接口设计

1. 接口设计

1.1 Eolink链接

（暂空）

1.2 接口文档

1.2.1 用户登录接口（OAuth2.0 规范实现）

接口地址：

`/api/login`

请求方法：

`POST`

请求头：

`Content-Type: application/json`

请求参数：

字段	名称	说明
user_id	用户ID	用户的唯一标识，用于链接对应的用户表
passwd	用户密码	登录密码
nick_name	账号	登录账号，与用户ID和用户密码绑定
reCAPTCHA（SDK）	人机验证	用于人机验证功能，防止强行攻破

字段名	类型	必填	说明
grant_type	string	是	授权类型，固定为password
username	string	是	用户名
password	string	是	使用公钥加密后的密码
public_key	string	是	客户端生成的公钥

请求体示例：

```
1 {"grant_type": "password","username": "user123","password":
  "encrypted_password","public_key":
  "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA..."}
```

响应参数：

字段名	类型	说明
access_token	string	用于身份验证的访问令牌
token_type	string	令牌类型，通常为Bearer
expires_in	int	令牌的有效期，单位为秒
refresh_token	string	用于刷新访问令牌的刷新令牌
encrypted_data	string	加密后的响应数据

响应示例：

```
1 {"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9","token_type":
  "Bearer","expires_in": 3600,"refresh_token": "8xLOxBtZp8","encrypted_data":
```

```
  "U2FsdGVkX1+..."}
```

完整代码实现：

```
const express = require('express');
const bodyParser = require('body-parser');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');
const crypto = require('crypto');

const app = express();
app.use(bodyParser.json());

const users = [
  {id: 1,username: 'user123',password:
'$2b$10$V1lOmywvj1xBMSHG3JnlKOm3ZP6Ph0IOfvVoF3/5ypVZonf76ENcW'
  }
];

const JWT_SECRET = 'supersecretkey';
const PRIVATE_KEY = `-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBALe...
-----END RSA PRIVATE KEY-----`;
app.post('/api/login', async (req, res) => {const { grant_type, username,
password, public_key } = req.body;
if (grant_type !== 'password') {return res.status(400).json({error:
'unsupported_grant_type',error_description: 'The grant type is not supported'
    });
  }
const user = users.find(u => u.username === username);if (!user) {return
res.status(400).json({error: 'invalid_grant',error_description: 'The user does
not exist'
    });
  }
let decryptedPassword;try {
    decryptedPassword =
crypto.privateDecrypt(PRIVATE_KEY,Buffer.from(password, 'base64')
    ).toString('utf8');
  } catch (error) {return res.status(400).json({error:
'invalid_request',error_description: 'Password decryption failed'
    });
  }
const passwordMatch = await bcrypt.compare(decryptedPassword, user.password);if
 (!passwordMatch) {return res.status(400).json({error:
'invalid_grant',error_description: 'Invalid username or password'
```

```
    });
  }
const accessToken = jwt.sign(
    { user_id: user.id, username: user.username },JWT_SECRET,
    { expiresIn: '1h' }
  );
const response = {access_token: accessToken,token_type: 'Bearer',expires_in: 3600,refresh_token: 'mock_refresh_token'
  };
const encryptedData = crypto.publicEncrypt(public_key, Buffer.from(JSON.stringify(response)));
  res.json({
    ...response,encrypted_data: encryptedData.toString('base64')
  });
});
app.listen(3000, () => {console.log('Server is running on port 3000');
});
```

代码解释：

1. **请求体解析**：使用 `body-parser` 库解析客户端发送的 JSON 格式请求体，提取出 `grant_type`、`username`、`password` 和 `public_key` 字段。
2. **授权类型验证**：检查 `grant_type` 是否为 `password`，符合 OAuth 2.0 中 `password` 模式的要求。
3. **用户验证**：根据提供的 `username` 查找用户数据库。如果未找到用户或密码不匹配，则返回错误响应。
4. **密码解密**：使用服务器端的私钥对加密的密码进行解密。
5. **JWT 令牌生成**：如果用户验证成功，使用 `jsonwebtoken` 库生成 JWT 访问令牌，并设置有效期为 1 小时。
6. **响应结果加密**：使用客户端提供的公钥对响应数据进行加密，防止中间人攻击。
7. **返回加密后的响应数据**：返回包含访问令牌、令牌类型、有效期、刷新令牌和加密后的响应数据的 JSON 响应。

前端代码示例（使用 JavaScript 和 CryptoJS 库）：

1. **引入 CryptoJS 库**：

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.0.0/crypto-js.min.js"></script>
```

1. **加密密码的函数：**

```
async function encryptPassword(password, publicKey) {const encrypt = new JSEncrypt();
  encrypt.setPublicKey(publicKey);return encrypt.encrypt(password);
}
```

1. **处理用户登录的函数：**

```
async function handleLogin() {const username = document.getElementById('username').value;const password = document.getElementById('password').value;const publicKey = 'MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA...'; // 从服务器获取的公钥const encryptedPassword = await encryptPassword(password, publicKey);
const loginData = {grant_type: 'password',username: username,password: encryptedPassword,public_key: publicKey
  };
const response = await fetch('/api/login', {method: 'POST',headers: {'Content-Type': 'application/json'
    },body: JSON.stringify(loginData)
  });
const result = await response.json();console.log(result);
}
```

1. **HTML 表单示例：**

```
<form onsubmit="handleLogin(); return false;"><label for="username">Username:</label><input type="text" id="username" name="username" required><label for="password">Password:</label><input type="password" id="password" name="password" required><button type="submit">Login</button></form>
```

代码解释：

1. **引入 CryptoJS 库**：在 HTML 文件中引入 CryptoJS 库，用于加密操作。
2. **加密密码的函数**：使用 `JSEncrypt` 库对用户输入的密码进行加密。`JSEncrypt` 是一个用于 RSA 加密的 JavaScript 库。
3. **处理用户登录的函数**：获取用户输入的用户名和密码，使用公钥对密码进行加密，然后将加密后的密码和其他登录数据发送到服务器。

4. **HTML 表单示例**：一个简单的登录表单，用户输入用户名和密码后，调用 `handleLogin` 函数进行处理。

这样，用户输入的密码在前端被加密后再发送到服务器，确保密码在传输过程中不会以明文形式出现，从而提高了安全性.

1.2.2 用户上传文件到阿里云OSS、服务器在OSS下载并转码JPG

接口地址：

```
/api/upload-and-download
```

请求方法：

```
POST
```

请求参数：

字段名	类型	描述
file	file	要上传的文件
task_name	string	创建的任务名称
user_id	string	用户 ID
model	string	选择的 model，二选一（例如 "modelA" 或 "modelB"）

请求体示例：

```
{
  "task_name": "上传并下载文件任务",
  "user_id": "user123"
}
```

响应参数：

字段名	类型	描述

status	number	响应状态码
message	string	响应信息
task_id	string	创建的任务 ID

响应体：

```
{
  "status": 200,
  "message": "任务创建成功，文件已上传并下载处理完毕",
  "task_id": "task123"
  "model": "modelA"
}
```

代码：

```
// 导入必要的库
const express = require('express'); // 导入 Express 框架
const OSS = require('ali-oss'); // 导入 ali-oss SDK 用于与阿里云 OSS 交互
const { v4: uuidv4 } = require('uuid'); // 导入 uuid 生成唯一任务 ID
const dotenv = require('dotenv'); // 导入 dotenv 用于加载环境变量
const multer = require('multer'); // 导入 multer 用于处理文件上传
const ffmpeg = require('fluent-ffmpeg'); // 导入 ffmpeg 库用于视频处理
const fs = require('fs'); // 导入文件系统模块用于文件操作
const path = require('path'); // 导入路径模块用于处理文件路径
const app = express(); // 创建 Express 应用

dotenv.config(); // 加载环境变量

// 初始化阿里云 OSS 客户端
const client = new OSS({
  region: process.env.OSS_REGION, // OSS 区域
  accessKeyId: process.env.OSS_ACCESS_KEY_ID, // 访问密钥 ID
  accessKeySecret: process.env.OSS_ACCESS_KEY_SECRET, // 访问密钥 Secret
  bucket: process.env.OSS_BUCKET // 存储空间名称
});

// 配置 multer 以处理文件上传
const storage = multer.memoryStorage(); // 使用内存存储
const upload = multer({ storage: storage }); // 创建 multer 实例

// 合并接口：用户上传文件到阿里云OSS并下载
```

```
app.post('/api/upload-and-download', upload.single('file'), async (req, res) => {
  const { task_name, user_id, model } = req.body; // 从请求体中获取参数

  if (!task_name || !user_id || !req.file) { // 如果缺少必要参数，返回错误
    return res.status(400).json({
      status: 400,
      message: '缺少必要参数'
    });
  }

// 验证 model 是否为允许的选项
    if (model !== 'modelA' && model !== 'modelB') {
        return res.status(400).json({
        status: 400,
        message: '无效的 model 参数'
         });
    }

  try {
    // 生成唯一任务 ID
    const taskId = uuidv4(); // 生成唯一任务 ID
    console.log(`任务创建: ${task_name}, 任务 ID: ${taskId}`); // 打印任务创建信息

    // 上传文件到阿里云OSS
    const result = await client.put(`uploads/${req.file.originalname}`, req.file.buffer); // 将文件上传到 OSS

    // 检查OSS返回的状态码
    if (result.res.status === 200) {
      console.log(`文件上传成功: ${result.url}`); // 打印上传成功信息

      // 下载文件
      const fileUrl = result.url; // 获取文件的 URL
      const fileName = req.file.originalname; // 获取文件名

      // 下载文件
      const downloadResult = await client.get(`uploads/${fileName}`); // 从 OSS 下载文件
      if (downloadResult.res.status === 200) {
        const fileContent = downloadResult.content; // 获取文件内容
        const tempVideoPath = path.join(__dirname, 'temp', fileName); // 创建临时视频路径
        fs.writeFileSync(tempVideoPath, fileContent); // 将视频内容写入临时文件

        // 转码为 JPG
```

```
        const tempJpgPath = path.join(__dirname, 'temp', `${path.parse(fileName).name}.jpg`); // 创建 JPG 临时路径
        ffmpeg(tempVideoPath) // 使用 ffmpeg 处理视频
          .on('end', () => {
            console.log('视频转码成功'); // 打印转码成功信息
            fs.unlinkSync(tempVideoPath); // 删除临时视频文件
          })
          .on('error', (err) => {
            console.error('视频转码失败:', err); // 打印转码失败信息
            res.status(400).json({ // 如果发生错误，返回 400 状态
              status: 400,
              message: '视频转码失败'
            });
          })
          .screenshots({
            count: 1, // 获取一帧
            folder: path.join(__dirname, 'temp'), // 存放临时 JPG 的文件夹
            filename: `${path.parse(fileName).name}.jpg` // JPG 文件名
          });

        // 返回成功响应
        res.json({
          status: 200,
          message: '任务创建成功，文件已上传并下载处理完毕',
          task_id: taskId // 返回任务 ID
        });
      } else {
        console.error('文件下载失败:', downloadResult.res.status); // 打印错误日志
        res.status(400).json({ // 如果发生错误，返回 400 状态
          status: 400,
          message: '文件下载失败'
        });
      }
    } else {
      console.error('文件上传失败:', result.res.status); // 打印错误日志
      res.status(400).json({ // 如果发生错误，返回 400 状态
        status: 400,
        message: '文件上传失败'
      });
    }
  } catch (error) {
    console.error('处理失败:', error); // 打印错误日志
    res.status(500).json({ // 如果发生错误，返回 500 错误
      status: 500,
      message: '处理失败，请稍后再试'
    });
  }
```

```
});

// 启动服务器
app.listen(3000, () => {
  console.log('Server is running on port 3000'); // 在控制台输出服务器启动信息
});

```

1.2.3 模型计算完成后上传.obj到阿里云OSS、并生成下载链接供用户下载

接口地址：

```
/api/download-model
```

请求方法：

```
POST
```

请求参数：

参数名称	类型	必填	描述
user_id	string	是	用户的唯一标识
task_id	string	是	任务的唯一标识

请求体：

```
{
  "user_id": "12345",
  "task_id": "67890"
}
```

响应参数：

参数名称	类型	描述
download_url	string	提供给前端的模型下载链接

响应体：

```
{
  "download_url": "https://oss.aliyun.com/download?task_id=67890"
}
```

代码：

```
// 引入必要的模块
const express = require('express'); // Express 框架用于创建 HTTP 服务
const OSS = require('ali-oss'); // 阿里云 OSS SDK，用于文件存储操作

// 初始化 Express 应用
const app = express();
app.use(express.json()); // 支持 JSON 格式的请求体解析

// 配置阿里云 OSS 客户端
const client = new OSS({
  region: 'oss-cn-your-region', // OSS 地区
  accessKeyId: 'your-access-key-id', // 你的阿里云 AccessKeyId
  accessKeySecret: 'your-access-key-secret', // 你的阿里云 AccessKeySecret
  bucket: 'your-bucket-name' // OSS 存储空间名称
});

// 定义下载模型的 API 路由
app.post('/api/v1/download-model', async (req, res) => {
  const { user_id, task_id } = req.body; // 从请求体中获取 user_id 和 task_id

  try {
    // 构造文件的路径，假设文件命名为 task_id 对应的 .obj 文件
    const fileName = `${task_id}.obj`;

    // 从 OSS 中获取文件的签名下载链接，有效期为 1 小时
    const downloadUrl = client.signatureUrl(fileName, {
      expires: 3600 // 链接有效时间，单位为秒，这里设置为1小时
    });

    // 返回下载链接给前端
    res.json({
      download_url: downloadUrl // 将生成的下载链接返回
    });
  } catch (err) {
    // 如果出现错误，返回错误信息
```

```
    console.error('下载模型时出错: ', err);
    res.status(500).json({ message: '生成下载链接失败' });
  }
});

// 启动服务器，监听指定端口
app.listen(3000, () => {
  console.log('服务器正在3000端口监听');
});
```

1.2.4 状态更新逻辑

1.2.4.1 OSS与服务器的状态更新

- **同步方式**：服务器在收到OSS状态码后，立即根据OSS状态码转换为服务器自定义状态码并更新任务状态。
- **状态码转换逻辑**：
  - OSS状态码 200 转换为（我们的）服务器状态码 102（进行中）或 200（已完成），具体取决于任务阶段。
  - OSS状态码 400、403、404、500 转换为服务器状态码 400（任务出错）。

1.2.5 OSS与服务器与前端的状态更新设计

1. **OSS到服务器：同步更新**
   - 任务状态实时同步到服务器，服务器在收到OSS的反馈状态码后立刻更新对应的任务状态。
   - 例如，OSS上传完成时，立即将OSS状态码200转换为服务器状态码102（进行中）或200（已完成）。
2. **服务器到前端：异步更新**
   - 通过长轮询的方式，前端定期发送请求获取服务器的任务状态更新，服务器只有在任务状态发生变化时返回更新的数据。
   - 前端通过轮询方式，每5秒向服务器发起状态检查请求。

1.2.5.1 服务器与前端的状态更新

- **长轮询实现**：前端发送请求以获取任务状态，服务器在状态更新时响应请求。
- **请求体**：前端定期（例如每5秒）发送请求以检查任务状态。
- **响应体**：根据任务状态返回不同的响应内容。

接口地址

```
POST /task/status
```

请求方法

```
POST
```

请求参数

参数名	类型	描述
task_id	string	任务的唯一标识符

请求体示例

```
{
  "task_id": "task123"
}
```

响应参数

参数名	类型	描述
task_id	string	任务的唯一标识符
status	integer	服务器定义的状态码
message	string	状态描述

响应体：

1. 任务进行中

```
{
  "task_id": "task123",
  "status": 102,
  "message": "任务进行中"
}
```

2. 任务已完成

```
{
  "task_id": "task123",
  "status": 200,
  "message": "任务已完成"
}
```

3. 任务出错

```
{
  "task_id": "task123",
  "status": 400,
  "message": "任务出错：处理过程中出现错误"
}
```

4. 任务已失效

```
{
  "task_id": "task123",
  "status": 404,
  "message": "任务已失效：超过有效期"
}
```

5. 连接成功

```
{
  "message": "连接成功"
}
```

6. 任务不存在

```
{
  "task_id": "task999",
  "status": 404,
  "message": "任务不存在：未找到指定的任务"
}
```

代码：

```
const express = require('express'); // 引入express框架
const bodyParser = require('body-parser'); // 引入body-parser中间件
const app = express(); // 创建express应用
const PORT = 3000; // 定义端口号

app.use(bodyParser.json()); // 使用body-parser解析JSON请求体

let taskStatus = {}; // 存储任务状态的对象

// 模拟任务状态更新的函数
function updateTaskStatus(task_id, status, message) {
  taskStatus[task_id] = { status, message }; // 更新任务状态
}

// 状态更新示例
updateTaskStatus('task123', 102, '任务进行中'); // 初始化任务为进行中
setTimeout(() => { // 模拟任务完成
  updateTaskStatus('task123', 200, '任务已完成');
}, 5000);

// 状态查询接口
app.post('/task/status', (req, res) => {
  const { task_id } = req.body; // 从请求体中获取任务ID
  if (taskStatus[task_id]) { // 如果任务存在
    return res.json(taskStatus[task_id]); // 返回任务状态
```

```
  }
  return res.status(404).json({ task_id, status: 404, message: '任务不存在：未找到指定的任务' }); // 返回任务不存在的响应
});

app.listen(PORT, () => { // 启动服务器
  console.log(`Server is running on port ${PORT}`);
});
```

1.2.6 ER关系表

1.2.6.1 用户表 (User)

字段名	类型	描述
user_id (PK)	string	用户唯一标识符
username	string	用户名
password	string	用户密码
email	string	用户电子邮件
created_at	timestamp	用户创建时间
updated_at	timestamp	用户信息更新时间

1.2.6.2 任务表 (Task)

字段名	类型	描述
task_id (PK)	string	任务唯一标识符
user_id (FK)	string	所属用户的唯一标识符
file_name	string	上传的文件名
oss_url	string	文件在OSS上的存储地址
task_status	integer	当前任务状态码
start_timecode	timestamp	任务开始时间
finish_timecode	timestamp	任务结束时间
created_at	timestamp	任务创建时间

updated_at	timestamp	任务更新时间

1.2.6.3 文件上传进度表 (UploadProgress)

字段名	类型	描述
progress_id (PK)	string	上传进度唯一标识符
task_id (FK)	string	任务唯一标识符
progress	integer	上传进度（百分比）
status	integer	上传状态
created_at	timestamp	上传进度记录创建时间
updated_at	timestamp	上传进度记录更新时间

1.2.6.4 模型文件表 (ModelFile)

字段名	类型	描述
model_id (PK)	string	模型文件唯一标识符
task_id (FK)	string	任务唯一标识符
model_url	string	模型文件在OSS上的存储地址
created_at	timestamp	模型文件创建时间
updated_at	timestamp	模型文件更新时间

1.2.6.5 ER 关系图

```
+-----------------+          +-----------------+          +-----------------+
|     User        |          |      Task       |          |  UploadProgress |
+-----------------+          +-----------------+          +-----------------+
| user_id (PK)    |<-------->| task_id (PK)    |<-------->| progress_id (PK)|
| username        | 1/N      | user_id (FK)    | 1/1      | task_id (FK)    |
| password        |          | file_name       |          | progress        |
| email           |          | oss_url         |          | status          |
| created_at      |          | task_status     |          | created_at      |
| updated_at      |          | start_timecode  |          | updated_at      |
+-----------------+          | finish_timecode |          +-----------------+
```

```
                                | created_at      |
                                | updated_at      |
                                +-----------------+
                                        |
                                        |1/1
                                        v
                                +-----------------+
                                |   ModelFile     |
                                +-----------------+
                                | model_id (PK)   |
                                | task_id (FK)    |
                                | model_url       |
                                | created_at      |
                                | updated_at      |
                                +-----------------+
```

1.2.7 说明

1. **用户表 (User) 和任务表 (Task) 之间的关系是 1对N**：
   - 一个用户可以拥有多个任务。
   - 每个任务只能属于一个用户。
2. **任务表 (Task) 和文件上传进度表 (UploadProgress) 之间的关系是 1对1**：
   - 每个任务对应一个上传进度记录。
   - 每个上传进度记录只能属于一个任务。
3. **任务表 (Task) 和模型文件表 (ModelFile) 之间的关系是 1对1**：
   - 一个任务对应一个模型文件。
   - 每个模型文件只能属于一个任务。