

1、系统概述

1.1、系统简介

(1) 项目背景及基本情况介绍

- 1. 时间，这位无声无息的存在，自宇宙诞生之初便存在，见证了无数变迁。人类生命虽短暂，却拥有无比珍贵的东西。时间无情地带走了我们的体力与记忆，却加深了对故人的思念。因此，我们应珍惜与亲人朋友相处的每一刻，并学会释放过去的遗憾和痛苦。
- 2. 修复老照片系统是保存珍贵记忆的重要工具。随着时间的推移，老照片逐渐模糊和受损。该系统通过先进的数字图像处理技术，为这些记忆提供了修复、保护和传承的可能。它能够去除污渍、划痕和褪色，修复破损的边缘和细节，让照片重现光彩。这不仅是对美好瞬间的延续，也是对家族文化传统的传承。
- 3. 修复老照片系统的重要性不仅在于技术价值，更在于人文意义。通过修复这些照片，我们能更好地缅怀过去，回忆与亲人共度的时光。这些照片见证了我们的成长、家庭的变迁和时代的演进，是我们情感连接的纽带和文化认同的基石。
- 4. 因此，修复老照片系统不仅是对技术的运用，更是对人文精神的传承和发扬。它让我们在时间的长河中留住珍贵的过往与情感，让记忆得以永恒。通过这个系统，我们可以更好地纪念过去，陶冶心灵，为未来的生活留下更多宝贵的回忆。
- 5. 系统开发的过程中，我们贯彻的是模块化的编程思想，首先是前端代码和后端代码的分离，对于后端主要分为两个模块一个静态修复模块，一个是动态修复模块，将训练模型，调用，接口等文件解耦合，这种设计模式有助于提高代码的易维护性和易拓展性。具体实现的过程中，我们调用了腾讯和百度两个公司的大模型，采用超分算法用于处理图片，前后端连接采用 Flask 架构，前端使用 Vue 框架搭建网站。

1.2、术语表

序号	术语或缩略语	说明性定义
1.	源图片 source_image	用户上传的图片
2.	驱动视频 drive_video	用以动作. 表情迁移的视频
3.	Flask 框架	<div>1. Flask 框架的全称是“A Python Microframework based on Werkzeug and Jinja2”，通常简称为 Flask。它是一个用 Python 编写的轻量级 Web 应用框架。Flask 的设计哲学是提供一个简单、易扩展的核心，并允许开发者根据自己的需求添加其他功能。</div> <div>2. Flask 基于两个主要库： (1) Werkzeug: 一个 WSGI (Web Server Gateway Interface) 工具库，用于处理网络请求和响应 (2) Jinja2: 一个现代且设计友好的模板引擎，用于生成 HTML 等文本输出。</div> <div>3. Flask 因其简洁、灵活和易于上手而受到许多 Python</div>

		开发者的喜爱，特别适合于快速开发小型到中型的 Web 应用。
4.	HTTP 响应	<p>1. HTTP 响应 (HTTP Response) 是指当客户端 (如 Web 浏览器) 向服务器发送 HTTP 请求后, 服务器返回给客户端的应答信息。HTTP 响应包含了服务器对请求的处理结果, 以及可能伴随的数据或状态指示。</p> <p>2. HTTP 响应主要包含以下几个部分, 状态行 (Status Line), 响应头 (Response Headers), 空行 (Blank Line), 响应体 (Response Body)。</p>
5.	@app.route	Flask 的装饰器, 用于定义路由 (URL 路径) 和处理该路径的函数。
6.	RESTful API	(Representational State Transfer Application Programming Interface), 是一种设计风格, 用于构建可扩展的 Web 服务。REST 是一种架构风格, 它定义了一组用于创建 Web 应用程序的约束。当一个服务遵循这些约束时, 它可以被称为是 RESTful 的。
7.	cors	跨域资源共享 (Cross-Origin Resource Sharing), 是一种机制, 允许在某些受限的情况下, 不同域之间的网页脚本可以相互访问资源。
8.	Vue 框架	<p>1. Vue 框架是一款轻量且强大的 JavaScript 框架, 专注于构建用户界面。它采用声明式编程和组件化开发, 使开发者能高效构建复杂的前端应用。Vue 的响应式系统能自动跟踪数据变化并更新视图, 简化开发流程。</p> <p>2. Vue.js (通常简称 Vue) 是一个用于构建用户界面的渐进式 JavaScript 框架。它由前谷歌工程师尤雨溪 (Evan You) 创建, 并于 2014 年发布。Vue 的设计目标是采用自底向上增量开发的设计, 使得它既可以用于简单的网页交互, 也可以驱动复杂的单页应用程序 (SPA)。</p> <p>3. Vue.js 的核心库专注于视图层, 并且非常容易学习, 同时也便于与第三方库或现有项目集成。Vue 还提供了配套工具, 如 Vuex 用于状态管理, Vue Router 用于路由管理, 以及 Vue CLI 用于快速搭建项目脚手架。</p>
9.	Mysql	基于 SQL 查询的开源跨平台数据库管理系统), 在我们的系统中, 用于存储用户账号中的各种信息, 并计划将其部署到云端服务器上便于远程访问。
10.	GFPGAN	<p>1. 腾讯在人像复原、超分等方面的开源模型, 我们的系统准备利用它的照片超分功能实现老照片的修复功能。</p> <p>2. GFP-GAN (Generative Facial Prior-Generative Adversarial Network) 是一种先进的图像处理技术, 特别是在人脸图像增强和修复领域。这个技术结合了生成对抗网络 (GAN) 和人脸先验知识, 以实现高质量的人脸图像恢复。</p>

		<p>3. GAN 是一种深度学习模型，由两个主要部分组成：生成器（Generator）和判别器（Discriminator）。生成器的任务是生成与真实数据相似的样本，而判别器的任务是区分生成的样本和真实样本。在训练过程中，生成器和判别器相互竞争，生成器试图生成更真实的样本以欺骗判别器，而判别器则试图提高其区分真伪的能力。这种竞争过程最终导致生成器能够生成高质量的图像。</p> <p>4. GFP-GAN 的关键创新在于它利用了人脸的先验知识，这些知识通常来自于大量的人脸数据集，包含了人脸的结构、特征点、表情等信息。这些先验知识帮助生成器更好地理解 and 重建人脸图像，即使在输入图像质量较差或部分缺失的情况下也能产生逼真的结果。</p>
11.	PaddleGAN	<p>1. 百度生成对抗网络开发的模型套件，包括多种功能，包括但不限于超分、插帧、上色、换妆、图像动画生成等功能，我们打算利用其中的动作迁移模型（first-order-demo）实现照片动起来的功能。</p> <p>2. PaddleGAN 是 百度 开 源 的 一 个 基 于 飞 桨（PaddlePaddle）深度学习平台的生成对抗网络（GAN）工具库。它提供了一系列的 GAN 模型和相关的工具，用于图像和视频的生成、编辑、修复等任务。PaddleGAN 旨在简化 GAN 模型的开发和部署，使得研究人员和开发者能够更容易地利用 GAN 技术进行各种创意和应用开发。</p>
12.	Json	JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成。它基于 JavaScript 语言的一个子集，但已经成为一种独立于语言的数据格式，广泛用于网络通信和数据存储。
13.	Element Plus	Element Plus 是一个基于 Vue.js 的组件库，它提供了一套丰富的 UI 组件，用于帮助开发者快速构建美观、易用的 Web 应用程序。它是 Element UI 的升级版本，专为 Vue 3 设计，同时也支持 Vue 2。Element Plus 继承了 Element UI 的设计理念和组件风格，同时进行了性能优化和功能增强。

1.3、系统运行环境

操作系统	Windows 64 位操作系统
数据库系统	MySQL 数据库
编程平台	本地使用 Pycharm WebStorm Vscode 进行编程
网络协议	TCP 传输控制协议 ,HTTP 协议, post get

	请求
硬件平台	Dell G15 5510 笔记本电脑, WD Elements SE 固态硬盘

1.4、开发环境

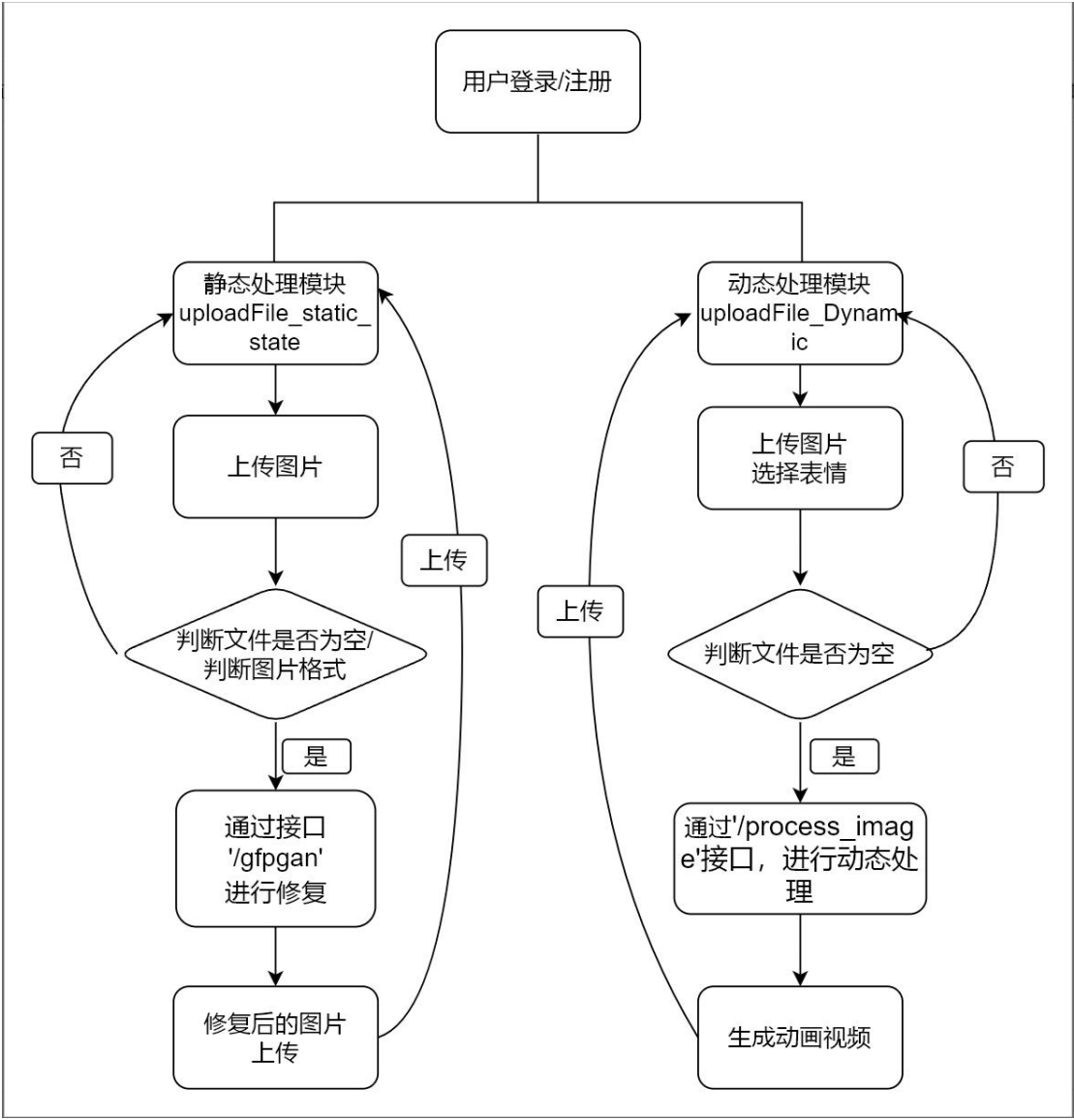
工具软件	PyCharm 2019.3.3 x64 WebStorm 2023.1 Vscode1.88.0
开发语言	Python3.7 .Vue3.2.37 .Web 开发 (HTML5 CSS3) MySQL8.0
模块(库)版本	后端: tqdm~=4.66.2 PyYAML==5.3 scikit-image~=0.20.0 scipy~=1.9.1 opencv-python==4.9.0.80 imageio==2.6.1 imageio-ffmpeg==0.4.9 librosa==0.8.1 numba==0.48.0 easydict==1.12 munch==4.0.0 natsort==8.4.0 matplotlib==3.1.3 basicsr==1.4.2 facexlib==0.3.0 lmbd==1.4.1 numpy~=1.24.4 tb-nightly==2.12.0a20230113 torch==1.13.1 torchvision==0.14.1 yapf==0.28.0 前端: @element-plus/icons-vue 2.0.9 axios 0.27.2 body-parser 1.20.2 cors 2.8.5 echarts 5.5.0 express 4.19.2 jsonwebtoken 9.0.2 md-editor-v3 2.2.1 pinia 2.0.20 vue: ^3.2.37, vue-cropperjs: ^5.0.0, vue-router: ^4.3.0,

	vue-schart: ^2.0.0, vuex: ^4.1.0, wangeditor: ^4.7.15, xlsx: ^0.18.5
--	---

2、系统说明

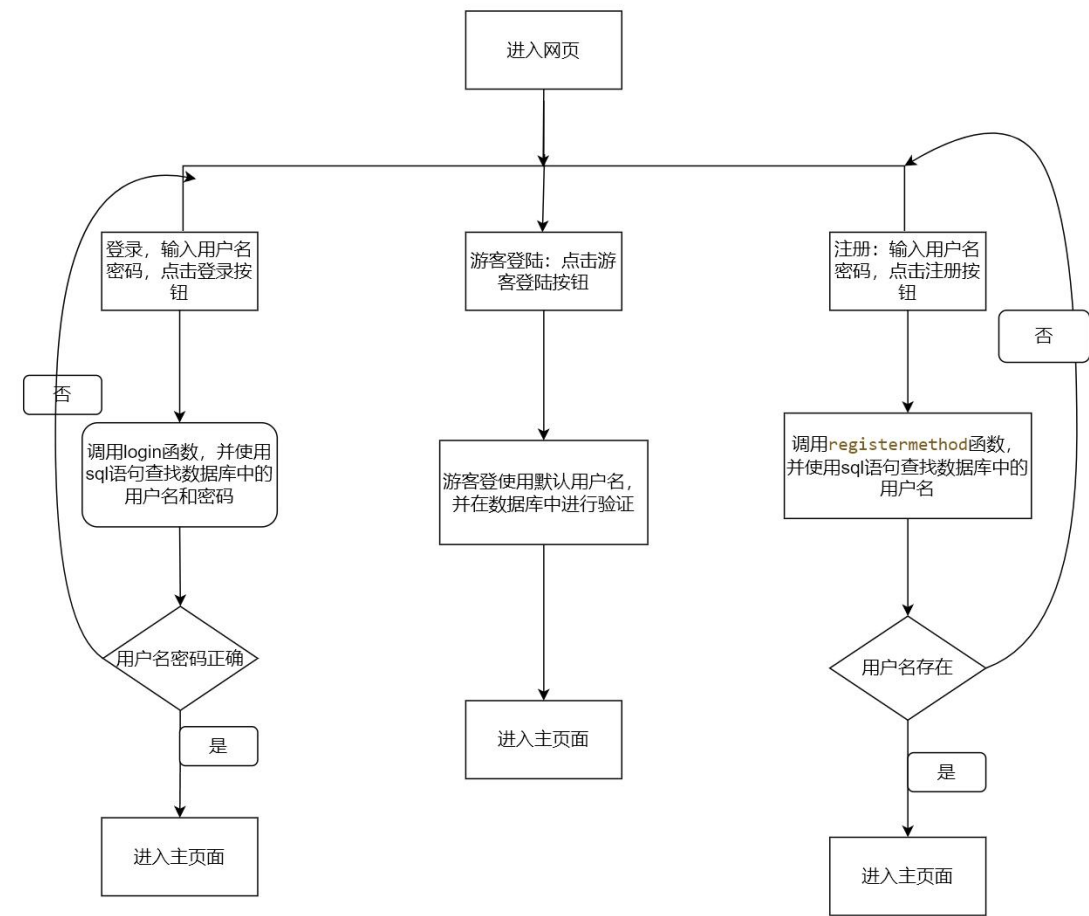
2.1、整体说明

我们的系统总体由三个模块组成，登录注册模块，静态处理模块，动态处理模块。用户登录过后，可以通过边栏选择对应模块。选用静态处理模块，用户上传自己的图片，然后图片传输到后端，后端进行超分处理然后将超分后的结果上传到网页。选用动态处理模块，用户上传照片，并选择自己想要的实现的表情，网页会将图片和用户的选择传到后端，后端再通过驱动视频生成动画，并返回生成的动画视频给网页。



2.2、模块 1 登录注册模块

算法流程



2.2.1、功能描述

本模块主要实现用户登录，注册，以及游客登陆功能的实现，以及与数据库的连接。

2.2.2、使用说明

介绍该模型的输入（输入数据、有效性检测规则等），输出信息（输出信息、表现形式、信息意义等）。

输入数据：

用户名：username，校验规则：不能为空，不能超过 50 个字符

密码：password，校验规则：不能为空，不能超过 50 个字符

密码验证：confirmpassword，校验规则：是否与密码一致

输出信息：

登陆成功注册成功：alert(' 登录成功' / '注册成功')

登陆注册失败：alert(' 注册/登录失败')

用户名密码错误：alert(' 用户名密码错误')

用户名存在：alert(' 用户名已存在', '注册失败')

意义：提示用户操作

表现形式：



2.2.3、重要代码分析

结合算法，对模块中的重要代码进行分析。

1. 数据库连接代码

```
const mysql = require('mysql');  
  
// 2 创建链接配置  
const db = mysql.createPool({  
  host: 'localhost',  
  user: 'root',  
  password: '',  
  database: 'gan'  
})  
  
module.exports = db
```

配置数据库的连接地址 localhost，用户名 root，密码 password，以及所用数据库的名字 gan，然后导出数据库的实例 db

2. 数据库后端登录函数

```

exports.login = (req, res) => {
  var sql = 'SELECT * FROM user WHERE username = ? AND password = ?';
  console.log("req.query.username:", req.body.username, req.body.password)
  db.query(sql, [req.body.username, req.body.password], (err, result) => {
    console.log("result.length:", result.length)
    if(err) {
      return res.send({
        status: 400,
        message: "登录失败"
      })
    }
    if(result.length === 1) {
      res.send({
        status: 200,
        message: "登录成功"
      })
    } else {
      res.send({
        status: 202,
        message: '用户名或密码错误'
      })
    }
  })
}

```

该函数的参数为 `req`（前端请求），`res`（后端返回值），`login` 函数先定义 `sql` 的查询语句，通过 `req` 获取前端用户输入的用户名和密码。使用数据库实例，调用 `query` 函数使用 `sql` 语句查询用户输入的用户名和密码是否存在，若 `query` 函数调用失败则证明连接数据库失败，若 `query` 函数有返回值，这证明用户名密码正确，返回登陆成功的信息给前端，若 `query` 函数无返回值，则说明用户名密码错误，并返回相应的值给前端。

3. 数据库后端注册函数


```
exports.register = (req, res) => {  
  const sql1 = 'SELECT * FROM user WHERE username = ?;'  
  const sql2 = 'insert into user (username, password) value (?, ?)'  
  
  db.query(sql1, [req.body.username], (err, result) => {  
    console.log("result:", result)  
    if(err) {  
      return res.send({  
        status: 500,  
        message: "操作失败"  
      })  
    }  
    if(result.length > 0) {  
      return res.send({  
        status: 202,  
        message: '用户名已存在'  
      })  
    }else{  
      db.query(sql2, [req.body.username, req.body.password], (err, result) => {  
        if(err) {  
          return res.send({  
            status: 400,  
            message: "注册失败"  
          })  
        }  
        res.send({  
          status: 200,  
          message: "注册成功"  
        })  
      })  
    }  
  })  
}
```

```
}
```

注册函数 `register`，跟 `login` 一样有两个参数，`req`（前端请求），`res`（后端返回值），先定义了两个 sql 语句，一个查找，一个插入。`register` 函数先根据 `req` 得到前端用户输入值，用数据库的实例化执行 `sql1` 语句，验证用户输入的用户名是否重复。若没有重复，再次调用 `query` 函数执行 `sql2` 语句，插入用户注册的用户名和密码到数据库里。在上述不同的情况中，返回不同的 `res` 给前端。

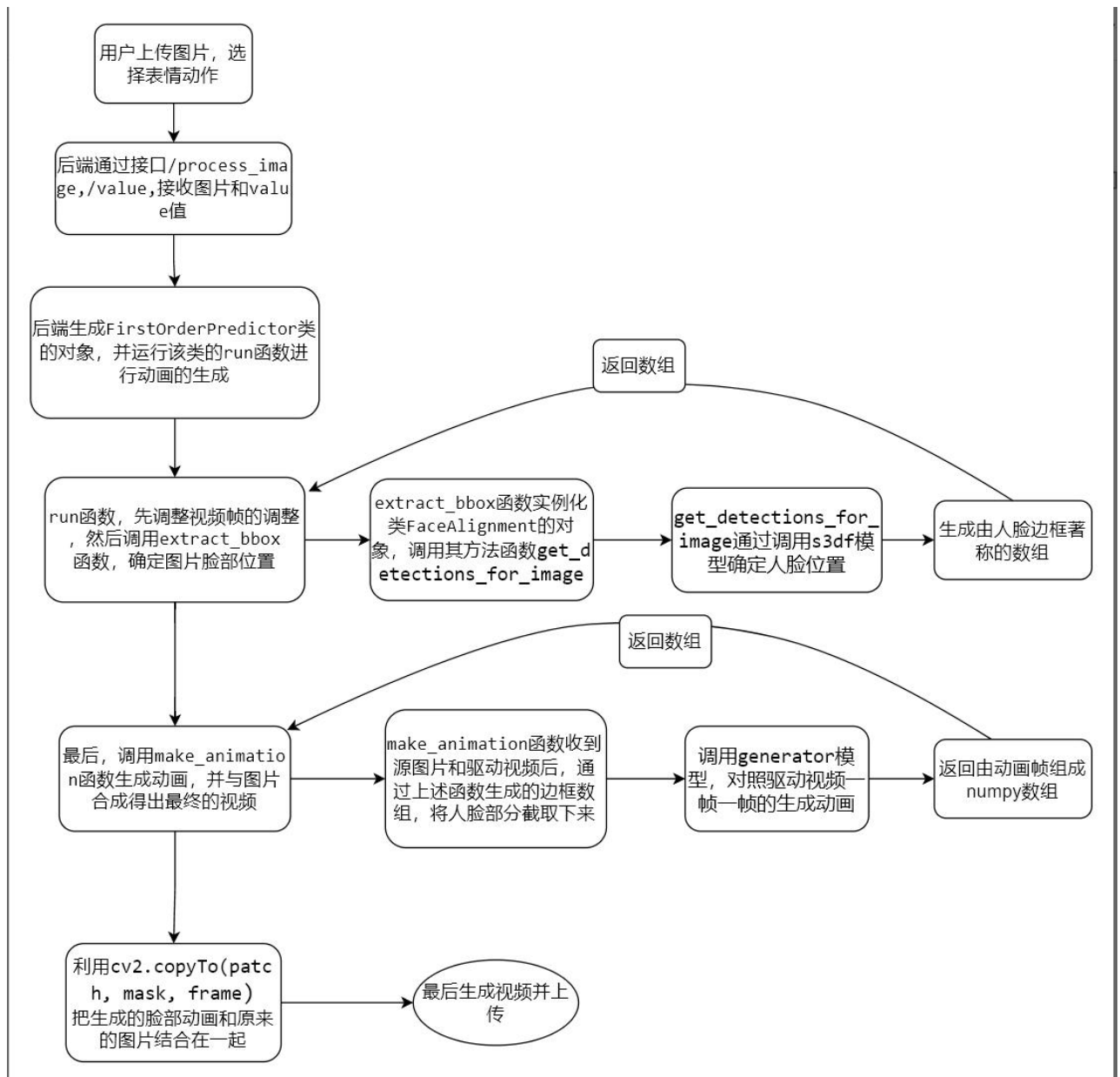
4. 前端登录注册函数

```
async loginmethod() {  
  try {  
    // 发送登录请求  
    axios.defaults.baseURL = "http://localhost:3000"  
    const response = await axios.post('/login', {  
      username: this.Loginform.username,  
      password: this.Loginform.password  
    });  
    console.log("response.data.status", response.data.status)  
    // 根据响应状态处理结果  
    if (response.data.status === 200) {  
      alert('登录成功')  
      axios.defaults.baseURL = "http://localhost:5000"  
      this.toPage();// 跳转到登录页面  
      .....  
    }  
  }  
}
```

前端登录函数与注册函数思路相同，先更改 axios 访问路径，使他的访问端口为后端数据库服务器的运行端口，使用 `axios.post` 方法访问数据库后端函数 login，并传输用户的输入，更具后端函数不同的返回值，抛出不同的信息框 `alert('登录成功')`，并跳转到不同的页面 `this.toPage()`；

2.3、模块 2 动态处理模块

算法流程：



2.3.1、功能描述

结合前端上传操作，后端根据用户的选择，对用户上传的图片进行不同的动态处理，生成不同的表情动画。

2.3.2、使用说明

输入数据

value: 与单选组件 radio 相连接，根据不同的选项，获得不同的 value 值

对于上传的 value 值，代码会进行以处理：

接口方式：'/value'，methods=['POST']，后端接口以及请求方式

引入全局变量：global value

定义变量 data 获取 json 变量：request.json。

获取传输值：value = data.get('selectedValue')

1. 图片：用户上传与需要动态处理的的图片

对于上传的图片文件，代码会进行以下有效性检验：

接口方式：'/process_image'，methods=["GET","POST"]

引入全局变量：global value

文件部分：'file' not in request.files 确保请求中包含文件部分。

文件名：file.filename == '' 检查文件名是否为空。

文件名不为空，对于 '/process_image'，接口中的图片路径，将直接存入 UPLOAD_FOLDER 指向的文件夹中

输出数据

输出根据上传图片 and value 值生成的动态视频。

定义结果保存的路径 result_file_path，

通过 base64.b64encode，将视频重新编码，

生成 json 字典，并发往前端，json.dumps({"result_base64": result_base64})

2.3.3、重要代码分析

结合算法，对模块中的重要代码进行分析。

1. 前端上传函数，前端 uploadFile_Dynamic 文件中

上传图片部分：

```
<el-upload class="upload-demo" drag :limit="1" :show-file-list="true"
          action="http://127.0.0.1:5000/process_image"
          v-loading.fullscreen.lock="fullscreenLoading" multiple
          :on-success="dealimage" :on-error="uploadFileError"
          @change="handleUploadChanged">
    <el-icon class="el-icon--upload"><upload-filled /></el-icon>
    <div class="el-upload__text">
      将图片拖到此处，或
      <em>点击上传</em>
    </div>
</el-upload>
```

前端组件连接了三个事件。

```
const fullscreenLoading = ref(false)
const openFullScreen1 = () => {
  fullscreenLoading.value = false
  // setTimeout(() => {
```

```

    // fullscreenLoading.value = false
    // }, 2000)
}

const dealimage = (response: any, file: any, fileList: any) => {
    console.log(response)
    console.log(response.result_base64)
    videoValue.value = response.result_base64
};

const uploadFileError = (err: any, file: any, fileList: any) => {
    console.log("上传失败")
}

```

用户通过上传框，上传图片后，fullscreenLoading 作为一个响应式引用，控制全屏加载动画的显示。dealimage 函数是上传成功后的回调函数，将 videoValue 引用的值设置为结果视频 result_base64。这意味着 videoValue 用于在 Vue 组件中显示生成的动画。

openFullScreen1 函数用于关闭全屏加载动画，但当前的实现直接将

2. 前端 value 值传输函数

```

const radioValue = ref<number | null>(null); // 使用 ref 来定义响应式数据

const updateRadioValue = () => {
    radioValue.value = radio.value;
};

const sendToBackend = async () => {
    console.log("value:",radioValue.value)
    if (radioValue.value !== null) { // 确保有一个值被选中
        try {
            const response = await axios.post('http://127.0.0.1:5000/value', {
                selectedValue: radioValue.value, // 将选中的值发送到后端
            });
            // 处理响应...
            console.log("发送成功");
            console.log(response.data);
        } catch (error) {

```

```

        // 处理错误...

        console.error(error);

    }

} else {

    console.log('请先选择一个选项! ');

}

};

```

`radio.value` 的值与单选框绑定在一起，随着用户的选着而改变自身的值，通过 `sendToBackend` 方法将 `value` 之传入后端，后端可以根据不同的 `value` 值选择不同的驱动视频，以生成不同的动画效果。

3. 后端接收图片和 `value` 值

```
app = Flask(__name__)
```

```
CORS(app)
```

```
value=0
```

```
app.config['UPLOAD_FOLDER'] =
```

创建一个 Flask 应用实例。

使用 `cors(app)` 初始化跨域资源共享（CORS），允许跨域请求。

定义全局变量 `value`

定义上传图片的储存路径 `UPLOAD_FOLDER`

```
@app.route('/value', methods=['POST'])
```

```
def value():
```

```
    data = request.json
```

```
    global value
```

```
    value = data.get('selectedValue')
```

```
    print("value:", value)
```

```
return jsonify({'message': 'Received selected value: {}'.format(value)})
```

建立你后端服务器接口 `/value` 接收 `value` 值，应用全局变量，将前端传入的 `value` 值赋值给全局变量，并返回相应信息

```
@app.route('/process_image', methods=["GET", "POST"])
```

```
def process_image():
```

```
    global value
```

```
    file.save(os.path.join(app.config['UPLOAD_FOLDER'], file.filename))
```

```
    default_filepath = app.config['UPLOAD_FOLDER'] + '/' + file.filename
```

建立后端接口 `/process_image` 接收前端传来的图片，并将其保存到指定路径 `UPLOAD_FOLDER`

4. 后端上传结果视频

```
result_file_path =
```

```
r'D:\python\project-finally\GFPGAN\GFPGAN\applications\output\result.mp4'
```

```
with open(result_file_path, 'rb') as file:
```

```
    result_data = file.read()
```

```
    # 将文件内容转换为 base64 编码
```

```
result_base64 = 'data:video/mp4;base64,' +
```

```
base64.b64encode(result_data).decode('utf-8')
```

```
    # 输出 base64 编码后的结果
```

```
result_json = json.dumps({"result_base64": result_base64})
```

```
return result_json
```

生成的动画视频将被保存到 `result_file_path` 结果保存路径里。然后后端通过 `with open` 函数访问结果视频，并使用 `base64.b64encode` 修改视频编码形式，一 `json` 字典的形式将结果视频传回前端网页并显示

5. run 函数

位于 ppgan 文件夹中的 first_order_predicter 文件中，功能：调用其他函数模块，根据驱动视频生成图片的动态处理动画。参数：source_image（用户上传的图片），driving_video（用户选的表情对应的驱动视频）。

```
driving_video = [

    cv2.resize(frame, (self.image_size, self.image_size)) / 255.0

    for frame in driving_video

]

results = []
```

run 函数首先通过 cv2 模块读取视频，并改变视频帧的大小，使得其与图片大小归一化处理，定义结果数组 results，生成的动画结果将会保存到这里

```
bboxes = self.extract_bbox(source_image.copy())

print(str(len(bboxes)) + " persons have been detected")
```

调用 extract_bbox 函数检测人脸，并返回人脸所在区域边框的数组，并打印出检测出了几张人脸。bboxes 五个元素的数组 [x1, y1, x2, y2, area]，其中 (x1, y1) 是边界框左上角的坐标，(x2, y2) 是右下角的坐标，area 是边界框的面积若有多张人脸 bboxes 为多维数组

```
for rec in bboxes:

    face_image = source_image.copy()[rec[1]:rec[3], rec[0]:rec[2]]

    face_image = cv2.resize(face_image,

                            (self.image_size, self.image_size)) / 255.0

    predictions = get_prediction(face_image)

    results.append({
```

```

        'rec':

        rec,

        'predict':

        [predictions[i] for i in range(predictions.shape[0])]

    })

```

```

if len(bboxes) == 1 or not self.multi_person:

```

```

    break

```

循环遍历 bboxes，为所有的人脸生成动画。face_image 为通过 copy 函数截取的图片的人脸面部部分，将其大小归一化处理后，通过调用 get_prediction 函数，调用函数 make_animation 对照视频每一帧生成每个人脸的动画，并将动画结果存入 results 中

```

for i in range(len(driving_video)):

```

```

    frame = source_image.copy()

```

```

    for result in results:

```

```

        x1, y1, x2, y2, _ = result['rec']

```

```

        h = y2 - y1

```

```

        w = x2 - x1 //计算边框大小

```

```

        out = result['predict'][i]

```

```

        out = cv2.resize(out.astype(np.uint8), (x2 - x1, y2 - y1))

```

```

    if len(results) == 1:

```

```

        frame[y1:y2, x1:x2] = out

```

```

        break

```

```

    else:

```

```

patch = np.zeros(frame.shape).astype('uint8')

patch[y1:y2, x1:x2] = out

mask = np.zeros(frame.shape[:2]).astype('uint8')

cx = int((x1 + x2) / 2)

cy = int((y1 + y2) / 2)

cv2.circle(mask, (cx, cy), math.ceil(h * self.ratio),

            (255, 255, 255), -1, 8, 0)

frame = cv2.copyTo(patch, mask, frame)

```

外层循环嵌套层循环，外层循环为选用的驱动视频 driving_video 的帧数内层循环为 results（每个人脸生成的动画帧）计算人脸边框大小 h, w 后，读取动画帧，并调整动画帧的大小，存储到 out 中，若 results 列表中只有一个人脸的预测结果，那么直接将调整大小后的动画帧 out 合成到 frame 中相应的边界框区域，并退出内层循环。如果视频中有多个脸，它将使用遮罩确保每个人脸动画帧只替换相应边界框内的区域。

```

out_frame.append(frame)

imageio.mimsave(os.path.join(self.output, self.filename),

                [frame for frame in out_frame],

                fps=fps)

```

将外层循环中合成的当前帧 frame 添加到 out_frame 列表中。out_frame 用于存储整个视频序列中每一帧的合成结果。在处理完所有的帧之后，这个列表包含了整个视频的所有帧。最后使用 imageio.mimsave 函数来保存视频。imageio 是一个用于读写图片和视频的 Python 库。

6. extract_bbox 函数

生成类 FaceAlignment 的对象，通过调用类 FaceAlignment 的 get_detections_for_batch 方法调用类 SFDDetector 的方法，使用 s3fd 模型，进行人脸位置的识别，返回人脸边框组成的数组 [x_min, y_min, x_max, y_max,

```
confidence]
```

x_min 和 y_min 是边界框左上角的坐标。

x_max 和 y_max 是边界框右下角的坐标。

confidence 是一个表示检测到的人脸置信度的值。

```
def extract_bbox(self, image):
```

```
    detector = face_detection.FaceAlignment(
```

```
        face_detection.LandmarksType._2D,
```

```
        flip_input=False,
```

```
        face_detector=self.face_detector)
```

这里创建了一个 FaceAlignment 实例，它是用于人脸检测和对齐的类

```
    frame = [image]
```

```
    predictions = detector.get_detections_for_image(np.array(frame))
```

图像被添加到一个列表中，然后转换为 NumPy 数组，传递给 detector 的 get_detections_for_image 方法进行人脸检测。

```
    person_num = len(predictions)
```

```
    if person_num == 0:
```

```
        return np.array([])
```

获取检测到的人脸数量，并检查是否有人脸被检测到。如果没有检测到人脸，则返回一个空的 NumPy 数组。

```
    results = []
```

```
    face_boxes = []
```

创建两个列表，results 用于存储初步的边界框和相关计算结果，face_boxes 用于存储最终的边界框。

```
    h, w, _ = image.shape
```

```
    for rect in predictions:
```

```
bh = rect[3] - rect[1]

bw = rect[2] - rect[0]

cy = rect[1] + int(bh / 2)

cx = rect[0] + int(bw / 2)
```

遍历每个检测到的人脸，根据检测结果 `rect` 计算边界框的坐标。`rect` 包含了人脸区域的原始坐标，代码计算了人脸的高度 `bh` 和宽度 `bw`，以及人脸中心的坐标 `cy` 和 `cx`。

```
margin = max(bh, bw)

y1 = max(0, cy - margin)

x1 = max(0, cx - int(0.8 * margin))

y2 = min(h, cy + margin)

x2 = min(w, cx + int(0.8 * margin))

area = (y2 - y1) * (x2 - x1)

results.append([x1, y1, x2, y2, area])
```

为边界框添加一个外边界 `margin`，这个边界是人脸高度和宽度中较大的一个。然后计算边界框的新坐标 (`x1`, `y1`, `x2`, `y2`)，确保它们不会超出图像的边界。

```
sorted(results, key=lambda area: area[4], reverse=True)

results_box = [results[0]]

for i in range(1, person_num):

    num = len(results_box)

    add_person = True

    for j in range(num):

        pre_person = results_box[j]
```

```

        iou = self.IOU(pre_person[0], pre_person[1], pre_person[2],
                        pre_person[3], pre_person[4], results[i][0],
                        results[i][1], results[i][2], results[i][3],
                        results[i][4])

        if iou > 0.5:

            add_person = False

            break

    if add_person:

        results_box.append(results[i])

```

对 results 列表按面积降序排序，并初始化 results_box 列表，首先添加面积最大的边界框。然后遍历剩余的边界框，使用交并比（IoU）检查它们是否与 results_box 中的已有边界框重叠过多（IoU > 0.5）。如果不重叠，则添加到 results_box 中。

```

boxes = np.array(results_box)

return boxes

```

7. make_animation 函数

算法原理：函数 extract_bbox 读取视频后和源图后，则将驱动视频分割成多个批次以进

行批量处理，使用 kp_detector 检测源图像和驱动视频中的关键点，遍历驱动视频的每一帧，将源图像的关键点与当前帧的关键点结合起来，通过 generator 生成动画帧。最后返回由动画帧组成的 NumPy 数组。

```

predictions = []

source = paddle.to_tensor(source_image[np.newaxis]).astype(
    np.float32).transpose([0, 3, 1, 2])

driving_video_np = np.array(driving_video).astype(np.float32)

```

```
driving_n, driving_h, driving_w, driving_c = driving_video_np.shape
```

将源图像和驱动视频转换为适合模型输入的格式

```
driving_slices = []
```

```
# whole driving as a single slice
```

```
driving = paddle.to_tensor(
```

```
    np.array(driving_video).astype(np.float32)).transpose(
```

```
        [0, 3, 1, 2])
```

```
frame_count_in_slice = driving_n
```

```
driving_slices.append(driving)
```

则将整个驱动视频作为一个批次。

```
kp_source = kp_detector(source)
```

```
kp_driving_initial = kp_detector(driving_slices[0][0:1])
```

使用 `kp_detector` 检测源图像和驱动视频中的关键点。

```
kp_source_batch = {}
```

```
kp_source_batch["value"] = paddle.tile(
```

```
    kp_source["value"], repeat_times=[self.batch_size, 1, 1])
```

```
kp_source_batch["jacobian"] = paddle.tile(
```

```
    kp_source["jacobian"], repeat_times=[self.batch_size, 1, 1, 1])
```

```
source = paddle.tile(source,
```

```
    repeat_times=[self.batch_size, 1, 1, 1])
```

为了适应批量处理，重复关键点数据以匹配批量大小。

```
begin_idx = 0
```

```

for frame_idx in tqdm(
    range(int(np.ceil(float(driving_n) / self.batch_size))):

    frame_num = min(self.batch_size, driving_n - begin_idx)

    slice_id = int(frame_idx * self.batch_size /

                    frame_count_in_slice)

    internal_start = frame_idx - slice_id * frame_count_in_slice

    internal_end = frame_idx - slice_id * frame_count_in_slice + frame_num

    driving_frame = driving_slices[slice_id][

        internal_start:internal_end]

    kp_driving = kp_detector(driving_frame)

    kp_source_img = {}

    kp_source_img["value"] = kp_source_batch["value"][0:frame_num]

    kp_source_img["jacobian"] = kp_source_batch["jacobian"][

        0:frame_num]

```

遍历驱动视频的每一帧，生成动画帧，从驱动视频中提取当前批次的帧，并使用关键点检测器对这些帧进行处理。它计算批次索引、批次内的帧数和帧索引，并从预先处理好的源图像关键点数据中提取相应数量的关键点数据。这些信息将用于后续的动画帧生成过程。

```

kp_norm = normalize_kp(

    kp_source=kp_source,

    kp_driving=kp_driving,

    kp_driving_initial=kp_driving_initial,

```



```

        use_relative_movement=relative,

        use_relative_jacobian=relative,

        adapt_movement_scale=adapt_movement_scale)

```

根据相对运动和适应运动比例的设置，归一化关键点数据

```

out = generator(source[0:frame_num],

                kp_source=kp_source_img,

                kp_driving=kp_norm)

img = np.transpose(out['prediction'].numpy(),

                  [0, 2, 3, 1]) * 255.0

```

使用 generator 模型和归一化的关键点数据生成动画帧。

```

        predictions.append(img)

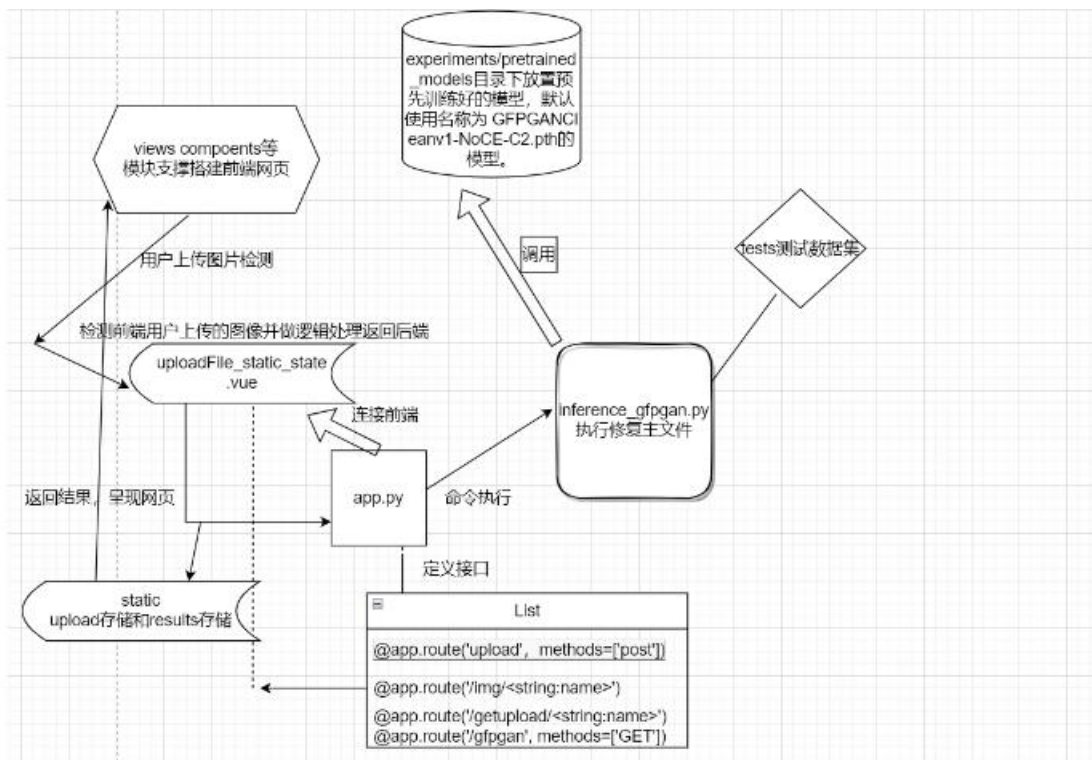
        begin_idx += frame_num

return np.concatenate(predictions)

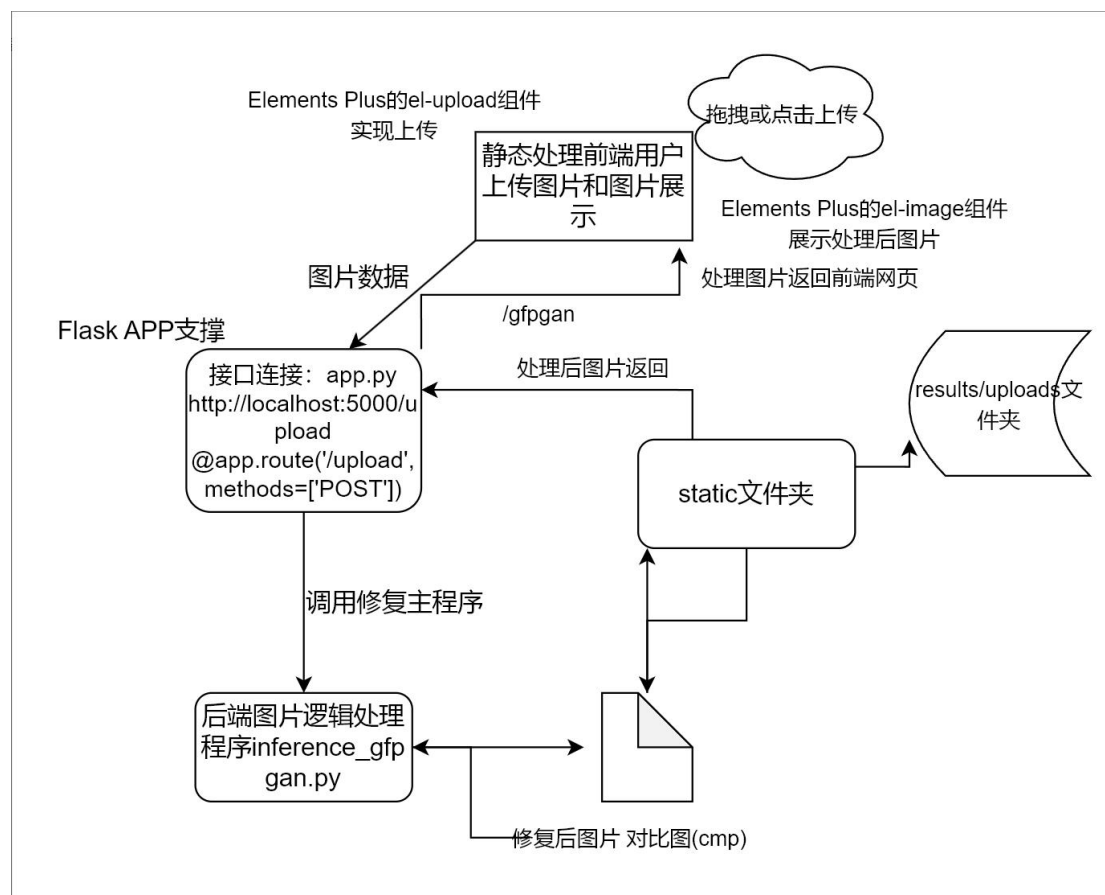
```

2.4、模块3 静态处理模块

模块流程:



算法流程:



2.4.1、功能描述

结合前端上传操作，后端根据用户上传的照片，对用户上传的图片进行静态人像的修

复效果处理最后将得到的结果，即成功修复之后的图像返回给前端网页，供用户进行保存。

2.4.2、使用说明

(1)输入：

1. 图片：用户上传与需要处理的的图片
2. 文件路径：gfpgan 接口中输入的图片路径

对于上传的图片文件，代码会进行以下有效性检验：

文件部分： `'file' not in request.files` 确保请求中包含文件部分。

文件名： `file.filename == ''` 检查文件名是否为空。

对于 gfpgan 接口中的图片路径，代码会进行以下有效性检验：

路径是否存在： `not os.path.exists(input_path_full)` 检查提供的路径是否实际存在。

图片格式的有效性检验：直接将上传的图片文件与 JPG 和 PNG 格式进行比较。如果上传的文件格式不符合要求，将提示用户选择正确的图片格式。

从物理模型中的数据库表获取数据的相关信息如下：

在 `upload_pic` 路由中，图片文件被保存到 `UPLOAD_FOLDER` 目录下的一个临时文件中。

在 `show_img` 路由中，从 `current_selectimage` 路径下获取显示的图片数据。

在 `get_upload` 路由中，从 `current_uploadimage` 路径下获取上传的图片数据。

在 `gfpgan` 路由中，根据输入的图片路径从文件系统中读取图片，并使用 `image_handle` 处理图片，处理后的结果存储在 `static/images/results` 目录下，并通过 `result_image` 返回。

(2)输出：

1. 图片数据：输出根据上传图片生成的修复图片。
2. 响应数据：在处理图片或其他操作完成后，会返回相应的 JSON 数据，包括处理结果、输入图片的链接和输出图片的链接等。

下面以计算机之父为例子，展示一下，来复活一下这位伟大人物的笑貌！

2.4.3、重要代码分析

1. 先来分析一下 `app.py` 这个项目文件。这段代码是一个 Flask Web 应用程序，用于处理图像上传、显示和图像恢复操作。

```
from flask import Flask, request, jsonify, make_response

import os

import subprocess

from PIL import Image

import time

import uuid

from werkzeug.utils import secure_filename
```

```
from extension import db, cors
```

```
from inference_gfpgan2 import image_handle
```

```
app = Flask(__name__)
```

```
cors.init_app(app) # 注册跨域请求伪造相关
```

1. 初始化和配置

创建一个 Flask 应用实例。

使用 `cors.init_app(app)` 初始化跨域资源共享（CORS），允许跨域请求。

获取当前文件的绝对路径和所在目录，用于后续文件操作。

```
# 获取当前文件的绝对路径
```

```
current_file_path = os.path.abspath(__file__)
```

```
print("current_file_path", current_file_path)
```

```
# 获取当前文件所在的目录
```

```
project_root_path = os.path.dirname(current_file_path)
```

```
print("current_dir", project_root_path)
```

```
UPLOAD_FOLDER = r'static\images\uploads'
```

```
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
```

```
output_path = "custom_results"
```

```
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

```
app.config['current_selectimage'] = os.path.join(project_root_path, output_path,  
"restored_imgs")
```

```
app.config['current_uploadimage'] = os.path.join(project_root_path, "uploads")
```

2. 上传文件夹设置

定义上传文件夹 `UPLOAD_FOLDER` 和输出结果文件夹 `output_path`。

使用 `os.makedirs(UPLOAD_FOLDER, exist_ok=True)` 确保上传目录存在。

在 Flask 配置中设置上传文件夹和选择图片、上传图片的路径。

```

@app.route('/upload', methods=['POST'])

def upload_pic():

    print("进行上传")

    if 'file' not in request.files:

        return jsonify({'error': '没有文件部分'}), 400

    file = request.files['file']

    if file.filename == "":

        return jsonify({'error': '没有选择文件'}), 400

    if file:

        filename = secure_filename(file.filename).split('.')[0] + "_" +
str(int(time.time())) + '.' + secure_filename(file.filename).split('.')[1]

        print("filename:", filename)

        file_path =

os.path.join(r'D:\python\project-finally\GFPGAN\GFPGAN\static\images\uploa
ds', filename)

        file.save(os.path.join(project_root_path, file_path))

        return jsonify({'message': '文件上传成功', 'path': file_path}), 200

```

3. 图像上传处理

定义/upload 路由，处理 POST 请求，实现图像上传功能。

检查请求中是否包含文件部分，如果没有返回错误。

使用 secure_filename 确保文件名安全。

生成基于时间戳的唯一文件名，避免同名文件冲突。

保存文件到指定路径，并返回成功消息和文件路径。

```

@app.route('/img/<string:name>', methods=['GET'])

```

```

def show_img(name):

    img_url = os.path.join(app.config['current_selectimage'], name)

    if name:

        image_data = open(img_url, "rb").read()

        response = make_response(image_data)

        response.headers['Content-Type'] = 'image/jpg'

    return response

@app.route('/getupload/<string:name>', methods=['GET'])

def get_upload(name):

    img_url = os.path.join(app.config['current_uploadimage'], name)

    if name:

        image_data = open(img_url, "rb").read()

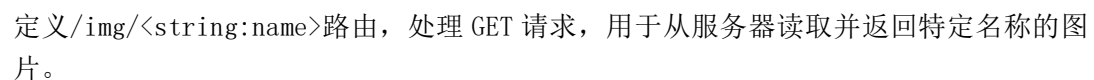
        response = make_response(image_data)

        response.headers['Content-Type'] = 'image/jpg'

    return response

```

4. 图像显示处理

定义路由，处理 GET 请求，用于从服务器读取并返回特定名称的图片。

定义/getupload/<string:name>路由，处理 GET 请求，用于从上传目录读取并返回特定名称的图片。

两个路由都使用 make_response 创建响应对象，并设置内容类型为 image/jpg。

```

@app.route('/gfpgan', methods=['GET'])

def gfpgan():

    input_path = request.args.get('input')

    input_path_full = os.path.join(project_root_path, input_path)

```

```

if not input_path_full:

    return jsonify({'error': 'Input path is required'}), 400

getupload =

f"http://localhost:5000/getupload/{os.path.basename(input_path)}"

handle_res = image_handle({

    'input': input_path_full,

    'output': 'static/images/results',

})

result_image = f"http://127.0.0.1:5000/{handle_res[0]}"

return jsonify({'message': 'Face restoration completed', 'input': getupload,

'output': result_image}), 200

if __name__ == '__main__':

    app.run(debug=True, port=5000)

```

5. 图像恢复处理

定义/gfpgan 路由，处理 GET 请求，调用 image_handle 函数进行图像恢复。

从请求参数中获取输入路径。

调用 image_handle 函数处理图像，返回处理结果的 URL。

返回一个包含成功消息、输入图片 URL 和输出图片 URL 的 JSON 对象。

6. 辅助函数

image_handle 函数（在 inference_gfpgan2 模块中定义）用于执行实际的图像恢复算法。

7. 运行应用

如果直接运行此脚本，则启动 Flask 应用，开启调试模式，并设置监听端口为 5000。

代码亮点

安全性：使用 secure_filename 来清理文件名，防止安全问题。

唯一性：通过时间戳生成唯一文件名，避免文件覆盖。

模块化：图像恢复逻辑被封装在 image_handle 函数中，易于维护和测试。

跨域支持：通过 CORS 支持跨域请求，提高 Web 服务的灵活性。

潜在改进

错误处理：可以增加更全面的错误处理，例如处理文件读写权限问题。
配置管理：使用环境变量或配置文件来管理路径和端口等配置。
性能优化：对于图像处理，可以考虑使用异步任务队列来提高响应性能。
安全性增强：增加对上传文件类型的检查，限制文件大小，防止恶意文件上传。

2. 再来分析一下 `uploadFile_static_state.vue` 这个项目文件，这段代码是一个使用 Vue 3 和 Element Plus UI 框架构建的单文件组件（SFC），用于实现图片上传和展示检测结果的功能。以下是对重要代码的分析：

模板部分（<template>）

容器结构：使用 `<div class="container">` 定义了整个组件的容器。

图片数据上传：

使用 `<el-upload>` 组件实现拖拽上传功能，限制最多上传 1 张图片，并且显示文件列表。

`action="http://localhost:5000/upload"` 指定了上传的后端 API 地址。

`v-loading.fullscreen.lock="fullscreenLoading"` 用于在上传时显示全屏加载动画。

检测结果展示：

使用 `<el-image>` 组件来展示图片，`:src="url"` 绑定了要展示的图片 URL。

`:preview-src-list="srcList"` 提供了图片预览列表，允许用户查看不同的图片。

```
<template>

  <div class="container"><div>

    <div class="content-title">图片数据上传</div>

    <div class="plugins-tips">

      <!--      <a href="https://element-plus.org/zh-CN/component/upload.html"
target="_blank">Element Plus Upload</a>-->

      <p style="line-height: 20px">
        建议上传的图片格式为 JPG & PNG </p></div>

    <el-upload
      class="upload-demo"
      drag
      :limit="1"
      :show-file-list="true"
      action="http://localhost:5000/upload"
      v-loading.fullscreen.lock="fullscreenLoading"
      multiple
      :on-success="dealimage"
      :on-error="uploadFileError">

      <!--      action 是将图片 file 文件直接上传到后端服务器上，要想将图片附带的数据信息上
传必须要实现额外函数 -->

      <el-icon class="el-icon--upload"><upload-filled /></el-icon>
```



```

<div class="el-upload__text">
  将图片拖到此处，或
  <em>点击上传</em>
</div>
</el-upload>
<div>
  <div class="content-title">检测结果</div>
  <div class="demo-image__preview">
    <el-image
      style="width: 100px; height: 100px"
      :src="url"
      :zoom-rate="1.2"
      :max-scale="7"
      :min-scale="0.2"
      :preview-src-list="srcList"
      :initial-index="4"
      fit="cover" /> </div> </div></div> </div></template>

```

脚本部分 (<script>)

导入依赖：

导入了 Vue 的 ref、reactive、onMounted 等 API。

导入 Element Plus 的组件和方法，例如 ElMessage、ElMessageBox。

响应式状态：

query：一个响应式对象，用于存储分页和搜索相关的状态。

pageTotal：一个响应式引用，用于存储总页数。

fullscreenLoading：控制全屏加载动画的显示。

url：用于存储当前展示图片的 URL。

srcList：存储图片 URL 列表，用于图片预览。

生命周期钩子：

onMounted：组件挂载后执行的生命周期钩子，当前为空。

上传处理：

dealimage：图片上传成功后的回调函数，用于触发后端处理并更新图片 URL。

uploadFileError：图片上传失败时的回调函数，打印上传失败信息。

API 请求：

使用 axios 库在 dealimage 函数中发送 GET 请求到/gfpagan，传递上传的图片路径，处理完成后更新图片展示。

```

<script lang="ts" setup>
import {ref, reactive, onMounted} from 'vue';

```

```

import { ElMessage, ElMessageBox } from 'element-plus';
import { Delete, Edit, Search, Plus } from '@element-plus/icons-vue';
import axios from "axios";

const query = reactive({
  curPage: 1,
  pageSize: 10,
  tableName: "file",
  keyword: ""
});

const pageTotal = ref(0);
const fullscreenLoading = ref(false)
const openFullScreen1 = () => {
  fullscreenLoading.value = false
  // setTimeout(() => {
  //   fullscreenLoading.value = false
  // }, 2000)
}

const url = ref('../src\\assets\\img\\null.png')
const srcList = [
  "http://localhost:5000/img/10045_1710698433.png", ""
]

onMounted(() => {
  // getNum()
})

// 本地文件上传相关操作
const dealimage = (response: any, file: any, filelist: any) => {
  console.log(response.path)
  fullscreenLoading.value = true
  axios.get(
    `/gfpagan`, {
      params: { input: response.path }
    }
  )
}

```

```

).then(res=>{
  if(res.data.message == 'Face restoration completed'){
    console.log(1919810)

    fullscreenLoading.value = false

    url.value = res.data.output

    srcList[0] = res.data.output
    srcList[1] = res.data.input
  }
})
};

const uploadFileError=(err:any, file:any, fileList:any)=>{
  console.log("上传失败")
}

```

<el-upload>组件是实现文件上传的核心，它通过属性配置了上传的行为和样式。fullscreenLoading 的使用是提升用户体验的关键，它在上传过程中提供了加载反馈。dealimage 函数是处理上传后图片的关键函数，它通过调用后端 API 并处理响应来更新图片 URL。srcList 数组用于存储用于预览的图片 URL 列表，它与<el-image>组件结合使用，允许用户在不同的图片间切换预览。

3. 我们最后分析一下 inference_gfpgan.py 这个文件，这段代码是一个 Python 脚本，用于使用 GFPGAN (Generative Facial Prior GAN) 模型进行图像恢复的推理演示。

1. 解析命令行参数

使用 argparse 库来解析命令行参数，允许用户指定输入输出路径、模型版本、放大比例等选项。

```

parser = argparse.ArgumentParser()
# ... 添加参数 ...
args = parser.parse_args()

```

2. 准备输入和输出

确定输入图像的路径，可以是单个文件或文件夹，并将输出目录创建为 results (或用户指定的路径)。

```

if os.path.isfile(args.input):

```

```

    img_list = [args.input]

```

```

else:

```

```

    img_list = sorted(glob.glob(os.path.join(args.input, '*')))

```

```
os.makedirs(args.output, exist_ok=True)
```

3. 设置背景上采样器

根据用户选择，设置背景上采样器。如果使用 RealESRGAN，且 GPU 可用，则初始化 RealESRGAN 模型。

```
if args.bg_upsampler == 'realesrgan':  
    # ... 初始化 RealESRGAN 模型 ...
```

4. 设置 GFPGAN 恢复器

根据用户指定的版本选择 GFPGAN 模型，并下载（如果本地不存在）或加载预训练模型。

```
if args.version == '1':  
    # ... 设置不同版本的 GFPGAN 模型 ...  
  
    restorer = GFPGANer(  
  
        model_path=model_path,  
  
        upscale=args.upscale,  
  
        arch=arch,  
  
        channel_multiplier=channel_multiplier,  
  
        bg_upsampler=bg_upsampler)
```

5. 图像恢复

遍历所有输入图像，使用 GFPGAN 恢复器进行图像恢复，并将恢复的图像保存到指定的输出目录。

```
for img_path in img_list:  
  
    # ... 读取图像 ...  
  
    input_img = cv2.imread(img_path, cv2.IMREAD_COLOR)  
  
    # ... 恢复图像 ...  
  
    cropped_faces, restored_faces, restored_img = restorer.enhance(  
  
        input_img,  
  
        # ... 参数 ...)  
    # ... 保存恢复的图像 ...
```

6. 保存恢复结果

将恢复的人脸图像、裁剪的图像和比较图像保存到磁盘。

```
# save faces
```

```
for idx, (cropped_face, restored_face) in enumerate(zip(cropped_faces,
restored_faces)):

    # ... 保存裁剪和恢复的人脸图像 ...

    imwrite(cropped_face, save_crop_path)

    imwrite(restored_face, save_restore_path)

    # ... 保存比较图像 ...
```

7. 主函数入口

如果脚本被直接运行，则调用 `main()` 函数。

```
if __name__ == '__main__':
    main()
```

关键点分析

参数解析：允许用户灵活地指定不同的配置选项。

输入输出处理：支持单个图像或批量图像处理，自动创建输出目录。

模型选择：根据用户选择加载不同版本的 GFPGAN 模型。

背景上采样器：可选地使用 RealESRGAN 对背景进行上采样，提高最终图像质量。

图像恢复：使用 GFPGAN 模型对输入图像进行人脸恢复。

结果保存：将恢复的图像和相关信息保存到磁盘，供用户查看。

3、功能测试

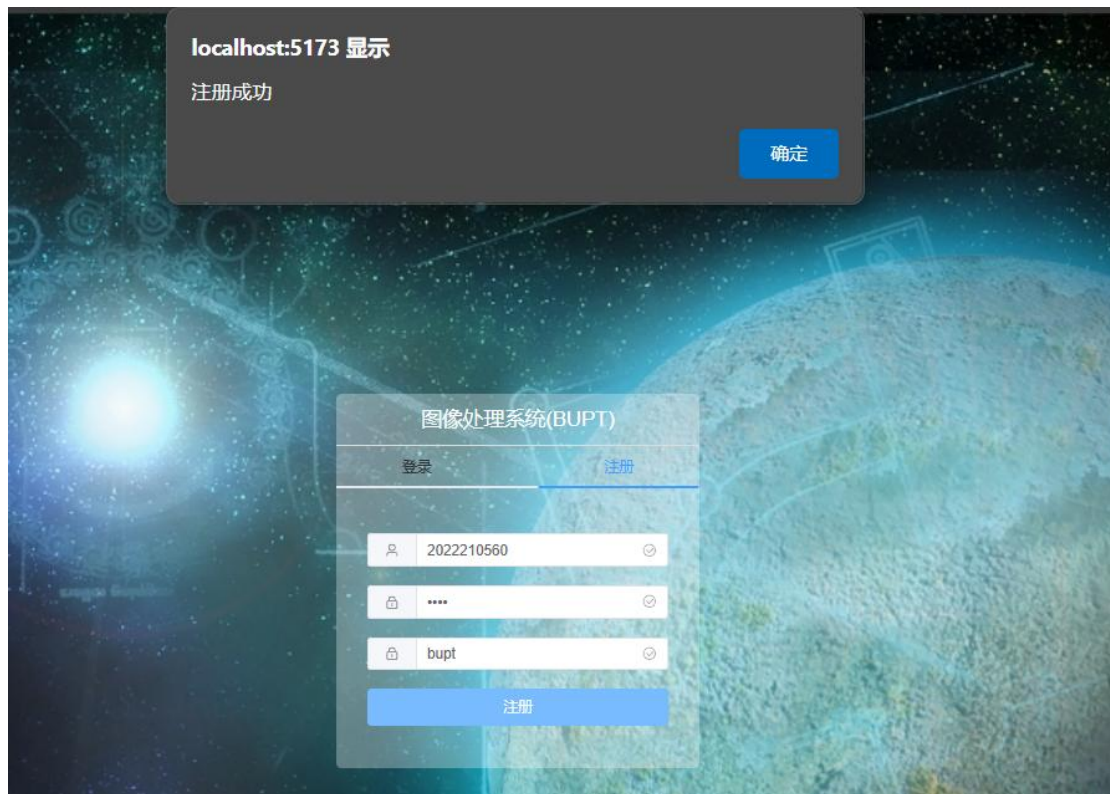
3.1、模块1 登录注册模块/功能1 实现用户的账号密码登录.注册.游客登录的基本功能

3.1.1、测试用例

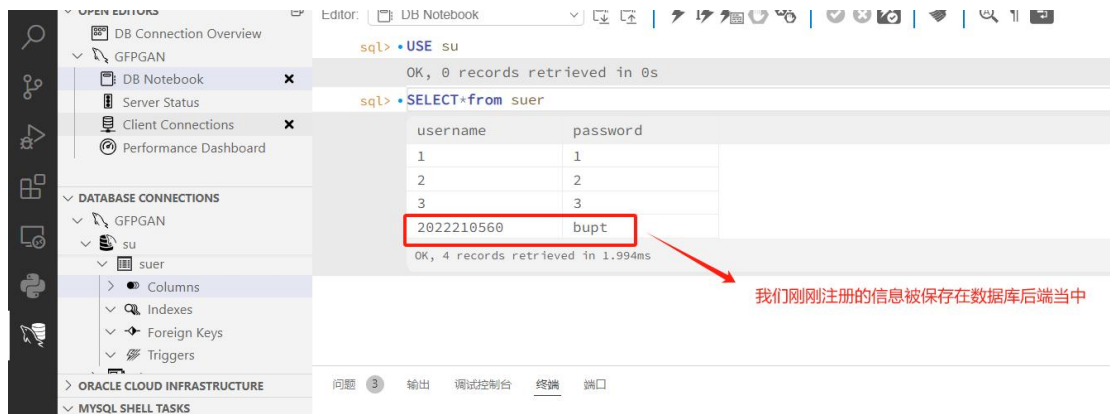
- (1) 我们先进行注册的测试，用户名为 2022210560 密码为 bupt
- (2) 我们再利用我们刚才注册的账号进行登录操作
- (3) 我们再进行游客登录操作

3.1.2、测试结果

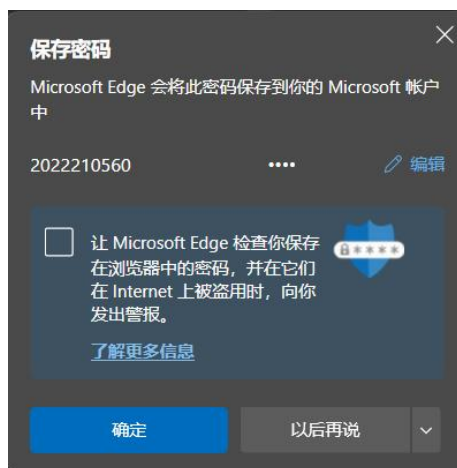
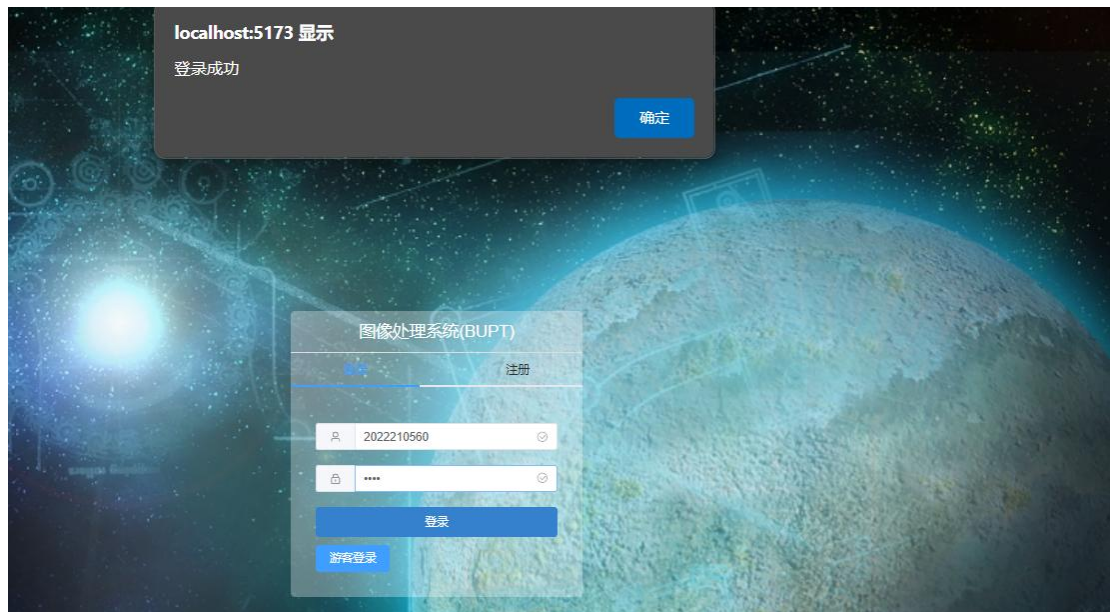
1. 进行注册测试，使用我们刚才设置的注册测试用例用户名为 2022210560 密码为 bupt
2. 如下图所示，我们测试是成功的.



3. 下面我们再来检测一下后台数据库中的信息，来检验一下, 如下图。可以正确的查询到我们刚才注册的信息。



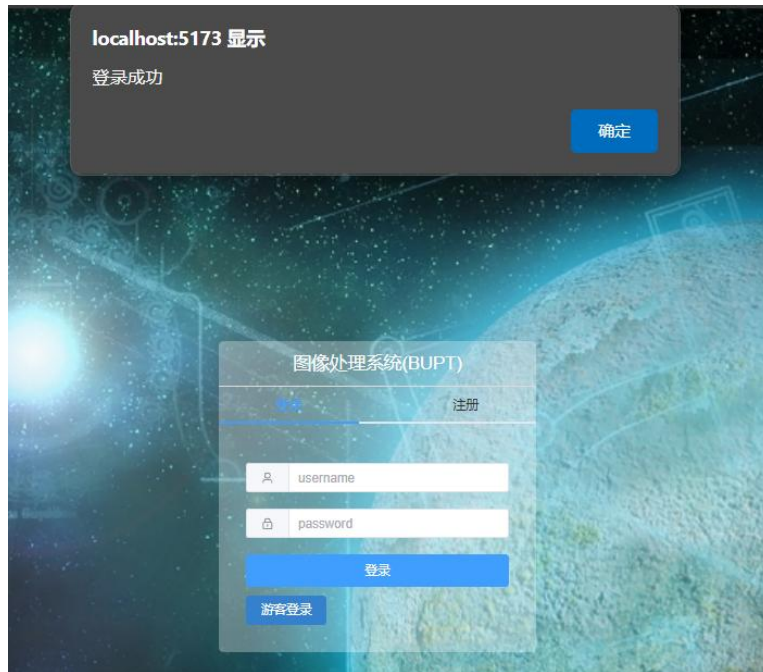
4. 接下来我们进行，登录测试，就是使用我们刚才注册的信息进行登录操作. 结果如下图所示。也可以与浏览器进行关联，如下图。



```
PS D:\python\project-finally\gfpVUE\gfpVUE> cd .\server\  
PS D:\python\project-finally\gfpVUE\gfpVUE\server> node .\app.js  
服务器启动成功  
req.query.username: 1 1  
result.length: 1  
result: []  
req.query.username: 2022210560 bupt  
result.length: 1  
[]
```

可见测试成功，用户可以当输入正确的账号和密码时进行登录网站。同时后端调试台也能检测到相关的信息。

5. 进行游客登录测试，如下图。



3.1.3、结果分析

经过分析，登录注册模块能够呈现出很好的效果，能够完成基本的功能。

3.2、模块 2 动态处理模块/功能 1 实现图像中人脸的动态修复效果/用户可以选择不同的动作效果迁移

3.2.1、测试用例

本电脑模拟用户在网页进行上传图片的操作，采用上传的图片是我们的计算机之父阿兰·麦席森·图灵 (Alan Mathison Turing)，我们希望重现这位伟大的科学家的笑貌。

具体图片详见 3.3.1 中图片。

3.2.2、测试结果

1. 微笑眨眼 (value=3) 效果测试：

(1) 首先前端能够正确实现用户正常地上传图片，如上图所示。



(2) 并且用户前端上传的图片能够正确的存储在指定的文件夹当中。

2. 困惑 (value=6) 效果测试:

由于上面我们已经展示的很详细了，我们只展示 1 中的 (3) 部分。

(3) 后端检测到前端用户上传的图片，并且根据用户选择的 value 值，选择对应的驱动视频调用模型进行处理图片。

```
Success!
6
[05/23 22:15:04] ppgan INFO: Found C:\Users\18722\.cache\ppgan\vox-cpk-512.pdparams
[05/23 22:15:08] ppgan INFO: Found C:\Users\18722\.cache\ppgan\GPEN-512.pdparams
D:\python\project-finally\GFPGAN\GFPGAN\applications\input\source_image\tuling_restore.jpg
D:\python\project-finally\GFPGAN\GFPGAN\applications\input\driving_video\困惑.mp4
True
True
True
1 persons have been detected
100%|██████████████████████████████████████████████████████████████████████████████| 133/133 [1:09:55<00:00, 31.55s/it]
[2024-05-23 23:25:28,333] [ WARNING ] _io.py:561 - IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from
(1000, 750) to (1008, 752) to ensure video compatibility with most codecs and players. To prevent resizing, make your input image divisible by the
macro_block_size or set the macro_block_size to 1 (risking incompatibility).
[swscaler @ 00000225ffc40500] Warning: data is not aligned! This can lead to a speed loss
[2024-05-23 23:25:33,169] [ INFO ] _internal.py:96 - 127.0.0.1 - - [23/May/2024 23:25:33] "POST /process_image HTTP/1.1" 200 -
```

3. 点头 (value=9) 效果测试:


(3) 后端检测到前端用户上传的图片，并且根据用户选择的 value 值，选择对应的驱动视频调用模型进行处理图片。

```
Success!
9
[05/23 23:30:19] ppgan INFO: Found C:\Users\18722\.cache\ppgan\vox-cpk-512.pdparams
[05/23 23:30:23] ppgan INFO: Found C:\Users\18722\.cache\ppgan\GPEN-512.pdparams
D:\python\project-finally\GFPGAN\GFPGAN\applications\input\source_image\Tuling_restore.jpg
D:\python\project-finally\GFPGAN\GFPGAN\applications\input\driving_video\同意点头.mp4
True
True
True
1 persons have been detected
100% |██████████████████████████████████████████████████████████████████████████| 121/121 [2:28:32<00:00, 73.66s/it]
[2024-05-24 01:59:15,318] [ WARNING ] _io.py:561 - IMAGEIO FFmpeg_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from
(1000, 750) to (1008, 752) to ensure video compatibility with most codecs and players. To prevent resizing, make your input image divisible by the
macro_block_size or set the macro_block_size to 1 (risking incompatibility).
[suscaler @ 000002b5da0ae0500] warning: data is not aligned! This can lead to a speed loss
[2024-05-24 01:59:20,043] [ INFO ] _internal.py:96 - 127.0.0.1 - - [24/May/2024 01:59:20] "POST /process_image HTTP/1.1" 200 -
```

4. 摇头(value=12)效果测试:

(3) 后端检测到前端用户上传的图片, 并且根据用户选择的 value 值, 选择对应的驱动视频调用模型进行处理图片。

```
D:\python\project-finally\GFGPGAN\applications\input\source_image\tuling_restore.jpg  
D:\python\project-finally\GFGPGAN\GFGPGAN\applications\input\driving_video\摇头.mp4  
  
True  
True  
True  
  
1 persons have been detected  
100% ████████████████████████████████████████████████████████████ 116/116 [2:33:19<00:00, 79.30s/it]  
[2024-05-24 16:52:26,194] [ WARNING ] _io.py:561 - IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from  
(1000, 750) to (1008, 752) to ensure video compatibility with most codecs and players. To prevent resizing, make your input image divisible by the  
macro_block_size or set the macro_block_size to 1 (risking incompatibility).  
[swscaler @ 0000028e9c4f0500] Warning: data is not aligned! This can lead to a speed loss  
[2024-05-24 16:52:27,340] [ INFO ] internal.py:96 - 127.0.0.1 - - [24/May/2024 16:52:27] "POST /process_image HTTP/1.1" 200 -
```



5. 如果用户不进行选择, 则为默认驱动视频:

(3) 后端检测到前端用户上传的图片，并且根据用户选择的 value 值，选择对应的驱动视频调用模型进行处理图片。

```
D:\python\project-finally\GFGPGAN\GFGPGAN\applications\input\source_image\Tuling_restore.jpg
D:\python\project-finally\GFGPGAN\GFGPGAN\applications\input\driving_video\驱动视频.MOV
True
True
True
1 persons have been detected
100%|██████████████████████████████████████████████████████████████████████████| 251/251 [2:09:49<00:00, 31.03s/it]
[2024-05-24 11:53:44,955] [ WARNING ] _io.py:561 - IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_size=16, resizing from
(1000, 750) to (1008, 752) to ensure video compatibility with most codecs and players. To prevent resizing, make your input image divisible by the
macro_block_size or set the macro_block_size to 1 (risking incompatibility).
[swscaler @ 0000259496d0500] Warning: data is not aligned! This can lead to a speed loss
[2024-05-24 11:53:46,789] [ INFO ] _internal.py:96 - 127.0.0.1 - - [24/May/2024 11:53:46] "POST /process_image HTTP/1.1" 200 -
```

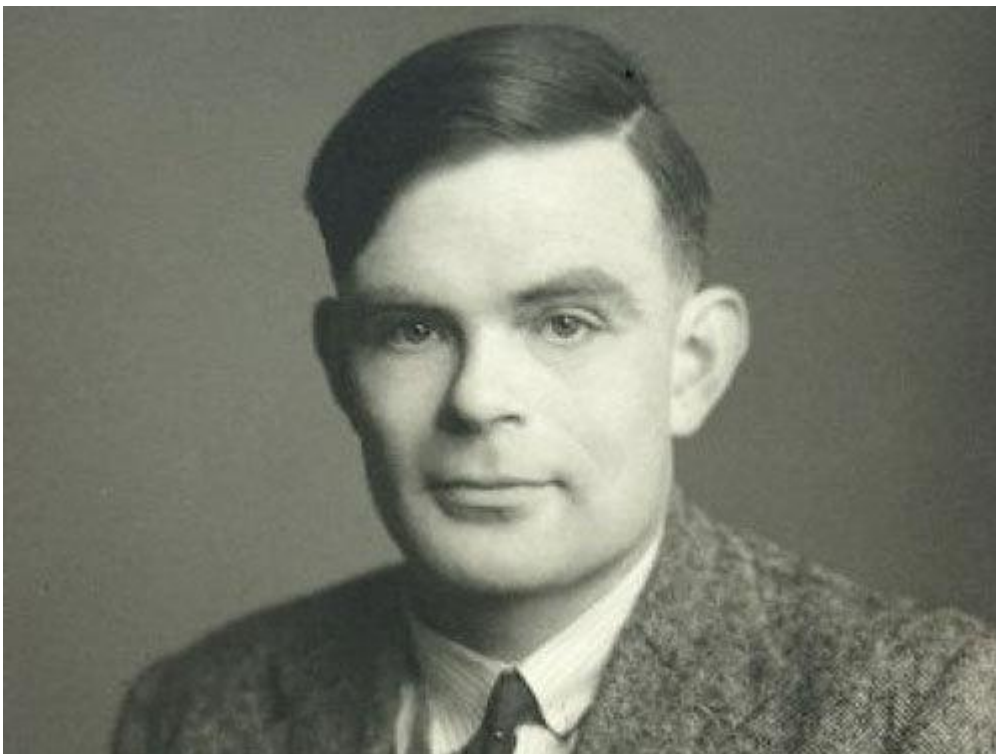
3.2.3、结果分析

分析得到的结果，特别是测试结果与预计结果不相符合的样例，应重点分析。将得到的图片动态视频与原来的驱动视频做对比，可见我们将驱动视频的呈现出来的效果还是不错的，但是算法参数还需要进一步地调整，可见视频的两侧部分几乎还是呈现出原来老照片的背景效果，这就使得最终生成的视频看起来两侧部分比较生硬，融洽度不够。同时脸部匹配的不是最佳，个别细节部分，比如耳朵等部分，存在些许偏移和变形，但整体上效果达到基本要求。

3.3、模块3 静态处理模块/功能1 实现用户上传图片并且成功返回修复之后的图片.

3.3.1、测试用例

本电脑模拟用户在网页进行上传图片的操作，采用上传的图片是我们的计算机之父阿兰·麦席森·图灵(Alan Mathison Turing),我们希望重现这位伟大的科学家的笑貌。下面进行测试分析:我们先来展示以下原图片:



不难看到原图片的人脸部分还是有一点点模糊，细节方面还不是特别清晰。下面我们进行修复.

3.3.2、测试结果

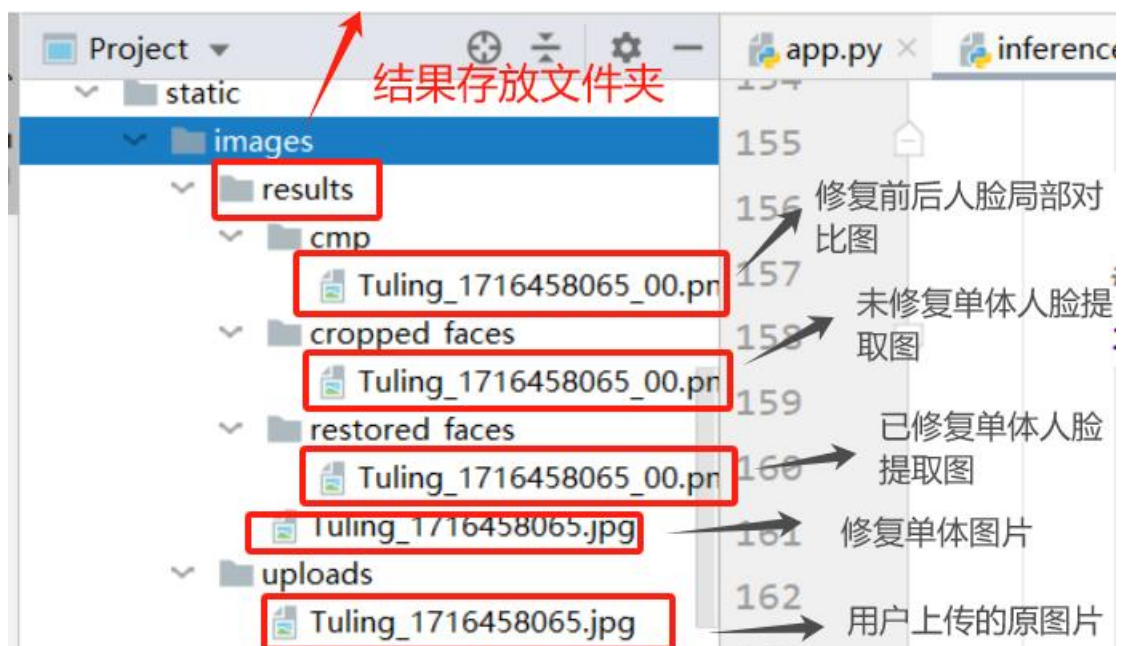
1. 能够实现正常的用户上传图片，并且后端能够检测到用户的操作，并且将上传的图片存储到对应的 static/images/uploads 这个指定的文件夹之下。并且附上上传的时间戳。如下图:

图片数据上传

建议上传的图片格式为JPG & PNG



用户输入(上传)图片



2. 后端程序能够正常的对前端用户的操作进行响应, 如下图:


```
Terminal: Local Local (2) +
* Restarting with stat
current_file_path D:\python\project-finally\GFPGAN\GFPGAN\app.py
current_dir D:\python\project-finally\GFPGAN\GFPGAN
* Debugger is active!
* Debugger PIN: 119-145-315
127.0.0.1 - - [23/May/2024 17:54:25] "OPTIONS /upload HTTP/1.1" 200 -
进行上传
filename: Tuling_1716458065.jpg
127.0.0.1 - - [23/May/2024 17:54:25] "POST /upload HTTP/1.1" 200 -
D:\python\project-finally\GFPGAN\GFPGAN\inference_gfpgan2.py:48: UserWarning: The unoptimized RealESRGAN is slow on CPU. We do not use it. If you really want to use it, please modify the corresponding codes.
warnings.warn('The unoptimized RealESRGAN is slow on CPU. We do not use it. ')
D:\ProgramData\envs\su\lib\site-packages\torchvision\models\utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
warnings.warn(
D:\ProgramData\envs\su\lib\site-packages\torchvision\models\utils.py:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavior is equivalent to passing `weights=None`.
warnings.warn(msg)
Processing Tuling_1716458065.jpg ...
127.0.0.1 - - [23/May/2024 17:54:34] "GET /gfpgan?input=D:\\python\\project-finally\\GFPGAN\\GFPGAN\\static\\images\\uploads\\Tuling_1716458065.jpg HTTP/1.1" 200 -
127.0.0.1 - - [23/May/2024 17:54:34] "GET /static/images/results/Tuling_1716458065.jpg HTTP/1.1" 200 -
```

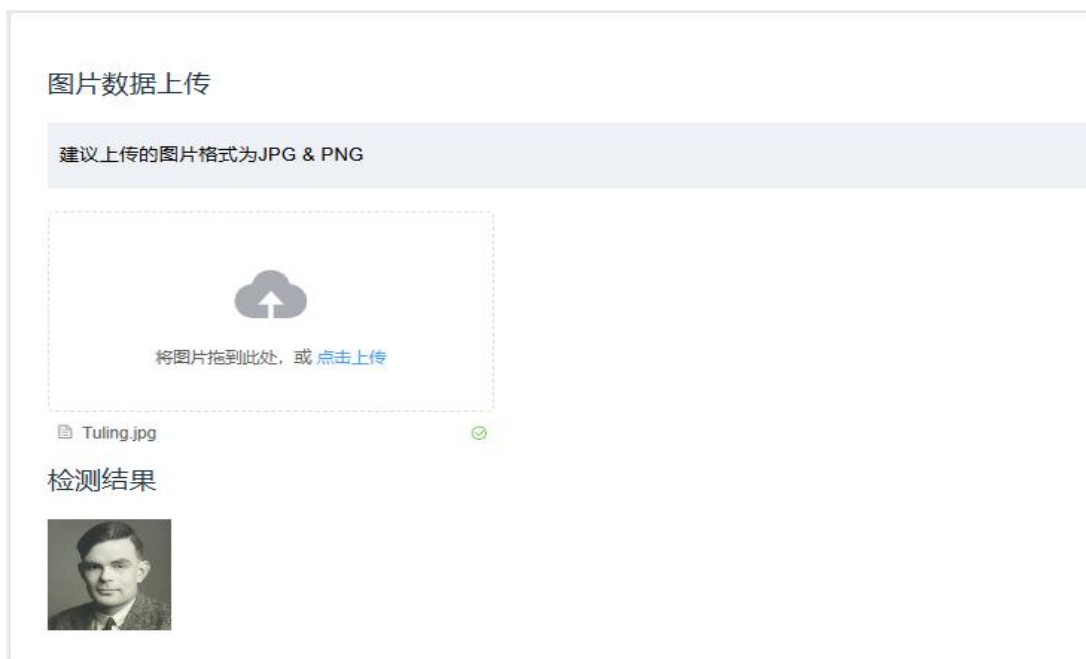
→ 后端程序成功执行

→ 检测到用户在指定时间在前端进行了图片上传

→ 用户进行上传，依据时间戳给图片命名

→ 调用模型，执行修复代码，并且进行存储

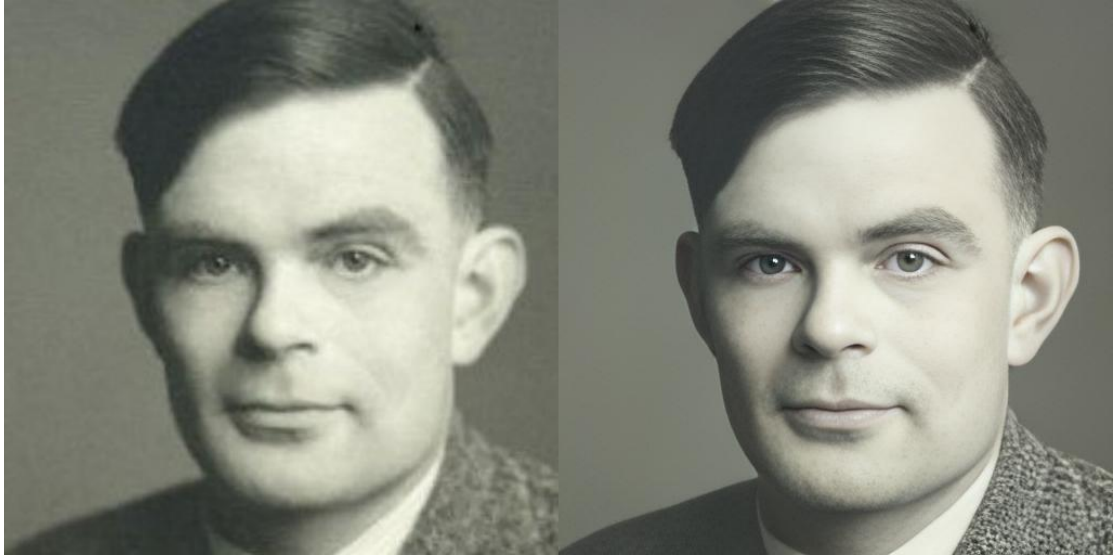
3. 前端网页能够正常的呈现出后端修复后的图片



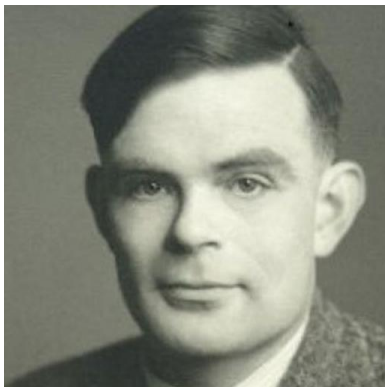
3.3.3、结果分析

我们将结果进行分析如下：

(1) 修复前后人脸局部对比图（cmp）：



(2) 未修复单体人脸提取图(cropped_faces):



(3) 已修复单体人脸提取图(restored_faces):



(4) 最终呈现给用户的最终修复图:



经过分析对比测试前后的图片效果，我们可以看到右侧修复后的图片在细节和清晰度上有了显著的提升。首先，从面部特征来看，修复后的图片（右侧）使得男子的脸部特征更加清晰，包括眼睛、鼻子和嘴巴等部位的轮廓都更为鲜明。这使得观者能够更加容易地辨认出图片中人物的细节，增加了图片的可读性和辨识度。其次，从光线和色调上来看，修复后的图片显得更加明亮，黑色和白色的对比更加鲜明，这使得整个图片在视觉效果上更加吸引人。这种明亮的光线处理不仅凸显了人物的面部特征，还增强了图片的整体质感。此外，从服装和发型上来看，修复后的图片在细节上也有所改善。男子的西装和领带更加清晰，毛衣在修复后不再出现在图片中，这使得图片的层次感更加丰富。同时，男子的发型也变得更加蓬松，这可能与修复技术中对头发细节的增强有关。最后，从背景来看，虽然两侧图片的背景都是单一色调的，但修复后的图片在背景处理上更加细腻，没有出现明显的噪点或瑕疵。这进一步增强了图片的整体美观度和专业性。

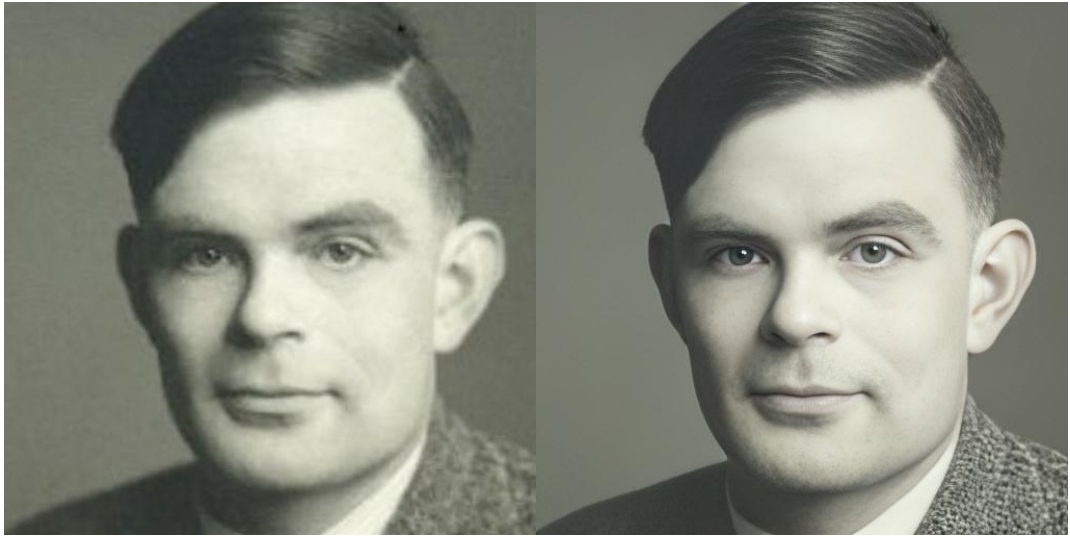
3.4、总体功能测试

各模块间合作完成的功能实际上就是，先进行登录或者注册或者游客登录，在进入网页之中进行图片的动态和静态修复效果，上面已经演示的非常清晰。总体的核心功能就是完成图像的两种处理效果。

4、性能测试

4.1、精度分析

4.1.1 静态精度分析：经过精心修复，男子的脸部图像焕然一新，细节之处彰显匠心独运。发丝纤毫毕现，面部特征更是精致入微。右侧的修复后图像，将男子的五官轮廓勾勒得更加立体鲜明，眼睛、鼻子和嘴巴等关键部位的线条流畅而生动。肌肤质感细腻光滑，透出青春的光泽与活力。尽管仔细观察后，我们不难发现鼻梁略显“偏移”，修复后的鼻子更显高挺，立体感更强。然而，这些微小的偏差仍在可接受的范围内，并不影响整体的修复效果。凝视修复后的图像，我们依然能够感受到这位计算机科学之父——艾伦·图灵的独特气质和深邃眼神。



4.1.2 动态精度分析

对比驱动视频和我们生成的视频，可以明显动态视频精度的准确性。在连续的视频帧中，算法不仅能够精确捕捉并传递驱动视频中的人物动作，还能够在每一帧中保持源图像人脸特征的一致性和稳定性。这种精确的特征对齐确保了面部动作的自然流畅，无论是微小的表情变化还是复杂的头部运动，都能被算法精准捕捉并重现。

此外，算法在处理视频帧之间的过渡时表现出了极高的平滑性，没有出现任何明显的抖动或失真现象。这意味着在动态视频中，人物的面部轮廓、眼睛、鼻子、嘴巴等关键特征始终保持清晰和连贯，即使是在快速运动或表情变化的场景下，也不会产生模糊或断裂的效果。但是视频两侧的部分修复后不是效果明显，略有一点误差的存在，同时如何进行大幅度的运动，会有一定的人脸扭曲，这一点有待调整。

更重要的是，尽管人脸被赋予了动态的动作，但原始图像中的人物面部特征并未发生任何改变。算法巧妙地将动态信息与静态特征融合在一起，使得生成的视频既具有生动的动作表现力，又不失原始人物的真实感和辨识度。

总之，动态视频展现出了优秀的精度特性。它不仅能够精确传递人物动作，保持面部特征的一致性和稳定性，还能够创造出自然、流畅、逼真的动态视觉效果。

4.2、时间特性分析

4.2.1 静态时间特性

静态图像处理以其时间花费短，原因在于它仅需对单张图片进行处理。得益于预先训练好的本地模型，我们无需再投入额外的时间进行模型训练或加载，这一优势使得静态照片修复在时间成本上极大减少。整个处理过程快速且无缝，大大提升了用户体验，令其在可接受的时间范围内轻松完成。展现了静态图像修复技术在时间特性上的优越性。

4.2.2 动态时间特性

动态处理则需要花费的时间比较长。因为动态处理的是视频，需要根据视频的每一帧里生成动画，以此来实现让图片中的人物动起来的功能，一个 3-4 秒的视频，一共 120 多帧，相当于静态连续处理 120 张照片，并且其中还涉及到动作迁移、人脸捕捉、超分辨率、图像分析、动画生成、运动跟踪处理等算法运行的时间，因此动态处理的周期很长。

4.3、灵活性分析

系统具备有良好的数据与处理能力，能够对用户输入的不同精度的图像进行处理，最终

返回给用户一个修复完好的版本。同时在运行环境变化时，系统具有良好的跨平台兼容性，能够在不同的操作系统（如 Windows、MacOS、Linux）上运行。同时具有硬件适应性，系统能够根据不同的硬件配置（如 CPU、GPU、RAM）进行优化，以确保在各种设备上都能有效运行。在用户界面设计上提供直观易用的界面，同时图像处理是自动化的，不需要用户手动操作，同时动态处理支持用户进行效果的选择，具有灵活性，最终致力于为用户提供稳定可靠的服务。

5、测试结论和建议

根据性能和功能测试给出系统的测试结论，对于系统在测试中的不足和尚未解决的问题，给出针对性建议。

1. 首先就是动态修复和静态修复的精度问题，对于用户上传的照片，最终修复的人脸还是比较不错的，但是对于照片的背景效果的修复与修复后的人脸放在一起就显得不是十分的搭配，也就是说对于照片背景的修复精度还有待提升，并且如果用户上传的照片色彩丢失严重，划痕太多，人物撕裂等严重损坏，系统的修复效果就不是那么的好。这需要我们继续调整模型的参数，来不断的调整，同时如果有必要还需要引入新的修复模型和设计新的算法。
2. 同时某些方面性能表现不是特别的优秀，谈一下处理的性能，内存占用是最基本的，不会产生很大的问题。静态处理的时间也是比较短的，基本都能在 5s 之内完成指定照片的修复效果，但是动态修复则需要将近 1 小时多的时间，这样肯定是不行的，因为如果等待的时间过于长，无论是对于调试还是反馈用户效果，都是极为不方便的，用户也很有可能因此变得不耐烦。因此对于动态处理时间的问题，我们计划于购置服务器，将代码部署在云端，来增加算力，增强代码的运行速度。
3. 用户反馈无法及时，同时没有给用户提供详细的说明文档，包括操作指南等。并且用户的反馈我们技术人员无法及时接收到用户在使用过程中反馈的信息，无法及时解决用户发现的问题或者提出的建议，我们对此的解决办法就是增加互动论坛似的反馈功能页面，能够让用户反馈信息，及时提供技术支持服务。