胡铮老师的反馈建议:

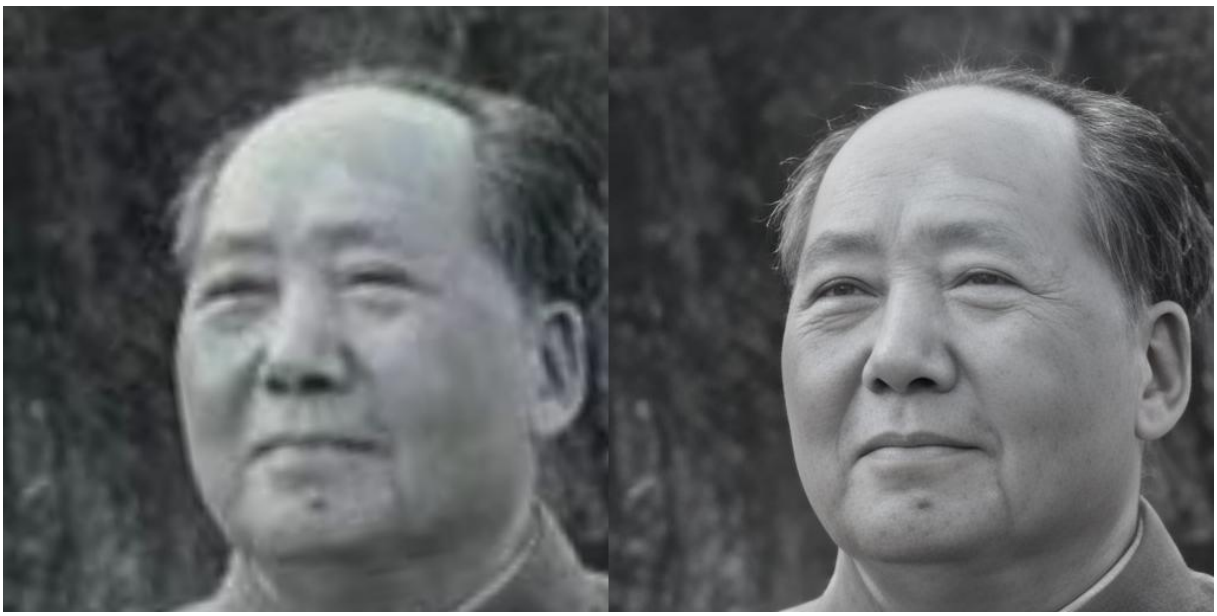
(1)静态修复功能的技术细节--挖掘一下非主脸目标的全幅优化?

(2)动态功能的技术细节?

(3)技术经验的总结和分享。

一.静态技术细节

静态修复模块，接收到照片之后，会将照片裁剪，只将照片的人脸部分传进模型进行修复，再将修复好的人脸部分贴回原来的照片



对比修复前后的人脸部分我们发现，我们之用的模型对人脸后的背景也有一定程度上修复，于是我们修改代码，将一整张照片都传入模型中，将修复，最后将人脸部分贴会修复后的人脸照片上，以此来达成较为简陋的全局修复效果。

```
//先生成这个类的实例化对象
restorer = GFPGANer(
        model_path=model_path,  //调用模型的本地路径
        upscale=args.upscale, //图像放大的比例
        arch=arch,//模型的架构选择
        channel_multiplier=channel_multiplier,
```

```
    bg_upsampler=bg_upsampler) //背景放大器，用于增强图像背景部分
//调用这个类的 enhance 函数进行照片的修复
cropped_faces, restored_faces, restored_img = restorer.enhance(
    input_img,  //修复的照片
    has_aligned=args.aligned, //是否对齐面部，由命令行参数 --aligned 控制
    paste_back=True,//是否将恢复后的面部粘贴回原始图像中，这里设置为 True，表示需要执行此操作。
    weight=args.weight)
```

含有这个参数 bg_upsampler，说明该模型有一定的背景的修复效果，上述的方法可行。

enhance 函数分析

首先调整原本的图片的大小，方便处理

```
img = cv2.resize(img, (512, 512))
self.face_helper.cropped_faces = [img]
```

将原本的改为合适模型处理的格式

```
img_resized = cv2.resize(img, (512, 512))
img_tensor = img2tensor(img_resized / 255., bgr2rgb=True, float32=True)  //将图像数据转换为 PyTorch 张量
normalize(img_tensor, (0.5, 0.5, 0.5), (0.5, 0.5, 0.5), inplace=True)  //归一化
```

```
img_tensor = img_tensor.unsqueeze(0).to(self.device)
```

调用 cropped_faces 函数将人脸部分裁剪下来，并将其改为合适模型处理的格式

```
for cropped_face in self.face_helper.cropped_faces:
    # prepare data
    cropped_face_t = img2tensor(cropped_face / 255., bgr2rgb=True, float32=True)  //将图像数据转换为 PyTorch 张量
    normalize(cropped_face_t, (0.5, 0.5, 0.5), (0.5, 0.5, 0.5), inplace=True)  //归一化
    cropped_face_t = cropped_face_t.unsqueeze(0).to(self.device)
```

调用 gfpgan 模型，传入人脸部分 cropped_face_t,原本的照片 img_tensor，进行修复，最后再将修复结果拼在一起。

```
try:
    output1 = self.gfpgan(cropped_face_t, return_rgb=False, weight=weight)[0]
    output2 = self.gfpgan(img_tensor, return_rgb=False, weight=weight)[0]
    # convert to image
    restored_face = tensor2img(output.squeeze(0), rgb2bgr=True, min_max=(-1, 1))
except RuntimeError as error:
    print(f'\tFailed inference for GFPGAN: {error}.')
    restored_face = cropped_face
    restored_img = img_resized
```

改进前后结果的对比图：

毛主席

改进前

改进前后对比效果：改进后的毛主席所站的地面更清晰，后面的房屋跟立体，因为我们调用的模型在人脸修复方面跟突出，所以改进修复后，衣服的细节尚未还原。

林徽因

改进前

改进后

改进后林徽因的所座的沙发更清晰

周总理

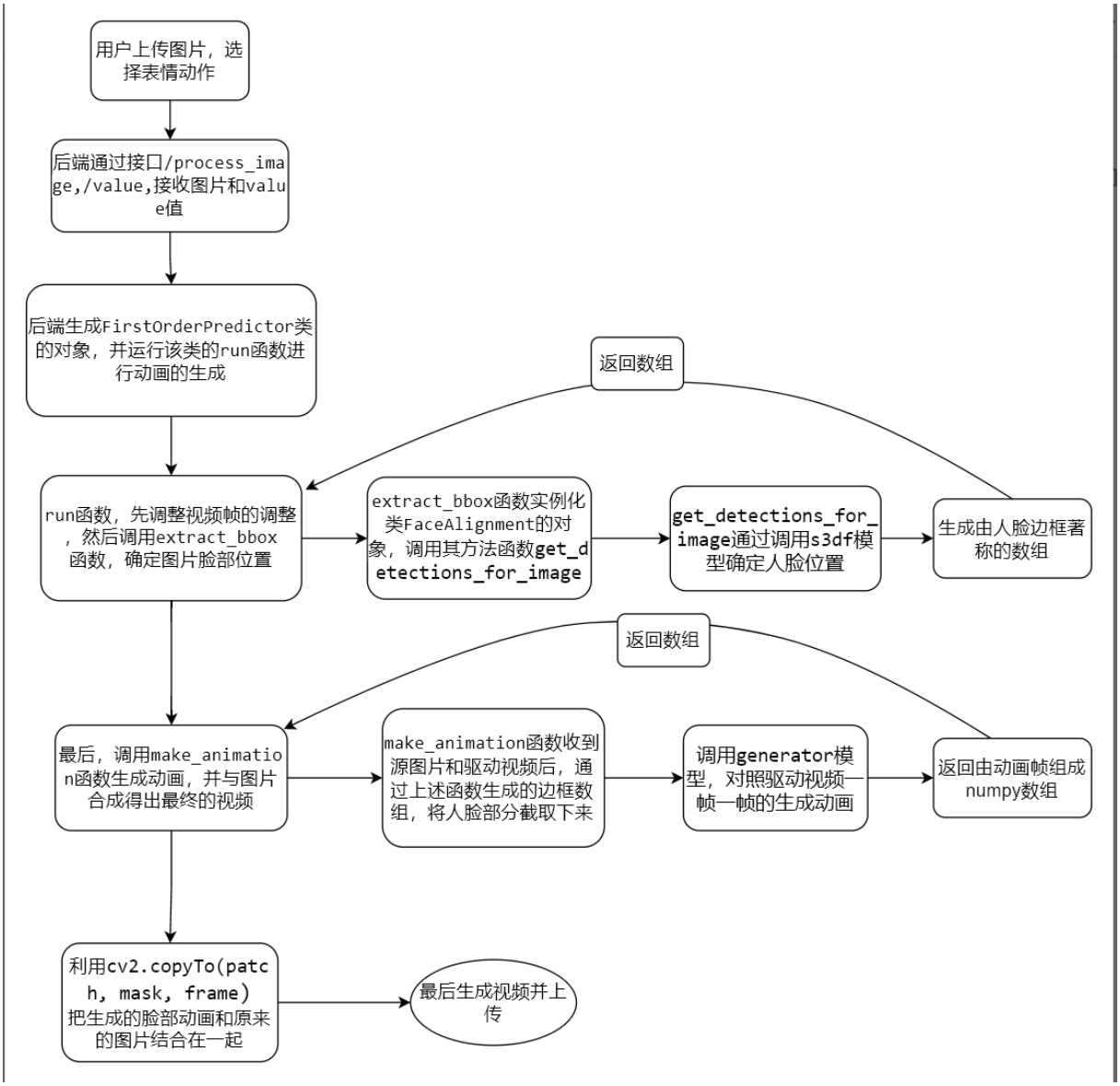
改进前

改进后周总理前的话筒，鲜花都更清晰

改进后任存在模型上的缺点：比如本来就非常模糊的地方无法完美的修复，比如毛主席身后的树，林徽因身后的树等。

二.动态技术细节

算法流程：



1. 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

2. 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

3. 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，外城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。如果视频中有多个人脸，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。

4. 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_boxs = []
```

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_boxs 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
```

5. 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。最后返回由动画桢组成的 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)
```

这里说明，动画实际的效果是比较丝滑的，答辩展示时，由于驱动视频被经过剪辑的原因导致生成动画效果的抖动，产生抖动的原因并非是算法因素，而是驱动视频的被剪辑了的原因。

这是答辩时展示的视频：

这是未经剪辑的视频，都放在如下网页中：

https://windows-dance-tb6.craft.me/RaajntwT7XeQs8

对比两个视频，可以看出，由没有经过剪辑的驱动视频生成的照片动画是连续且没有明显波动的。

三.技术细节经验总结分享

(1)在本项目中，我们通过静态和动态修复技术，实现了对老旧照片的修复和动态化处理。以下是我们在开发过程中积累的一些技术经验和教训，以及我们认为可以分享的要点：

在项目初期，我们高度重视技术选型，选择了 Vue 作为前端框架，Flask 作为后端框架，并辅以 MySQL 数据库以确保数据的可靠存储。这些选择基于它们强大的社区支持、高灵活性和易用性。我们采用了模块化设计思路，将前后端代码分离，并细分为静态修复和动态修复模块，极大提升了系统的可维护性和扩展性。此外，我们还集成了腾讯的 GFP-GAN 和百度的 PaddleGAN 模型，用于提升老照片的超分辨率修复效果和动态化处理，为用户带来了卓越的视觉体验。

前后端之间通过 Flask 定义的路由和 Vue 的 API 调用实现高效通信，保证了前端请求的灵活处理和后端结果的准确传递。在开发过程中，我们特别关注用户体验，设计了简洁直观的界面、易于导航的布局，并提供了响应式设计和交互式教程以帮助用户更好地使用系统。

为了确保系统的高效运行，我们进行了性能优化，包括利用 GPU 加速图像处理、优化内存管理和 I/O 操作。同时，我们也非常重视用户数据的安全和隐私保护，计划采用加密存储和安全的传输协议来保障用户数据安全。

在开发周期内，我们进行了严格的功能测试、性能测试和用户接受测试，以确保系统的稳定性和可靠性。我们坚信，只有持续学习和改进，才能跟上技术发展的步伐，不断优化和完善系统。此外，我们还成功解决了前后端分离开发中的跨域问题，通过 CORS 机制实现了资源的跨域共享。

在最后和老师答辩交流的过程当中，我们准备考虑人像和其背景的修复问题，我们上传了一整张的图片，进而来修复背景部分，进行最大程度的还原，考虑全局和局部的优化，我们会将继续完善和学习。

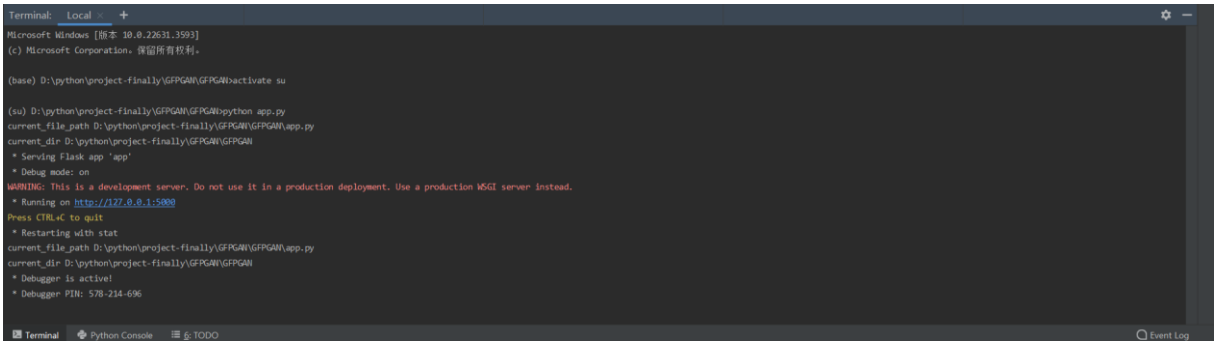
四.项目使用说明

1. 首先计算机安装 python，node.js，vue，Mysql 数据库，并且配置好相应的环境，安装相应的编译器，推荐:python3.7，Vue3.2.37，Mysql8.0，nodev18.19.1,Pycharm2019.3.3 x64，WebStorm2023.1，vscode1.88.0；接下来安装后端项目所需要的库，我们列举如下，当然也可以查看后端的/GFPGAN/documentation/requirements.txt,安装所需要的库。

```
facexlib
realesrgan
basicsr>=1.4.2
facexlib>=0.2.5
lmdb
numpy
opencv-python
pyyaml
scipy
tb-nightly
torch>=1.7
torchvision
tqdm
yapf
tqdm
PyYAML>=5.1
scikit-image>=0.14.0
scipy>=1.1.0
opencv-python<=4.6.0.66
imageio==2.9.0
imageio-ffmpeg
librosa==0.8.1
numba
easydict
munch
natsort
matplotlib
```

2.以上安装完成之后，在编译器中打开前后端项目，启用静态功能请运行 app.py 文件，如果启用动态功能，请运行/GFPGAN/applications/tools/first-oreder-demo.py 文件，我们给出运行之后的效果图。

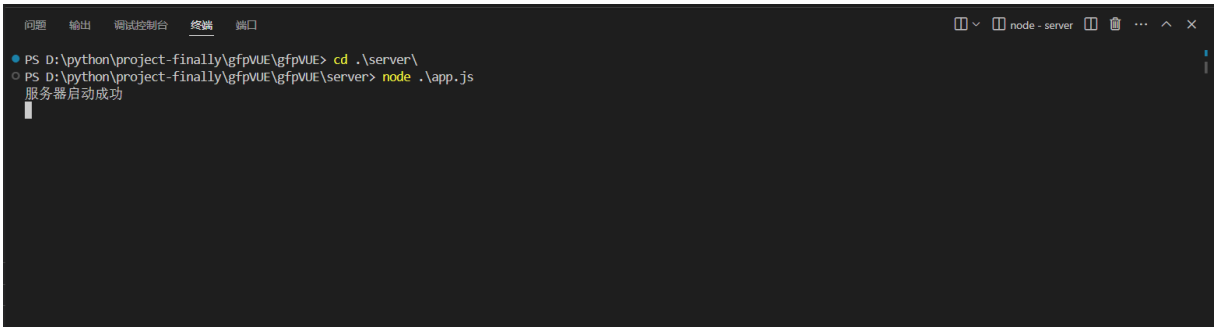
运行成功 app.py 文件:



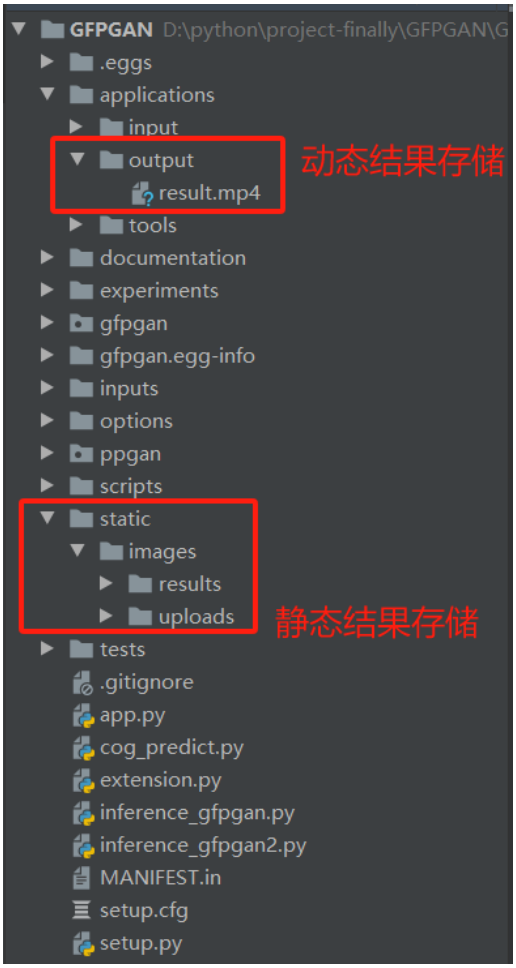
运行成功 first-order-demo.py 文件

下面我们再看前端代码，用编译器打开之后，在终端命令行窗口执行命令 npm run dev ，然后等待，执行成功效果如下图:

接着我们再来启动数据库服务器，在一个新的终端端口执行 cd .\server\ ，进入指定文件夹之下，接着执行 node .\app.js ,执行成功效果如下图:



接下来就可以进入网页执行相应的操作，这里就不在介绍，登录/游客登录/注册,静态效果，动态效果.....最后说明返回结果的存储。具体网页的演示我们这里就不在详细说明了。


GFPGAN D:\python\project-finally\GFPGAN\G
.eggs
applications
input
output
result.mp4
tools
documentation
experiments
gfpgan
gfpgan.egg-info
inputs
options
ppgan
scripts
static
images
results
uploads
tests
.gitignore
app.py
cog_predict.py
extension.py
inference_gfpgan.py
inference_gfpgan2.py
MANIFEST.in
setup.cfg
setup.py
动态结果存储
静态结果存储