1. 请求钩子

```
# @app.before_request 装饰请求预处理的钩子
# def xxx(): 返回 None
@app.before_request
def before_request_1():
    print('before_request_1')
```

```python
# @app.after_request 装饰响应预处理的钩子
# def xxx(response):  response 参数是视图函数或者前一个钩子的返回值
#     需要返回一个 response
@app.after_request
def after_request_1(response):
    print('after_request_1')
    return response
```

```python
# @app.before_first_request 装饰第一个请求处理之前执行的钩子
# 这个钩子只会被运行一次，一般用于web应用初始化处理
@app.before_first_request
def before_first_request_1():
    print('before_first_request_1')
```

```python
# @app.teardown_request 装饰每个视图调用之后的钩子
# def xxx(error): 如果视图函数执行失败了，这里的 error 是错误信息，否则 error 是 None
@app.teardown_request
def teardown_request_1(error):
    print(error)
    print('teardown_request_1')
```

2. 蓝图

基本使用

```python
# 1. 创建蓝图对象 在 __init__.py 文件中 flask.Blueprint(名称, __name__)

from flask import Blueprint

home_blue = Blueprint('home', __name__)

# 4.  导入home.views 模块
# from home import views
# import home.views
__import__('home.views')
```

```python
# 2. 在 views.py 文件中使用蓝图绑定视图函数
#    @蓝图对象.route(路径,...)
#    def 函数():
#        返回响应

from home import home_blue


@home_blue.route('/home_index')
def home_index():
    return 'home index'
```

```python
from home import home_blue

# 2. 创建Flask应用对象 固定格式 app = Flask(__name__)
app = Flask(__name__)

# 3. 在 main.py 注册蓝图 app.register_blueprint(蓝图对象)

app.register_blueprint(home_blue)

# 4. 启动web服务 app.run()

# app.run()
```

使用细节

```python
# 添加蓝图访问的前缀 url_prefix=前缀 要以 / 开头

home_blue = Blueprint('home', __name__, url_prefix='/home')
```

```python
# 定义蓝图模块的请求钩子
# @蓝图对象.before_request
@home_blue.before_request
def home_blue_before_request():
    print('home_blue_before_request')
```

3. 上下文

介绍

演示使用

```python
# 3. 定义一个接口 读取上一个接口设置的 g 的属性
#    由于是不同的接口，也就属于不同的请求范围，所以第一个接口设置的属性，第二个接口是读取不到的
@app.route('/read_g_var')
def read_g_var():
    # getattr(对象,'字符串', 默认值)
    # getattr(g, 'name','没有 name 属性')
    name = getattr(g, 'name', '没有 name 属性')
    print(name)
    return 'read g var'
```

```python
46
47      print(session)          报错
48      app.run(host='0.0.0.0', port=8000, debug=True)
49
```

机制

4. 综合认证

统一处理

```python
# 定义一个请求钩子 通过 @app.before_request
# 逻辑: 如果 session 中包含 username 字段就说明用户已经登录，设置 g.is_login = True,否则 g.is_login = False

@app.before_request
def set_g_islogin():
    if 'username' in session:
        g.is_login = True
    else:
        g.is_login = False
```

装饰器

```python
# ....
# 定义  login_required 装饰器
# 如果 g.is_login==True，就调用被装饰函数，否则 abort(401)
def login_required(func):
    def __wrapper(*args, **kwarg):
        if g.is_login:
            return func(*args, **kwarg)
        else:
            abort(401)

    return __wrapper


# 定义一个接口，模拟更新用户信息，
# 使用 login_required 装饰器装饰，来限制必须登录才能够访问
# @app.route(....)
# @login_required
# def update_info():
#     ......
# 注意装饰的顺序
@app.route('/userinfo/update'1
@login_required 2
def update_userinfo():
    return 'update success'
```

装饰器问题解决

```python
# ....
# 定义  login_required 装饰器
# 如果 g.is_login==True，就调用被装饰函数，否则 abort(401)
def login_required(func):
    @wraps(func)        from functools import wraps
    def __wrapper(*args, **kwarg):
        if g.is_login:
            return func(*args, **kwarg)
        else:
            abort(401)

    return __wrapper
```

5. 配置

基本配置

```python
app = Flask(__name__)

# app.config[配置项名大小] = 配置项值
app.config['SECRET_KEY'] = 'test'
app.config['PERMANENT_SESSION_LIFETIME'] = timedelta(days=7)


@app.route('/read_config')
def read_config():
    # 通过 current_app 读取配置
    print(current_app.config.get('PERMANENT_SESSION_LIFETIME', '没有该配置'))
    return 'read config'
```

从对象中加载配置

```python
from datetime import timedelta


# 定义父类 1
class BaseConfig():
    SECRET_KEY = 'test'
    PERMANENT_SESSION_LIFETIME = timedelta(days=7)


# 定义 dev 子类 2
class DevConfig(BaseConfig):
    ENV = 'development'


# 定义 prod 子类 3
class ProdConfig(BaseConfig):
    ENV = 'production'
```

```python
from flask import current_app
from config import DevConfig, ProdConfig

# 2. 创建Flask应用对象 固定格式 app = Flask(__name__)
app = Flask(__name__)

# 从类中加载配置
app.config.from_object(ProdConfig)


# 定义路由读取配置
@app.route('/read_config')
def read_config():
    print(current_app.config.get('ENV'))
    print(current_app.config.get('PERMANENT_SESSION_LIFETIME'))
    return 'read_config'
```

工厂函数

```python
# config_dict= {'dev':DevConfig,'prod':ProdConfig}
# 定义工厂函数
def create_flask_app(config_key):
    """
    flask 工厂函数
    :param config_key: 配置类对应的key, dev|prod
    :return:
    """
    #   1. 实例化 app
    flask_app = Flask(__name__)  1
    #   2. 读取配置类
    config_class = config_dict[config_key]  2
    #   3. 加载配置类
    flask_app.config.from_object(config_class)  3
    #   4. 返回 app
    return flask_app  4
```

```python
# 调用工厂函数创建app
app = create_flask_app('prod')  # 'prod'
```

动态创建应用

```python
    #   4. 返回 app

    # 绑定路由和视图函数
    flask_app.route('/read_config')(read_config)
    return flask_app


def read_config():
    print(current_app.config.get('ENV'))
    print(current_app.config.get('PERMANENT_SESSION_LIFETIME'))
```

read_config()

```
(py3) ➜  Flask-Codes export FLASK_APP="main20_工厂函数:create_flask_app('prod')"
(py3) ➜  Flask-Codes flask run -h 0.0.0.0 -p 8000
 * Serving Flask app "main20_工厂函数:create_flask_app('prod')"
```

加载隐私配置

```python

# 加载隐私配置 export ENV_CONFIG='secret_config.py'

app.config.from_envvar('ENV_CONFIG')

```

```
py3) ➜  Flask-Codes
py3) ➜  Flask-Codes export FLASK_APP="main21_隐私配置" 1
py3) ➜  Flask-Codes export ENV_CONFIG='secret_config.py' 2
py3) ➜  Flask-Codes flask run -h 0.0.0.0 -p 8000 3
* Serving Flask app "main21_隐私配置"
```

```python
app.config.from_envvar('ENV_CONFIG', silent=True)
```

如果环境变量不存在，默认会报错，silent=True 即使没有环境变量也不报错

```python
@app.route('/read_config')
def read_config():
    print(current_app.config.get('SECRET_KEY'))
```

6. falsk-restful

7. 基本使用

```python
app = Flask(__name__)

# 1. 创建扩展/组件对象 Api(flask 应用对象 app)
api = Api(app)


# 2. 定义类视图继承自 Resource
#    定义 视图方法 def get|post|...
#    视图方法可以直接返回字典
class TestResource(Resource):
    def get(self):
        return {'method': 'get'}

    def post(self):
        return {'method': 'post'}

    def put(self):
        return {'method': 'put'}


# 3. 组件添加类视图 api.add_resource(类视图, 路径)
api.add_resource(TestResource, '/test')
```

8. 类视图装饰器

```python
# 3. 定义类视图
#    定义 get 和 post 方法
#    通过  method_decorators = 装饰器列表              装饰全部视图方法
#    通过 method_decorators = {方法名:装饰器列表}      装饰指定的视图方法

class TestResource(Resource):
    # method_decorators = [deco2, deco1]
    method_decorators = {'get': [deco1], 'post': [deco2]}

    # @deco2
    # @deco1
    def get(self):
        print('get')
        return {'method': 'get'}
```

0. 练习题

定义 goods 蓝图模块(蓝图模块)

需求:

- 蓝图的路由前缀是 goods
- 在 goods 模块下定义一个接口，可以通过 http://localhost:8000/goods/index 访问到该接口

定义一个装饰器(单个知识点)

需求:

- 打印当前的时间、当前请求的 url、method 信息
- 使用改装饰器装饰多个接口，测试不同接口是否能够正确打印信息

定义用户类视图(rest风格)

需求:

- 定义 get 方法用于模拟获取用户信息
- 定义 post 方法用于模拟用户登录
- 定义 delete 方法用于模拟退出登录

提示:

- 使用 session 来模拟用户的登录，信息获取，退出登录