flask扩展第三方包

参考：https://blog.csdn.net/shifengboy/article/details/114274271

- Flask-SQLalchemy：操作数据库，ORM
- Flask-script：终端脚本工具，脚手架
- Flask-migrate：管理迁移数据库
- Flask-Session：Session存储方式指定
- Flask-WTF：表单
- Flask-mail：邮件
- Flask-Bable：国际化
- Flask-Login：认证用户状态
- Flask-OpenID：认证，OAuth
- Flask-RESTful：开发REST API的工具
- Flask JSON-RPC：开发rpc远程服务[过程]调用

flask与django的区别

flask	django
轻量级	大而全
路由，视图，模板（jinja2）/session/中间件，第三方组件齐全。django的请求是逐一封装和传递；flask的请求是利用上下文管理来实现的	django提供orm，session,cookie,admin,form.modelform，路由，视图，模板，中间件，分页，auth,contenttype,缓存，信号，多数据库链接

1. flask的快速使用

安装：pip3 install flask

1.1 依赖wsgi WerKzeug

```
from werkzeug.serving import run_simple

def func(environ,start_response):
    print("请求来了")
    pass

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

1.2 快速使用flask

```
from flask import Flask,render_template,jsonify

app=Flask(__name__)

#返回页面
@app.route('/index')
def index():
```

```
    return render_template("login.html")

#返回json字符串
@app.route('/json')
def json():
    return jsonify({"code":200,'data':[1,2,3]})

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

1.3 session相关

```
#设置session
session["xxx"] = "1234"
session.permanent = True
app.permanent_session_lifetime = timedelta(minutes=10)  # 设置session到期时间
#取session
session.get("xxx")
#清理session
session.pop("xxx")
session.clear()
```

1.3.1 登录功能快速实现

```
from flask import Flask,render_template,jsonify

app=Flask(__name__)

#返回页面
@app.route('/index')
def index():
    return render_template("login.html")

#返回json字符串
@app.route('/json')
def json():
    return jsonify({"code":200,'data':[1,2,3]})

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

1.4 简单登陆案例

```
from flask import Flask, render_template, jsonify, request
from werkzeug.utils import redirect

app=Flask(__name__)

@app.route('/index')
def index():
    return jsonify({"code":200})

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

```
    if request.method=='GET':
        return render_template('login.html')
    user=request.form.get('user')
    pwd=request.form.get('pwd')

    if user=='admin' and pwd=='dsb':
        return redirect("/index")
    error='用户名或密码错误'
    return render_template("login.html",error=error)

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

1.5 取值操作

html端：render_template返回的值{%for key,value in data_dict.items()%}　{%endfor%}

服务端：

- /edit? nid=1　　request.args.get('nid')
- /edit/1　　@app.route('/edit/<int:nid>') 方法传nid

1.6 总结

1.flask路由

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

2.路由的参数

```
@app.route('/login',methods=['GET','POST'],endpoint='login')
def login():
    pass
#endpoint是/login路径的别名 redirect('/index')==redirect(url_for(login))
#未定义endpoint默认等于函数名
```

3.动态路由

```
@app.route('/login')
def login():
    pass

@app.route('/login/<name>')
def login(name):
    pass

@app.route('/login/<int:nid>')
def login(nid):
    pass
```

4.获取提交的数据

```
@app.route('/login')
def login():
    request.args #GET形式传递的参数
    request.form #POST形式传递的参数
```

5.返回数据

```
@app.route('/index')
def login():
    return render_template('模板文件',)
    return jsonfiy()
    return redirect('url路径')
    return redirect(url_for('路径别名endpoint'))
    return "……"
```

6.模板处理中

```
{{value}}
{{%for i in list%}}
    {{i}}
{% endfor %}
```

7.html模板注释

```
{#html#}
```

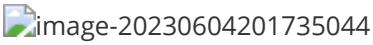
1.7 装饰器相关

```
import functools
def auth(func):
    @functools.wraps(func)
    def inner(*args,**kwargs):
        return func(*args,**kwargs)
    return inner

@auth
def login():
    pass
```

2. 蓝图

2.1 构建目录结构



2.2 创建蓝图

```
from flask import Blueprint

login=Blueprint('view1',__name__)

@login.route('/f1')
def f1():
    return 'f1'

@login.route('/f2')
def f2():
    return 'f2'
```

注册使用蓝图

```
from flask import Flask
from .views.view1 import login

def create_app():
    app=Flask(__name__)
    app.secret_key="asdasfwrfwdas"

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

    app.register_blueprint(login)#注册蓝图
    return app
```

3. 数据库连接池

3.1 普通数据库连接

```
conn=pymysql.connect("host:127.0.0.1",port=3306,user='root',password='123456',db
='myobject')
    cursor=conn.cursor()
    cursor.execute("select * from url_collection")
    result=cursor.fetchall()
    conn.commit()
    cursor.close()
    conn.close()
```

3.2 数据库连接池

安装

```
pip install pymysql
pip install dbutils
```

模式一：为每个线程创建一个连接，线程即使调用了close方法，也不会关闭，只是把连接重新放到连接池，供自己线程再次使用。当线·程终止时，连接自动关闭。

```
POOL = PersistentDB(
    creator=pymysql,  # 使用链接数据库的模块
```

```
    maxusage=None,  # 一个链接最多被重复使用的次数，None表示无限制
    setsession=[],  # 开始会话前执行的命令列表。如：["set datestyle to ...", "set time zone ..."]
    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
    closeable=False,
    # 如果为False时， conn.close() 实际上被忽略，供下次使用，再线程关闭时，才会自动关闭链接。如果为True时， conn.close()则关闭链接，那么再次调用pool.connection时就会报错，因为已经真的关闭了连接（pool.steady_connection()可以获取一个新的链接）
    threadlocal=None,  # 本线程独享值得对象，用于保存链接对象，如果链接对象被重置
    host='127.0.0.1',
    port=3306,
    user='root',
    password='123',
    database='pooldb',#选择数据库
    charset='utf8'
)

def func():
    conn = POOL.connection(shareable=False)
    cursor = conn.cursor()
    cursor.execute('select * from tb1')
    result = cursor.fetchall()
    cursor.close()
    conn.close()

func()
```

模式二：创建一批连接到连接池，供所有线程共享使用

PS：由于pymysql、MySQLdb等threadsafety值为1，所以该模式连接池中的线程会被所有线程共享

```
import time
import pymysql
import threading
from DBUtils.PooledDB import PooledDB, SharedDBConnection
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空闲的链接，0表示不创建
    maxcached=5,  # 链接池中最多闲置的链接，0和None不限制
    maxshared=3,  # 链接池中最多共享的链接数量，0和None表示全部共享。PS: 无用，因为pymysql和MySQLdb等模块的 threadsafety都为1，所有值无论设置为多少，_maxcached永远为0，所以永远是所有链接都共享。
    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
    maxusage=None,  # 一个链接最多被重复使用的次数，None表示无限制
    setsession=[],  # 开始会话前执行的命令列表。如：["set datestyle to ...", "set time zone ..."]
    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='123',
    database='pooldb',
    charset='utf8'
)


def func():
    # 检测当前正在运行连接数的是否小于最大链接数，如果不小于则：等待或报raise
TooManyConnections异常
    # 否则
    # 则优先去初始化时创建的链接中获取链接 SteadyDBConnection。
    # 然后将SteadyDBConnection对象封装到PooledDedicatedDBConnection中并返回。
    # 如果最开始创建的链接没有链接，则去创建一个SteadyDBConnection对象，再封装到
PooledDedicatedDBConnection中并返回。
    # 一旦关闭链接后，连接就返回到连接池让后续线程继续使用。
    conn = POOL.connection()

    # print(th, '链接被拿走了', conn1._con)
    # print(th, '池子里目前有', pool._idle_cache, '\r\n')

    cursor = conn.cursor()
    cursor.execute('select * from tb1')
    result = cursor.fetchall()
    conn.close()

func()
```

3.3 数据库连接池函数写法

```
#数据库连接池函数写法
import pymysql
from DBUtils.PooledDB import PooledDB

Pool = PooledDB(
    creator=pymysql,
    maxconnections=6,
    mincached=2,
    blocking=True,
    ping=0,
    host='127.0.0.1',
    port=3306,
    user='root',
    password="123456",
    database="myobject",
    charset='utf8'
)

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

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

```
def fetchall(sql,*args):
    conn,cursor=open()
    cursor.execute(sql,args)
    result=cursor.fetchall()
    close(cursor, conn)
    return result

def fetchone(sql,*args):
    conn,cursor=open()
    cursor.execute(sql, args)
    result = cursor.fetchone()
    close(cursor,conn)
    return result
```

3.4 数据库连接池对象写法（推荐）

```
from DBUtils.PooledDB import PooledDB
import pymysql


class Pool():
    def __init__(self):
        self.pool = PooledDB(
            creator=pymysql,
            maxconnections=6,
            mincached=2,
            blocking=True,
            ping=0,
            host='127.0.0.1',
            port=3306,
            user='root',
            password="123456",
            database="myobject",
            charset='utf8'
        )

    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):
        conn,cursor=self.open()
        cursor.execute(sql, args)
        result = cursor.fetchall()
        self.close(conn,cursor)
        return result

    def fetchone(self,sql,*args):
        conn,cursor=self.open()
        cursor.execute(sql,* args)
        result=cursor.fetchone()
        self.close(cursor,conn)
```

```
        return result

db=Pool()
```

3.5 上下文管理

```
class Foo(object):
    def do_something(self):
        pass

    def close():
        pass

class Context(object):
    def __enter__(self):
        self.foo=Foo()
        return self.foo

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

with Context() as ctx:
    ctx.do_something()
```

4. 静态文件处理

static_folder：静态文件目录，默认`static`

static_url_path：前端页面的文件访问路径。默认`static`

网页访问静态资源路径：

- /static/文件 == /static_url_path/文件 例：<img src="/static/1.jpg">（不推荐）
- 推荐{{url_for('static' , filename= 'mm.jpg')}} 例：<img src="{{url_for('static' , filename= 'mm.jpg')}}">（推荐）

5. 配置文件

5.1 全局变量的配置

```
#localsettings(只在本地存在，git提交排除，服务器端自定义localsettings)
username='test'
password='test'
```

```
#settings
try:
    from .localsettings import *
except:
    pass
```

```
#app
app.config.fromobject('config.setting')
```

5.2 类的配置文件

```
#BaseSettings 存放公共配置
class BaseSettings(object):
    common='123'

#存放开发环境配置
class DevSettings(BaseSettings):
    Host='127.0.0.1'

#存放生产环境配置
class ProdSettings(BaseSettings):
    Host='192.168.1.1'
```

6. 路由

路由的两种写法:

```
# 方式一
def index():
    return render_template("index.html")

app.add_url_rule('/index','index',index)

# 方式二
@app.route('/login')
def login():
    return render_template('login.html')
```

路由的加载源码流程

```
- 将url和函数打包成rule对象
- 将rule对象添加到map对象中
- app.url_map = map对象
```

动态路由

```
@app.route('/login')
def login():
    return render_template('login.html')

@app.route('/login/<name>')
def index(name):
    print(type(name))
    return render_template('login.html')

@app.route('/login/<int:name>')
def index(name):
    print(type(name))
    return render_template('login.html')
```

- @app.route('/user/<username>')
- @app.route('/post/<int:post_id>')
- @app.route('/post/<float:post_id>')

- @app.route('/post/<path:path>')
- @app.route('/login',methods=['GET', 'POST'])

7. 模板

继承

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

{% block content %}
    <h1>MD</h1>
    {% include 'form.html'%}
{% endblock %}
```

全局函数

```
from flask import Flask,render_template

app=Flask(__name__)

@app.template_global()
def func(arg):
    return 'hello'+arg

@app.route(/md/hg)
def index():
    return render_template('md_hg.html')

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

```
<body>
    {{func('world')}}
</body>
```

8. 特殊装饰器

9. Flask-SQLalchemy

`常见数据类型：`

- Float：浮点型。
- Boolean：传递True/false进去。
- DECIMAL:定点类型。
- enum：枚举类型
- Date：传递datetime.date()进去。
- DateTime：传递datetime.datetime()进去。
- Time：传递datetime.time()进去。
- String:字符类型，使用时需要指定长度，区分Text类型。
- Text：文本类型。
- LONGTEXT：长文本类型

`Column常用参数：`

- default：默认值。

- nullable：是否为空。
- primary_key：是否是为主键。
- unique：是否唯一。
- autoincrement：是否自增长。
- onupdate：更新的时间执行的函数。
- name：该属性在数据库中的字段映射。

聚合函数：

- func.count:统计行的数量。
- func.avg:求平均值。
- func.max:求最大值。
- func.min:求最小值。
- func.sum:求和。

9.1 FlaskSQLchemy

- SQLALchemy实际上是对数据库的抽象，让开发者不用直接和sql语句打交道，而是通过Python对象来操作数据库，在舍弃一些性能开销的同时，换来的是开发效率的较大提升
- SQLALchemy是关系型数据库框架，它提供了高层的ORM和底层的原生数据库的操作，flask-sqlalchemy是一个简化SQLAlchemy操作flask的扩展

9.2 安装flask-sqlalchemy

```
pip install flask-sqlalchemy
```

如果连接的是mysql数据库，需要安装mysqldb

```
pip install pymysql
```

9.3 使用flask-sqlAlchemy管理数据库

在flask-sqlalchemy中，数据库使用URL指定，而且程序使用的数据库必须保存到Flask配置对象的SQLALCHEMY_DATABASE_URI键中。

```
#数据库配置
app.config['SQLALCHEMY_DATABASE_URI']="mysql+pymysql://root:123456@127.0.0.1:3306/myobject"

#其他配置
#动态追踪修改设置，如果未设置只会提示警告，不建议开启
app.config['SQLALCHEMY_TRACK_MODIFICATIONS']=False
#查询时会显示原始SQL语句
app.config['SQLALCHEMY_ECHO']=True
```

9.4 定义数据模型

```
#数据库模型，需要继承db.model
class Role(db.model):
    #定义表名
    __tablename__='roles'
    #定义字段
    #db.Column表示一个字段
    id=db.Column(db.Interger,primary_key=True)
    name=db.Column(db.String(16),unique=True)
```

```
    #在1的1方，写关联（role：User是一对多）
    users1=db.relationship('User'):#表示和User模型发生关联，增加一个users属性
    #backref='role':表示role是User要用的一个属性
    users2=db.relationship('User',backref='role')


class User(db.Model):
    __tablename__='users'
    id=db.Columu(db.Integer,primary_key=True)
    name=db.Column(db.String(16),unique=True)
    #db.ForeignKey（'roles.id'）表示外键，表名.id
    role_id=db.Column(db.Integer,db.ForeignKey('roles.id'))

    #user希望有role属性，但是这个属性的定义，需要在另外一个模型中定义
```

9.5 增删改操作

- 插入，修改，删除操作，均由数据库会话管理（会话用db.session表示，在准备把数据库写入数据库前，要先将数据添加到会话中然后调用`commit()`方法提交会话）
- 查询操作是通过`query对象`操作数据

```
db.session.add(role)
db.session.add_all([user1,user2])#添加多条数据到session中
db.session.commit()#提交数据库的修改(包括增/删/改)
db.session.rollback()#数据的回滚操作
db.session.delete(user)#删除数据库（需要跟上commit）
```

9.6 查询操作

过滤器	说明
filter()	把过滤器添加到原查询上，返回一个新查询
filter_by()	把等值过滤器添加到原查询上，返回一个新查询
limit()	使用指定的值限定原查询返回的结果
offset()	偏移原查询返回的结果，返回一个新查询
order_by()	根据指定条件对原查询结果进行排序，返回一个新查询
group_by()	根据指定条件对原查询结果进行分组，返回一个新查询

方法	说明
all()	以列表形式返回所有查询的结果
first()	返回查询的第一个结果，如果未查到，返回None
first_or_404()	返回查询的第一个结果，如果未查到，返回404
get()	返回指定主键对应的行，如果不存在，返回None
get_or_404()	返回指定主键对应的行，如果不存在，返回404
count()	返回查询结果的数量

10. 项目部署

10.1 安装gcc

```
yum install gcc -y
```

10.2 安装Python 依赖

```
yum install zlib zlib-devel -y
yum install bzip2 bzip2-devel -y
yum install ncurses ncurses-devel -y
yum install readlines readlines-devel -y
yum install openssl openssl-devel -y
yum install xz lzma xz-devel -y
yum install sqlite sqlite-devel -y
yum install gdbm gdbm-devel -y
yun install tk tk-devel -y
yum install mysql-devel -y
yum install python-devel -y
yum install libffi-devel -y
```

10.3 下载源码Python3.9源码

```
yum install wget -y
```

```
wget https://www.python.org/ftp/python/3.8.8/Python-3.8.8.tgz
```

10.4 解压&编译&安装

```
#解压
tar -xvf Python-3.9.5.tgz
#进入目录并编译安装
cd Python-3.8.8
./configure
make all
make install

#测试
python -V

/usr/local/bin/python3
/usr/local/bin/pip3
/usr/local/bin/pip3.8

#配置豆瓣源
pip3.8 config set global.index-url https://pypi.douban.com/simple
```

10.5 安装虚拟环境

```
#按装virtulenv
pip3.8 install virtualenv

#创建虚拟环境
```

```
mkdir /envs
virtualenv /env/nb --python=python3.8

#激活虚拟环境
source /envs/nb/bin/activate

#修改centos默认python版本
mv /usr/bin/python /usr/bin/python22

#建立软连接
ln -s /usr/local/bin/python3 /usr/bin/python

#修改yum配置文件（把文件头部的#!/usr/bin/python改成#!/usr/bin/python22）
vi /usr/bin/yum
```

10.6 安装uwsgi

```
#激活虚拟环境
source /evns/nb/bin/activate

#安装uwsgi
pip install uwsgi

#基于uwsgi运行flask项目
cd 项目目录

- 命令方式
    uwsgi -http :8080 --wsgi-file app.py --callable app

- 配置文件(推荐)
nb_uwsgi.ini
    [uwsgi]
    socket=127.0.0.1:8082
    chdir= /data/项目目录
    wsgi-file = app.py
    callable = app
    processes=1
    virtualenv = /evns/nb/

#在虚拟环境中启动uwsgi
uwsgi --ini nb_uwsgi.ini &

#停止
ps -ef|grep nb_uwsgi
kill -9 pid
```