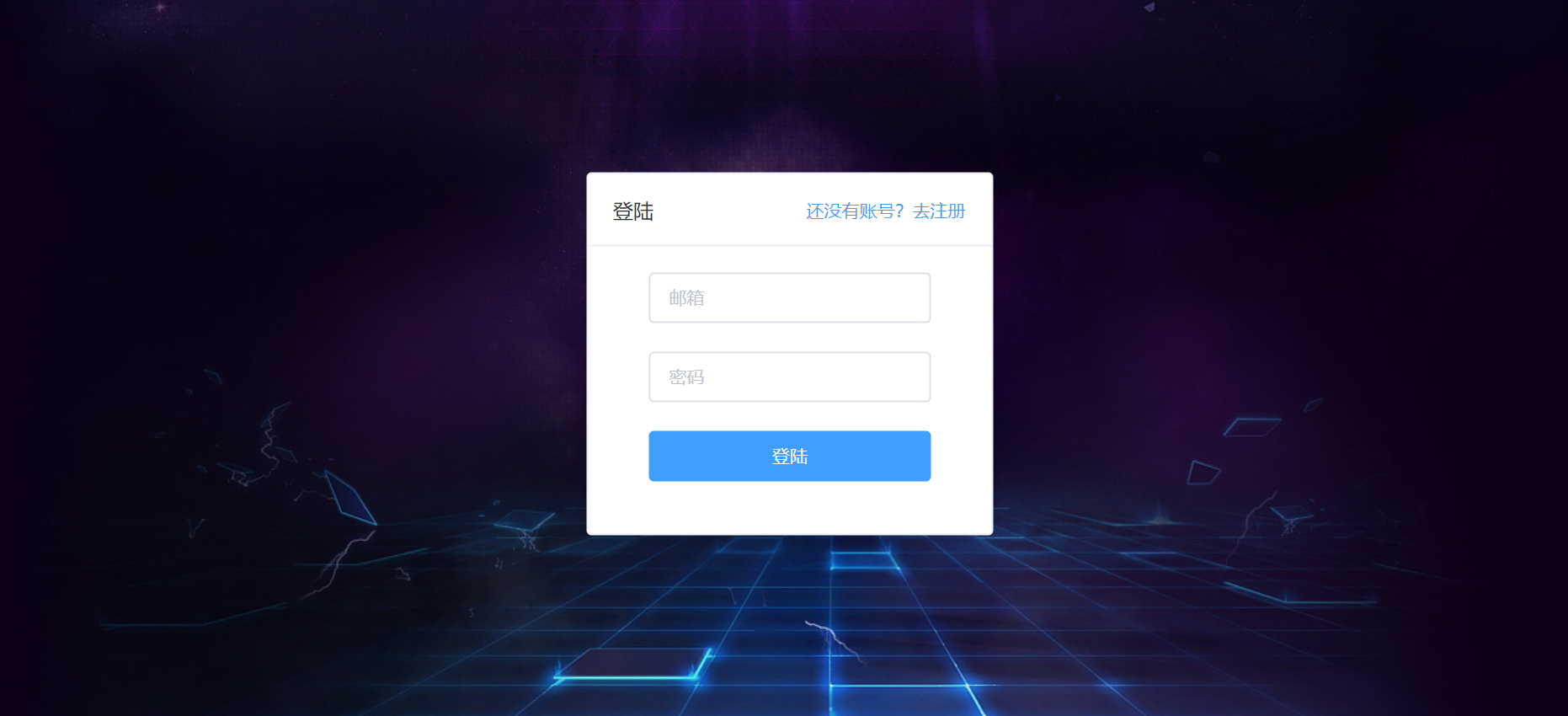
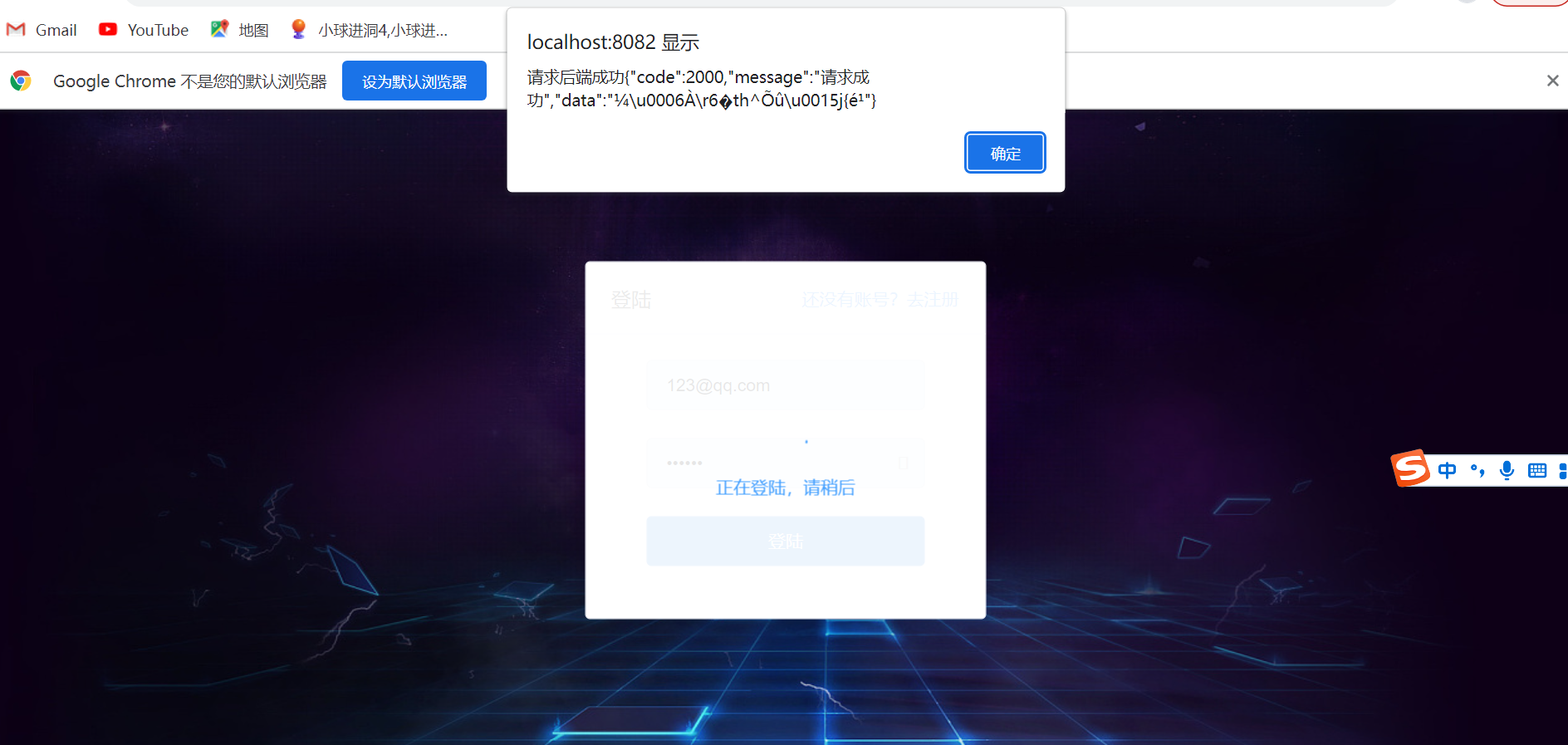
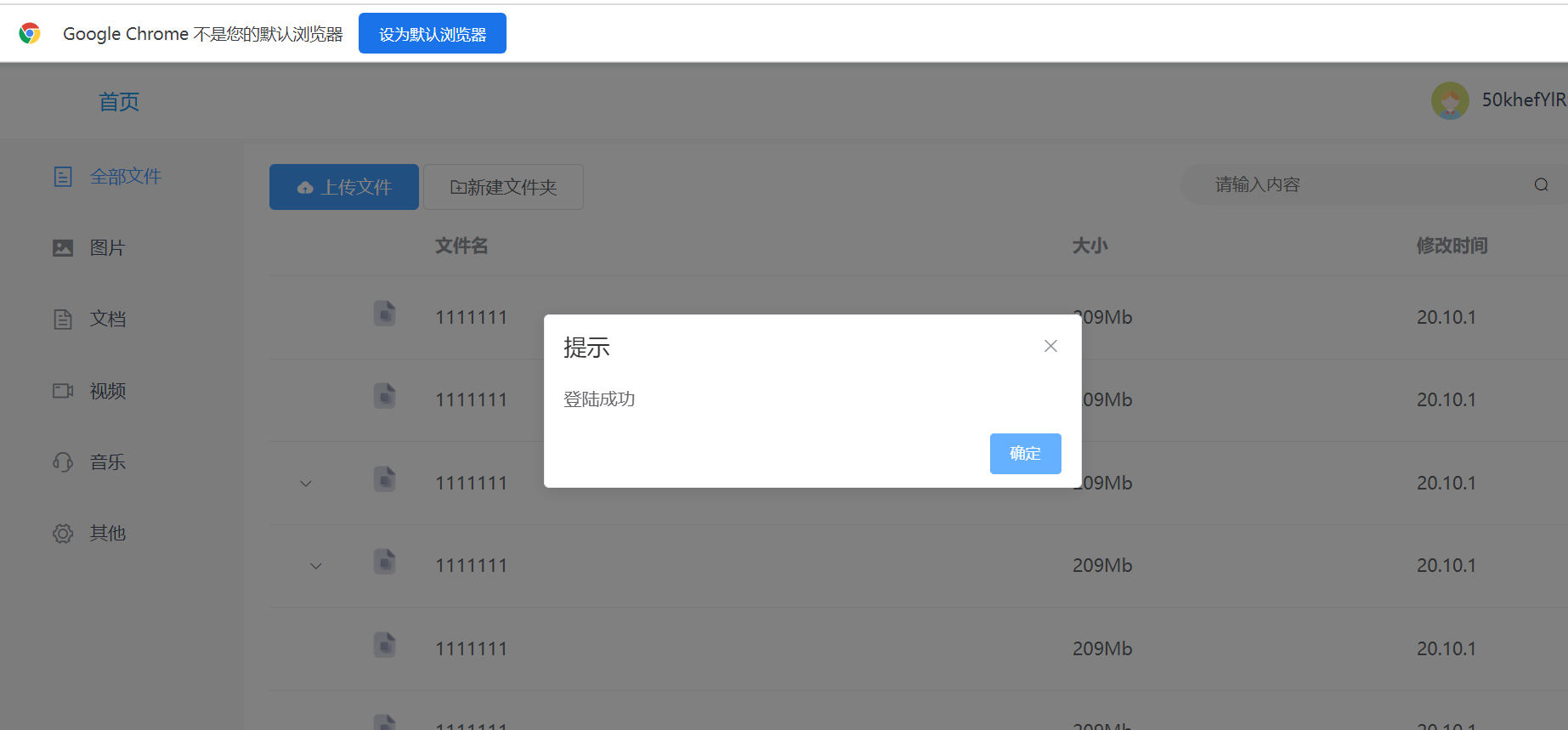
登录实现

登录流程方案：

1. 用户在前端填写用户名（Email）和密码，点击登录按钮进行登录；
2. 前端将用户名发送到后端，在前端开辟一个临时储存空间，将密码储存在前端；
3. 后端接收前端发送的用户名，在数据库中找到对应的客户端随机数，发送给前端；
4. 前端接收后端发送的客户端随机数，用字符串”SZTUBIGDATA” 连接对齐字符”A”作为填充字符然后连客户端随机数，使用SHA-256生成Salt，利用用户输入的密码，随机数生成的盐值（Salt），使用PBKDF2算法生成256位哈希值（与注册时加密方法相同）；
5. 将PBKDF2算法生成的256位哈希值的后128位用SHA-256生成256位的哈希值；
6. 将256位哈希值发送到后端进行验证；
7. 后端接收256位哈希值，利用之前接收的用户名在数据库找到对应的256位哈希值进行比对；
8. 比对成功则将该用户加密的主密钥及文件系统根目录相关信息发送给前端；
9. 比对失败则在前端显示登录失败；
10. 验证成功后，用户在前端解密主密钥，并将后端发送的文件信息展示在前端页面上；

登录界面实现：



登录流程实现代码：

用户通过web浏览器进入登录页面，通过表单填写email和密码，点击登陆功能，进行登录验证 。如果没有账号，就点击“还没有账号？去注册”，跳转到注册界面进行注册。

登入页面部分代码：

1. 通过form表单收集用户登入信息，用户点击“登陆”按钮进行登录，在前端通过login函数对用户信息进行验证。

<template>

<div class="main">

<el-card class="box-card" v-loading="loading" element-loading-text="正在登陆，请稍后">

<div slot="header" class="clearfix">

<span>登陆</span>

<el-button style="float: right; padding: 3px 0" type="text" @click="register">还没有账号？去注册

</el-button>

</div>

<el-row type="flex" justify="center">

<el-form :model="ruleForm" :rules="rules" ref="ruleForm" class="demo-ruleForm">

<el-form-item prop="name">

<el-input v-model="ruleForm.name " placeholder="邮箱"></el-input>

</el-form-item>

<el-form-item prop="pass">

<el-input v-model="ruleForm.pass" show-password placeholder="密码"></el-input>

</el-form-item>

<el-form-item>

<el-button type="primary" @click="login('ruleForm')" class="login">登陆</el-button>

</el-form-item>

</el-form>

</el-row>

</el-card>

</div>

</template>

login函数如下：

通过export default导出login，在login函数中收集表单的邮箱和密码信息，首先检查邮箱格式是否正确，如果不正确重置表单并提醒用户重新填写正确的邮箱并返回。如果邮箱格式正确，前端就将用户名发送给后端并将密码存储在前端。

export default {

name: "Login",

data() {

var name = (rule, value, callback) => {

if (value === '') {

callback(new Error('请输入邮箱'));

} else if (!isEmail(value)) {

callback(new Error('邮箱格式不正确'));

} else {

callback();

}

};

var pass = (rule, value, callback) => {

if (value === '') {

callback(new Error('请输入密码'));

} else {

callback();

}

};

1. 请求发送邮箱：

Methods是前端处理数据、代码逻辑的地方，通过async await异步请求方法调用web端接口，通过发送的请求数据获得接口返回值，然后通过数据处理实现数据渲染，将数据发送到后端。

methods: {

async login(ruleForm) {

// 防止以前的数据没有清除，影响调试

localStorage.clear();

var randomvalue = null;

var verify = null;

let \_this = this;

// 先验证账号

await this.$refs.ruleForm.validate(async (valid) => {

if (valid) {

\_this.loading = true

} else {

console.log('error submit!!');

return false;

}

});

await request.post("/users/check", JSON.stringify({

"email": \_this.ruleForm.name

})

1. 后端接收前端发送的用户名，后端找到客户端随机数，并发送给前端：

后端接收login界面发来的email，并与数据库进行对比，判断账号和客户端随机数是否存在。若都存在则将该用户的客户端随机数返回给前端；若email不存在则返回请求无效；若客户端随机数不存在则返回用户不存在。

 /\*\*

     \* 检查用户是否存在，并返回用户的随机码

     \*

     \* @param email 邮箱

     \* @return 随机码

     \*/

    @PostMapping("/check")

    @ApiOperation(value = "检查用户是否存在")

    @ApiImplicitParams({@ApiImplicitParam(name = "email", value = "用户邮箱", dataType = "String")})

    @ApiResponses({@io.swagger.annotations.ApiResponse(code = 2000,message = "响应成功",response = ApiResponse.class)})

    public ApiResponse checkUser(String email) {

        String randomValue = null;

        if (email == null || email.length() == 0) {

            return ApiResponse.ofStatus(StatusEnum.INVALID\_REQUEST);

        }

        randomValue = userService.findRandomValueByEmail(email);

        if (randomValue == null) {

            return ApiResponse.ofStatus(StatusEnum.NO\_USER);

        }

        log.info("------------" + randomValue);

        return ApiResponse.ofSuccess(randomValue);

    }

1. 前端请求接收客户随机数、邮箱或客户端随机数不存在状态的信息：

接收后端返回的客户端随机数或者邮箱不存在的信息，若获取到客户端随机数则进行加密；若获取到邮箱或客户端随机数不存在状态的信息，则在前端页面上提示用户登录失败。

// 先验证账号

await this.$refs.ruleForm.validate(async (valid) => {

if (valid) {

\_this.loading = true

} else {

console.log('error submit!!');

return false;

}

});

await request.post("/users/check", JSON.stringify({

"email": \_this.ruleForm.name

})

).then(async function (res) {

alert("请求后端成功" + JSON.stringify(res))

if (res.code === 2000) {

randomvalue = res.data;

// sessionStorage只能存储字符串数据，无法直接存储数组类型和JSON对象,TODO关注

// 可以无需存储

sessionStorage.setItem('randomValue', randomvalue)

// 计算256验证哈希

}

})

1. 通过客户随机数生成salt并和密码进行pbkdf2加密，然后后128位用SHA-256哈希：

用后端返回的客户端随机数和密码利用PBKDF2加密产生256位的密钥，后128用SHA-256生成哈希值。用注册时定义好的函数，先将客户端随机数用SHA-256生成Salt，再将Salt、密码用PBKDF2生成256位的密钥。再将后128位用SHA-256生成哈希值。

let pbkdf2Key = await pbkdf2Function(\_this.ruleForm.pass, randomvalue);

const rawKey = await crypto.subtle.exportKey('raw', pbkdf2Key);

const rawKeyArray = new Uint8Array(rawKey)

const left128Bits = rawKeyArray.subarray(0, 16);

// 只能保存字符串

sessionStorage.setItem('left128Bits', uint8ArrayToString(left128Bits))

const right128Bits = rawKeyArray.slice(16, rawKeyArray.length);

const hashBuffer = await crypto.subtle.digest('SHA-256', right128Bits)

console.log("-----------" + hashBuffer + "---------------")

var sha256VerifyKey = new Uint8Array(hashBuffer);

console.log("sha256VerifyKey---" + sha256VerifyKey);

verify = uint8ArrayToString(sha256VerifyKey)

alert("验证哈希"+verify)

1. 将密钥后128位用SHA-256哈希的哈希值并发送给后端：

sync await异步请求方法将256位哈希值发送到后端。

await request.post("/users/login", JSON.stringify({

"email": \_this.ruleForm.name,

"verifyKey": verify,

})).then(async function (res) {

console.log((JSON.stringify(res)))

1. 后端接收前端发送的哈希值，在数据库中找到该用户对应的哈希值，将从前端接收哈希值与找到的相比较。若相同则验证成功，向前端返回验证成功的响应，并找到该用户加密的主密钥和文件根目录，并将其发送给前端：

@PostMapping("/login")

    @ApiOperation(value = "用户登录")

    public ApiResponse login(@Valid @RequestBody AuthenticationRequest authenticationRequest) {

        // 验证用户

        UserPo userPo = authenticationService.authentication(authenticationRequest);

        String token = MyJWTUtil.createToken(userPo);

        userPo.setToken(token);

        return ApiResponse.ofSuccess(userPo);

    }

@GetMapping("/{userPath}")

    @ApiOperation(value = "获取用户详细信息")

    @ApiImplicitParams({@ApiImplicitParam(name = "userPath", value = "用户路径", dataTypeClass = String.class)})

    public ApiResponse findInfoById(@PathVariable(value = "userPath") String userId) {

        UserPo userPo = userService.findById(MyStringUtil.getId(userId));

        return ApiResponse.ofSuccess(userPo);

    }

1. 前端接收后端发送的主密钥及文件系统根目录，获取加密后的主密钥及文件系统根目录的信息，就将主密钥解密并将所有信息储存在前端。在前端web浏览器跳转到主页面，在主页面上展示根目录信息，并提示用户登陆成功。

将加密的主密钥解密并将所有的数据储存在前端：

//获取加密后的主密钥然后解密存储,TODO

var encryptedMasterKey = stringtoUint8Array( user.encryptedMasterKey)

let randomvalue = stringtoUint8Array( user.clientRandomValue);

const left128Bits = stringtoUint8Array(sessionStorage.getItem("left128Bits"));

// 将返回的数据全部放到 localStorage 中

unction saveUser(user){

localStorage.setItem("uid",user.id);

localStorage.setItem("email",user.email);

localStorage.setItem("name",user.name);

localStorage.setItem("clientRandomValue",user.clientRandomValue);

localStorage.setItem("verifyKey",user.verifyKey);

// 这里应该存解密好了的主密钥

localStorage.setItem("encryptedMasterKey",user.encryptedMasterKey);

localStorage.setItem("curLoadTime",user.curLoadTime);

localStorage.setItem("lastLoadTime",user.lastLoadTime);

localStorage.setItem("registerTime",user.registerTime);

localStorage.setItem("usedCapacity",user.usedCapacity);

localStorage.setItem("totalCapacity",user.totalCapacity);

}

saveUser(user)

在前端页面提示用户登陆成功，跳转到主页面。

\_this.$alert('登陆成功', '提示', {

confirmButtonText: '确定',

callback: action => {

\_this.$router.push({path: 'Home'})

}

})

setTimeout(() => {

\_this.$router.push({path: '/Home'})

}, 1000)

} else {

\_this.loading = false

\_this.$alert(res.message, '提示', {

confirmButtonText: '确定',

})

}

若需源代码，请访问https://github.com/darKmili/cloud