day93 flask上下文管理

内容回顾

1. django和flask的区别

```
- 概括的区别
- django中提供功能列举
- 请求处理机制不同，django是通过传参的形式，flask是通过上下文管理的方式实现。
```

2. wsgi

```
django和flask内部都没有实现socket，而是wsgi实现。
wsgi是web服务网管接口，他是一个协议，实现它的协议的有：wsgiref/werkzurg/uwsgi
```

```python
# django之前
from wsgiref.simple_server import make_server

def run(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [bytes('<h1>Hello, web!</h1>', encoding='utf-8'), ]

if __name__ == '__main__':
    httpd = make_server('127.0.0.1', 8000, run)
    httpd.serve_forever()
```

```python
# flask之前
from werkzeug.serving import run_simple
from werkzeug.wrappers import BaseResponse


def func(environ, start_response):
    print('请求来了')
    response = BaseResponse('你好')
    return response(environ, start_response)


if __name__ == '__main__':
    run_simple('127.0.0.1', 5000, func)
```

3. web框架都有的功能：路由、视图、模板
4. before_request/after_request

```
相当于django的中间件，对所有的请求定制功能。
```

5. tempalte_global / template_filter

```
定制在所有模板中都可以使用函数
```

6. 路由系统处理本质 @app.route

```
将url和函数打包成rule，天剑到map对象，map再放到app中。
```

7. 路由
    - 装饰器实现 / add_url_rule
    - endpoint
    - 动态路由
    - 如果给视图加装饰器：放route下面 、 functools
8. 视图
    - FBV
    - CBV（返回一个view函数，闭包的应用场景）
    - 应用到的功能都是通过导入方式：request/session
9. flask中支持session

```
默认将session加密，然后保存在浏览器的cookie中。
```

10. 模板比django方便一点

```
支持python原生的语法
```

11. 蓝图

```
帮助我们可以对很多的业务功能做拆分，创建多个py文件，把各个功能放置到各个蓝图，最后再将蓝图注册到flask对象中。

帮助我们做目录结构的拆分。
```

12. threading.local对象

```
自动为每个线程开辟空间，让你进行存取值。
```

13. 数据库链接池 DBUtils （SQLHelper）
14. 面向对象上下文管理（with）

概要

- flask上下文源码
- flask的扩展

内容详细

1. 栈

后进先出，通过列表可以实现一个栈。

```
v = [11,22,33]
v.append(44)
v.pop()
```

应用场景：

- 节流

2. 面向对象

```
class Foo(object):

    def __setattr__(self, key, value):
        print(key,value)

    def __getattr__(self, item):
        print(item)

obj = Foo()
obj.x = 123
obj.x
```

- drf中request

  ```
  request.data
  request.query_params
  request._request
  request._request.POST
  request._request.GET

  #
  request.data
  request.query_params
  request.POST
  request.Data
  ```

```
class Local(object):
    def __init__(self):
        # self.storage = {}
        object.__setattr__(self,"storage",{})

    def __setattr__(self, key, value):
        self.storage[key] = value

    def __getattr__(self, item):
        return self.storage.get(item)

local = Local()
local.x1 = 123
print(local.x1)
```

3.线程唯一标识

```
import threading
from threading import get_ident

def task():
    ident = get_ident()
    print(ident)
for i in range(20):
    t = threading.Thread(target=task)
    t.start()
```

4.自定义threading.local

```
import threading
"""
storage = {
    1111:{'x1':[0,1,2,3]},
    1112:{'x1':1}
    1113:{'x1':2}
    1114:{'x1':3}
    1115:{'x1':4}
}
"""
class Local(object):
    def __init__(self):
        object.__setattr__(self,'storage',{})

    def __setattr__(self, key, value):
        ident = threading.get_ident()
        if ident in self.storage:
            self.storage[ident][key] = value
        else:
            self.storage[ident] = {key:value}

    def __getattr__(self, item):
        ident = threading.get_ident()
        if ident not in self.storage:
            return
        return self.storage[ident].get(item)

local = Local()

def task(arg):
    local.x1 = arg
    print(local.x1)

for i in range(5):
    t = threading.Thread(target=task,args=(i,))
    t.start()
```

5.加强版threading.local

```
import threading
"""
storage = {
    1111:{'x1':[]},
```

```
    1112:{'x1':[]}
    1113:{'x1':[]}
    1114:{'x1':[]}
    1115:{'x1':[]},
    1116:{'x1':[]}
}
"""
class Local(object):
    def __init__(self):
        object.__setattr__(self,'storage',{})

    def __setattr__(self, key, value):
        ident = threading.get_ident()
        if ident in self.storage:
            self.storage[ident][key].append(value)
        else:
            self.storage[ident] = {key:[value,]}

    def __getattr__(self, item):
        ident = threading.get_ident()
        if ident not in self.storage:
            return
        return self.storage[ident][item][-1]

local = Local()

def task(arg):
    local.x1 = arg
    print(local.x1)

for i in range(5):
    t = threading.Thread(target=task,args=(i,))
    t.start()
```

6.flask源码关于local的实现

```
try:
    # 协程
    from greenlet import getcurrent as get_ident
except ImportError:
    try:
        from thread import get_ident
    except ImportError:
        from _thread import get_ident
"""
__storage__ = {
    1111:{"stack":[汪洋] }
}
"""
class Local(object):

    def __init__(self):
        # self.__storage__ = {}
        # self.__ident_func__ = get_ident
        object.__setattr__(self, "__storage__", {})
        object.__setattr__(self, "__ident_func__", get_ident)
```

```python
    def __iter__(self):
        return iter(self.__storage__.items())

    def __release_local__(self):
        self.__storage__.pop(self.__ident_func__(), None)

    def __getattr__(self, name):
        try:
            return self.__storage__[self.__ident_func__()][name]
        except KeyError:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        ident = self.__ident_func__() # 1111
        storage = self.__storage__
        try:
            storage[ident][name] = value
        except KeyError:
            storage[ident] = {name: value}

    def __delattr__(self, name):
        try:
            del self.__storage__[self.__ident_func__()][name]
        except KeyError:
            raise AttributeError(name)

class LocalStack(object):
    def __init__(self):
        self._local = Local()
    def push(self, obj):
        """Pushes a new item to the stack"""
        # self._local.stack == getattr
        # rv = None
        rv = getattr(self._local, "stack", None)
        if rv is None:
            self._local.stack = rv = []
        rv.append(obj)
        return rv

    def pop(self):
        stack = getattr(self._local, "stack", None)
        if stack is None:
            return None
        elif len(stack) == 1:
            # release_local(self._local)
            # del __storage__[1111]
            return stack[-1]
        else:
            return stack.pop()

    @property
    def top(self):
        try:
            return self._local.stack[-1]
        except (AttributeError, IndexError):
            return None

obj = LocalStack()
```

```
obj.push('汪洋')
obj.push('成说')

print(obj.top)

obj.pop()
obj.pop()
```

总结:

```
在flask中有个local类，他和threading.local的功能一样，为每个线程开辟空间进行存取数据，他们两个的内部实现机制，内部维护一个字典，以线程(协程)ID为key，进行数据隔离，如：
    __storage__ = {
        1211:{'k1':123}
    }

    obj = Local()
    obj.k1 = 123

在flask中还有一个LocalStack的类，他内部会依赖local对象，local对象负责存储数据，localstack对象用于将local中的值维护成一个栈。
    __storage__ = {
        1211:{'stack':['k1',]}
    }

    obj= LocalStack()
    obj.push('k1')
    obj.top
    obj.pop()
```

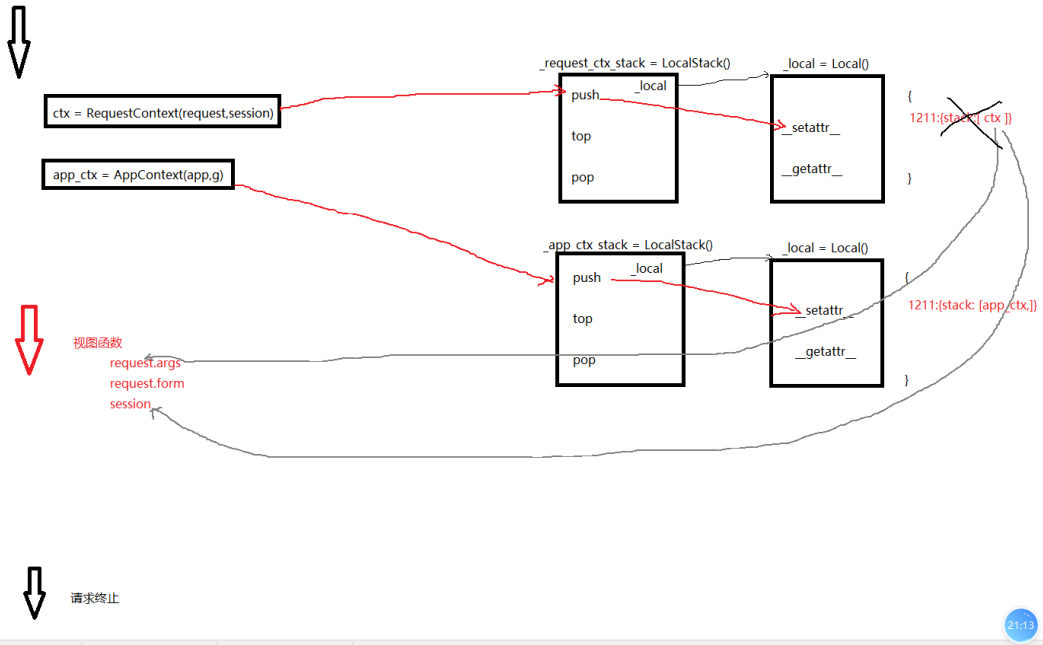
7.flask源码中总共有2个localstack对象

```
# context locals
__storage__ = {
    1111:{'stack':[RequestContext(reqeust,session),]},
    1123:{'stack':[RequestContext(reqeust,session),]},
}
_request_ctx_stack = LocalStack()


__storage__ = {
    1111:{'stack':[AppContenxt(app,g),]}
    1123:{'stack':[AppContenxt(app,g),]},
}
_app_ctx_stack = LocalStack()
```

```
_request_ctx_stack.push('小魔方')
_app_ctx_stack.push('大魔方')
```

- 上下文管理
  - 请求上下文管理
  - 应用上下文管理



7.源码初识

7.1 项目启动

- 实例化Flask对象

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

```
1. 对app对象封装一些初始化的值。
    app.static_url_path
    app.static_folder
    app.template_folder
    app.view_functions = {}
2. 添加静态文件的路由
    self.add_url_rule(
        self.static_url_path + "/<path:filename>",
        endpoint="static",
        host=static_host,
        view_func=self.send_static_file,
        )

3. 实例化了url_map的对象，以后在map对象中放 【/index/ 函数的对象应观】
    class Flask(object):
        url_rule_class = Rule
        url_map_class = Map

        def __init__(self...):
            self.static_url_path
            self.static_folder
            self.template_folder
            self.view_functions = {}
            self.url_map = self.url_map_class()
    app = Flask()
    app.view_functions
app.url_rule_class
```

- 加载配置文件（给app的config进行赋值）

```
from flask import Flask

app = Flask(__name__,static_url_path='/xx')

app.config.from_object('xx.xx')
```

```
1. 读取配置文件中的所有键值对，并将键值对全都放到Config对象。（Config是一个字典）
2. 把包含所有配置文件的Config对象，赋值给 app.config
```

- 添加路由映射

```
from flask import Flask

app = Flask(__name__,static_url_path='/xx')

@app.route('/index')
def index():
    return 'hello world'
```

```
1. 将 url = /index  和  methods = [GET,POST]  和 endpoint = "index"封装到Rule
对象

2. 将Rule对象添加到 app.url_map中。

3. 把endpoint和函数的对应关系放到 app.view_functions中。
```

- 截止目前

```
app.config
app.url_map
app.view_functions
```

- 运行flask

```
from flask import Flask

app = Flask(__name__,static_url_path='/xx')

@app.route('/index')
def index():
    return 'hello world'

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

```
1. 内部调用werkzeug的run_simple，内部创建socket，监听IP和端口，等待用户请求到来。

2. 一旦有用户请求，执行app.__call__方法。

    class Flask(object):
        def __call__(self,envion,start_response):
            pass
        def run(self):
            run_simple(host, port, self, **options)

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

7.2 有用户请求到来

- 创建ctx = RequestContext对象，其内部封装了 Request对象和session数据。
- 创建app_ctx = AppContext对象，其内部封装了App和g。
- 然后ctx.push触发将 ctx 和 app_ctx 分别通过自己的LocalStack对象将其放入到Local中，Local的本质是以线程ID为key，以{“stack”:[]}为value的字典。

  ```
  {
      1111:{“stack”:[ctx,]}
  }

  {
      1111:{“stack”:[app_ctx,]}
  }
  ```

  注意：以后再想要获取 request/session / app / g时，都需要去local中获取。
- 执行所有的before_request函数
- 执行视图函数
- 执行所有after_request函数（session加密放到cookie中）
- 销毁ctx和app_ctx

8.了解源码流程之后，使用：session、request、app、g

- 偏函数

  ```
  import functools

  # 偏函数
  """
  def func(a1,a2):
      print(a1,a2)

  new_func = functools.partial(func,123)
  new_func(2)
  """
  ```

- 私有成员

```
class Foo:
    def __init__(self):
        self.name = 'alex'
        self.__age = 123


obj = Foo()

print(obj.name)
print(obj._Foo__age)
```

- setattr
- setitem

```
# session, request, current_app, g 全部都是LocalProxy对象。
"""
session['x'] = 123     ctx.session['x'] = 123
request.method         ctx.request.method
current_app.config    app_ctx.app.config
g.x1                  app_ctx.g.x1
"""
```

9.g到底是个什么鬼?

```
在一次请求请求的周期，可以在g中设置值，在本次的请求周期中都可以读取或复制。
相当于是一次请求周期的全局变量。
```

```
from flask import Flask,g

app = Flask(__name__,static_url_path='/xx')

@app.before_request
def f1():
    g.x1 = 123

@app.route('/index')
def index():
    print(g.x1)
    return 'hello world'

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

总结

- 第一阶段：启动flask程序，加载特殊装饰器、路由，把他们封装 app= Flask对象中。
- 第二阶段：请求到来

    - 创建上下文对象：应用上下文、请求上下文。
    - 执行before / 视图 / after
    - 销毁上下文对象

扩展

```
class Foo:
     #

obj = Foo()
obj() # __call__

obj[x1] = 123 # __setitem__
obj[x2]  # __getitem__

obj.x1 = 123 # __setattr__
obj.x2  # __getattr__
```

SQLhelper

- 方式一

    ```
    import pymysql
    import threading
    ```

```python
from DBUtils.PooledDB import PooledDB

"""
storage = {
    1111:{'stack':[]}
}
"""

class SqlHelper(object):
    def __init__(self):
        self.pool = PooledDB(
            creator=pymysql,  # 使用链接数据库的模块
            maxconnections=6,  # 连接池允许的最大连接数，0和None表示不限制连接数
            mincached=2,  # 初始化时，链接池中至少创建的链接，0表示不创建
            blocking=True,  # 连接池中如果没有可用连接后，是否阻塞等待。True，等待；
False，不等待然后报错
            ping=0,
            # ping MySQL服务端，检查是否服务可用。# 如：0 = None = never, 1 =
default = whenever it is requested, 2 = when a cursor is created, 4 = when a
query is executed, 7 = always
            host='127.0.0.1',
            port=3306,
            user='root',
            password='222',
            database='cmdb',
            charset='utf8'
        )
        self.local = threading.local()

    def open(self):
        conn = self.pool.connection()
        cursor = conn.cursor()
        return conn, cursor

    def close(self, cursor, conn):
        cursor.close()
        conn.close()

    def fetchall(self, sql, *args):
        """ 获取所有数据 """
        conn, cursor = self.open()
        cursor.execute(sql, args)
        result = cursor.fetchall()
        self.close(conn, cursor)
        return result

    def fetchone(self, sql, *args):
        """ 获取所有数据 """
        conn, cursor = self.open()
        cursor.execute(sql, args)
        result = cursor.fetchone()
        self.close(conn, cursor)
        return result

    def __enter__(self):
        conn,cursor = self.open()
        rv = getattr(self.local,'stack',None)
        if not rv:
```

```
            self.local.stack = [(conn,cursor),]
        else:
            rv.append((conn,cursor))
            self.local.stack = rv
        return cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        rv = getattr(self.local,'stack',None)
        if not rv:
            # del self.local.stack
            return
        conn,cursor = self.local.stack.pop()
        cursor.close()
        conn.close()

db = SqlHelper()
```

```
from sqlhelper import db


# db.fetchall(...)
# db.fetchone(...)

with db as c1:
    c1.execute('select 1')
    with db as c2:
        c1.execute('select 2')
    print(123)
```

- 方式二

```
import pymysql
import threading
from DBUtils.PooledDB import PooledDB

POOL = PooledDB(
            creator=pymysql,  # 使用链接数据库的模块
            maxconnections=6,  # 连接池允许的最大连接数，0和None表示不限制连接数
            mincached=2,  # 初始化时，链接池中至少创建的链接，0表示不创建
            blocking=True,  # 连接池中如果没有可用连接后，是否阻塞等待。True，等待；
False，不等待然后报错
            ping=0,
            # ping MySQL服务端，检查是否服务可用。# 如：0 = None = never, 1 =
default = whenever it is requested, 2 = when a cursor is created, 4 = when a
query is executed, 7 = always
            host='127.0.0.1',
            port=3306,
            user='root',
            password='222',
            database='cmdb',
            charset='utf8'
        )

class SqlHelper(object):
```

```
    def __init__(self):
        self.conn = None
        self.cursor = None

    def open(self):
        conn = POOL.connection()
        cursor = conn.cursor()
        return conn, cursor

    def close(self):
        self.cursor.close()
        self.conn.close()

    def __enter__(self):
        self.conn,self.cursor = self.open()
        return self.cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()
```

```
# ################# 使用 ##################

with SqlHelper() as c1:
    c1.execute('select 1')
    with SqlHelper() as c2:
        c2.execute('select 2')
    print(666)

with SqlHelper() as cursor:
    cursor.execute('select 1')

with SqlHelper() as cursor:
    cursor.execute('select 1')
```

作业

1.drf源码分析系列

- 01 restful规范
- 02 从cbv到drf的视图 / 快速了解drf
- 03 视图
- 04 版本
- 05 认证
- 06 权限
- 07 节流
- 08 jwt
- 持续更新中...

2.flask源码分析系列

- 01 werkzurg 了解wsgi
- 02 快速使用
- 03 threading.local和高级
- 04 LocalStack和Local对象实现栈的管理
- 05 Flask源码之：配置加载
- 06 Flask源码之：路由加载
- 持续更新中...

印象笔记 、 有道云笔记