == 0) {

          clearInterval(this.timer);

          this.updataButtonStatus({

            status: false,

            text: "获取验证码"

          });

          // this.codeButtonStatus.status = false;

          // this.codeButtonStatus.text = "获取验证码";

        } else {

          this.codeButtonStatus.text = `倒计时${time}秒`;

        }

      }, 1000);

    },

下面我们对点击注册的时候发送请求给后台进行一个验证，在login.js文件中定义请求方法Register其实和获取验证码的请求相似，获取的url接口是/register/

export *function* Register(*data*){

    return service.request({

        method:"post",

        url:"/register/",

        data //data: data

    })

}

然后在login.vue文件中引入Register方法，下面对底部的登录/注册按钮进行发送一个注册请求，将表单的信息作为参数发送请求，当请求成功时使用弹窗组件将后台返回的信息弹出

/\*\*

 \* 提交表单

 \*/

submitForm(*formName*) {

      this.$refs[formName].validate(*valid* *=>* {

        if (valid) {

*let* registerData = {

            username: this.ruleForm.username,

            password: this.ruleForm.password,

            code: this.ruleForm.code,

            module:"register"

          };

          Register(registerData)

            .then(*response* *=>* {

*let* data = response.data;

*console*.log(data);

              this.$message({

              showClose: true,

              message: data.message,

              type: "success"

            });

            })

            .catch(*error* *=>* {

              this.$message.error(data.message);

            });

        } else {

*console*.log("error submit!!");

          return false;

        }

      });

    }

注册成功跳转到登陆界面，我们只需要更改model的值改为login即可，清除获取验证码的倒计时，,点击登陆注册的切换界面的按钮的时候也需要进行一次清除调用下面的clearCountDown方法

//清除倒计时的方法

    clearCountDown() {

      //验证码按钮高亮

      this.updataButtonStatus({

        status: false,

        text: "获取验证码"

      });

      // this.codeButtonStatus.status = false;

      // this.codeButtonStatus.text = "获取验证码";

      //清除定时器

      clearInterval(this.timer);

    },

下面我们进行登录状态的请求

在login.js文件中定义Login方法，然后在login.vue中进行引入，然后我们需要底部的登录/注册按钮进行一个区分，就是登录的时候使用Login请求，注册的时候使用Register请求，此处我们对两个方法进行一个封装，然后当通过表单后执行三目运算符，判断model的值，来执行对应的方法

/\*\*

  \* 登陆

 \*/

export *function* Login(*data*){

    return service.request({

        method: "post",

        url: "/login/",

        data

    })

}

下面在通过表单后使用三目运算符，执行login方法和register方法，分别是对登录和注册的请求进行的一个封装

/\*\*

     \* 提交表单

     \*/

    submitForm(*formName*) {

      this.$refs[formName].validate(*valid* *=>* {

        //表单验证通过

        if (valid) {

          // console.log(this.model)

          this.model === "login" ? this.login() : this.register();

        } else {

*console*.log("error submit!!");

          return false;

        }

      });

    }

  }

};

下面是对Login、Register方法的一个封装

 /\*\*

     \* 注册请求

     \*/

    register() {

*let* registerData = {

        username: this.ruleForm.username,

        password: this.ruleForm.password,

        code: this.ruleForm.code,

        module: "register"

      };

      Register(registerData)

        .then(*response* *=>* {

*let* data = response.data;

*console*.log(data);

          this.$message({

            showClose: true,

            message: data.message,

            type: "success"

          });

          //注册成功后跳转到登录的界面

          this.toggleMenu(this.menuTab[0]);

          this.clearCountDown();

        })

        .catch(*error* *=>* {

*console*.log(error)

          // this.$message.error(error);

        });

    },

    /\*\*

     \* 登录请求

     \*/

    login() {

*let* LoginDate = {

        username: this.ruleForm.username,

        password: this.ruleForm.password,

        code: this.ruleForm.code,

      };

      Login(LoginDate)

        .then(*response* *=>* {

*let* data = response.data;

*console*.log(data);

          this.$message({

            showClose: true,

            message: data.message,

            type: "success"

          });

         //清理定时器

          this.clearCountDown();

        })

        .catch(*error* *=>* {

*console*.log(error);

          // this.$message.error(error);

        });

    },

密码加密：sha1的使用

加密方法有base64、md5（加密后不可逆）、sha1（加密后不可逆），其实后台也会对密码进行的一个加密

我们前端这里进行一次加密，这里我们使用sha1加密，首先我们安装sha1依赖，见官网npm install sha1

然后我们将sha1引入login.vue文件，然后在发送注册和登录请求的参数传入的密码进行加密

import sha1 from "sha1";

password: sha1(this.ruleForm.password)

然后当发送注册请求的时候，我们可以在Chrome的开发者工具里面的的Network在Name中找到对应的请求信息---Headers----Request Playload里面可以看到你的密码被加密了

请求头传的相关参数，这里我们需要在没此发送请求的时候，我们需要传一些什么参数给后台，我们需要进行请求之前需要做点什么，在request.js文件中的拦截器，config就是我们需要返回出去给后台的东西，然后这个可以在Chrome的开发者工具里面的的Network在Name中找到对应的请求信息---Headers----Request Headers中找到我们添加的请求头的信息Tokey，userId，sui

//请求接口前，做一些数据处理（请求拦截器）

 service.interceptors.request.use(*function* (*config*) {

    // 在发送请求之前需要做什么

    //在请求头添加参数

    //Tokey

    //userId

    //sui

    //业务需求

*config*.headers['Tokey'] = "123";

*config*.headers['userId'] = "MZH";

*config*.headers['sui'] = "456";

    return *config*;

  }, *function* (*error*) {

    // Do something with request error

    return *Promise*.reject(*error*);

  });

我们登陆注册就到此结束了

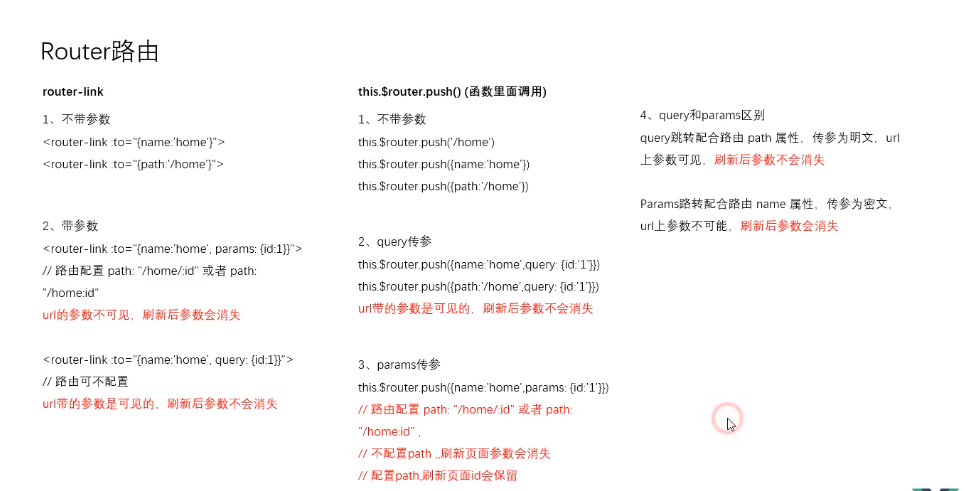
**登陆注册结束**

**后台管理页面开始**

登录页面路由的跳转到后台

然后我们进行一个后台的首页的制作，下面我们来做登陆成功后进入到后台管理页面

看一下下面的路由,这里我们使用的是this.$router.push(),进行路由的跳转



然后我们先在view是文件夹中创建一个Layout文件夹来存放我们布局的组件,在Layout文件（布局组件）中创建一个Layout.vue文件，再创建一个components文件夹其中创建Nav.vue、Main.vue、Header.vue文件，然后我们再views文件夹中创建一个Console文件夹创建console.vue文件用来在Main.vue中显示



在router文件的index.js文件中添加一个路由，调到布局的组件layout.vue

{

    path: "/layout",

    name: "Layout",

    component: ()*=>*import("../views/Layout/layout.vue")

  }

然后我们在login.vue文件中的登陆的请求中,在返回登陆成功后调用this.$router.push()，传入name为Layout，便会跳转到Layout的路由，这里也可以传入一些id 或者user等

//后台页面的跳转

          this.$router.push({

            name:"Layout",

            params:{

              id:'',

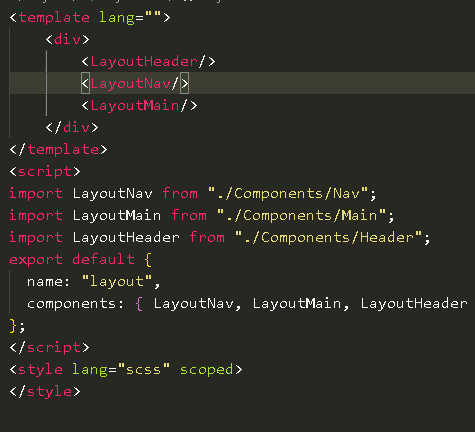
              user:''

            }

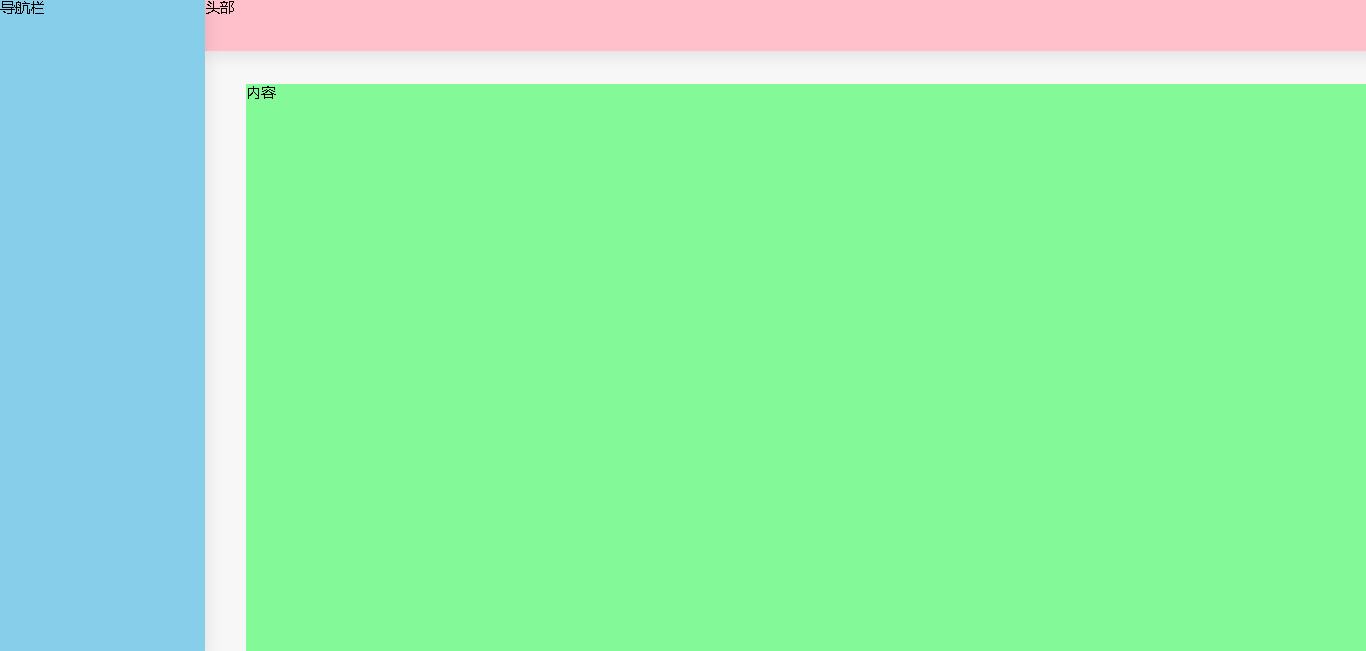
          })

**下面写后台页面的布局**

在layout引入Nav.vue、Main.vue、Header.vue



然后在layout中显示这三个组件，分别设置这三个组件的样式布局，如下



然后我们对Layout进行加入children组件，这里我对layout路由进行一个重定向到console，因为要打开就要在Main.vue组件中显示出来，然后在Main.vue中加入 <router-view /> 这样console.vue就会在Main.vue中被显示出来

{

        path: "/layout",

        name: "Layout",

redirect: "console",

        component: () *=>* import("../views/Layout/layout.vue"),

        children: [

            {

                path: "/console",

                name: "Console",

                component: () *=>* import("../views/Console/console.vue")

            }

        ]

    }



我们使用变量是形式，因为如果导航栏的宽度就要改动头部和内容的样式，所以我们在styles文件中创建一个config.scss文件然后引入main.scss中，在config.scss中定义一个变量$navMenu，然后在Nav.vue、Header.vue、Main.vue文件中引入config.scss文件，然后给导航定义宽度

导航栏

然后我们先对导航栏入手,然后我们这里也是先到Element UI的NavMenu导航菜单中找到我们要用的---折叠---展开---导航一，我们只用两级菜单栏,我们只要里面的一级导航< template > 和一个二级的< el-menu-item> 和相关的数据和方法粘贴到我们的Nav.vue文件里面，然后我们对其进行一些样式修改（见Element UI），在el-menu标签中修改background-color和text-color等，我们将背景设置为透明，然后将isCollapse的值变为true（展开导航）

下面我们再进行将路由遍历到我们的导航栏中（打印出this.$router就可以看到我们的路由了），然后我们先去路由编写一些信息在router文件的index.js文件，在**各个**路由下面添加一个meta对象里面存放一个name属性，包括我们children，因为我们的登录和首页是不需要显示出来所以在首页和登录加多一个hidden属性，例如下面

path: "/",

        redirect: "login",

hidden: true,

        meta:{

          name:"首页"

        }

然后回到我们的Nav.vue文件，然后我们要对遍历出来的首页和登录使用v-if，因为v-for和v-if最好就是不要放在同一标签（原因去Vue官网看看v-for），所以我们使用template便签将el-submenu便签包裹起来，然后在template便签中使用v-for将routers（routers：this.$router.options.routes）遍历出来，然后key值的绑定是要在el-submenu，因为template是不会被解析出来的。然后children遍历到子菜单（这里el-submenu标签绑定的index需要转成字符串类型，否则会报错）

<template v-for="(item,index) in routers">

 <el-submenu v-if="!item.hidden" :key="item.id" :index="index + ''

">

       <!-- 一级菜单 -->

        <template slot="title">

           <i class="el-icon-location"></i>

           <span slot="title">{{ item.meta.name}}</span>

        </template>

         <!-- 子菜单 -->

      <el-menu-item v-for="(subItem,index) in item.children"  :key="subItem.id" :index="subItem.path">{{subItem.meta.name}}

</el-menu-item>

 </el-submenu>

</template>

然后我们现在新加几个路由到router文件夹的index.js文件中,上面这样遍历每个路由是为了我们在点击的时候进行路由的跳转就是对控制台的这个内容进行的一个切换所做的准备

添加一个信息管理的路由（包含children meta对象 name:信息列表、name:信息分类）、用户（包含children meta对象 name:用户列表），然后创建对应的文件夹和vue文件，



Router文件的index.js文件中添加路由，例如下面 -------用户（注意这里的父组件,必须是layout.vue，否则会进行整个页面的跳转）

// 用户------------------------

    {

        path: "/user",

        name: "User",

        meta: {

            name: "用户"

        },

        component: () *=>* import("../views/Layout/layout.vue

"),

        children: [

            {

                path: "/userIndex",

                name: "UserIndex",

                meta: {

                    name: "用户列表"

                },

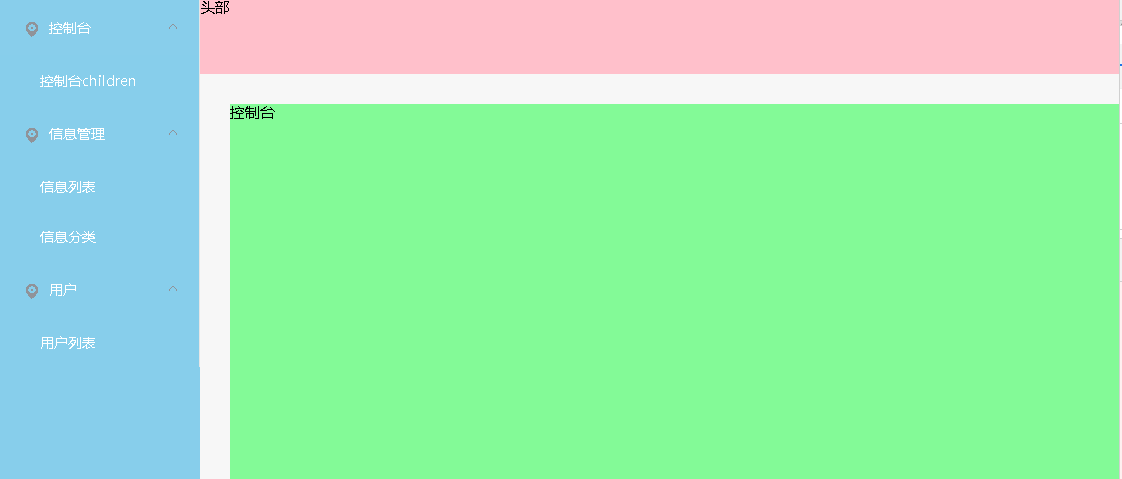
                component: () *=>* import("../views/User/userIndex.vue")

            }

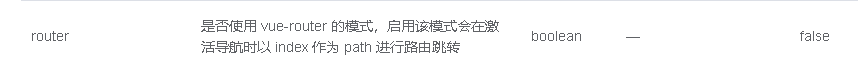
        ]

    }

**此时的页面**



然后我们回到Element UI，导航菜单是有一个router的属性的，所以我们要在el-menu标签中激活我们的路由



然后子菜单需要绑定index，将我们的子路由的path绑定然后就能进行跳转了

<!-- 子菜单 -->

      <el-menu-item v-for="(subItem,index) in item.children"  :key="subItem.id" :index="subItem.path">{{subItem.meta.name}}

</el-menu-item>

然后下面我们对导航栏的样式进行修改，然后这里我们需要在浏览器的开发者工具中找到我们要修改的样式对应的标签类名，这里我们需要对Element UI进行一个全局的修改，因为这里局部修改不生效，然后我们需要在style文件创建一个element.scss文件，我们对边距和选中的样式进行了修改，然后这里使用的$mainColor变量需要在config.scss文件中定义，然后我们要在main.scss文件中引入element.scss

//修改导航栏，二级选中和字体、边距

.el-submenu .el-menu-item {

*font-size*:14px;

*padding-left*: 59px !important;

    &.is-active{*background-color*: rgb(250, 232, 235) !important;}

}

//修改一级菜单栏

li.el-submenu {

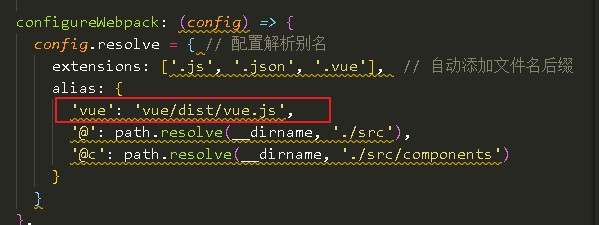
    &.is-active.is-opened .el-submenu\_\_title{ *background-color*: $mainColor !important;}

}

（这里随自己喜欢改颜色样式，上面的样式搭的并不怎么好，只是随意改，后面我会进行一个修改）

然后我们下面对icon图标进行一个修改，这里我们用的是阿里巴巴的矢量图标库（<https://www.iconfont.cn/>），我们需要在路由文件router文件夹的index.js文件中在对应的路由的meta对象里面添加一个icon属性，用来在便签中遍历出来

然后这里我们使用一个全局组件，然后我们使用的是运行模式，所以我需要解决模板编译的问题，需要在vue.config.js文件中修改，让vue指向我们的我们的依赖包里的vue.js文件





Svg的使用和全局组件----使用全局组件编写svg组件，配置好svg获取svg文件下的所有svg图标，引入main.js-----全局引入

下面我们需要安装svg的依赖包

npm install svg-sprite-loader --save-dev

然后下面我们需要在vue.config.js文件进行svg的相关配置

 chainWebpack: (*config*) *=>* {

*const* svgRule = *config*.module.rule("svg");

    svgRule.uses.clear();

    svgRule

    .use("svg-sprite-loader")

    .loader("svg-sprite-loader")

    .options({

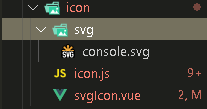
      symbolId: "icon-[name]",

      include: ["./src/icons"]

    })

  },

然后我们在src文件夹里面创建一个icon文件夹,这个文件是用来存放我们的icon文件和我们的自定义全局路由组件的，然后在该文件夹下创建icon.js文件然后我们在这个文件创建我们的自定义全局路由，



然后我们需要在**main.js**文件中引入我们的icon.js文件。

//引入自定义全局组件

import "./icon/icon";

然后我们在icon文件再创建一个svgIcon.vue文件这个文件用来编写我们的全局组件的模板，然后我们再icon.js文件中进行引入（模块化），我们只用在svgIcon.vue编写我们的组件模板即可，**下面为icon.js文件的代码**

//自定义全局组件

import Vue from "vue";

import svgIcon from "./svgIcon"

Vue.component('svg-icon',svgIcon)

/\*\*

 \* require.context :读取指定目录的所有文件

 \* 第一个：目录

 \* 第二个：是否遍历子目录

 \* 第三个: 定义遍历文件规则

 \*/

*const* req = require.context('./svg', false, /\.svg$/);

*const* requireAll = (*requireContext*) *=>* {

    return *requireContext*.keys().map(*requireContext*)

};

requireAll(req);

然后我们需要去下载我们svg图标，去阿里巴巴矢量图标库下载svg图标，然后我们在icon文件夹创建一个svg文件夹用来存放我们的svg文件的将我们下载的svg图标放在这个文件夹中，然后我们将应用svg的标签写在svgIcon.vue文件中，下面为svgIcon.vue的代码

<template>

    <svg :class="svgClass" aria-hidden="true">

        <use :xlink:href="iconName"></use>

    </svg>

</template>

<script>

export default {

  name: "svgIcon",

  props: {

    iconClass: {

      type: *String*, //定义接受类型

      default: ""

    },

    className: {

      type: *String*, //定义接受类型

      default: ""

    }

  },

  data() {

    return {

      msg: ""

    };

  },

  created() {

*console*.log(this.iconClass); //父组件中iconClass传入的值，这里我用来调试

*console*.log(this.className);

  },

  computed: {

    iconName() {

      return `#icon-${this.iconClass}`;

    },

    svgClass() {

      if (this.className) {

        return "svg-icon " + this.className;

      } else {

        return "svg-icon";

      }

    }

  }

};

</script>

<style lang="scss" scoped>

.svg-icon {

}

</style>

然后在我们的组件中使用

<svg-icon iconClass="console" className="console" />

这样我们下载的console.svg文件就能被显示出来了

然后下面我们在Nav.vue中使用svg-icon标签，替换调我们一级菜单的li标签就是那个图标的标签，那么这样这里就可以使用我们的下载svg图标的了，然后我们把对应的svg图标的文件名在meta对象中添加到icon的属性值，然后我们这里使用遍历路由，添加对应的svg图标

<template v-for="(item,index) in routers">

<el-submenu v-if="!item.hidden" :key="item.id" :index="index + ''">

<!-- 一级菜单 -->

<template slot="title">

<svg-icon :iconClass="item.meta.icon" :className="item.meta.icon" />

<span slot="title">{{ item.meta.name}}</span>

</template>

<!-- 子菜单 -->

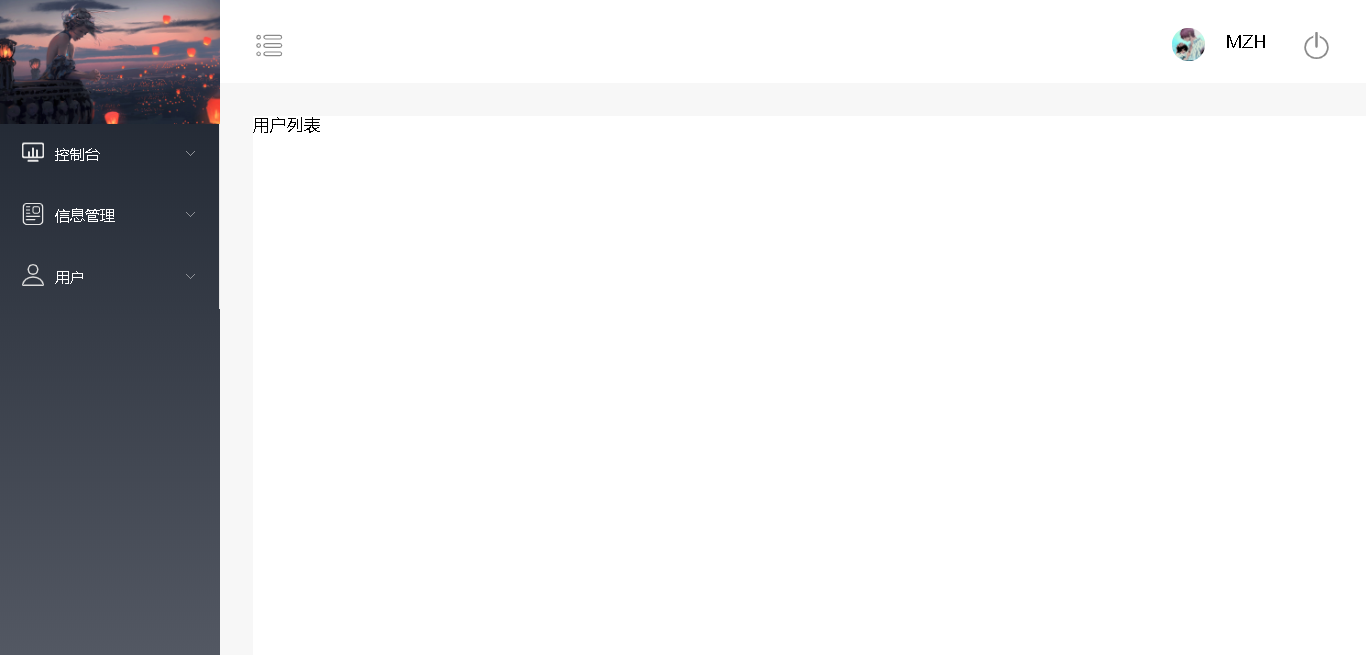
<el-menu-item v-for="subItem in item.children" :key="subItem.id" :index="subItem.path">{{subItem.meta.name}}</el-menu-item>

</el-submenu>

</template>

然后我们回到Header.vue头部添加左侧的信息列表的图标和右侧的退出的svg图标和头像

再回到Nav.vue菜单导航栏添加我们的logo图片，大致布局如下



Vuex

现在我们使用Vuex来实现,点击Header.vue头部的信息列表的svg图标来控制我们的Nav.vue导航栏的收起和展开,之前我们了解过这个导航栏的收起和展开是绑定的一个collapse的属性,然后我们在store文件的index.js文件,因为我们创建项目的时候就已经对Vuex进行了安装和引入(所以没有安装的话就上网找找安装吧)

state 是存储初始化数据的

getters 用来获取我们的存储在store里的state的数据的，对state的数据进行二次处理，获取键值得方法： this.$store.state.xxx this.$store.getters.xxx

mutations(同步) 对数据进行计算的方法都写在这里面(类似computed)，在页面中触发时使用this.$store.commit(“方法名”)触发mutations方法改变state的值

actions (异步) 处理mutations中已经写好的方法，直接触发方式是this.$store.dispatch(方法名)，功能是和mutations是类似的，都是去变更store里的state，不过actions和mutations有两个不同：

1. actions主要是处理的是异步的操作，mutations必须是同步执行，而actions就是不受这样的限制，也就是说actions中我们既可以处理同步（视图触发actions，actions再触发mutations），也可以处理异步的操作
2. actions改变状态，最后是通过提交mutations
3. 角色定位基于流程顺序，二者扮演不同的角色

mutations：专注于修改state、理论上是修改state的唯一途径

action：业务代码、异步请求

modules 模块化Vuex

Vuex存储属性

下面使用Vuex存储isCollapse属性，点击事件触发的方法改变isCollapse属性的值

  state: {

        isCollapse: false

    },

    getters: {

        isCollapse: *state* *=>* *state*.isCollapse

    },

    mutations: {

        SET\_COLLAPSE(*state*) {

*state*.isCollapse = !*state*.isCollapse;

            // console.log(state.isCollapse);

        }

    },

    actions: {},

    modules: {}

然后在Nav.vue中使用computed对赋值给collapse属性的isCollapse进行计算，监听改属性的变化，

computed: {

     isCollapse(){

       return this.$store.state.isCollapse;

    }

  },

然后我们需要在Header.vue文件中的信息列表的svg图标添加点击事件，该方法触发mutations的SET\_COLLAPSE方法改变isCollapse属性的值

<svg-icon iconClass="menu" className="menu" @click="navMenuState" />

 methods: {

    navMenuState(){

        this.$store.commit("SET\_COLLAPSE");

    }

  },

使用actions发送异步请求---在action中使用promise封装请求

在actions中定义方法，该方法传递两个参数，第一个是content对象所包含的东西，然后data是我们要传入的数据，content也可以使用解构的方式，然后在方法里面直接属性名调用

 actions: {

      login(*content*,*data*){ //login({state,getters},data)

*content*.state

*content*.getters

*content*.commit

*content*.rootGetters

*content*.rootState

      }

}

现在我们说说在actions中发送异步请求,这里我们就需要用到Promise了,在actions定义方法，我们我们先在该文件中引入登录请求的方法，然后第二个参数LoginDate使我们在Login.vue所要传入的数据

import { Login } from "../api/login";

 actions: {

      login(*content*,*LoginDate*){

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

          Login(*LoginDate*).then((*response*)*=>*{

*resolve*(*response*);

          }).catch((*error*)*=>*{

*reject*(*error*)

          })

        })

      }

    },

然后问我们回到login.vue文件中修改我们的登录请求也就是我们的Login方法，使用this.$store.dispatch(“方法名”，数据)，调用vuex中的actions的方法---login

其实就是我们在actions中定义的方法发送异步请求后返回的是成功还是失败返回response还是reject，然后回到login.vue中处理事情

login() {

*let* LoginDate = {

        username: this.ruleForm.username,

        password: sha1(this.ruleForm.password),

        code: this.ruleForm.code

      };

      this.$store

        .dispatch("login", LoginDate)

        .then(*response* *=>* {

*let* data = response.data;

*console*.log(response);

*console*.log("登录成功");

          this.$message({

            showClose: true,

            message: data.message,

            type: "success"

          });

          //清理定时器

          this.clearCountDown();

          //后台页面的跳转

          this.$router.push({

            name: "Layout",

            params: {

              id: "",

              user: ""

            }

          });

        })

        .catch(*error* *=>* {

*console*.log(error);

          // this.$message.error(error);

        });

}

modules (我们下面再讲它的使用----使vuex模块化)

绑定class类样式的使用和封装css3兼容样式方法

然后下面我们对导航栏展开进行优化,完善我们的交互,当展开和收起的时候,我们的Header头部和我们的内容Main的边距也要进行相应的改变

我们在布局文件Layout.vue进行绑定一个class

 <template lang="">

    <div id="layout" :class="[isCollapse ? 'close' : 'open']">

        <LayoutHeader />

        <LayoutNav />

        <LayoutMain />

    </div>

</template>

当isCollapse为true的时候我们的导航栏是收起绑定的class为close

然后我们分别需要在Nav.vue、Header.vue、Main.vue文件中定义open 和 close类样式

由于我们的导航栏和内容区的边距的变化是一样的，所以我们可以才config.scss文件中定义一个变量$navMenuMin来统一调用这个变量即可，然后我们的logo图片的大小也是要进行相应的改变的

然后我们添加一个过渡使变化的不用这么的僵硬，然后使用transition，需要对c3属性的兼容在config.scss文件中进行封装一个方法，使用@mixin 方法名（）{ }， 下面的$style为属性，$value为值

//c3属性兼容的的方法

@mixin webkit($style,$value) {

    -webkit-#{$style}: $value;

    -moz-#{$style}: $value;

    -o-#{$style}: $value;

    -ms-#{$style}: $value;

    #{$style}:$value;

}

调用方法 @include 方法名（$style,$value）

 @include webkit(*transition*, all 0.3s ease 0s);

Html5本地存储和Cookie的使用--解决兄弟组件同步相联的值的使用



**本地存储**

当我们刷新页面的时候，我们的导航栏是恢复到我们初始的状态，那么我们想要保留我们的状态，下面我们就使用一个本地H5的本地存储我们的isCollapse值，然后然后在Vuex中,下面我们使用html5的本地存储localStroage(需要手动清除) sessionStroage(关闭浏览器或页面后清除) ,下面我以sessionStroage为例

我们在Vuex的store文件的index.js文件中的mutations属性中将isCollapse用sessionStroage存储，使用JSON对象存储在我们的客户端，在开发者工具中的Application中的sessionStroage中可以看到，存储方法：sessionStroage.setItem(“name”,”value”)

然后我们在state属性中获取这个sessionStroage存储的isCollapse属性,这里我们需要将JSON对象转回js对象，获取属性值的方法：sessionStroage.getItem(“name”)

 state: {

        isCollapse: JSON.parse(sessionStorage.getItem("isCollapse")) || false

    },

 mutations: {

        SET\_COLLAPSE(*state*) {

*state*.isCollapse = !*state*.isCollapse;

            // console.log(state.isCollapse);

            //html5  localStorage存储   sessionStorage存储

            sessionStorage.setItem(

                "isCollapse",

                JSON.stringify(*state*.isCollapse)

            );

        }

    },

这样我们刷新浏览器也可以保留到了我们的导航栏的状态了

**Cookie**

我们先安装cookie的依赖（<https://www.npmjs.com/package/js-cookie>）

npm install js-cookie --save

然后我们在package.json中可以看到我们的cookie版本信息,然后我们在Vuex中引入在store文件的index文件中进行引入

import Cookie from "js-cookie";

然后使用方式和html5的方法有点相似下面是存储和获取isCollapse的方法然后我们在开发者工具中的Application中Cookies中可以看到我们存储的属性值(更多的使用方法见官网)

 state: {

        isCollapse: JSON.parse(Cookie.get("isCollapse")) || false

    }

 mutations: {

        SET\_COLLAPSE(*state*) {

*state*.isCollapse = !*state*.isCollapse;

            //cookie 存储

            Cookie.set("isCollapse", JSON.stringify(*state*.isCollapse));

        }

    }

上面对isCollapse的存储我们使用本地存储，cookie存储一般用来存储我们的token值

然后下面我们完善一下当导航栏收起的时候,鼠标经过导航栏时的样式，这里需要修改Element UI的样式因为他这里是透明的我们把他改成导航栏一样的颜色，我们需要在开发者工具找出我们需要修改类的样式，然后我们在开发者工具的Style----:hov把:hover勾选上我们好调试出来

然后我们修改Element UI的样式有一个element.scss文件

modules的使用-----vuex模块化

现在将我们的vuex进行模块化，我们在store文件夹下然后我们在store中创建一个文件加modules文件，用来放置我们的模块，在modules文件创建一个app.js，,在app.js文件中创建一个app对象，然后我们将index.js里面的 Vuex.Store里面的内容和引入的login请求方法全剪切到app.js文件中的app对象里面，将各个属性提取出来，使用const定义每个属性，然后在exoprt default将所有属性暴露出去，然后处理我们的命名空间的问题将namespaced:true暴露出去，然后我们在index.js文件中引入app.js，

//app.js

import { Login } from "../../api/login";

*const* state = {

isCollapse: JSON.parse(sessionStorage.getItem("isCollapse")) || false

    // isCollapse: JSON.parse(Cookie.get("isCollapse")) || false

};

*const* getters = {

    isCollapse: *state* *=>* *state*.isCollapse

};

*const* mutations = {

    SET\_COLLAPSE(*state*) {

*state*.isCollapse = !*state*.isCollapse;

        sessionStorage.setItem("isCollapse", JSON.stringify(*state*.isCollapse));

  //cookie 存储

        // Cookie.set("isCollapse", JSON.stringify(state.isCollapse));

    }

};

*const* actions = {

    login(*content*, *LoginDate*) {

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

            Login(*LoginDate*)

                .then(*response* *=>* {

*resolve*(*response*);

                })

                .catch(*error* *=>* {

*reject*(*error*);

                });

        });

    }

};

export default {

    namespaced: true,

    state,

    getters,

    mutations,

    actions

};

//index.js

import Vue from "vue";

import Vuex from "vuex";

// import Cookie from "js-cookie";

import app from "./app";

Vue.use(Vuex);

export default new Vuex.Store({

  modules:{

    app

  }

});

然后我们之前对在index.js中应用的state中的isCollapse的值的位置也发生了改变,所以我们在在layout.vue和Nav.vue中找到该代码进行修改,在isCollapse前加一个模块名称app ，还有我们在Header.vue中调用我们的app.js里mutations里面的SET\_COLLAPSE方法就要在它前面加我们模块名了 “app/SET\_COLLAPSE”，还有一处在login.vue中我们调用的登录请求Login方法也需要在前面加我们的模块名

//layout.vue 和 Nav.vue

this.$store.state.app.isCollapse

//Header.vue

this.$store.commit("app/SET\_COLLAPSE");

//login.vue

this.$store.dispatch("app/login", LoginDate)

后面我们还有有其他的模块,后面我们将其他模块放进modules中，引入store的index.js文件中中,区分我们的命名空间,使调用方法更直观

登陆安全问题---vue路由守卫配合token（cookie存储）值解决非法进入后台页面（并且在Vuex也存储一份包括用户信息后续可能有用）



然后我们这里我们之前跨域的接口是没返回token值的，所以我们在vue.config.js里面的devServer中的proxy修改跨域的api接口---target，在api接口后面加/token

这个api是有返回token值的和用户名，然后我们重启项目，每当我们修改vue.config.js文件的时候都需要重启一下项目

target: "http://www.web-jshtml.cn/productapi/token",

因为登录的安全问题，就是我们不允许直接改掉我们的url就直接进入我们的后台页面等（非法进入我们的后台首页），所以我们在路由改变的时候进行相关校验（cookie）-----使用router.beforeEach

我们在utils文件夹创建app.js文件用来存放我们的cookie值的操作的方法，在app.js中引入cookie，然后再router文件夹创建premit.js文件用来存放我们的路由守卫

//util—>app.js

import Cookie from "js-cookie";

*const* adminToken = "admin\_token";

*const* userName = "username"

export *function* getToken() {

    return Cookie.get(adminToken);

}

export *function* setToken(*token*) {

    return Cookie.set(adminToken,*token*);

}

export *function* removeToken(*token*) {

    return Cookie.remove(adminToken);

}

export *function* setUserName(*value*) {

    return Cookie.set(userName,*value*);

}

export *function* getUserName(){

    return Cookie.get(userName);

}

export *function* removeUserName(){

    return Cookie.remove(userName);

}

我们在store文件夹下的app.js操作我们的vuex，首先引入util->app.js的操作cookie值的方法，然后在store也存储一份username和token值，和在发送请求的时候调用cookie方法和mutations的方法，存储cookie和state

//store—>app.js

import { Login } from "../../api/login";

import { setToken,setUserName, getUserName} from "../../utils/app";

*const* state = {

    isCollapse: JSON.parse(sessionStorage.getItem("isCollapse")) || false,

    // isCollapse: JSON.parse(Cookie.get("isCollapse")) || false

    username: getUserName() || "",

    admin\_token: getToken() || ""

};

*const* getters = {

    isCollapse: *state* *=>* *state*.isCollapse

};

*const* mutations = {

    SET\_COLLAPSE(*state*) {

*state*.isCollapse = !*state*.isCollapse;

        sessionStorage.setItem("isCollapse", JSON.stringify(*state*.isCollapse));

},

//在state中存储我们要存放cookie

    SET\_USERNAME(*state*, *value*) {

*state*.username = *value*;

    },

    SET\_TOKEN(*state*, *value*) {

*state*.admin\_token = *value*;

    }

};

*const* actions = {

    login(*content*, *LoginDate*) {

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

            Login(*LoginDate*)

                .then(*response* *=>* {

*let* data = *response*.data.data;

//请求成功存储cookie和state

*content*.commit("SET\_USERNAME",data.username);

*content*.commit("SET\_TOKEN",data.token);

                    setToken(data.token);

                    setUserName(data.username);

*resolve*(*response*);

                })

                .catch(*error* *=>* {

*reject*(*error*);

                });

        });

    }

};

export default {

    namespaced: true,

    state,

    getters,

    mutations,

    actions

};

在premit.js文件中引入router文件夹的index.js(引入路由)，和引入app.js中的方法，然后使用router.beforeEach((to,from,next)={})方法

to:到哪个路由

from:离开哪个路由

next:最后到的哪个路由(如果不传参默认为to的路由)

(注意:如果next传参数,没有对next,没有使用判断什么条件终止,就会出现死循环,因为路由跳转就会执行router.beforeEach)

然后把premit.js在我们的main.js文件中全局引入

premit.js

首先，引入util->app.js中的cookie方法和引入store，然后定义一份白名单数组（一般是我们的登录页面），就是用来存放我们的路由，在路由守卫router.beforeEach中通过if语句判断如果Token不存在，则判断要跳转的路由是否存在白名单，如果不存在则执行next(‘/login’)跳转回登录页面，如果存在则执行next()跳转,如果我们跳转到登陆页面login则将我们的所有的token值删除,然后就不能回到后台页面了需要重新登陆

然后我们需要获取到后台给我们的token值，然后对它进行存储，来用于路由守卫的条件判断

使用cookie存储token值，使用token值作为进入我们后台页面的依据

//router->premit.js

import router from "./index";

import { getToken,removeToken,removeUserName } from "../utils/app";

import store from "../store/index";

//白名单

*const* whiteRouter = ['/login']; //indexof方法，判断数组中是否存在指定的某个对象，如果不存在，则返回-1

router.beforeEach((*to*, *from*, *next*) *=>* {

    if (getToken()) {

         if (*to*.path === "/login") {

            removeToken();

            removeUserName();

//记得添加命名空间名app

            store.commit("app/SET\_TOKEN", "");

            store.commit("app/SET\_USERNAME", "");

            // console.log("存在");

*next*();

        }else{

*next*();

        }

    } else {

        if(whiteRouter.indexOf(*to*.path) !== -1){

*next*(); //to

        }else{

*next*("/login");

        }

    }

    /\*\*

     \* 当进行路由跳转时判断获取到的Token值是否存在，不存在则判断我们要跳转的路由是否存在我们的白名单里面，如果存在则直接跳转，如果不存在则执行next("/login")回到登录页面，路由再次跳转再执行router.beforeEach，因为login存在我们的白名单，则调用next()直接跳转

    \*/

});

退出登录

下面再给我们的svg的退出图标添加点击事件，退出登录跳转回登录页面，然后将存储的cookie和store中的state的username和admin\_token删除

头像img旁边使用我们在store存储的username名称，然后在svg退出的图标添加点击事件exit，引入存放cookie的操作方法和store，在exit方法中进行路由跳转this.$router.push({name:”路由名称”})和删除相关cookie和store中的state（其实在路由守卫也进行了一次删除的，因为路由跳转就会触发路由守卫）

//Header.vue

<div class="user-info pull-left">

<img src="../../../assets/images/userHeader.png" alt="">{{username}}

</div>

<div class="header-info pull-left" @click="exit">

   <svg-icon iconClass="exit" className="exit" />

</div>

在

//Header.vue

<script>

import store from "../../../store/index";

import { removeToken, removeUserName } from "../../../utils/app";

export default {

  name: "Header",

  data() {

    return {

      username: store.state.app.username

    };

  },

  methods: {

    //设置控制导航菜单栏的展开收起的状态

    navMenuState() {

      this.$store.commit("app/SET\_COLLAPSE");

    },

    //退出登陆

    exit() {

*console*.log("退出登陆");

      this.$router.push({

        name: "Login"

      });

      store.commit("app/SET\_TOKEN", "");

      store.commit("app/SET\_USERNAME", "");

      removeToken();

      removeUserName();

    }

  }

};

</script>

Element ui弹窗踩坑

Element ui的弹窗使用踩坑----el-dialog的父级元素设置了position:fixed，并且其z-index比弹出框的遮罩层的小，所以就会出现遮罩层把内容挡住的问题。

解决方案:

1、给el-dialog设置modal-append-to-body=“false”，使遮罩层插入至 Dialog 的父元素上。（推荐）

2、给position:fixed的父元素设置一个z-index，并且要比遮盖层的大。

3、el-dialog父元素不使用fixed定位。

Echarts

npm install echarts –S

在main.js中全局引入

import echarts from 'echarts'

Vue.prototype.$echarts = echarts;

父子组件传值：

父组件传值到子组件:

父组件中在子组件的标签中绑定一个属性 例 :flag并将父组件的数据赋值给该属性

子组件中使用props,把父组件的绑定的属性添加到props中props:{ flag{type: Boolean,default: false} }

子组件传值到父组件:

在父组件中在子组件的标签中绑定一个方法 例 @func,然后将父组件的中定义个一个方法（要有参数）赋给@func

子组件中的标签绑定方法@func将子子组件的方法赋给他，在子组件的方法调用

this.$emit(“@func”,value) 把value传给父组件

子组件传值到父组件（另一个方法使用修饰符）:

父组件中在子组件的标签中绑定一个属性使用sync修饰符修饰 例 :flag.sync并将父组件的数据赋值给该属性

自组件中在方法中调用this.$emit("update:flag",value)，把value传给父组件

（但该方法只能是传一个值，不能进行其他的逻辑处理）