为了了解全栈知识，以及为了更多的学习一些后端方面的知识，而不仅仅局限于前端的方面，将在视频中学习到的一些知识整理成文章，方便日后查阅，回顾。

* flask 光速入门感受
* 一个行之有效的框架展示及从0开始完成一个网站的登陆和注册功能
* flask 中如何处理路由
* flask 中如何与数据库交互(M)
* flask 中view的编写(V)
* flask 中controller层编写(C)

**1. Flask 入门体验 -- hello world**

**//** 打开终端

mkdir part**-**1

cd part**-**1

touch index**.**py

touch requirement**.**txt

*# 在requirement.txt 写入 Flask 然后下载Flask*

pip install **-**r requirement**.**txt

*# 在Flask中 体验hello world*

**from** flask **import** Flask

*# 初始化一个Falsk实例*

app **=** Flask(\_\_name\_\_)

@app**.**route("/")

**def** **hello**():

**return** 'hello world‘

app**.**run(host**=**'127.0.0.1', port**=**3000, debug**=True**)

这样，当请求127.0.0.1的时候，页面就会出现了hello world。光速体验一波flask，但是认真理解一下，首先初始化一个flask对象，@app.route则是引入路由装饰器，然后定义一个方法，这个方法在请求到根节点路由时会自动执行。最后在本地的3000端口启动该服务。

**2. 一个行之有效的框架搭建**

# 框架骨架展示

common

-- models # 用来存放与数据库中的表相关联的文件

-- xxx.py

-- libs # 存放公共类

config # 用来存放本地环境和生产环境的变量

-- baseConfig.py

-- localConfig.py

-- productionConfig.py

controllers # MVC 中控制层的文件

application.py # 全局的变量类

www.py # 路由的核心文件

manager.py # 入口文件

requirement.txt # python 的扩展文件 用来安装依赖

static # 用来存放静态资源

-- js

-- css

-- img

templates # 模版文件即html结合flask的一些语法文件

-- common # 存放一些公共页面

-- layout.html

-- index.html

一点一点开始搭建。如下图所示，框架搭建的目的是能够形成一个首页，登陆注册页之间形成一个完整的闭环。

1.首先 在入口文件创建实例化对象

# application.py 全局变量类

from flask import Flask

app = Flask(\_\_name\_\_)

# manager.py

from application import app

app.run(host='127.0.0.1', port=3000, DEBUG=True)

# 此时 已经可以通过python manager.py 执行该文件启动一个web服务器 但没有任何返回

2.构建一个路由管理对象即处理 www.py文件

# 此时 引入一个新的名词 叫做蓝图 flask 通过蓝图Blueprint来管理注册这个整个应用的路由，

# 可以不需要，但是更规范

cd controllers

touch index.py member.py

# index.py

from flask import Blueprint

# 可以利用蓝图的前缀 然后将页面层次区分开

# 1. 首先实例化一个index\_page

index\_page = Blueprint( "index\_page", \_\_name\_\_ )

# 2. 使用实例的装饰器

# 即当请求到对应路径时 会自动触发相应方法

@index\_page.route('/')

def index():

return 'welcome to index page'

# member.py 是控制登陆和注册页面的路由 同上

然后在www.py文件中注册路由

# www.py

from controllers.index import index\_page

from controllers.member import member\_page

from application import app

app.register\_blueprint( index\_page, url\_prefix='/')

app.register\_blueprint( member\_page, url\_prefix='/member‘)

最后在manager.py(启动文件中)引入路由，这样路由就注册好了

# manager.py

from application import app

from www import \*

app.run(host='127.0.0.1', port=3000, DEBUG=True)

3.flask 通过 jinja2来返回模版页面，即请求注册页面时，页面显示的不再是字符串，而是html模版页面。

当需要用到jinja2的时候，就需要之前创建的templates文件夹。拿访问根路径举例

原来的代码，需要通过引入模版，代替返回的字符串，

# controllers/index.py

from flask import Blueprint, render\_template

# 可以利用蓝图的前缀 然后将页面层次区分开

# 1. 首先实例化一个index\_page

index\_page = Blueprint( "index\_page", \_\_name\_\_ )

# 2. 使用实例的装饰器

# 即当请求到对应路径时 会自动触发相应方法

@index\_page.route('/')

def index():

return 'welcome to index page'

@index\_page.route("/template")

def template():

# 传值

name = 'imooc'

return render\_template("index.html", name=name)

然后在template下新建一个index.html页面,，并写入一些html。

最后的展示如下

然后通过在template下新建一个common/layout.html

作为页面的一个整体的布局框架，然后通过路由去匹配每一个不同的页面。

# common/layout

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>layout页面</title>

<!--bootstrap 核心样式--->

{% block css %} {% endblock %}

</head>

<body>

<nav class="navbar navbar-inverse" style="border-radius: 0">

<div class="container-fluid">

<!-- Brand and toggle get grouped for better mobile display -->

<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

<ul class="nav navbar-nav">

<li class="active"><a href="/">首页</a></li>

<li class="active"><a href="/member/reg">注册</a></li>

<li class="active"><a href="/member/login">登陆</a></li>

</ul>

</div><!-- /.navbar-collapse -->

</div><!-- /.container-fluid -->

</nav>

<div class="container" style="min-height: 600px">

{% block content %} {% endblock %}

</div>

{% block js %}

{% endblock %}

</body>

</html>

然后对应的页面需要做一定的修改，使用extends关键字

# index.html

{% extends "common/layout.html" %}

{% block content %}

<p>这其实是首页</p>

{% endblock %}

最后的展示结果如下:



可以通过页面切换到不同的内容。然后把页面的样式稍微优化一下。

4. 与数据库进行交互

使用flask-sqlalchemy关联到数据库，整个库是flask内部封装好的数据库ORM。首先进行两个模块库的安装

flask-sqlalchemy

msyqlclient

# application.py 文件中 进行初始化

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

app = Flask( \_\_name\_\_ )

# 链接数据库 这里的xxx是本机的管理员密码

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'mysql://root:xxx@127.0.0.1/mysql'

db = SQLAlchemy( app )

这样其他地方就能使用连接到数据库进行使用了，使用举例如下：

# controllers/index.py 中返回一个数据库测试

@index\_page.route('/sql-test‘)

def sql\_test():

from sqlalchemy imoprt test

from application import db

sql = text( 'select \* from `user`')

result = db.engine.execute( sql )

context = {}

context['result'] = result

return render\_template('index.html', \*\*context)

最后修改一下index.html的文件 测试是否有返回

{% extends "common/layout.html" %}

{% block content %}

<p>这其实是首页</p>

{% for item in result %}

{{ item['User'] }}

{% endfor %}

{% endblock %}

展示如下：

成功将数据库中的用户名信息返回出来了。

升级～使用model来访问数据库

1.使用model关联数据库

2.也可以使用model来创建数据库中没有的表

# 如何使用model来升级数据库的操作

# 1. 安装 flask-sqlacodegen

# 2. 建立软连接 ln -s /usr/local/python3/bin/flask-sqlacodegen /usr/localbin

# 3. flask-sqlacodegen "mysql://root:xxx(密码)@127.0.0.1/mysql” --tables user --outfile “commom/m“““ls/user.py“ --flask

在common/model下会新生成一个user.py文件，里面是关于数据库中use表的信息,展示如下

# coding: utf-8

from sqlalchemy import Column, DateTime, Integer, LargeBinary, SmallInteger, String, Text

from sqlalchemy.schema import FetchedValue

from sqlalchemy.dialects.mysql.enumerated import ENUM

from flask\_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):

\_\_tablename\_\_ = 'user'

Host = db.Column(db.String(60, 'utf8\_bin'), primary\_key=True, nullable=False, server\_default=db.FetchedValue())

User = db.Column(db.String(32, 'utf8\_bin'), primary\_key=True, nullable=False, server\_default=db.FetchedValue())

......

然后将该文件的db改为从application中引入

from application import db

class User(db.Model):

\_\_tablename\_\_ = 'user'

Host = db.Column(db.String(60, 'utf8\_bin'), primary\_key=True, nullable=False, server\_default=db.FetchedValue())

User = db.Column(db.String(32, 'utf8\_bin'), primary\_key=True, nullable=False, server\_default=db.FetchedValue())

此时就可以将之前的sql语句升级了

# 同样通过请求 sql-test 页面，修改其中的连接数据库的查询语句

# controllers/index.py 中返回一个数据库测试

from common.model.user import User

@index\_page.route('/sql-test‘)

def sql\_test():

# from sqlalchemy imoprt test

# from application import db

# sql = text( 'select \* from `user`')

# result = db.engine.execute( sql )

# context = {}

# context['result'] = result

# 新写法

context = {}

result = User.query.all()

context.result = result

return render\_template('index.html', \*\*context)

这样就可以查找到了。

现在将登陆页面和注册页面优化一下，添置几个输入框。

# 注册逻辑如下

1. 用户首先在注册页面进行注册操作 注册时 填入login\_name login\_pwd，前端校验注册信息是否正确

2. 后端接受到请求之后，对请求体的数据进行校验，防止脏数据影响

3. 通过刚才升级的sql语法，在数据库中查找是否已经有注册该用户，如果用户已经注册，则返回提示

若当前注册的用户未存在，将该条数据插入到数据库中，并返回注册成功

# 登陆逻辑如下

1. 用户首先在登陆页面进行注册操作 注册时 填入login\_name login\_pwd 前端校验登陆信息是否正确

2. 后端校验登陆数据

3. 首先从数据库中查找是否有该登陆用户的数据，如果没有，提示登陆错误。如果有存在当前用户的数据，

则提示登陆成功。

上述是登陆注册的简单操作，代码如下：

# member.py

from flask import Blueprint, render\_template, request, jsonify

from common.libs.UrlManager import UrlManager

from common.models.user import User

from application import db

# 1. 首先实例化一个 member\_page

member\_page = Blueprint('member\_page‘, \_\_name\_\_)

# 2. 使用实例装饰器 当请求到对应路径时 会自动触发相应方法

@member\_page.route('/reg‘, methods=[“GET“, “POST“])

def reg():

if request.method == 'GET':

return render\_template('member/reg.html‘)

req = request.values

login\_name = req['login\_name'] if 'login\_name‘ in req else ''

login\_pwd = req['login\_pwd‘] if 'login\_pwd‘ in req else ''

# 3. 后端做校验 登陆时同理

if login\_name is None or len( login\_name ) < 1:

return jsonify({ “msg“: “用户名或密码错误“, “status“: 0 })

if login\_pwd is None or len( login\_pwd ) < 6:

return jsonify({ "msg": "用户名或密码错误", "status": 0 })

# 4. 查询数据库中是否存在该用户

user\_info = User.query.filter\_by( login\_name=login\_name ).first()

if user\_info:

return jsonify({ "msg": "用户名已存在", "status": 0 })

# 5. 将用户信息存到数据库中

model\_user = User()

model\_user.login\_name = login\_name

model\_user.login\_pwd = login\_pwd

...

db.session.add( model\_user )

db.commit()

return jsonify({ "msg": "注册成功", "status": 1 })

# 登陆操作

@member\_page.route('/login‘, methods=['GET', 'POST‘])

def login():

if request.methods = 'GET‘:

return render\_template('member/login.html‘)

req = request.values

login\_name = req['login\_name'] if 'login\_name' in req else ''

login\_pwd = req['login\_pwd'] if 'login\_pwd' in req else ''

# 后端做校验

if login\_name is None or len(login\_name) < 1:

return jsonify({"msg": "用户名或密码错误", "status": 0})

if login\_pwd is None or len(login\_pwd) < 6:

return jsonify({"msg": "用户名或密码错误", "status": 0})

# 上面步骤同上

# 查询数据中是否存在这个用户

user\_info = User.query.filter\_by(login\_name=login\_name).first()

if not user\_info:

return jsonify({ "msg": "登陆用户名或密码错误", "status": 0 })

return jsonify({ "msg": "登陆成功", "status": 1 })

目前将最基础的登陆和操作功能完成，但是存在一个问题，就是如何存储用户登陆信息。

关于登陆和注册的处理用户登陆状态的问题:

方案一: 利用flask中的session来记录

# 在member.py中的返回之前

from flask import session

# 直接存 使用session模式 但是不推荐

session['uid'] = user\_info.id

# 需要在application.py中添加secret\_key

app.config['SECRET\_KEY'] = '12321312321312‘

# 这样用户在请求登陆之后 就会在cookie中添加一个session字段

方法二：使用cookie 来实现

步骤一：

# 设置cookie

from flask import make\_response, jsonify

response = make\_response(jsonify({ “msg": “登陆成功", “status": 0 }))

# 设置头

response.set\_cookie(app.config['AUTH\_COOKIE\_NAME'], “%s" ('this is cookie‘), 60 \* 60 \* 24)

return response

步骤二：

然后在每次请求之前，去判断是否存在该用户对应的cookie，如果对应上了，则将用户信息 设置为全局变量，如果没对应，则返回登陆页面。

# 新建一个interceptors/Auth.py

# 在www.py中引入拦截器 from imterceptors.Auth import \*

from application import app

from flask import request, g

from common.modelss.user import User

@app.before\_request

def before\_request():

user\_info = check\_login()

g.current\_user = None

if user\_info:

g.current\_user = user\_info

return

@def check\_login():

cookies = request.cookies

cookie\_name = app.config['AUTH\_COOKIE\_NAME']

auth\_cookie = cookies[ cookie\_name ] if cookie\_name in cookies else None

if auth\_cookie is None:

return False

if auth\_cookie == 'this is cookie‘:

user\_info = User.query.filter\_by( login\_name='aaa‘ ).first()

return user\_info

步骤三：

# 在controller/index.py中 将current\_user 引入

@index\_page.route('/')

def index():

context = {}

if 'current\_user‘ in g:

context['current\_user'] = g.current\_user

return render\_template('index.html‘, \*\*context)

# 大功告成 最后在需要用到用户信息的时候 就能取到用户信息