使用Flask搭建图像处理API

V. 1.1

1、关于Flask

要实现基于Flask的接口，先要了解 **Flask** 。

➔ Flask是一个轻量级的后台框架，学习成本低，维护简单，传统java、Php太过笨重。如果只是从简单这个角度出发，Flask是开发后端的佼佼者，最快只需要7行代码完成一个Web应用。

1.1 从Hello Word开始

经典程序语言学习方法，从“Hello World”开始 。

环境要求

python3、pip、Flask

使用pip安装Flask

运行 `pip install Flask` 安装Flask；至此环境就安装好了。

下面正式进入主题

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def index():

return '<h1>Hello World</h1>'

if __name__ == '__main__':

app.run()
```

只需要7行代码，一个“Hello World”的Web应用的代码就写完了，保存为”hello.py“，运行

python hello.py

控制台打印会输出

* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

说明服务器启动成功。

说明：”http://0.0.0.0:5000/“是”app.run()”默认值即”app.run(host='0.0.0.0', post=5000)”的缩写，实际测试需要打开”http://127.0.0.1:5000/“。通过修改”host“和”post“的值可以完成访问地址和端口号设定。

测试

开启服务器之后，打开浏览器，访问”http://127.0.0.1:5000/“，可以看到”Hello World“

代码解释

- 先导入Flask包，并创建一个Web应用的实例”hello“

```
from flask import Flask

app = Flask(__name__)
```

这里的实例名称就是这个python模块名

- 定义路由规则

```
@app.route('/')
```

这个函数级别的注解指明了当地址是根路径时，就调用下面的函数。

- 处理请求

```
def index():

return '<h1>Hello World</h1>'
```

当请求的地址符合路由规则时，就会进入该函数。你可以在里面获取请求的request对象，返回的内容就是response。本例中的response就是大标题”Hello World”。

- 启动Web服务器

```
if __name__ == '__main__':

app.run()
```

当本文件为程序入口（也就是用python命令直接执行本文件）时，就会通过app.run()启动Web服务器。如果不是程序入口，那么该文件就是一个模块。Web服务器会默认监听本地的5000端口，但不支持远程访问。如果你想支持远程，需要在run()方法传入host=0.0.0.0，想改变监听端口的话，传入port=端口号，你还可以设置调试模式。具体例子如下：

```
if __name__ == '__main__':

app.run(host='0.0.0.0', port=8888, debug=True)
```

1.2 路由

从Hello World中，我们了解到URL的路由可以直接写在其要执行的函数上。这是一种把Model和Controller绑在一起的操作。如果你想灵活的配置Model和Controller，这样是不方便，但是对于轻量级系统来说，灵活配置意义不大，反而写在一块更利于维护。Flask路由规则都是基于Werkzeug的路由模块的，它还提供了很多强大的功能。

带参数的路由

在上面的Hello World基础上，加上下面的函数。并运行程序。

```
@app.route('/hello/<name>')

def hello(name):

return 'Hello %s' % name
```

当你在浏览器的地址栏中输入http://127.0.0.1:5000/hello/man，你将在页面上看到”Hello man”的字样。URL路径中/hello/后面的参数被作为hello()函数的name参数传了进来。

你还可以在URL参数前添加转换器来转换参数类型，我们再来加个函数：

```
@app.route('/user/<int:user_id>')

def get_user(user_id):

return 'User ID: %d' % user_id
```

试下访问http://127.0.0.1:5000/hello/man，你会看到404错误。但是试下http://127.0.0.1:5000/hello/123，页面上就会有”User ID: 123”显示出来。参数类型转换器int:帮你控制好了传入参数的类型只能是整形。

目前支持的参数类型转换器有：

类型转换器	作用
缺省	字符型，不能有转义斜线
int:	整型
float:	浮点型
path:	字符型，可以有斜线

多URL的路由

一个函数可以设置多个URL路由规则

```
@app.route('/')

@app.route('/hello')

@app.route('/hello/<name>')



def hello(name=None):

if name **is** None:

name = 'World'

return 'Hello %s' % name
```

这个例子接受三种URL规则，`/`和`/hello`都不带参数，函数参数`name`值将为空，页面显示”Hello World”；`/hello/<name>`带参数，页面会显示参数`name`的值，效果与上面第一个例子相同。

HTTP 请求方法

HTTP请求方法有Get, Post, Put, Delete，常用的有前两种。Flask路由规则也可以设置请求方法。

```
from flask import request

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

def login():

if request.method == 'POST':

return 'This is a POST request'

else:

return 'This is a GET request'
```

当你请求地址http://127.0.0.1:5000/login，”GET”和”POST”请求会返回不同的内容，其他请求方法则会返回405错误。

静态文件位置

一个Web应用的静态文件包括了JS, CSS, 图片等，Flask的风格是将所有静态文件放在”static”子目录下。并且在代码或模板中，使用`url_for('static')`来获取静态文件目录。上小节中第四个的例子就是通过`url_for()`函数获取”static”目录下的指定文件。如果你想改变这个静态目录的位置，你可以在创建应用时，指定`static_folder`参数。

```
app = Flask(__name__, static_folder='files')
```

1.3 （*）模板

Flask的模板功能是基于Jinja2模板引擎实现的。让我们来实现一个例子吧。创建一个新的Flask运行文件：

```
from flask import Flask

from flask import render_template
```

```
app = Flask(__name__)



@app.route('/hello')

@app.route('/hello/<name>')

def hello(name=None):

return render_template('hello.html', name=name)


if __name__ == '__main__':

app.run(host='0.0.0.0', debug=True)
```

这段代码同上一章节的多URL路由的例子非常相似，区别就是`hello()`函数并不是直接返回字符串，而是调用了`render_template()`方法来渲染模板。方法的第一个参数`hello.html`指向你想渲染的模板名称，第二个参数`name`是你要传到模板去的变量，变量可以传多个。

接下来我们创建模板文件。在当前目录下，创建一个子目录”templates”（注意，一定要使用这个名字）。然后在”templates”目录下创建文件`hello.html`，内容如下：

```
<!doctype html>

<title>Hello Sample</title>

{% if name %}

<h1>Hello {{ name }}!</h1>

{% else %}

<h1>Hello World!</h1>

{% endif %}
```

上面就是一个HTML模板，根据`name`变量的值，显示不同的内容。变量或表达式由`{{ }}`修饰，而控制语句由`{% %}`修饰，其他的代码，就是我们常见的HTML。打开浏览器，输入http://127.0.0.1:5000/hello/man，页面上即显示大标题”Hello man!”。按F12 查看页面源码，可以看到：

```
<!doctype html>

<title>Hello from Flask</title>

<h1>Hello man!</h1>
```

可以看出模板代码进入了Hello {{ name }}!分支，而且变量{{ name }}被替换为了”man”。

模板继承

一般我们的网站虽然页面多，但是很多部分是重用的，比如页首，页脚，导航栏之类的。对于每个页面，都要写这些代码，很麻烦。Flask的Jinja2模板支持模板继承功能，省去了这些重复代码。让我们基于上面的例子，在”templates”目录下，创建一个名为”layout.html”的模板：

```
<!doctype html>

<title>Hello Sample</title>

<link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='style.css') }}">

<div** class="page">

{% block body %}

{% endblock %}

</div>
```

再修改之前的”hello.html”，把原来的代码定义在{% block body %}中，并在代码一开始”继承”上面的”layout.html”：

```
{% extends "layout.html" %}

{% block body %}

{% if name %}

<h1>Hello {{ name }}!</h1>

{% else %}

<h1>Hello World!</h1>

{% endif %}

{% endblock %}
```

打开浏览器，再看下http://127.0.0.1:5000/hello/man 页面的源码。

```
<!doctype html>

<title>Hello Sample</title>

<link rel="stylesheet" type="text/css" href="/static/style.css">

<div class="page">

<h1>Hello man!</h1>

</div>
```

虽然render_template()加载了”hello.html”模板，但是”layout.html”的内容也一起被加载了。而且”hello.html”中的内容被放置在”layout.html”中{% block body %}的位置上。形象的说，就是”hello.html”继承了”layout.html”。

HTML自动转义

如果在路由函数中这样写：

```
@app.route('/')

def index():

return '<div>Hello %s</div>' % '<em>Flask</em>'
```

打开页面，可以看到”Hello Flask”字样，而且”Flask”是斜体的，因为我们加了 标签。（引入”Markup”类可以取消自动转义）

2、Flask实现图像处理API

后面使用了opencv，因此需要的环境有：python 、扩展包：flask 、opencv

2.1 文件上传

分4个步骤实现文件上传功能：

- 首先建立一个让用户上传文件的页面，我们将其放在模板”upload.html”中

-upload.html-

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Flask上传图片演示</title>

</head>

<body>
```

```
<h1>使用Flask上传本地图片Demo</h1>

<form action="" enctype='multipart/form-data' method='POST'>

<input type="file" name="file" style="margin-top:20px;"/>

<br>

<input** type="submit" value="上传" class="button-new"
style="margin-top:15px;"/>

</form>

</body>

</html>
```

这里主要就是一个enctype="multipart/form-data"的form表单；一个类型为file的input框，即文件选择框；还有一个提交按钮。

- 定义一个文件合法性检查函数

```
# 设置允许的文件格式

ALLOWED_EXTENSIONS = set(['png', 'jpeg', 'jpg', 'JPG', 'PNG',
'bmp'])


# 检查文件类型是否合法

def allowed_file(filename):

# 判断文件的扩展名是否在配置项ALLOWED_EXTENSIONS中

return '.' in filename and filename.rsplit('.', 1)[1] in
ALLOWED_EXTENSIONS
```

- 文件提交后，在POST请求的视图函数中，通过request.files获取文件对象

这个request.files是一个字典，字典的键值就是之前模板中文件选择框的”name”属性的值，上例中是”file”；键值所对应的内容就是上传过来的文件对象。

- 检查文件对象的合法性后，通过文件对象的save()方法将文件保存在本地 我们将第3和第4步都放在视图函数中，代码如下：

```
from flask import Flask, render_template, request

from werkzeug.utils import secure_filename

import os

app = Flask(__name__)

# 设置json显示中文

app.config['JSON_AS_ASCII'] = False

# 设置静态文件缓存过期时间

app.send_file_max_age_default = timedelta(seconds=1)

# 设置请求内容的大小限制，即限制了上传文件的大小（5M，可选）

app.config['MAX_CONTENT_LENGTH'] = 5 * 1024 * 1024

@app.route('/enhance/upload', methods=['POST', 'GET']) # 添加路由

def upload():

if request.method == 'POST': # 如果是POST方法

# 获取上传的文件对象

f = request.files['file']

# 检查文件对象是否存在，且文件名是否合法

if not (f and allowed_file(f.filename)):
```

```
# 不合法返回error

return jsonify({"error": 1001, "error_msg": "format error: 请检查上传的图片类型，仅限于png、jpg、jpeg、PNG、JPG、bmp"})

# 如果文件合法:

basepath = os.path.dirname(__file__) # 当前文件所在路径

# 去除文件名中不合法的内容

upload_path = os.path.join(basepath, 'static/images', secure_filename(f.filename)) # 注意：没有的文件夹一定要先创建，不然会提示没有该路径

 # 文件保存

f.save(upload_path)

# 上传成功

return '上传成功'

# 如果是GET方法

return render_template('upload.html')
```

代码解释

1. **app.config['JSON_AS_ASCII'] = False配置json显示中文**
2. **Flask的MAX_CONTENT_LENGTH配置项可以限制请求内容的大小默认是没有限制，上例中我们设为5M。**
3. **必须调用werkzeug.secure_filename()来使文件名安全。比如用户上传的文件名为"../../../../user/username/.xyz"，secure_filename()方法可以将其转为"userusername.xyz"。**

2.2 图像处理

将图片文件上传以后就可以按照常规的方式进行图像处理了。具体流程是：图片上传->处理图片->保存处理结果->返回处理后的图片到前端页面（如果有需要）返回结果的参数json（如果有需要）。

整体的文件框架：

Flask_project_name

```
    |___static
    |      |__images
    |___templates
    |      |__upload.html
    |      |__upload_ok.html
    |___enhance.py
    |___upload_pictures.py
```

这里介绍使用opencv完成简单的图像处理，在上传、保存以后就可以使用opencv进行操作了。关键代码如下：

```
#开始使用opencv从上传的文件中进行图像处理操作

src = cv2.imread(upload_path)

# 保存临时图片用于进行处理和输出前端展示

cv2.imwrite(os.path.join(basepath, 'static/images', 'src.jpg'), src)

# 调用enhance()处理输出保存

dst = enhance(src)

# 保存处理结果进行输出前端展示

cv2.imwrite(os.path.join(basepath, 'static/images', 'result.jpg'),
dst)
```

上面的enhance()函数是自己的处理函数，位于enhance.py中，使用from enhance import enhance导入。

如果我们将文件上传成功之后的返回值**return** '上传成功' 修改成返回HTML模板：

```
# 返回html模板

return render_template('upload_ok.html', val1=time.time())
```

然后在“upload_ok.html”中展示上传和处理后的图片，这里可以简单这样写：

-upload_ok.html-

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Flask上传图片演示</title>

</head>

<body>

<h1>使用Flask上传本地图片图像处理API演示Demo</h1>

<form action="" enctype='multipart/form-data' method='POST'>

<input type="file" name="file" style="margin-top:20px;"/>

<br>

<input type="submit" value="上传" class="button-new" style="margin-
top:15px;"/>

</form>

<table>

<tr>
```

```
<td>

<a><img src="{{ url_for('static', filename=
'./images/src.jpg',_t=val1) }}" width="600" height=auto

ALIGN=left"/></a>

</td>

<td>

<a><img src="{{ url_for('static', filename=
'./images/result.jpg',_t=val1) }}" width="600" height=auto

ALIGN=right alt="你的图片被外星人劫持了～～"**/></a>**

</td>

</tr>

</table>

</body>

</html>
```

上面的HTML中

```
<img src="{{ url_for('static', filename= './images/src.jpg',_t=val1)
}}" width="600" height=auto

ALIGN=left"/>
```

展示了使用cv2.imwrite(os.path.join(basepath, 'static/images', 'src.jpg'), src)保存的上传后的src.jpg；下面一个HTML标签中则展示了处理后的图像result.jpg。

完整的upload_pictures.py

-upload_pictures.py-

```
from flask import Flask, render_template, request
```

```
from werkzeug.utils import secure_filename

import os

import cv2

import time

from datetime import timedelta

from enhance import enhance

# 设置允许的文件格式

ALLOWED_EXTENSIONS = set(['png', 'jpeg', 'jpg', 'JPG', 'PNG',
'bmp'])


# 检查文件类型是否合法

def allowed_file(filename):

return '.' in filename and filename.rsplit('.', 1)[1] in
ALLOWED_EXTENSIONS

app = Flask(__name__)

# 设置json显示中文

app.config['JSON_AS_ASCII'] = False

# 设置静态文件缓存过期时间

app.send_file_max_age_default = timedelta(seconds=1)

# 设置请求内容的大小限制，即限制了上传文件的大小（5M，可选）

app.config['MAX_CONTENT_LENGTH'] = 5 * 1024 * 1024
```

```
@app.route('/enhance/upload', methods=['POST', 'GET']) # 添加路由

def upload():

if request.method == 'POST':

# 获取上传的文件对象

f = request.files['file']

# 检查文件对象是否存在，且文件名是否合法

if not (f and allowed_file(f.filename)):

# 不合法返回error

return jsonify({"error": 1001, "error_msg": "format error：请检查上传的图片类型，仅限于png、jpg、jpeg、PNG、JPG、bmp"})

# 如果文件合法：

basepath = os.path.dirname(__file__) # 当前文件所在路径

# 去除文件名中不合法的内容

upload_path = os.path.join(basepath, 'static/images', secure_filename(f.filename)) # 注意：没有的文件夹一定要先创建，不然会提示没有该路径

# 文件保存

f.save(upload_path)

#开始使用opencv从上传的文件中进行图像处理操作

src = cv2.imread(upload_path)

# 保存临时图片用于进行处理和输出前端展示
```

```
cv2.imwrite(os.path.join(basepath, 'static/images', 'src.jpg'), src)

# 调用enhance()处理输出保存

dst = enhance(src)

# 保存处理结果进行输出前端展示

cv2.imwrite(os.path.join(basepath, 'static/images', 'result.jpg'),
dst)

# 返回html模板

return render_template('upload_ok.html', val1=time.time())

# 如果是GET方法

return render_template('upload.html')

if __name__ == '__main__':

# 如果对这里的 host 和port有疑问可以参考前文

app.run(host='0.0.0.0', port=5000, debug=True)
```

当使用python upload_pictures.py运行服务打开以后，在浏览器打开http://127.0.0.1:5000/enhance/upload就可以看到：

)

其中的图像处理步骤：

开始使用opencv从上传的文件中进行图像处理操作

```
src = cv2.imread(upload_path)

# 保存临时图片用于进行处理和输出前端展示

cv2.imwrite(os.path.join(basepath, 'static/images', 'src.jpg'), src)

# 调用enhance()处理输出保存

dst = enhance(src)

# 保存处理结果进行输出前端展示

cv2.imwrite(os.path.join(basepath, 'static/images', 'result.jpg'),
dst)
```

这里enhance()来自enhance.py文件，是一个简单的限制对比度的自适应直方图均衡来进行图像暗光增强的处理过程。

-enhance.py-

```
# -- coding: utf-8 --

import cv2

def enhance(src):

# 分离BGR通道

src_B, src_G, src_R = cv2.split(src)

# 创建CLAHE对象（对每个通道）

clahe_B = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))

clahe_G = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))

clahe_R = cv2.createCLAHE(clipLimit=2.0,tileGridSize=(8,8))

# 限制对比度的自适应阈值均衡化（对每个通道）
```

```
    dst_B = clahe_B.apply(src_B)

    dst_G = clahe_G.apply(src_G)

    dst_R = clahe_R.apply(src_R)

    # 通道合并

    dst = cv2.merge([dst_B, dst_G, dst_R])

    # 返回图像数组

    return dst
```

在http://127.0.0.1:5000/enhance/upload打开文件，点击上传之后就会在页面展示出未处理图片和处理后图片的对比：

)

至此基于Flask的简单Web页面和进行图像处理完成。

总结

单纯做简单的图像处理接口，只需定义好函数（包括参数）、及其返回值（一个或多个），然后在Flask的server中调用这些函数就够了，所有的处理功能都可以在图像处理函数中完成，Flask的server只起到一个保存结果并request到前端的作用。

3、（*）使用json

有时候处理的结果中含有一些参数或属性需要输出，这个时候可以在Flask中使用json。

Flask已经内置了json：

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')

def root():

t = {

'a': 1,

'b': 2,

'c': [3, 4, 5]

}

return jsonify(t)


if __name__ == '__main__':

app.debug = True

app.run()
```

注意：在Flask中直接返回list或dict是不可行的，如：

```
from flask import Flask
```

```
app = Flask(__name__)

@app.route('/')

def root():

t = {

'a': 1,

'b': 2,

'c': [3, 4, 5]

  }

return t

if __name__ == '__main__':

app.debug = True

app.run()
```

这样会提示TypeError: 'dict' object is not callable

解决方法:

HTTP返回json格式数据主要有两个方面:

\1. 数据本身为json格式;

\2. Content-Type声明为json格式。

使用标准库**json**

```
from flask import Flask

import json


app = Flask(__name__)

@app.route('/')
```

```
def root():

t = {

'a': 1,

'b': 2,

'c': [3, 4, 5]

}

return json.dumps(t)

if __name__ == '__main__':

app.debug = True

app.run()
```

这样当访问时即能够正常得到json数据。但这么做有一个缺点，就是HTTP返回的Content-Type仍然是text/html，即HTTP认为内容是HTML。

声明Content-Type为json格式

在上面的解决方法上作一个加强，手动指定其Content-Type为application/json，通常采用的是修改Flask中的Response模块：

```
from flask import Flask, Response

import json

app = Flask(__name__)

@app.route('/')

def root():

t = {
```

```
        'a': 1,
        'b': 2,
         'c': [3, 4, 5]
    }
return Response(json.dumps(t), mimetype='application/json')

if __name__ == '__main__':
app.debug = True
app.run()
```

这样不仅HTTP返回的内容是json，而且返回的Content-Type也是application/json了。

使用Flask中的jsonify

实际上flask已经为json准备了专门的模块：jsonify。jsonify不仅会将内容转换为json，而且也会修改Content-Type为application/json。

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')

def root():

t = {

        'a': 1,

        'b': 2,

        'c': [3, 4, 5]

    }
```

```
return jsonify(t)


if __name__ == '__main__':

app.debug = True

app.run()
```

在Flask中使用jsonify和json.dumps的区别

Flask提供了jsonify函数供用户处理返回的序列化json数据，而python自带的json库中也有dumps方法可以序列化json对象，那么在Flask的视图函数中return它们的区别是什么?

区别在于jsonify可以接受和python中的dict构造器同样的参数，而json.dumps比jsonify可以多接受list类型和一些其他类型的参数。

V. 2.0

为什么需要2.0版本?

因为之前写的前后端捆绑在一起，且前端直接在页面间跳转，没有考虑和使用异步刷新，这会导致网页加载慢，前后端不分离不利于结构化开发。所以2.0版本主要工作放在Web后端独立，只在返回结果中提供前端需要参数。

0、 前后端分离

好处：前后端分离的意思是，前后端只通过 JSON 来交流，组件化、工程化不需要依赖后端去实现。也可以用在移动端开发。

1、 相同

环境要求同**V. 1.1**中环境要求：

环境要求

python3、pip、Flask、OpenCV-python

使用pip安装Flask

运行 `pip install Flask OpenCV-python` 安装Flask；至此环境就安装好了。

2、 不同

V. 1.1版本中路由函数返回为html页面，展示也是直接跳转到相对应的html，这就导致加载缓慢，前端移植复杂的情况。而考虑到批量处理的情况，**V. 2.0**版本增加了批量处理操作，批量处理采用上传zip压缩包再解压处理的方法，同时增加保存处理结果到服务器。

2.1 单张处理

依旧要检测上传文件格式 ：

```
# 设置允许的文件格式

ALLOWED_EXTENSIONS = set(['jpg', 'JPG', 'png', 'PNG', 'jpeg' 'bmp'])

# 查看上传图片是否是对应的图片

def allowed_pic(filename):

return '.' in filename and filename.rsplit('.', 1)[-1] in
ALLOWED_EXTENSIONS
```

对单张图片进行暗光增强处理：

过程同V. 1.1版本。首先接受上传文件，保存到tmp/，然后将保存到服务器的图像作为处理的输入进行图像处理，将处理结果保存到results/。这里多了一步将结果保存到远端服务器。

```
# 添加*单张*暗光增强接口的路由

@app.route('/enhance/solo_pic', methods=['POST', 'GET']) # 添加路由

def image_enhance_solo():

if request.method == 'POST':

start = time.time()

solo_pic = request.files['solo_pic']


if not (solo_pic and allowed_file(solo_pic.filename)):

RuntimeError("Please check the uploaded file!")

basepath = os.path.dirname(__file__) # 当前文件所在路径

# 修改上传的文件名

unix_time = int(time.time())
```

```
new_p_name = str(unix_time) + '.png'

# 上传文件保存路径

upload_path = os.path.join(basepath, 'tmp')

upload_dir = os.path.join(basepath + 'tmp/%s' % new_p_name)

# 清除历史文件

if os.path.isdir(upload_path):

shutil.rmtree(upload_path)

os.makedirs(upload_path)

# 将待处理的图片存为临时文件

solo_pic.save(upload_dir)

# 使用一个字典封装返回结果

solo_rlt = {}

solo_rlt['method'] = 'LowLightEnhance'

# 图像处理

pic_rtl = enhance.enhance(upload_dir)

# 本地保存位置

local_pic = basepath + '/results/%s' % new_p_name

cv2.imwrite(local_pic,pic_rtl)

print(local_pic)

remote_pic = '/home/XYZ/images/%s' % new_p_name
```

```
# 上传到服务器

upload_status = upload_pic(local_pic, remote_pic)

# 将图片服务器对应的地址放入结果中返回

if upload_status == 1:

solo_rlt['solo_path'] = '/results/%s' % new_p_name

elif upload_status == 0:

solo_rlt['solo_path'] = 'None'

end = time.time()

print("处理耗时: %s" % (end - start))

solo_rlt['cost time'] = (end - start)

return json.dumps(solo_rlt, ensure_ascii=False)

else:

return RuntimeError("Only Accept POST Request!")
```

上传服务器操作（*）

主要用到了paramiko包，通过ssh协议在计算机和服务器之间传输文件。

```
# 服务器上传单张图片

def upload_pic(localfile, remotefile):

# 远程连接服务器的ip，端口号

client = paramiko.Transport(("88.88.88.88", 88))
```

```
# 服务器的 账户名 ， 密码

client.connect(username="root", password="123456")

sftp = paramiko.SFTPClient.from_transport(client)

try:

sftp.put(localfile, remotefile)

print('服务器上传成功!')

client.close()

return 1

except Exception:

print("服务器上传失败")

client.close()

return 0
```

2.2 进行批处理操作

检测上传压缩包格式

```
# 查看上传文件是否是zip文件

def allowed_file(filename):

return '.' in filename and filename.rsplit('.', 1)[-1] == 'zip'
```

接受上传压缩包，并解压、处理、压缩、保存返回。

```
# 添加*暗光增强*的批处理图片的路由
```

```python
@app.route('/enhance/batch', methods=['POST', 'GET'])
def image_enhance_batch():
if request.method == 'POST':
start = time.time()
# 接收前端传过来的一个压缩文件
try:
batch = request.files.get("batch")
print("接收到上传压缩文件: ", batch.filename)
except Exception:
raise RuntimeError("Please check the uploaded file!")
# return '<h1>请正确上传文件</h1>'
if not allowed_file(batch.filename):
raise RuntimeError("请上传zip格式的压缩文件!")
# return '<h1>请检查zip格式的压缩包</h1>'

# 当前服务文件所在的路径
base_path = os.path.dirname(__file__)
print("当前运行的目录路径为:  ", base_path)

# 将接收到的压缩包*解压*到指定目录folder_path下
```

```
folder_path = os.path.join(base_path + '/tmp')

# 运行前把上次的结果清空

if os.path.isdir(folder_path):

shutil.rmtree(folder_path)

# pass

os.makedirs(folder_path)

# 创建处理结果文件夹

# 运行前把上次的结果清空

res_path = os.path.join(base_path + '/tmp/results')

if os.path.isdir(res_path):

shutil.rmtree(res_path)

# pass

os.makedirs(res_path)


# 解压

un_zip(batch, folder_path)


# 图像处理

for img in foler_path:

        pic_rtl = enhance.enhance(img)
```

```
            cv2.imwrite('res_path/%s % img.rsplit('/', 1)[-1]', pic_rtl)


        print('批量去模糊完毕，开始打包进行上传！')

        # 将结果文件夹压缩成zip包，命名为 "时间戳.zip"

        batch_name = int(time.time())

        zipped_file = go_zip(base_path + '/tmp/%d.zip' % batch_name,
res_path)

        end = time.time()

        print("处理耗时：%s" % (end - start))


        # 将图片上传到FTP服务器上

        print("分类结果的本地文件在：", zipped_file)

        remote_file = '/home/XYZ/images/%d.zip' % batch_name

        print('将文件传送到服务器上：%d.zip' % batch_name)

        upload_batch(zipped_file, remote_file)

        # 将原文件夹及当中的文件删除

        try:

            shutil.rmtree(folder_path)

            print('解压文件夹%s删除成功！' % folder_path)

        except Exception:

            print('解压文件夹%s删除失败！' % folder_path)
```

```
        # 使用一个字典封装返回结果
        batch_rlt = {}
        # method字段
        batch_rlt['method'] = 'LowLightEnhance'
        # 处理用时放入结果返回
        batch_rlt['cost time'] = (end - start)
        # 将图片服务器对应的地址放入结果中返回
        batch_rlt['batch_path'] = '/tmp/%d.zip' % batch_name
        return json.dumps(batch_rlt, ensure_ascii=False)
    else:
     raise RuntimeError("Only Accept POST Request!")
```

其中解压方法为un_zip():

```
        # 解压
     un_zip(batch, folder_path)
```

定义un_zip()函数

```
 # 将压缩包batch解压缩到folder_path
def un_zip(batch, folder_path):
zip_file = zipfile.ZipFile(batch)
 # folder_path是图片解压后存放的文件夹名
if os.path.isdir(folder_path):
```

```
pass

else:

 os.mkdir(folder_path)


# 遍历压缩包中的每个图片文件，检查格式，并且解压文件到解压文件夹

    for names in zip_file.namelist():

      if names.rsplit('.', 1)[-1] in ALLOWED_EXTENSIONS:

        zip_file.extract(names, folder_path)

zip_file.close()
```

处理完成后将结果打包压缩：go_zip()：

```
zipped_file = go_zip(base_path + '/tmp/%d.zip' % batch_name,
res_path)
```

定义go_zip()函数

```
# 将结果压缩

def go_zip(target_path, target):

enhance_pics = list(os.walk(target))[0][2]

with zipfile.ZipFile(target_path, 'w', zipfile.ZIP_DEFLATED) as
zipping:

'''''

zipfile.ZipFile(fileName [,model [, compression[, allowZip64]]])
```

```
model和一般的文件操作一样，'r'表示打开一个存在的只读ZIP文件，'w'表示清空并打开一个只写的ZIP文件，或创建一个只写的ZIP文件；

'a'表示打开一个ZIP文件，并添加内容。

compression表示压缩格式，可选的压缩格式只有2个：ZIP_STORE；ZIP_DEFLATED。ZIP_STORE是默认的，表示不压缩，ZIP_DEFLATED表示压缩。
allowZip64 为True时，表示支持64位的压缩，一般而言，在所压缩的文件大于2G时，会用到这个选项；默认情况下，该值为Flase，因为Unix系统不支持。

'''
for m in enhance_pics:

zipping.write(target+'/'+m, '/result/'+m)

print('压缩完毕')

return target_path
```

上传服务器和单张处理结果上传一样就可以：

```
# 将结果压缩包上传到服务器（与单张上传完全一样，可以只写一个上传函数）

def upload_batch(localfile, remotefile):

# 远程连接服务器的ip，端口号

client = paramiko.Transport(("88.88.88.88", 88))

# 服务器的 账户名 ，密码

client.connect(username="root", password="123456")

sftp = paramiko.SFTPClient.from_transport(client)

try:

sftp.put(localfile, remotefile)
```

```
print('批量处理结果上传服务器成功!')

except Exception:

print("批量处理结果上传服务器失败!")

client.close()
```

然后将解压文件夹删除：

```
# 将原文件夹及当中的文件删除

try:

shutil.rmtree(folder_path)

print('解压文件夹%s删除成功! ' % folder_path)

except Exception:

print('解压文件夹%s删除失败! ' % folder_path)
```

最后将结果返回，返回结果根据前端需要增删，暂时设置了’method’ ,‘cost time’ ,’batch_path’ 。

’method’为处理方式，‘cost time’为处理时长，’batch_path’为处理结果保存位置。

```
#使用一个字典封装返回结果

batch_rlt = {}

# method字段

batch_rlt['method'] = 'LowLightEnhance'

# 处理用时放入结果返回

batch_rlt['cost time'] = (end - start)

# 将图片服务器对应的地址放入结果中返回
```

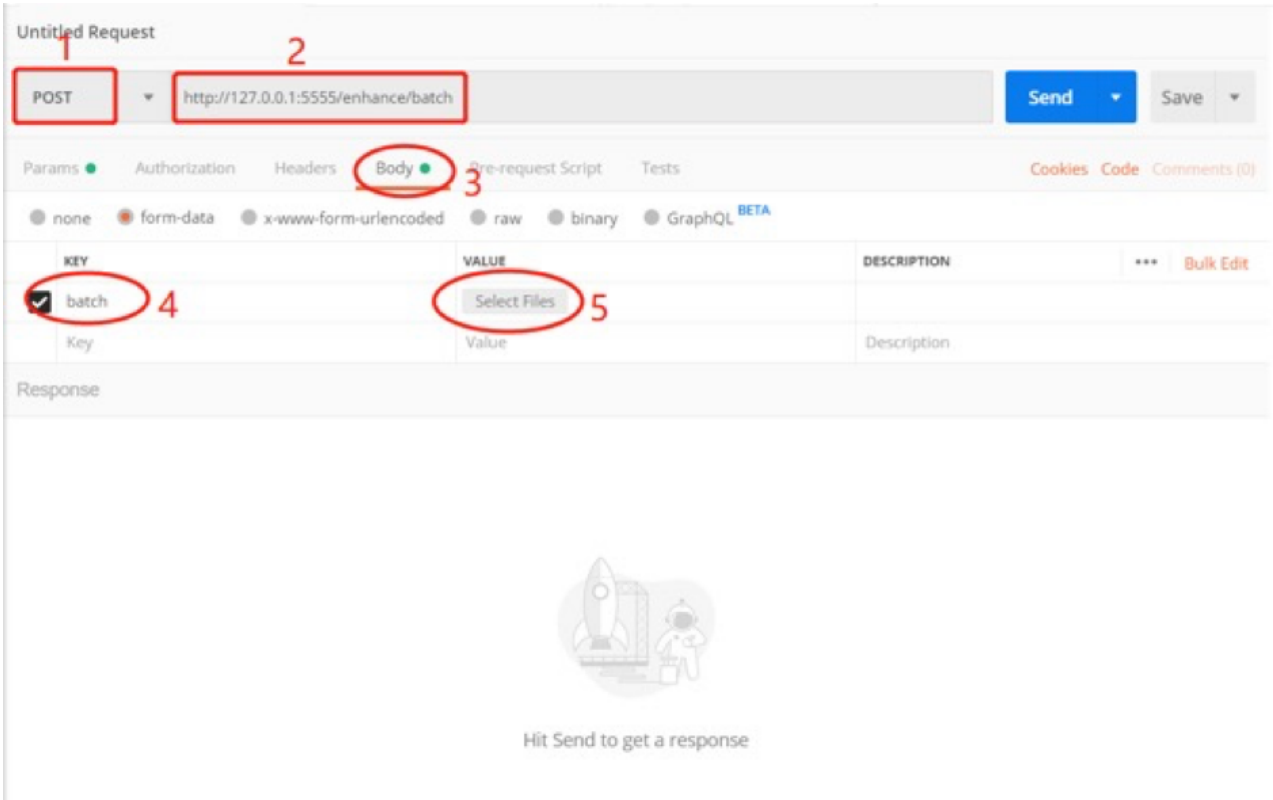
```
batch_rlt['batch_path'] = '/tmp/%d.zip' % batch_name

return json.dumps(batch_rlt, ensure_ascii=False)
```

3、测试

测试部分使用了[postman]模拟POST请求，下载安装打开。

比如测试批量暗光增强



如果测试单张，修改 **2** 处的 **url** 和 **4** 处的KEY值，在 **5** 处上传单张图片。最后点击右上角的 **SEND** 按钮等待返回结果。