Flask框架

静态网页与动态网页

- 静态网页：无法与服务器进行交互的网页
- 动态网页：能够与服务器进行交互的网页

web与服务器

- web：网页（html、css、js）
- 服务器：能够给用户提供服务的机器
    1. 软件和硬件：
        - 硬件范畴：一台主机
        - 软件范畴：一个能够接受用户请求并给出响应的程序
            1. apache
            2. tomcat
            3. iis
            4. nginx
    2. 作用
        - 存储web所需要的信息
        - 能够处理用户的请求（request）并给出响应（response）
        - 能够执行服务器端的一些程序
        - 具备一定的安全性功能

框架

- 什么是框架

  框架就是一个为了解决开放性问题而存在的一种结构。框架本身具备一些基本的功能。我们只需要在基础功能上搭建属于自己的操作即可

- Python web框架

  flask django tornado webpy ...

  请求 响应 数据

flask

- flask是一个基于Python并且依赖于Jinja2模板引擎和Werkzeug WSGI服务的一个微型框架
- WSGI：web server gateway interface --> web服务网关接口
- flask框架模式------MTV
    - M：models 模型层 负责数据库建模
    - T：Templates 模板层 用于处理用户显示的内容显示的内容
    - V：views 视图层 处理与用户交互的部分内容 处理用户的请求并给出响应
- 经典的三层结构：MVC

    - M：models 模型层 负责数据库建模
    - V：views 视图层 用于处理用户显示的部分内容
    - C：controller 控制器 处理与用户交互的部分内容 处理用户的请求与响应
- flask安装 ---> pip install flask==1.0.2
- 初始化

```
# run.py
from flask import Flask

app = Flask(__name__)
```

@app.route() flask中的路由定义 定义用户的访问路径

def index() 表示匹配上@app,route()路径后的处理程序---视图函数

@app.route('/') def index(): return "this is my first flask app"

if **name** == "**main**": # 运行flask应用（启动Flask的服务） # debug是将当前的启动模式改为调试模式（开发环境推荐使用，生产环境不可使用） app.run(debug=True)

访问路径：http://127.0.0.1:5000/

flask - 路由

1. 什么是路由

   客户端将请求发送给web服务器，web服务器再将请求发送给flask程序实例

   程序实例需要知道每个URL请求要运行那些代码，所以需要建立一个URL到Python函数的映射，处理URL和函数之间的关系的程序就是路由

   在flask中，路由是通过@app.route装饰器来表示的

2. 路由的体现
    - 路由的基本表示

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

- 带参数的路由

  http://localhost:5000/show/sf.zh

    1. 基本带参路由

```
  @app.route('/show/<name>')
```

    2. 带多个参数的路由

1.
```
@app.route('/show1/<name>/<age>')
def show2(name, age):
    return '<h3>姓名：%s 年龄：%d</h3>' % (name, age)
```

3.

3. 指定参数类型的路由

```
@app.route('/show3/<str name>/<int:age>') # age参数是一个整型的数值 而并非一个字符串
```

flask中支持的类型转换器（不能存在 / ）

- 缺省参数　字符串型
- float　　浮点型
- int　　整形
- path　　字符串 （允许有 /）

- 多URL的路由匹配

  允许在一个视图处理函数中设置多个url路由规则

```
@app.route('/')
@app,route('/index')
def index():
    return "xxxx"
```

- 路由中设置HTTP请求方法

  flask路由规则也允许设置对用的请求方法只有将匹配上请求方法的路径交给视图处理函数去执行

```
# 全写 全不写 表示均可访问
# 指定具体方式 只有该方式可以访问
@app.route('/post', methods=['GET','POST'])
```

- url的反向解析

  正向解析：程序自动解析，根据@app.route()中的访问路径匹配处理函数

  反向解析：通过视图处理函数的名称自动生成视图处理函数的访问路径

  flask中提供的函数为url_for，用于反向解析url

  - 第一个参数：指向函数名称（通过@app.route()修饰的函数）
  - 后续的参数：对应要构建的url上的变量

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

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

    return "name: %s" % name
```

```
    url_for('index') # 解析结果 /index
    url_for('show', name='', age=) # 解析结果 /show/<name>

@app.route('/url')
def url_views():
    # 将login()反向解析访问地址
    url_for('index')
```

- 静态文件解析方式

  url_for('static', filename='xxx.html')

模板 - Templates

1. 什么是模板

   模板是一个包含响应文本的文件 (通常是html文件)，该文件中允许包含'占位变量'来表示动态的内容，其具体值在请求中才能知道。“占位变量”最终会被真实的值所替换。

   模板最终也会被解析成响应的字符串，这一过程称为“渲染”

   flask实际上使用的死jinja2强大的模板引擎

2. 模板的设置

   默认情况下，flask会在程序文件夹中的templates文件夹中寻找模板

   - 程序文件夹下创建templates文件夹

3. 渲染模板

   在视图函数中，通过

   return render_template() 将模板渲染成字符串再返回给客户端

   render_template() 语法：

   render_template('xxx.html', arg1=value1, arg2=value2)

   参数1：xxx.html 要渲染给客户端的html模板文件

   参数2~n：要传递给模板动态显示的变量占位符，如果没有动态的变量占位符，则可以省略

   返回值：字符串

4. 模板中的语法规范

   - 变量------变量是一种特殊的占位符，告诉模板引擎该位置的值是从渲染模板时的数据中来获取的
   - 在视图中：

     ```
     @app.route('/')

     def index():

         return render_template('xxx.html',name='',age='')
     ```

     name和age就是要传递到xxx.html中的变量

     在模板中：

     {{变量名}}

```python
@app.route('/temp')
def temp():
    # 第一种方式
    # dic = {
    #         'title': '书本详情',
    #         'bookName': '西游记',
    #         'author': '吴承恩',
    #         'price': 60,
    #         'publish': '北京大学出版社',
    # }
    # return render_template('01_temp.html', params=dic)
    title = '书本详情'
    bookName = '西游记'
    author = '吴承恩'
    price = 60
    publish = '北京大学出版社'
    return render_template('01_temp.html', params=locals())
```

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{params.title}}</title>
</head>
<body>
<h2> 书名: {{ params.bookName }}</h2>
<h2> 作者: {{ params.author }}</h2>
<h2> 价格: {{ params.price }}</h2>
<h2> 出版社: {{ params.publish }}</h2>
<h2>list[1]: {{params.list[1]}}</h2>
<h2>list[1]: {{params.list.1}}</h2>
<h2>tup[2]: {{params.tup[2]}}</h2>
<h2>dic['yaoguai5']: {{params.dic['yaoguai5']}}</h2>
<h2>per.name:{{params.per.name}}</h2>
<h2>per.say():{{params.per.say()}}</h2>
</body>
</html>
```

- 过滤器

  1. 什么是过滤器

     过滤器是允许在变量输出现实之前改变变量的值

  2. 过滤器的语法

     {{变量|过滤器}}

     capitalize------首字符变大写，其他字符变小写

     lower------把值转换为小写

     upper------把值转换为大写

     title------把值中的每个单词的首字符变大写

trim------把值俩端的空格去掉

- 控制结构
  1. if结构

     {% if 条件 %}

     {% endif %}

     {% if 条件 %}

     满足条件时执行的代码

     {% else%}

     不满足条件时执行的代码

     {% endif %}
  2. for结构

     {% for 变量 in 元组|列表|字典 %}

     {{ endfor}}
  3. 宏

     {{% macro %}} 声明宏 那个地方调用 那个地方就会出现宏中的代码

     {% macro show(str) %}

     # {{str}}

     {% endmacro%}

     {{show(uname)}}

     为了方便一般会将宏放在一个单独的模板文件中声明定义

     创建一个macro.html

     ```
     {% macro show(str) %}

     <h1>{{str}}</h1>

     {% endmacro%}

     {% macro show_li(str) %}

     <h1>{{str}}</h1>

     {% endmacro%}
     ```

     在使用宏的网页中导入macro.html

     {% import macro.html as macros%}

4. 模板的包含

   在多处重复使用的代码可以放在单独的文件当中，可以被其他的模板所包含（引用）

   {% include 'xxx.html' %}

5. 静态文件

   在flask中不能与服务器动态交互的文件都是静态文件（css js image video media）

   - 静态文件处理

     所有静态文件都保存在项目文件夹中的static文件夹中

     在访问静态文件时，需要通过/static/资源路径进行访问

     ```
     <img src="/static/images/01.jpg">
     ```

     反向解析静态路径

     ```
     url_for('static', filename='<file_path>')
     <img src="{{url_for('static', filename='images/01.jpg')}}">
     ```

6. 模板的继承

   模板的继承类似于类的继承，如果在一个模板中出现的大量内容是另外一个模板的话，那么就可以使用继承的方式来简化开发

   语法：

   - 父模板中

     需要定义出 在子模板中可以被重写的内容

     {% block 块名 %}

     ……

     {% endblock %}

     block：定义允许在子模板中被修改的内容

     1. 在父模板中正常显示
     2. 在子模板中可以被重写

   - 子模版中

     1. 使用{% extends '父模板名称' %} 来完成继承
     2. 使用 {% block 块名 %} 来重写父模板中的同名内容

        {% block 块名 %}

        覆盖父模板中的内容

        {% endblock %}

        允许通过{{ super() }} 来调用父模板中的内容

   - 自定义错误页面

     以404为例

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html')
```

修改配置

```
app = Flask(__name__,
            template_folder='templates',  # 更改templates 即更改模板文件夹名称
            static_url_path='/static',
           static_floder='static',  # 更改templates 即更改模板文件夹名称
           debug=True)
```

请求和响应 request response

- http 协议

  请求对象------request 封装了所有与请求相关的信息

  在flask中，请求信息被封装到 request对象中

  ```
  from flask import request

  scheme = request.scheme
  method = request.method
  args = request.args
  form = request.form
  values = request.values
  cookies = request.cookies
  ```

  1. request的常用方法

     scheme：获取请求的方案（协议）

     method：获取请求的方式

     args：获取使用GET方式提交的数据

     form：获取使用POST方式提交的数据

     values：获取GET或POST请求方式提交的数据

     cookies：获取cookies中的信息

     headers：获取请求消息头的信息

     path：获取访问的url地址

     files：获取用户上传的文件

     referer：网页的源路径

     full_path：获取请求的完整路径

     url：获取访问地址

     ……

     2.获取请求提交的数据

- get请求提交的数据是存放在QueryString中的

  request.args 封装的就是get请求的数据，类型为字典

```
request.args['name']  # 获取name对应的值
request.args.get('name') # 获取name对应的值
request.args.getlist('name_list') # 获取列表数据

@app.route('/form')
def form_view():
    return render_template('03-form.html')
```

```
@app.route('/form_do')
def form_do():
    if request.method == 'GET':
        uname = request.args.get('uname')
        upwd = request.args.get('upwd')
        print(uname, upwd)
    return '获取表单数据成功'
```

```
```

•

* post请求提交的数据是存放在form中的

  request.form封装的就是post请求的数据，类型为字典

  ```Python
  request.form['name']  # 获取name对应的值
  request.form.get('name')  # 获取name对应的值
  request.form.getlist('name_list')  #获取列表数据

  # @app.route('/post')
  # def post_view():
  #     return render_template('04-form.html')
  ```

```
@app.route('/post', methods=['GET', 'POST'])
def post_do():
    if request.method == 'GET':
        return render_template('04-form.html')
    elif request.method == 'POST':
        uname = request.form.get('uname')
        upwd = request.form.get('upwd')
        uemail = request.form.get('uemail')
        trueName = request.form.get('trueName')
        print(uname, upwd, uemail, trueName)
        return 'post表单数据提交成功'
```

```
Method Not Allowed  通过在路由添加 method=['POST'] 解决
```

针对get请求，可以通过以下方式做提交

http://主机:端口/请求路径?参数1=值1&参数2=值2

* 响应 response

响应对象其实就是要响应给客户端的内容，可以是普通字符串，可以是模板 或者是重定向

```python
@app.route('/')
def index():
    # return 'xxxxxxx'
    return render_template('xxx.html')
'''以上俩种行为，本质上返回的都是字符串'''
```

1. 构建响应对象，再响应给客户端------不是直接响应字符串，而是响应对象

   响应对象，可以包含响应字符串，同时也可以实现其他的响应操作

   在flask中，使用make_response()构建响应对象

   ```Python
   from flask import make_response

   resp = make_response('响应内容')
   # 此处可以实现其他的响应操作 比如：添加cookies
   return

   @app.route('/response')
   def response_views():
       # resp = make_response('使用响应对象，响应给浏览器的内容')
       resp = make_response(render_template('04-form.html'))
       return resp
   ```

- 重定向

  由服务器端，通知客户端，重新向新的地址发送请求

  ```
  from flask import redirect

  resp = redirect('重定向地址')
  return resp
  # return redirect('/')
  ```

  发送俩次请求，第一次向请求地址发请求，第二次向跳转地址发请求

- flask文件上传
  - 表单中 提交方式 必须为 post
  - enctype 必须为 multipart/form-data

在服务器端

- request.files 获取上传的文件

  f = request.files['文件框name属性值']

- 通过 f.save(保存路径) 将文件保存到指定目录
- 通过 f.filename获取文件名称

  filename = f.filename

  f.save('static/' + filename)

```
@app.route('/file', methods=['GET', 'POST'])
def file_views():
    if request.method == 'GET':
        return render_template('05-file.html')
    elif request.method == 'POST':
        f = request.files['uimg']
        ftime = datetime.datetime.now().strftime('%Y%m%d%H%M%S%f') # 取当前时间并格式化为字符
串

        ext = f.filename.split('.')[1]
        filename = ftime + '.' + ext
        basedir = os.path.dirname(__file__)  # 查询文件当前路径
        upload_path = os.path.join(basedir, 'static/upload', filename)
        f.save(upload_path)
        # f.save('static/upload' + filename)
        return '文件保存成功'
```

大量数据上传时，不能使用网页上传（http协议不支持）需要使用单独的上传工具（C/S）

- 路径设置

```
import os
basedir = os.path.dirname(__file__)  # 查询文件当前路径
os.path.join(basedir, 'static/upload', filename)
```

模型 models

1. 模型是根据数据库中表的结构创建出来的class，每一张表到编程语言中就是一个class 表中每一个列，到编程语言中就是class的一个属性
2. 创建和使用模型 - ORM框架 （一类框架的总称）

   ORM --- Object Relational Mapping---对象关系映射

   - 数据表（table）到编程类（class）的映射

     数据库中的每一张 表 对应到编程语言中都有一个 类

     ORM中允许将数据表 自动生成一个 类

     也允许将类 自动生成一张 表

   - 数据类型的映射

     将数据库表中的字段以及数据类型 对应到 编程语言中 类 的属性

在ORM中允许将表中的字段和数据类型自动映射到编程语言中也允许将类中的属性和类型映射到数据库表中

- 关系映射

  允许将数据库表与表之间的关系对应到编程语言类之间的关系（一对一 一对多 多对多）

  一对一的实现：主外键关联，外键需要加唯一约束

  一对多的实现：主外键关联

  多对多的实现：通过第三张表建立联系

pip install sqlalchemy

pip install flask-sqlchemy

```
# 配置数据库
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
# app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://username:pwd@host:port/dbname'
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql://root:root@localhost:3306/flaskdemo'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)  # 创建一个SQLAlchemy的实例 表示程序正在使用的数据库同时获得了SQLAlchemy的所有功能
```

定义模型

- 模型：数据库中的表在编程语言中的体现 本质就是一个python类 可称为 模型类 或 实体类 类中的属性要和数据库中的列相对应

  ```
  class MODELNAME(db.Model):
      __tablename__ = "TABLENAME"
      COLUMN_NAME = db.Column(db.TYPE, OPTIONS)
  ```

  上段代码中，凡大写内容都可根据实际情况替换

  1. MODELNAME：定义模型类名称，根据表名设定
  2. TABLENAME：映射到数据库中表的名字
  3. COLUMN_NAME：属性名，映射到表中列的名字
  4. db.TYPE：映射到列的数据类型

     - 类型名 python类型 说明
     - Integer int 32位，普通整数
     - SmallInteger int 16位，小范围整数，通常16位
     - BigInteger int / long 不限精度的整数
     - Float float 浮点数
     - Numeric decimal.Decimal 定点数
     - String str 变长字符串
     - Text str 变长字符串，优化
     - Unicode unicode 变长Unicode字符串
     - UnicodeText unicode 变长Unicode字符串,优化
     - Boolean bool 布尔值

- Date　　datetime.date　　日期
- Time　　datetime.time　　时间
- DateTime　　datetime.datetime　日期和时间

5. OPTIONS：列选项

- 选项名　　说明
- primary_key　　设置为True 表示主键 默认自增（整数）
- unique　　设置为True表示唯一 允许为空 但只能一个为空
- index　　设置为True表示该列要创建索引
- nullable　　设置为True表示允许为空
- default　　设置默认值

6. 数据库的操作

- 插入

```
db.session.add(Models)
db.session.commit()
```

```
from flask import Flask, request, render_template
from flask_sqlalchemy import SQLAlchemy
import pymysql

app = Flask(__name__)
pymysql.install_as_MySQLdb()

app.config['SQLALCHEMY_DATABASE_URI'] =
'mysql://root:root@localhost:3306/flaskdemo'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SQLALCHEMY_COMMIT_ON_TEARDOWN'] # 执行完操作自动提交
db = SQLAlchemy(app)
```

```
class Users(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), nullable=False)
    age = db.Column(db.Integer)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, age, email):
        self.username = username
        self.age = age
        self.email = email

    def __repr__(self):
        return '<Users:%r>' % self.username

class Student(db.Model):
    __tablename__ = 'student'
    id = db.Column(db.Integer, primary_key=True)
    sname = db.Column(db.String(30), nullable=False)
```

```
    sage = db.Column(db.Integer)

    def __init__(self, sname, sage):
        self.sname = sname
        self.sage = sage

    def __repr__(self):
        return '<Student:%r>' % self.sname

class Teacher(db.Model):
    __tablename__ = 'teacher'
    id = db.Column(db.Integer, primary_key=True)
    tname = db.Column(db.String(30), nullable=False)
    tage = db.Column(db.Integer)

    def __init__(self, tname, tage):
        self.tname = tname
        self.tage = tage

    def __repr__(self):
        return '<Student:%r>' % self.tname

class Course(db.Model):
    __tablename__ = 'course'
    id = db.Column(db.Integer, primary_key=True)
    cname = db.Column(db.String(30))

    def __init__(self, cname):
        self.cname = cname

    def __repr__(self):
        return '<Student:%r>' % self.cname

db.create_all()

@ app.route('/register', methods=['GET', 'POST'])
def insert_views():
    if request.method == 'GET':
        return render_template('insert.html')
    elif request. method == 'POST':
        name = request.form.get('username')
        age = request.form.get('age')
        email = request.form.get('email')
        users = Users(name, age, email)
        db.session.add(users)
        # db.session.commit() 加了自动提交设置 可以省略
    return 'register success'

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

- flask-sqlalchemy 查询

1. 基于db.session进行查询

   db.session.query() 该函数返回一个query对象，类型为BaseQuery,包含了指定实体类对应的表中的所有数据 该函数可以接收多个参数，参数表示的是查询那个实体类

   查询执行函数：在查询的基础上 得到最终想要的结果

   - all()：以列表的方式返回所有的查询结果
   - first()：返回查询中的第一个结果 没有结果则返回none
   - first_or_404()： 返回查询中的第一个结果 没有结果则返回404
   - count()：返回查询结果的数量

2. 查询过滤器函数

   在查询的基础上筛选部分列出来

   db.session.query().过滤器函数.查询执行函数

   filter()：按指定条件进行过滤 （单表 多表 定值 不定值...）

   filter_by()：按照等值条件进行过滤

   limit()：按限制行数获取

   order_by()：根据指定条件进行排序

   group_by()：根据条件进行分组

```
# filter()
db.session.query(Users).filter(Users.age>30).all()    # , 隔开多个条件 表示 且
查询 或 引导多个条件需要借助 or_()函数（需要导入）
db.session.query(Users).filter(or_(条件1, 条件2)).all()
# 等值查询必须用 ==
# 模糊查询 like
db.session.query(Users).filter(Users.email.like('%W%')).all()
# 查询id 在[1,2,3]之间的信息
db.session.query(Users).filter(Users.id.in_([1,3])).all()
# 在整个查询过程中获取前五条数据
db.session.query(Users).limit(5).all()
db.session.query(Users).order_by('id desc', 'age asc').all()
db.session.query(Users).group_by('age').all()
```

   3. 通过model.query()查询

```
Users.query.filter(Users.id > 3).all()
```

   4. flask-sqlalchemy 删除和修改

      1. 先查询出要删除的实体

```
user = db.session.query(Users).filter_by(id=5).first()
```

      2. 根据所提供的删除方法将信息删除

```
db.session.delete(user)
```

3. 查

```
user = Users.query.filter_by(id=1).first()
```

4. 改

```
user.username = ""
user.age =
```

5. 保存

```
db.session.add(user)
```

3. flask-sqlalchemy 关系映射

1. 一对多

在“多”的实体中增加

外键列名 = db.Column(db.Integer, db.ForeignKey('主表.主键'))

在“一”的实体中增加反向引用关系

属性名 = db.relationship('多的实体类', backref='属性名', lazy='dynamic')

- 常用的关系选项

选项名 说明

backref 在关系的另一个模型中添加反向引用

lazy 指定如何加载相关记录

select 首次访问时加载

immediate 原对象加载后 立马加载关联数据

subqurey 立即加载 使用子查询

noload 不加载

dynamic 不加载记录 但是提供加载记录的查询（动态加载）

uselist 如果设置为False 则不使用列表 使用标量 （一对一时使用）

secondary 指定多对多关系中关联表的名字

```
class Teacher(db.Model):
    __tablename__ = 'teacher'
    id = db.Column(db.Integer, primary_key=True)
    tname = db.Column(db.String(30), nullable=False)
    tage = db.Column(db.Integer)
    # 增加一列 : course_id 外键列 要引用自主键表course的主键列 id
    course_id = db.Column(db.Integer, db.ForeignKey('course.id'))
```

```
    def __init__(self, tname, tage):
        self.tname = tname
        self.tage = tage

    def __repr__(self):
        return '<Student:%r>' % self.tname
```

```
class Course(db.Model):
    __tablename__ = 'course'
    id = db.Column(db.Integer, primary_key=True)
    cname = db.Column(db.String(30))
    # 反向引用
    # backref: 定义反向关系 本质上会在Teacher实体中增加一个course属性 该属性可替代course_id
    # 来访问course模型此时得到的是模型对象 而不是外键值
    teachers = db.relationship('Teacher', backref='course', lazy='dynamic')

    def __init__(self, cname):
        self.cname = cname

    def __repr__(self):
        return '<Student:%r>' % self.cname
```

2. 一对一 关系映射

   * A表中的一条记录只能与B表中的一条记录关联
   * B表中的一条记录只能与A表中的一条记录关联

   在数据库中的体现

```python
class Users(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), nullable=False)
    age = db.Column(db.Integer)
    email = db.Column(db.String(120), unique=True)
    position = db.relationship('Positions', backref='user', lazy='dynamic', uselist=False)

    def __init__(self, username, age, email):
        self.username = username
        self.age = age
        self.email = email

    def __repr__(self):
        return '<Users:%r>' % self.username
```

```
class Positions(db.Model):
    __tablename__ = 'positions'
    id = db.Column(db.Integer, primary_key=True)
    pname = db.Column(db.String(30))
    user_id = db.Column(db.Integer, db.ForeignKey('users.id'))
```

```
        # 增加反向应用

        def __init__(self, pname):
            self.pname = pname

        def __repr__(self):
            return '<Position %r>' % self.pname
    ```
```
```