Flask记录

P6 Flask对象初始化参数（一）

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

传入__name__是告诉Flask对象根目录的地址，Flask会在这个目录下寻找static、templates等文件夹

P7 Flask对象初始化参数（二）

P8 Flask的工程配置方式

一般有3种方式

1、创建一个类

```python
class DefaultConfig(object): # 新建一个类，类名随便
    """
    默认配置信息
    """
    SECRET_KEY = 'TaTaHshHsh'

app.config.from_object(DefaultConfig) # 通过这个api告诉app配置信息
```

2、创建一个文件

```python
app.config.from_pyfile('setting.py')
```

3、从环境变量里加载

环境变量一般指在操作系统中用来指定操作系统运行环境的一些参数

先在终端中执行如下命令

Python

```
export PROJECT_SETTING='~/setting.py'
```

然后再运行如下代码

Python

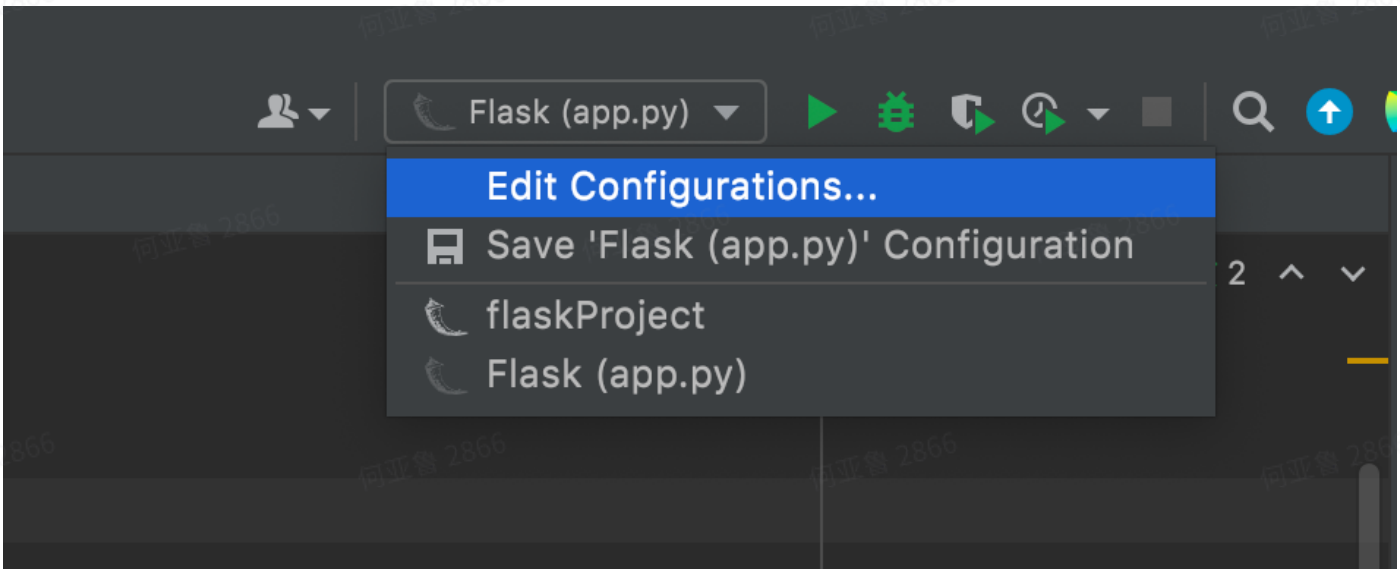
```
app.config.from_envvar('PROJECT_SETTING', silent=True)
# silent默认为False,如果环境变量里没有对应的值,就会报错。如果为True，不会报错
```

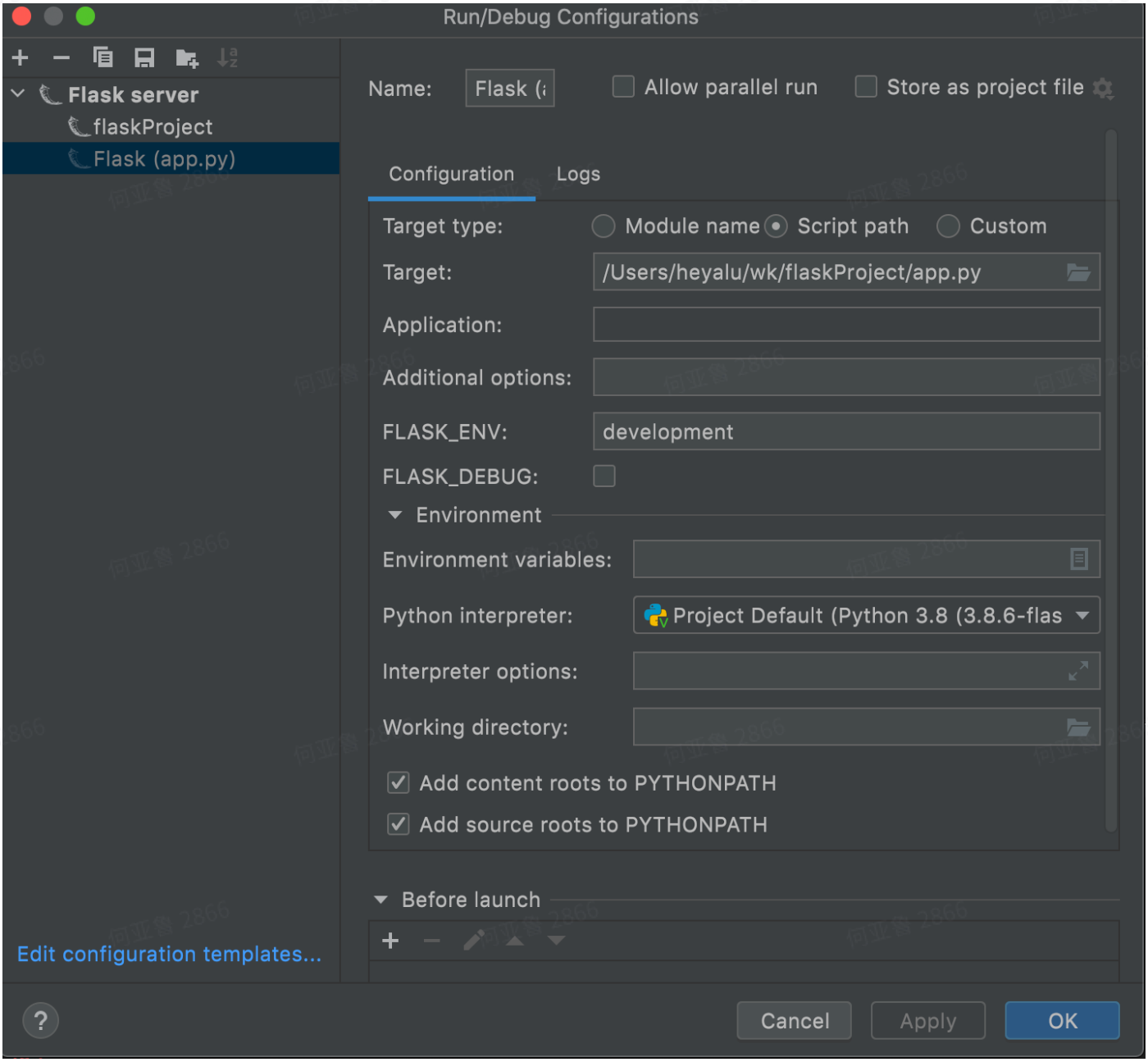
P9 从配置对象和配置文件加载

P10 从环境变量加载

1、在终端里执行时，可以在终端里 export 环境变量，然后在终端里执行 py 脚本

2、在pycharm里执行时，可以在pycharm里添加环境变量





P11 工程中Flask的配置的实践方法

三种方式对比：

1、使用类的方式：

优点是可以继承，缺点是敏感信息暴露在代码中。

一般用来写默认配置，让别人知道有这么个key，value可以为空。

2、使用文件：

优点是独立文件，可以保护敏感信息。缺点是文件路径固定，切换系统时可能有问题。

3、环境变量：

优点是既是独立文件，保护敏感数据。文件路径又不固定，比较灵活

缺点是不方便，还得设置环境变量

实战中的方式：

一般是方式1和方式3结合使用。

方式1加载默认数据，方式3加载真实数据。

```python
app.config.from_object(DefaultConfig)
app.config.from_envvar('PROJECT_SETTING', silent=True)
```

P12 新版Flask的运行方式

```python
flask run
```

这个run跑的是环境变量 FLASK_APP 对应的文件

```python
export FLASK_APP=helloworld.py
```

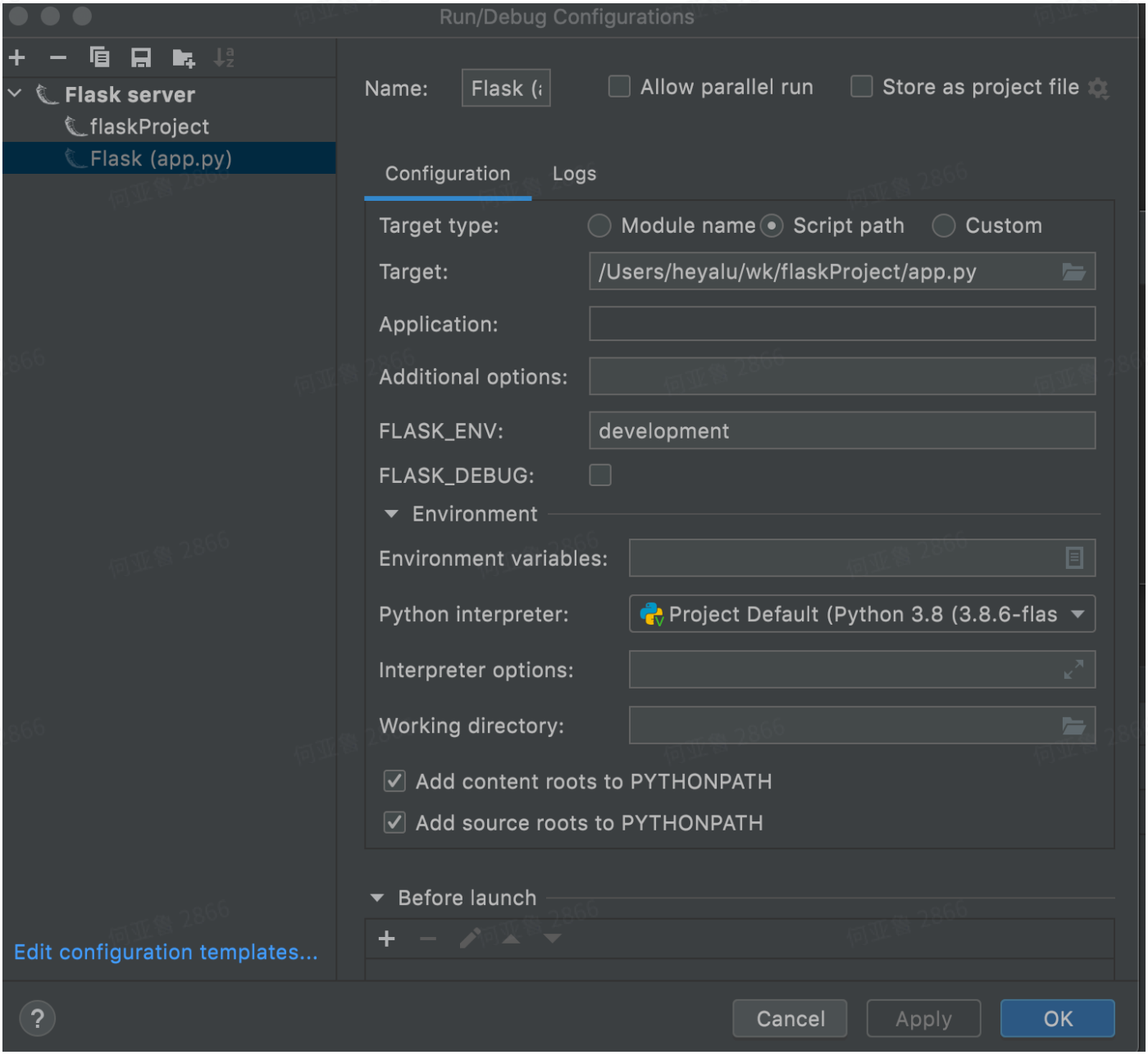
控制ip和端口号

```python
flask run -h 0.0.0.0 -p 8000
```

还有一个环境变量 FLASK_ENV 来控制生产模式和开发模式

```coffeescript
(3.8.6-flaskjson) ➜  flaskProject flask run
 * Environment: production  # 这个就是环境为生产模式
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

P13 Pycharm里新版Flask的运行方式



P14 查询路由的方式

在命令行里执行 flask routes

Visual Basic

```
(3.8.6-flaskjson) ➜  flaskProject flask routes
Endpoint     Methods  Rule
-----------  -------  ------------------------
hello_world  GET      /
query_data   GET      /query_data/<data_id>
static       GET      /static/<path:filename>
```

在代码里运行

Python

```python
print(app.url_map)
print(app.url_map.iter_rules())
```

P16 options请求方式与CORS跨域解决原理

路由默认的请求方式为：

GET

OPTIONS：简化版的GET，用于询问服务器某个接口信息的。跨域的时候往往会先发一个OPTIONS方式的请求，来查询另外一个URL可不可以访问

HEAD：也是简化版的GET，只返回GET请求处理时的响应头，不返回响应体

Rust

```
(3.8.6-flaskjson) ➜  flaskProject flask routes
Map([<Rule '/' (GET, HEAD, OPTIONS) -> hello_world>,
 <Rule '/query_data/<data_id>' (GET, HEAD, OPTIONS) -> query_data>,
 <Rule '/static/<filename>' (GET, HEAD, OPTIONS) -> static>])
```

P17 使用methods限制请求方式

Python

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

Rust

```
(3.8.6-flaskjson) ➜  flaskProject flask routes
Map([<Rule '/' (POST, OPTIONS) -> hello_world>,
 <Rule '/query_data/<data_id>' (HEAD, OPTIONS, GET) -> query_data>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>])
```

P18 蓝图的说明

Blueprint 是一个存储视图方法的容器，这些操作在这个Blueprint 被注册到一个应用之后就可以被调用，Flask 可以通过Blueprint来组织URL以及处理请求。

Flask使用Blueprint让应用实现模块化。

P19 蓝图的定义和引用

在单个py文件里定义，然后再app.py里引用

也可以在py包里定义，然后应用。（参照代码）

P22 请求与响应-问题说明

- Flask的debug模式的作用
  - 后端出现错误的话，会返回真实的报错信息给前端
  - 修改代码后，自动重启开发服务器

Python

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

- 支持OPTIONS方式的请求（但不等同于实现了CORS跨域）

P23 请求与响应-URL路径参数获取

Flask使用转换器的方式来获取url里面的路径参数

- 基本使用方式

Python

```python
@app.route('/query_data/<data_id>')
def query_data(data_id):
    return 'query data {}'.format(data_id)
```

- 带类型的使用方式，还会进行类型转换

Python

```python
@app.route('/query_data/<int: data_id>') # 设定为int类型
def query_data(data_id):
    return 'query data {}'.format(data_id)
```

- 可以自定义转换类型

自己声明一个类，继承自 BaseConverter ，实现里面的方法，然后注册给 Flask 就可以了。

P25 请求与响应-request对象的使用

- url路径之外的其它地方的参数都是由request对象获取，比如get请求?后面跟的参数或者post请求请求体里的参数

P26 请求与响应-模板响应

- 如何返回一个网页模板?
  - 模板都放在 templates 文件夹下，创建 templates
  - 在 templates 创建 .html 文件
  - 使用 render_template 渲染模板，返回即可。最终模板是以字符串的形式存在 body 里面

```Python
from flask import Flask, render_template
```

P27 请求与响应-重定向和 jsonify

- 重定向，使用 redirect

```CoffeeScript
from flask import redirect
```

- 返回json
  - 可以使用 json.dumps()
  - 也可以使用 jsonify （推荐使用）

```CoffeeScript
from flask import jsonify
```

后者不但把body转换为json，还会设置响应头 Content-Type 为 json

P28 请求与响应-构造状态码和响应头

- 可以返回元组

Python

```python
@app.route('/demo')
def demo():
    return 'demo', 200, {'temp': 'Python'}  # {response, status, headers}
    # 函数返回多值，本质上还是转换为一个元组
```

- 使用 make_response

Python

```python
from flask import make_response
```

P29 请求与响应-cookie使用

- 给响应设置cookie，使用 make_response

Python

```python
@app.route('/set_cookie')
def set_cookie():
    rsp = make_response('set_cookie_test')
    rsp.set_cookie('token', 'xxxxxxxx')
    # 会在Header里加上这个key-value，浏览器收到响应后会种上cookie
```

还可以设置过期时间

Bash

```
rsp.set_cookie('token', 'xxxxxxxx', max_age=3600)
```

- 读取cookie，使用request
- 删除cookie

Python

```python
@app.route('/cookie')
def set_cookie():
    rsp = make_response('cookie_test')
    rsp.delete_cookie('token')
    return rsp
    # 删除cookie并没有办法在Header里加上 Delete-Cookie，因为Header里只有Set-Cookie，一般框架的做法
```

删除cookie并没有办法在Header里加上 Delete-Cookie，因为Header里只支持Set-Cookie，一般框架的做法是修改过期时间，设置成前1秒或者1970年。

P30 请求与响应-session使用

- 设置session

Python

```python
from flask import Flask, session

@app.route('/set_session')
def set_session():
    session['username'] = 'heyalu'
    return 'set session sucess'
```

- 读取session

Python

```python
from flask import Flask, session

@app.route('/get_session')
def set_session():
    username = session['username']
    return 'get session sucess'.format(username)
```

- Flask的session存在哪里了？

Flask其实是浏览器session，存在了浏览器的cookie里。

P31 异常处理-abort的使用

立即停止视图函数的执行，并且把相对应的信息返回到前端中

Python

```python
abort(404)
```

P32 异常处理-异常捕获

使用 errorhandler ，可以在服务出现某个错误时，自定义一些返回信息。

Python

```python
@app.errorhandler(400)
def bad_request_err():
    return '网络请求出错了'

@app.errorhandler(ZeroDivisionError)
def zero_division_error():
    return '不能除以0哦'
```

P33 异常处理-中间件的使用

P34 异常处理-请求钩子的使用

Python

```python
# 在第一次请求之前运行.
# 例子：比如连接数据库操作，只需要执行一次
@app.before_first_request
def before_first_request():
    print('before_first_request')

# 在每一次请求都会执行
# 例子：可以在这里做权限校验操作，比如说某用户是黑名单用户，黑名单用户登录系统将遭到拒绝访问，可以使用
# before_request进行权限校验
@app.before_request
def before_request():
    print('before_request')

# 在请求之后运行
@app.after_request
def after_request(response):
    # response：就是前面的请求处理完毕之后，返回的响应数据
    # 如果需要对响应做额外处理,可以再这里进行
    # json.dumps 配置请求钩子
    # response.headers["Content-Type"] = "application/json"
    print('after_request')
    return response

# 每一次请求之后都会调用，会接受一个参数，参数是服务器出现的错误信息
@app.teardown_request
def teardown_request(error):
    # 数据库的扩展，可以实现自动提交数据库
    print('teardown_request: error %s' % error)

```