**注册：**

**注册流程方案：**

1. 用户在前端填写用户名（Email）和密码，点击注册按钮进行注册；
2. 在前端利用随机数生成函数分别生成128位 客户端随机数 和 128位 主密钥；
3. 使用字符串”SZTUBIGDATA”连接对齐字符”A”作为填充字符然后连接客户端随机数生成长度为200的字符串，使用SHA-256对该字符串进行哈希生成256位的哈希值，该哈希值作为盐值（Salt）；
4. 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，生成的函数原型如下，其中100000表示迭代十万次，256表示PBKDF2算法生成的哈希值为256位：

PBKDF2-HMAC-SHA-512( 密码 , Salt , 100000, 256 )

1. 将PBKDF2生成的256位分成前128位（左边）和后128位（右边）
2. 将前128位作为AES密钥对主密钥进行加密生成128位的加密主密钥
3. 将后128位利用SHA-256算法进行哈希，生成256位的哈希值用于身份验证
4. 将用户名、128位的客户端随机数、128位的加密主密钥、用于身份验证的256位哈希值一起发送到服务器，服务器存储成功完成注册

注册流程实现

前端界面：

图形用户界面, 应用程序

描述已自动生成

电脑萤幕的截图

描述已自动生成

图形用户界面, 应用程序

描述已自动生成

用户通过web浏览器进入注册页面，通过表单填写email并两次输入密码，点击注册功能，进行注册，表单代码如下：

注册页面部分代码：

通过form表单收集用户注册信息，用户点击“注册”按钮进行注册，前端执行register的函数完成注册功能。

**<template>**

**<div class="main">**

**<el-card class="box-card" v-loading="loading" :element-loading-text="loading\_text">**

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

**<span>注册</span>**

**<el-button style="float: right; padding: 3px 0" type="text" @click="login">已经有账号？去登陆**

**</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 prop="repass">**

**<el-input v-model="ruleForm.repass" show-password placeholder="确认密码"></el-input>**

**</el-form-item>**

**<el-form-item>**

**<el-button type="primary" @click="register('ruleForm')" class="register">注册</el-button>**

**</el-form-item>**

**</el-form>**

**</el-row>**

**</el-card>**

**</div>**

**</template>**

Register的函数如下，在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、检查密码是否大于六个字符以及两次输入的密码是否一致，如果不正确或不一致则重置表单并提醒用户重新填写正确的邮箱或密码并返回。如果邮箱格式正确、密码大于六个字符并且密码一致，则接下来分别生成两个128位的随机数：

验证邮箱格式是否正确、检查密码是否大于六个字符及验证再次输入的密码是否正确：

**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 if (value.length < 6) {**

**callback(new Error('至少为六个字符'));**

**} else {**

**callback();**

**}**

**};**

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

**if (value === '') {**

**callback(new Error('请输入密码'));**

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

**callback(new Error('两次输入密码不一致!'));**

**} else {**

**callback();**

**}**

**};**

在register.js文件的doregister()函数中用Uint8Array（2 \*\* 4）对象创建8个8位无符号整形数组，用crypto.getRandomValues();方法随机生成符合密码学要求安全的客户随机数和主密钥：

    let clientRandomValue = new Uint8Array(2 \*\* 4);

    let masterKey = new Uint8Array(2 \*\* 4);

window.crypto.getRandomValues(clientRandomValue);

window.crypto.getRandomValues(masterKey);

在doregister()函数中用pbkdf2Functio()函数对客户端随机数和密码进行pbkdf2加密。用exportkey()方法将生成的256位哈希值以原格式导出。将导出的哈希值转化成32个8位无符号整形数组。子数组left128Bits截取rawKeyArray数组的前16个数，子数组right128Bits截取rawKeyArray数组的后16个数。将right128Bits用digest()方法生成256位哈希值，将left128Bits用encryptKey()函数对主密钥进行AES加密：

    let pbkdf2Key = await pbkdf2Function(password, clientRandomValue);

    const rawKey = await crypto.subtle.exportKey('raw', pbkdf2Key); //将密钥以原格式导出

    const rawKeyArray = new Uint8Array(rawKey)

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

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

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

    var sha256VerifyKey = new Uint8Array(hashBuffer);

    var encryptedMasterKeyHashValue = await encryptKey(left128Bits,clientRandomValue, masterKey)

    var encryptedMasterKey = new Uint8Array(encryptedMasterKeyHashValue)

    var deKeyBuffer = await decryptKey(left128Bits,clientRandomValue, encryptedMasterKey)

    var decryptedMasterKey = new Uint8Array(deKeyBuffer)

    //Avoid password stole by server before interaction

    registerForm.elements['password'].value = "OverwritePassword";

    registerForm.elements['passwordAgain'].value = "OverwritePassword";

利用客户端随机数和密码使用PBKDF2生成256位哈希值：

在pbkdf.js文件中的pbkdf2Function()函数中，先将客户端随机数由unit8Array类型转换为String类型，再使用字符串”SZTUBIGDATA”连接对齐，字符”A”作为填充字符。生成200位的字符串shastring。使用SHA-256对shastring进行哈希生成256位的哈希值，该哈希值作为盐值salt。用importKey()方法生成一个PBKDF2的加密密钥，传递一个 Pbkdf2Params 对象给deriveKey ()方法实现PBKDF2算法生成256位哈希值。返回生成的哈希值。

    var randomValueLength = 16;

    var clientRandomValueBinaryString = uint8ArrayToString(clientRandomValue)//Convert clientRandomValue to String

    //Generate Padding String ("SZTUBIGDATA"|| Padding || clientRandomValue)with length 200

    var shaString = "SZTUBIGDATA";

    var padLength = 200 - randomValueLength;

    shaString = shaString.padEnd(padLength, 'A');

    shaString = shaString + clientRandomValueBinaryString;

console.log("shaString Length:"+shaString.length)

//对shastring进行SHA-256，生成salt

const textEncoder = new TextEncoder();

    const shaMessage = textEncoder.encode(shaString);

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

var salt = new Uint8Array(hashBuffer);

const enPassword = await window.crypto.subtle.importKey(

        "raw",

        textEncoder.encode(password),

        {name: "PBKDF2"},

        false,

        ["deriveBits", "deriveKey"]

    );

    return window.crypto.subtle.deriveKey(

      {

        "name": "PBKDF2",

        "hash": "SHA-256",

        "salt": salt,

        "iterations": 100000

      },

      enPassword,

      { "name": "AES-GCM", "length": 256},

      true,

      [ "encrypt", "decrypt"]

    );

利用left128Bits作为密钥对主密钥进行AES加密：

在encryptKey()函数中，先建立一个AES的加密对象params，再用import Key()方法生成一个加密密钥对象aesKey。最后将params、aesKey和主密钥导入encrypt()方法中进行加密。返回生成的加密的主密钥。

async function encryptKey(keyArr, clientRandomValue, theKeyToEnc){

    const params = {

        name: "AES-CBC",

        iv: clientRandomValue,

        length: 128

    };

    const aesKey = await crypto.subtle.importKey(

                        "raw",

                        keyArr.buffer,

                        "AES-CBC",

                        true,

                        ["encrypt", "decrypt"]);

    const encryptedData = await crypto.subtle.encrypt(params, aesKey, theKeyToEnc);

    return encryptedData

}

将Email、客户端随机数、加密的主密钥和哈希值等数据打包成json文件，向后端发送请求，将打包好的文件发送给后端并接收后端返回的数据，在前端页面上显示注册成功或者注册失败的信息：

**request.post("/users/register", JSON.stringify(value)).then(res => {**

**if (res.code === 2000) {**

**this.$alert('注册成功', '提示', {**

**confirmButtonText: '确定',**

**callback: action => {**

**this.$router.push({path: '/'})**

**}**

**})**

**} else {**

**this.loading = false**

**this.$refs.ruleForm.resetFields();**

**this.$alert(res.msg, '提示', {**

**confirmButtonText: '确定',**

**})**

**}**

**})}**

后端接收前端发送的Email、客户端随机数、加密的主密钥和哈希值：

后端利用@PostMapping注解接收前端的发送的数据，UserPo类定义多个对象，用@param将前端发送的多个数据映射到新建的UserPo对象。

    /\*\*

     \* 用户注册

     \*

     \* @param userPo 用户对象

     \* @return ApiResponse

     \*/

    @PostMapping("/register")

    @ApiOperation(value = "用户注册")

    public ApiResponse register(@Valid @RequestBody UserPo userPo) {

UserPo类部分代码，在UserPo类中定义用户的各种信息：

@Basic

    @Column(name = "email")

    @Email

    @ApiModelProperty(value = "用户邮箱",required = true)

  @Basic

    @Column(name = "client\_random\_value")

    @ApiModelProperty(value = "客户端随机数",required = true)

    @NotNull

    private String clientRandomValue;

    @Basic

    @Column(name = "sha256verify\_key")

    @ApiModelProperty(value = "256位验证哈希",required = true)

    @NotNull

    private String sha256VerifyKey;

    @Basic

    @Column(name = "encrypted\_master\_key")

    @ApiModelProperty(value = "已加密的主密钥",required = true)

将获取的Email与数据库相比对，相同就注册失败，并在前端提示已存在；不同则将Email、客户端随机数、加密的主密钥和哈希值存入数据库：

从数据库中获取email判断email是否存在，若存在就将“用户已经存在”返回给前端；如果不存在，就将Email、客户端随机数、加密的主密钥和哈希值等封装到UserPo对象里，将user对象的值插入到数据库中，并向前端返回注册成功的信息。

// 查看用户是否存在

        UserPo userPoByEmail = userService.findUserByEmail(userPo.getEmail());

        if (userPoByEmail != null) {

            return ApiResponse.ofStatus(StatusEnum.INVALID\_REQUEST).setMessage("用户已经存在");

        }

// 初始化用户

        UserPo initUser = userService.initUser(userPo);

        return ApiResponse.ofSuccess(initUser);

数据库的三张表格：

数据库用户信息表：



文件系统表：



文件块信息表：

