

Flask

Flask是一个用于web开发的微型框架，它具有一个包含基本服务的核心，其他功能可以通过扩展实现。

Flask有三个依赖：路由，调试，和web服务器网关接口。

虚拟环境

虚拟环境是python解释器的一个私有副本，可以避免安装的Python版本与系统预装的发生冲突，为每个项目单独创建虚拟环境，可以保证应用只能访问虚拟环境中的包，从而保持全局解释器的干净整洁，并且不需要管理员权限。

vscode中需要安装第三方库帮助创建虚拟环境

```
1 pip3 install virtualenv
```

创建虚拟环境

```
1 virtualenv venvname
```

选择解释器

```
1 virtualenv -p D:\python311\python.exe venvname
```

启动

```
1  venv\Scripts\activate
```

退出

```
1  deactivate
```

基本概念

域名

域名是网站的名字或地址，用于代替难记的IP地址

端口号

每台服务器可能同时多个服务(网页，邮件，数据库)，端口号用于指定具体的服务

Session和cookie

HTTP请求是无状态协议，不保存状态

`session` 允许你在**不同的请求之间存储和共享数据**。它实际上存储在服务器端，但会通过一个 `cookie` 在客户端传递给浏览器。

通过 `session` ，你可以在多个请求之间**持久化用户的状态和信息**，比如：用户名、登录状态、购物车数据等。

当客户发起第一次请求时，服务器会建立与客户端的会话并将会话的唯一标识符Session ID传送给客户，服务器会将一些信息保存在会话中，当客户第二次发起请求时会携带Session ID到服务器中，服务器通过Session ID获取会话中的信息

跨域限制

跨域限制(同源策略)防止一个网站上的脚本与另一个不同来源上的资源进行不受限制的交互

不同的来源指的是协议（例如 http 或 https）、域名（例如 example.com）和端口号相同

- 同源：

- <https://example.com/index.html> 与 <https://example.com/about.html>

- 不同源：

- <https://example.com> 与 <http://example.com>（协议不同）
- <https://example.com> 与 <https://api.example.com>（子域名不同）
- <https://example.com> 与 <https://example.com:8080>（端口不同）

CSRF

CSRF——跨站请求伪造，恶意网站通过已认证的用户浏览器在受信任站点上执行非正常操作，一般通过外部链接实现

CSRF防护机制会基于secret_key生成token，服务器再生成页面时会给用户一个token，在提交时用户需要携带token一起，然后服务器会检查token是否正确，外部网站是无法访问网站页面的token。

CRUD

CRUD (Create/Read/Update/Delete)增删改查操作

WSGI

全称Python web server Gateway interface，指定了web服务器和python web应用或web框架之间的标准接口，以提高web应用在一系列web服务器间的移植性

应用基本结构

初始化

应用实例是整个网站的心脏，所有的请求都会有Flask实例处理

```
1 from flask import Flask
2 app=Flask(__name__)
```

路由和视图函数

客户端把请求发送给web服务器，Web服务器再把请求发送给Flask应用实例，应用实例需要知道对每个URL请求去要运行那些代码，所以保存了一个URL到Python函数的映射关系，处理URL和函数之间关系的程序称为路由。

将index函数(视图函数)注册为根地址的处理程序，函数的放回置称为响应

```
1 @app.route('/')
2 def index():
3     return '<h1>Hello World!</h1>'
```

匹配URL中的参数部分，使用尖括号围起

```
1 @app.route('/user/<name>')
2 def user(name):
3     return f'<h1>Hello {name}!</h1>'
```

启动方式

终端输入

```
1 set FLASK_APP=app.py # 入口
2 PS D:\vscodeProject\flask> flask run
```

其他参数有

```
1  --host=0.0.0.0  允许外部访问
2  --port=8080  指定端口（默认 5000）
3  --debug  开启调试模式（等价于设置 FLASK_ENV=development）
4  --reload  源码变更自动重启服务器
```

```
1  flask run --host=0.0.0.0 --port=8080 --debug
```

这些参数可以在根目录下创建.flaskenv存在，在启动程序时自动读取

```
1  FLASK_APP=app.py          # 指定Flask入口文件
2  FLASK_ENV=development     # 设置运行环境：开发（development）或生产（production）
3  FLASK_DEBUG=1            # 是否开启调试模式（1 开启，0 关闭）
4  SECRET_KEY=' '          # 应用的主密钥，用于加密session，CSRF，flash
```

需安装插件

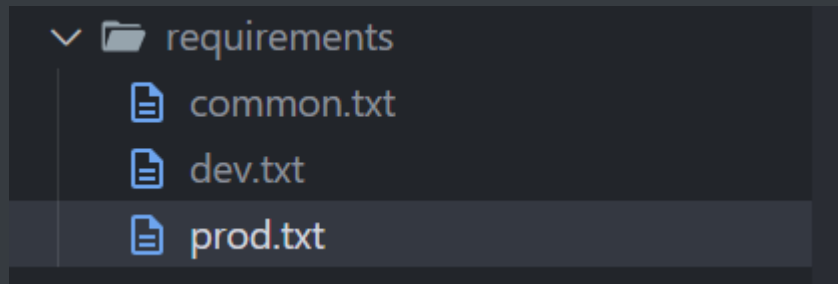
```
1  pip install python-dotenv
```

依赖

项目中需要生成依赖文件

```
1  pip freeze > requirements.txt
```

在开发模式下有些依赖是用于测试的不应加入到生产模式下，因此需要区分依赖文件



共有的依赖存放在common中，在其他文件中导入

```
1 -r common.txt
```

```
1 # 生产环境
2 pip install -r requirements/base.txt
3
4 # 开发环境
5 pip install -r requirements/dev.txt
```

上下文

Flask中的上下文能在处理请求时方便地访问应用相关的信息，而不需要单独传递到函数中。

上下文分为请求上下文和应用上下文

应用上下文绑定了Flask应用实例，能够在全局范围内访问应用相关的对象

- `current_app` ——当前正在处理请求的应用实例
- `g` ——在请求期间存储临时数据

请求上下文绑定了每一个HTTP请求，确保每个请求可以访问当前请求的相关内容，比如请求的参数，头信息，cookies

- `request` —— 包含当前请求的所有信息，如请求方法（GET、POST）、请求数据、表单数据、URL 参数等
- `session`：用于在多个请求之间存储和读取用户的会话数据

这些扩展将自己与Flask app对象绑定，其中**Bootstrap** 和 **Moment** 提供了Jinja2模板中可以直接使用的函数

```
1 moment = Moment(app)
2 bootstrap.init_app(app)
3 mail.init_app(app)
4 moment.init_app(app)
5 db.init_app(app) # 将数据可模型加载到上下文中
```

因此可以直接在模板中使用

```
1 {{ bootstrap.load_css() }}
2 {{ bootstrap.load_js() }}
```

视图函数中会自动推入上下文，需要访问实例属性时调用`current_app`

```
1 from flask import current_app
2 @app.route('/send')
3 def send_mail():
4     # 创建邮件对象，第一个参数hi是标题，第二个参数是收件人列表
5     msg = Message('Hello from Flask',
6                   recipients=['2228632512@qq.com'])
7     # 设置邮件内容
8     msg.body = 'This is a test message sent from Flask-Mail.'
```



```
9     msg.html = '<b>This is a test message sent from Flask-Mail.</b>'
10
11     current_app.config['name']....
12
13     # 发送邮件，视图函数中自动引入上下文获取mail对象
14     mail.send(msg)
15     return '邮件发送成功！'
```

在测试脚本中请求之外需要手动创建应用上下文

```
1 with app.app_context():
2     secret = current_app.config['SECRET_KEY']
```

请求钩子

请求钩子是在请求生命周期中的某些阶段执行的函数

- `before_request` 在每次请求之前运行。
- `before_first_request` 尽在处理第一个请求函数之前执行
- `after_request` 请求成功执行之后执行
- `teardown_request` 不管请求是否成功运行都执行

响应

视图函数默认返回状态码200，也可以手动设置

```

1 @app.route('/not_found')
2 def not_found():
3     return 'Page not found', 404 # 返回 404 状态码

```

还可以返回第三个参数——由HTTP响应首部组成的字典

如果不想返回元组则可以使用Response对象

```

1 from flask import make_response
2 @app.route('/')
3 def index():
4     response = make_response('<h1>This document carries a cookie!</h1>')
5     response.set_cookie('answer', '42')
6     return response

```

属性或方法	说 明
status_code	HTTP 数字状态码
headers	一个类似字典的对象，包含随响应发送的所有首部
set_cookie()	为响应添加一个 cookie
delete_cookie()	删除一个 cookie
content_length	响应主体的长度
content_type	响应主体的媒体类型
set_data()	使用字符串或字节值设定响应
get_data()	获取响应主体

响应的一种特殊类型是重定向

```

1 from flask import redirect
2 @app.route('/')
3 def index():
4     return redirect('http://www.example.com')

```

模板

模板中可以访问到render_template传入的变量，上下文中注入的变量，以及自行注入的变量

对于模板中经常需要用到的变量，可以自行注入。flask每次渲染模板时，都会自动调用所有注册的context_processor 函数，将它们返回的字典合并进模板上下文。

```
1  # 蓝图中注册，只能在该蓝图渲染的模板中使用
2  @main.app_context_processor
3  def inject_permissions():
4      return dict(Permission=Permission)
5
6  # 全局注册
7  @app.context_processor
8  def inject_permissions():
9      return dict(Permission=Permission)
```

Jinja2

Jinja2用于将python中的数据和逻辑渲染到HTML(txt也可以)模板中

变量输出

```
1  <h1>Welcome, {{ user.name }}!</h1>
```

条件语句

```
1 {% if user.is_admin %}  
2     <h1>Admin Dashboard</h1>  
3 {% else %}  
4     <h1>User Dashboard</h1>  
5 {% endif %}
```

循环

```
1 <ul>  
2 {% for item in items %}  
3     <li>{{ item }}</li>  
4 {% endfor %}  
5 </ul>
```

过滤器

```
1 <p>{{ name|capitalize }}</p>  
2 <p>{{ description|length }}</p>
```

`capitalize` 会将字符串的首字母大写，`length` 会返回字符串的长度。

模板继承

在base.html中放置block，block的名称自定义

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>{% block title %}My Web Application{% endblock %}</title>
</head>
<body>
  <header>
    <h1>Welcome to My Web Application</h1>
    <nav>
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/about">About</a></li>
        <li><a href="/contact">Contact</a></li>
      </ul>
    </nav>
  </header>

  <div class="content">
    {% block content %}
    <!-- 子模板中会替换这个部分 -->
    {% endblock %}
  </div>

```

在index.html中继承base.html，覆写block

html

复制

编辑

```

{% extends "base.html" %}

{% block title %}Home Page{% endblock %}

{% block content %}
  <h2>Welcome to the Home Page!</h2>
  <p>This is the content of the home page.</p>
{% endblock %}

```

最后通过render_template渲染同时传入参数，render_template默认从templates文件夹下寻找Html，子模版同时继承父模板引入的CSS，JS文件

```
1 return render_template('index.html', name=name)
```

如果父模板的block原本就有内容并且子模版想要追加而不是覆写，就需要使用super()

base.html

```
<body>
    {% block content %}{% endblock %}

    {% block scripts %}
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.min.js">
    {% endblock %}
</body>
```

child.html

```
{% block scripts %}
{{ super() }}
<script type="text/javascript" src="{{ url_for('static', filename='my-script.js') }}">
{% endblock %}
```

对于重复使用的html片段，可以单独保存为一个文件，然后在需要用到的地方引入，可以继承调用模板的变量

```
1 {% include 'example.html' %}
```

除了include，模板复用还可以使用宏，宏可以传入参数，宏可以接收未定义的参数将其置于**kwargs，用于传入到内部的url_for

```
1 {% macro pagination_widget(pagination, endpoint) %}
2     <!-- 动态生成分页控件 HTML -->
3 {% endmacro %}
```

使用如下

```
1 {% import "_macros.html" as macros %}
2
3 {% if pagination %}
4 <div class="pagination">
5     {{ macros.pagination_widget(pagination, '.index') }}
6 </div>
7 {% endif %}
```

自定义错误页面

@app.errorhandler是用来处理特定HTTP错误的装饰器

```
1 @app.errorhandler(404)
2 def page_not_found(e):
3     return render_template('404.html'),404
4
5 @app.errorhandler(500)
6 def internal_server_error(e):
7     return render_template('500.html'),500
```

链接

`url_for()`接受视图函数的名称，返回对应的url

```
1 url_for('index') # 返回/  
2 url_for('index', _external=True) # 返回 _external生成的 URL 会包含完整的协议、域名和端口  
3 url_for('user', name='John') # 接收参数，返回/user/John  
4 url_for('user', name='john', page=2, version=1) # 不仅限于动态路由中的参数，/user/john?  
   page=2&version=1
```

另一种应用是生成静态文件的URL

```
1 {% block head %}  
2 {{super()}}  
3 <link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}"  
   type="image/x-icon">  
4 <link rel="icon" href="{{ url_for('static', filename='favicon.ico') }}" type="image/x-  
   icon">  
5 {% endblock %}
```

日期和时间

web的用户可能来自世界各地，因此服务器需要统一处理时间

安装外部库

```
1 pip install flask-moment
```


需要在模板中导入，注意scripts标签要放到最顶部

```
1 {% block scripts %}
2 {{ super() }}
3 {{ moment.include_moment() }} # Flask-Moment提供的自动插入Moment.js的<script>标签的函数
4 {{ moment.locale('zh-cn') }} # 设置中文
5 {% endblock %}
```

作为参数传入

```
1 from datetime import datetime, timezone
2 from flask_moment import Moment
3 moment = Moment(app)
4 @app.route('/')
5 def index():
6     return render_template('base.html', current_time=datetime.now(timezone.utc))
```

模板中添加

```
1 <p>The local date and time is {{ moment(current_time).format('LLL') }}.</p>
2 <p>That was {{ moment(current_time).fromNow(refresh=True) }}</p>
```

- `moment().format('YYYY-MM-DD HH:mm:ss')` 格式化日期 常见格式有 'LLL' : Apr 11, 2025 9:08 PM
 'LLLL' : Friday, April 11, 2025 9:08 PM
- `moment().fromNow` 显示相对时间 `refresh` 设置刷新
- `moment().from()` 和指定时间比较
- `moment(current_time).calendar()` 日历格式 —— Today at 9:08 PM

Web表单

处理表单所需的库

```
1 pip install flask-wtf
```

Flask-WTF 默认启用了 **CSRF防护机制**，需要给secret_key来加密，生成令牌

```
1 import os
2 app.secret_key=os.urandom(24) # 最好是一个从环境变量中读取的固定值
```

表单类

表单类继承FlaskForm基类，其中每个字段代表一个表单项

```
1 from flask_wtf import FlaskForm
2 from wtforms import StringField, SubmitField # 文本框，提交按钮
3 from wtforms.validators import DataRequired # 验证器，确保输入不为空
4
5 class NameForm(FlaskForm):
6     # validators验证函数组成的列表 第一个参数为label—表单中显示的标签文字
7     name=StringField('What is your name?', validators=[DataRequired()])
8     submit=SubmitField('Submit')
9
10 from wtforms.validators import Email
11 class InfoForm(FlaskForm):
```

```
12     username = StringField('Username', validators=[DataRequired()])
13     email = StringField('Email', validators=[DataRequired(), Email()])
14     submit = SubmitField('Submit')
```

字段类型	说 明
BooleanField	复选框，值为 True 和 False
DateField	文本字段，值为 datetime.date 格式
DateTimeField	文本字段，值为 datetime.datetime 格式
DecimalField	文本字段，值为 decimal.Decimal
FileField	文件上传字段
HiddenField	隐藏的文本字段
MultipleFileField	多文件上传字段
FieldList	一组指定类型的字段
FloatField	文本字段，值为浮点数
FormField	把一个表单作为字段嵌入另一个表单
IntegerField	文本字段，值为整数
PasswordField	密码文本字段
RadioField	一组单选按钮
SelectField	下拉列表
SelectMultipleField	下拉列表，可选择多个值
SubmitField	表单提交按钮
StringField	文本字段
TextAreaField	多行文本字段

验证函数

验证函数	说 明
DataRequired	确保转换类型后字段中有数据
Email	验证电子邮件地址
EqualTo	比较两个字段的值；常用于要求输入两次密码进行确认的情况
InputRequired	确保转换类型前字段中有数据
IPAddress	验证 IPv4 网络地址
Length	验证输入字符串的长度
MacAddress	验证 MAC 地址
NumberRange	验证输入的值在数字范围之内
Optional	允许字段中没有输入，将跳过其他验证函数
Regexp	使用正则表达式验证输入值
URL	验证 URL
UUID	验证 UUID
AnyOf	确保输入值在一组可能的值中
NoneOf	确保输入值不在一组可能的值中

下拉框，choices的元组(值，显示的文本)

```
1 class EditProfileForm(FlaskForm):
2     gender = SelectField('Gender',
3                           choices=[('M', 'Male'),
4                                     ('F', 'Female'),
5                                     ('O', 'Other')],
6                           default='M',
7                           validators=[DataRequired()])
8
```

邮件验证器需要安装扩展

```
1 pip install email-validator
```

```

1  # 验证两次密码是否一致，如果不一致
2  password=PasswordField('Password',validators=[DataRequired(),EqualTo('password2',\
3      message='Passwords must match'])])
4
5  # 邮件格式字段
6  email=StringField('Email',validators=[DataRequired(),Length(1,64),Email()])
7
8  # 用户名需满足正则表达式，Regexp(正则表达式，标志位:0不忽略大小写，不匹配多行,错误提示信息 )
9  username=StringField('Username',validators=[DataRequired(),Regexp('^[A-Za-z][A-Za-
10     z._]*$'),0,'Usernames must have only letters, numbers, \
        dots or underscores'])

```

以 `validate_字段名` 命名的函数是自定义验证方法，会用于验证字段

```

1  from wtforms import ValidationError
2  def validate_email(self, field):
3      if User.query.filter_by(email=field.data).first():
4          raise ValidationError('Email already registered.')

```

表单渲染为HTML

通过视图函数传入form参数(表单对象)

```

1  <form method="POST">
2      {{ form.hidden_tag() }}    防护机制使用
3      {{ form.name.label }} {{ form.name() }} # 文本框
4      {{ form.submit() }} # 提交按钮
5  </form>

```

或者使用Bootstrap的表单样式，传入form快速渲染

```
1 {% import "bootstrap/wtf.html" as wtf %}
2 {{ wtf.quick_form(form) }}
```

validate_on_submit会判断当前是不是POST请求，以及表单是否通过过了验证器

```
1 @app.route('/', methods=['GET', 'POST'])
2 def index():
3     name=None
4     form=NameForm()
5     if form.validate_on_submit():
6         name=form.name.data
7         form.name.data= ''
8     return
9 # 注意form传入的是实例
10 render_template('index.html', form=form, name=name, current_time=datetime.now(timezone.u
    tc))
11
```

重定向

表单提交后再次刷新浏览器会重复提交表单，浏览器会出现警告

确认重新提交表单

您所查找的网页要使用已输入的信息。返回此页可能需要重复已进行的所有操作。是否要继续操作？

继续

取消

因此对于POST操作，使用重定向作为请求的响应，但是由于HTTP协议是无状态的，即每个请求和响应都是独立的，服务器无法记住不同请求的数据，所以需要session储存数据

闪现消息

有时请求完成后需要让用户知道状态发生了变化，可以通过Flask内置的flash()实现，必须配合模板中的 `get_flashed_messages()`

第一个参数是消息，第二个参数是消息类别('success'、'error'、'warning')，默认是'message'

```
1 from flask import flash
2 flash('Registration successful!', 'success')
```

Flask会把消息临时存在session里(需设置SECRET_KEY)，然后在下一次渲染是可以通过 `get_flashed_messages()`获取

```
1 {% block content %}
2 <div class="container">
3     {%for message in get_flashed_messages()%}
4     <div class="'alert alert_warning">
5         <button type="button" class="close" data-dismiss="alert">&times;</button>
6         {{message}}
7     </div>
8     {% endfor %}
9     {% block page_content %}{% endblock %}
10 </div>
11 {% endblock %}
```

数据库

SQLAlchemy是python中最流行的ORM框架，ORM框架指对象关系映射，将数据库表结构映射为python类。

flask_sqlalchemy是一个在flask项目中使用SQLAlchemy更便捷的扩展

初始化

flask中选用SQLite数据库，**SQLite** 是一个轻量级的嵌入式的关系型数据库，它不需要独立的数据库服务器，所有的数据都保存在一个本地文件中，它是“零配置”的数据库 —— 安装 Python 后就可以直接使用。

初始化

```
1 basedir = os.path.abspath(os.path.dirname(__file__)) # 获取当前程序的目录
2 app.config['SQLALCHEMY_DATABASE_URI'] = \
3     'sqlite:/// ' + os.path.join(basedir, 'data.sqlite')# 配置数据库
4 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False # 不储存对象的修改信息
5 db = SQLAlchemy(app)# 创建数据库实例
```


定义模型

继承基类

```
1 class Role(db.Model):
2     __tablename__='roles' # 表名
3     id=db.Column(db.Integer,primary_key=True) # 属性
4     name=db.Column(db.String(64),unique=True)
5
6     # 建立一对多关系 backref指定在User表中添加role对象 lazy设置查询返回的结果还可以继续查询
7     users = db.relationship('User', backref='role', lazy='dynamic')
8
9
10    # 用于表示对象的字符串方法
11    def __repr__(self):
12        return '<Role %r>'%self.name
13
14 class User(db.Model):
15     __tablename__='User' # 表名
16     id=db.Column(db.Integer,primary_key=True) # 属性
17     username=db.Column(db.String(64),unique=True,index=True)
18
19    # 创建外键引用 roles指的是表名
20    role_id=db.Column(db.Integer,db.ForeignKey('roles.id'))
21
22    # 用于表示对象的字符串方法
23    def __repr__(self):
24        return '<Role %r>'%self.username
```

常用的列选项

选项名	说 明
primary_key	如果设为 True，列为表的主键
unique	如果设为 True，列不允许出现重复的值
index	如果设为 True，为列创建索引，提升查询效率
nullable	如果设为 True，列允许使用空值；如果设为 False，列不允许使用空值
default	为列定义默认值

数据库操作

操作可以在flask shell中进行(避免代码重复执行)，使用exit()退出shell

创建表

```
(venv) $ flask shell
>>> from hello import db
>>> db.create_all()
```

插入行

先创建

```
>>> from hello import Role, User
>>> admin_role = Role(name='Admin')
>>> mod_role = Role(name='Moderator')
>>> user_role = Role(name='User')
>>> user_john = User(username='john', role=admin_role)
>>> user_susan = User(username='susan', role=user_role)
>>> user_david = User(username='david', role=user_role)
```

再插入

```
>>> db.session.add(admin_role)
>>> db.session.add(mod_role)
>>> db.session.add(user_role)
>>> db.session.add(user_john)
>>> db.session.add(user_susan)
>>> db.session.add(user_david)
```

或者简写成：

```
>>> db.session.add_all([admin_role, mod_role, user_role,
...                      user_john, user_susan, user_david])
```

为了把对象写入数据库，我们要调用 `commit()` 方法提交会话：

```
>>> db.session.commit()
```

修改行

修改后重新提交

```
>>> admin_role.name = 'Administrator'
>>> db.session.add(admin_role)
>>> db.session.commit()
```

删除行

```
1 >>> user = User.query.filter_by(username='alice').first()
2 >>> user
3 <User 'alice'>
4
5 >>> db.session.delete(user)
6 >>> db.session.commit()
7 >>> print("已删除")
8 已删除
```

清空表

```
1  # 删除 User 表中的所有数据
2  db.session.query(User).delete()
3
4  # 删除 Role 表中的所有数据
5  db.session.query(Role).delete()
6
7  # 提交更改
8  db.session.commit()
```

查询行

query是SQLAlchemy中的基础查询方法，它返回的对象可以进一步链式调用，支持QLAlchemy的各种查询方法，如 `filter()`，`filter_by()`，`order_by()`，`limit()` 等，`filter_by`是简化的查询方法只支持等值查询，`filter`支持更多的操作(`in()` `like()` `>` `<`)

```
1  # 查询 User 表中的所有记录
2  users = db.session.query(User).all()
3
4  # 查询 User 表中符合特定条件的记录
5  users = db.session.query(User).filter(User.age > 18).all()
6
7  # 使用多个条件
8  users = db.session.query(User).filter(User.age > 18, User.username == 'john').all()
9
10 # 排序
11 users = db.session.query(User).order_by(User.age.desc()).all()
12
13 # 简化的查询方式
14 users = User.query.filter(User.name == 'Alice').all()
```

表 8-8 最常用的 SQLAlchemy 查询方法

方 法	说 明
<code>all()</code>	以列表形式返回查询的所有结果
<code>first()</code>	返回查询的第一个结果，如果没有结果，则返回 <code>None</code>
<code>first_or_404()</code>	返回查询的第一个结果，如果没有结果，则终止请求，返回 404 错误响应
<code>get()</code>	返回指定主键对应的行，如果没有对应的行，则返回 <code>None</code>
<code>get_or_404()</code>	返回指定主键对应的行，如果没找到指定的主键，则终止请求，返回 404 错误响应
<code>count()</code>	返回查询结果的数量
<code>paginate()</code>	返回一个 <code>Paginate</code> 对象，包含指定范围内的结果

集成python shell

避免重复导入数据库实例与模型，可以预先配置shell

```
1 # 自定义shell启动时自动导入的变量
2 @app.shell_context_processor
3 def make_shell_context():
4     return dict(db=db, User=User, Role=Role)
```

数据迁移

在开发的过程中有时需要修改数据库模型，并且修改后还要更新数据库。

仅当数据库表不存在时，Flask-SQLAlchemy才会更根据模型创建，因此更新表的方式就是先删除旧表，但是这样会丢失数据库一种的全部数据。

另一种更新表的方式是使用数据库与迁移框架

```
1 pip install flask-migrate
```

初始化

```
1 from flask_migrate import Migrate
2 migrate=Migrate(app,db)
```

操作方式类似于git

```
1 初始化 flask db init
2 对模型做修改, flask db migrate -m "initial migration"
3 把改动应用到数据库 flask db upgrade
```

撤销上一次改动flask db downgrade，可能会导致数据丢失

电子邮件

初始化

电子邮件服务由Flask-Mail 扩展提供，使用SMTP协议将邮件交给服务器发送

```
1 pip install flask-mail
```

配置，使用QQ服务器

```

1 from flask_mail import Mail
2
3 # 配置邮箱服务器信息
4 app.config['MAIL_SERVER'] = 'smtp.qq.com' # 邮件服务器，比如QQ邮箱
5 app.config['MAIL_PORT'] = 465 # SSL端口号
6 app.config['MAIL_USE_SSL'] = True # 使用SSL
7 app.config['MAIL_USERNAME'] = os.environ.get('MAIL_USERNAME') # 发件人邮箱
8 app.config['MAIL_PASSWORD'] = os.environ.get('MAIL_PASSWORD') # 授权码
9
10 也可以自定义设置一些键值对比如标题前缀...
11
12 mail=Mail(app) # 注意配置写到初始化前面

```

邮箱授权码等不应直接写入代码，可以放置到环境变量中再读取

QQ邮箱获取授权码方法如下

登录 [QQ邮箱网页版](#) → 设置 → 账户 → 开启 SMTP/IMAP → 获取授权码（会发送短信验证）

示例

发送邮件示例

```

1 def send_mail(to,subject,template,**kwargs):
2     # 创建邮件对象，第一个是标题，sender发件人，recipients收件人列表
3     msg = Message(app.config['FLASKY_MAIL_SUBJECT_PREFIX']+subject,
4                   sender=app.config['FLASKY_MAIL_SENDER'],recipients=[to])
5     # 渲染文本和HTML内容
6     msg.body=render_template(template+'.txt',**kwargs)
7     msg.html=render_template(template+'.html',**kwargs)
8     mail.send(msg)

```

异步发送

网页如果不使用异步发送邮件，那么调用mail.send()时会卡住直到邮件发送完成

```
1 def send_async_email(app,msg):
2     with app.app_context():
3         mail.send(msg)
4
5 def send_email(to, subject, template, **kwargs):
6     msg = Message(current_app.config['FLASKY_MAIL_SUBJECT_PREFIX'] + subject,
7     sender=current_app.config['FLASKY_MAIL_SENDER'], recipients=[to])
8     msg.body = render_template(template + '.txt', **kwargs)
9     msg.html = render_template(template + '.html', **kwargs)
10    # 使用多线程的时候需要_get_current_object()传入真正的实例对象
11    tr=Thread(target=send_async_email,args=[current_app._get_current_object() ,msg])
12    tr.start()
13    return tr
```

项目结构

总体概览

```
1 your_project/
2 |
3 └─ app/
4 |   └─ __init__.py      # 应用工厂，注册蓝图和扩展
5 |   └─ models.py        # 数据库模型
6 |   └─ email.py          # 邮件发送逻辑（如果有）
7 |   └─ main/             # 一个蓝图（模块）
8 |       └─ __init__.py  # 注册蓝图
9 |       └─ views.py      # 路由和视图函数
```



```

10 |   |   └─ forms.py      # 表单
11 |   |   └─ ...          # 其他
12 |   └─ templates/      # HTML 模板（可子目录区分模块）
13 |   |   └─ ...
14 |   |
15 |   └─ statics/         # 静态文件(CSS JS 图片 字体) 不需要服务器处理，由浏览器请求加载
16 |       └─ ...
17 |
18 └─ migrations/         # flask-migrate生成的迁移目录
19 |
20 └─ config.py           # 配置文件（你已经写好的）
21 └─ manage.py           # 启动入口（flask run / shell用它）
22 └─ requirements.txt    # 项目依赖
23 └─ README.md
24

```

config.py

config.py是一个Python配置文件，用于集中管理应用程序的不同环境

```

1  import os
2
3  basedir=os.path.abspath(os.path.dirname(__file__))
4
5  # 配置基类
6  class Config:
7      SECRET_KEY = os.environ.get('SECRET_KEY') or os.urandom(24) # 设置密钥，最好是固定的
8      SQLALCHEMY_TRACK_MODIFICATIONS = False                    # 不追踪数据库修改
9
10
11     '''邮件设置 以QQ邮件服务器为例'''
12     MAIL_SERVER='smtp.qq.com'    # SMTP服务器
13     MAIL_PORT=465
14     MAIL_USE_SSL=True
15     MAIL_USERNAME=os.environ.get('MAIL_USERNAME') # 凑够环境变量中取得
16     MAIL_PASSWORD=os.environ.get('MAIL_PASSWORD')

```

```
17 MAIL_DEFAULT_SENDER = '2228632512@qq.com' # 默认发送者
18 FLASKY_MAIL_SUBJECT_PREFIX = '[Flasky]' # 前缀
19 FLASKY_MAIL_SENDER = 'Flasky Admin <2228632512@qq.com>' # 发送者
20
21
22 '''数据库设置'''
23 SQLALCHEMY_DATABASE_URI='sqlite:/// ' + os.path.join(basedir, 'data.sqlite')
24
25 #
26 @staticmethod
27 def init_app(app):
28     """可选：在应用创建后执行额外的初始化"""
29     pass # 默认不执行任何操作
30 # 下面是几个子类
31
32 '''
33 开发环境，启用调试模式
34 '''
35 class DevelopmentConfig(Config):
36     DEBUG = True # 开启调试模式
37
38 '''
39 测试环境，使用内容数据库
40 '''
41 class TestingConfig(Config):
42     TESTING = True # 启用测试模式
43     SQLALCHEMY_DATABASE_URI = 'sqlite:///memory:' # 使用内存数据库
44
45
46 '''
47 生产环境，关闭调试模式
48 '''
49 class ProductionConfig(Config):
50     DEBUG = False # 关闭调试模式
51
52 # 配置字典
53 config = {
54     'development': DevelopmentConfig,
```

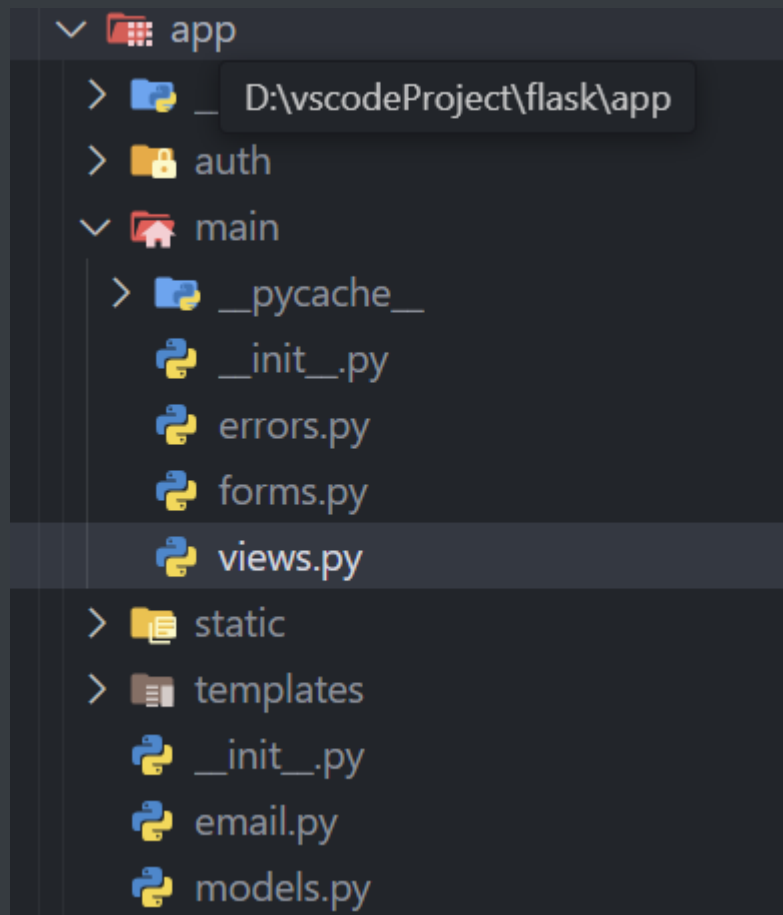
```
55     'testing': TestingConfig,  
56     'production': ProductionConfig,  
57     'default': DevelopmentConfig  
58 }
```

在主程序中加载配置

```
1  from flask import Flask  
2  from config import config # 导入 config 字典  
3  
4  app = Flask(__name__)  
5  
6  app.config.from_object(config['default'])
```

应用包

应用包存放应用的所有代码，模板静态文件(templates, statics)



在单个文件中开发应用很方便，但是应用在全局作用域创建无法动态修改配置，而在测试中需要使用不同的配置，使用应用工厂函数可以延迟创建实例，在创建前自由修改配置

app.__init__

```
1 from flask import Flask,render_template
2 from flask_bootstrap import Bootstrap
3 from flask_mail import Mail
4 from flask_moment import Moment
5 from config import config
6 from flask_sqlalchemy import SQLAlchemy
7 from flask_migrate import Migrate
8
9 bootstrap=Bootstrap()
10 mail=Mail()
11 moment=Moment()
12 db=SQLAlchemy()
13 migrate=Migrate()
14
```

```

15 def create_app(config_name):
16     app=Flask(__name__)
17     app.config.from_object(config[config_name])
18     config[config_name].init_app(app)
19
20     bootstrap.init_app(app)
21     mail.init_app(app)
22     moment.init_app(app)
23     db.init_app(app)
24     migrate.init_app(app,db)
25     '''
26     添加路由和自定义的错误页面
27     '''
28     return app

```

转换为应用工厂函数的操作让定义路由变得复杂了，在单脚本应用中，实例存在于全局作用域中，路由可以直接使用装饰器定义，但是用了应用工厂函数后，实例在运行时创建，只有在调用create_app后才能用装饰器定义路由，这样就太晚了。

而蓝图可以解决这个问题，蓝图定义的路由和错误处理程序处于休眠状态，等主程序调用create_app()时会把这些函数“注册”上去

主蓝本(app/main/__init__.py)

```

1 from flask import Blueprint
2
3 # 第一个参数是蓝图的名称，第二个参数是蓝图所在的包
4 main=Blueprint('main',__name__)
5
6 # 能把路由和错误处理程序与蓝图关联起来，但是要在最后导入防止循环依赖
7 from . import views,errors # 相对导入

```

主蓝图下的视图函数和错误处理程序

错误处理程序(app/main/errors.py)

```

1 from flask import render_template
2 from . import main
3
4 @main.app_errorhandler(404)
5 def page_not_found(e):
6     return render_template('404.html'),404
7
8
9 @main.app_errorhandler(500)
10 def internal_server_error(e):
11     return render_template('500.html'),500

```

视图函数app/main/views.py，修改路由装饰器app.route->main.route，实例修改为app->current_app，url_for参数index->main.index即主蓝图的名称

```

1 @main.route('/',methods=['GET','POST'])
2 def index():
3     form=NameForm()
4     if form.validate_on_submit():
5         user = User.query.filter_by(username=form.name.data).first()
6         if user is None:
7             user = User(username=form.name.data)
8             db.session.add(user)
9             session['known'] = False
10            if current_app.config['MAIL_USERNAME']:
11                send_email(current_app.config['MAIL_USERNAME'], 'New User',
12                    'email', user=user)
13            else:
14                session['known']=True
15                session['name']=form.name.data
16                form.name.data=''
17                return redirect(url_for('index'))
18            return
19
20 render_template('index.html',form=form,name=session.get('name'),current_time=datetime
21     .now(timezone.utc))

```

注册主蓝本

```
1 def create_app(config_name):
2     app=Flask(__name__)
3     app.config.from_object(config[config_name])
4     config[config_name].init_app(app)
5
6     bootstrap.init_app(app)
7     mail.init_app(app)
8     moment.init_app(app)
9     db.init_app(app)
10
11     '''
12     添加路由和自定义的错误页面
13     '''
14     from .main import main as main_blueprint
15     app.register_blueprint(main_blueprint)
16
17
18     from .auth import auth as auth_blueprint
19     app.register_blueprint(auth_blueprint,url_prefix='/auth') # url_prefix给蓝图下的路由加上公共前缀
20     return app
```

实例

```
1 import os
2 from app import create_app,db
3 from app.models import User,Role
4 from flask import Migrate
5
6 app=create_app('default')
7 migrate=Migrate()
8
9 @app.shell_context_processor
10 def make_shell_context():
11     return dict(db=db,User=User,Role=Role)
```

生成需求文件

```
1 pip freeze >requirements.txt
```

单元测试

```
1 import unittest
2 from flask import current_app
3 from app import create_app,db
4
5 class BasicTestCase(unittest.TestCase):
6     '''
7     测试前执行
8     '''
9     def setUp(self):
10         self.app=create_app('testing') # 创建falsk app
11         self.app_context=self.app.app_context() # 创建上下文
```


用户身份验证

密码

保证密码的安全性关键不在于存储密码本身，而是存储密码的散列值，计算密码散列值的函数接受密码作为输入，添加随即内容之后使用多种单向加密算法转换密码，最终得到一个和原始密码没有关系的字符序列，且无法还原成原始密码，核对密码时再次计算散列值(因为转换函数是可以复现的)，这样就可以储存散列密码以代替原始密码。

使用Werkzeug扩展来实现

```
1 pip install Werkzeug
```

重新定义模型，添加密码字段

```
1 from werkzeug.security import generate_password_hash, check_password_hash
2
3 class User(db.Model):
4     __tablename__ = 'User' # 表名
5     id = db.Column(db.Integer, primary_key=True) # 属性
6     username = db.Column(db.String(64), unique=True, index=True)
7     password_hash = db.Column(db.String(128))
8     # 创建外键引用 roles指的是表名
9     role_id = db.Column(db.Integer, db.ForeignKey('roles.id'))
10
11     # 用于表示对象的字符串方法
12     def __repr__(self):
13         return '<User %r>' % self.username
14
15     '''
16     @property装饰器将方法伪装为属性，当有人访问password时，会
```

```

17     触发异常
18     '''
19     @property
20     def password(self):
21         raise AttributeError("password is not a readabel attribute")
22
23     '''@password.setter与@property需成对出现 当有人设置password时调用此方法加密'''
24     @password.setter
25     def password(self,password):
26         self.password_hash=generate_password_hash(password)
27
28     def verify_password(self,password):
29         return check_password_hash(self.password_hash,password)
30

```

验证身份

使用Flask-Login 扩展

```

1 pip install flask-login

```

Flask-Login 需要应用中有User对象，且User模型必须实现以下的属性和方法

属性/方法	说 明
is_authenticated	如果用户提供的登录凭据有效，必须返回 True，否则返回 False
is_active	如果允许用户登录，必须返回 True，否则返回 False。如果想禁用账户，可以返回 False
is_anonymous	对普通用户必须始终返回 False，如果是表示匿名用户的特殊用户对象，应该返回 True
get_id()	必须返回用户的唯一标识符，使用 Unicode 编码字符串

可以继承满足大部分需求

```

1 from flask_login import UserMixin
2 class User(db.Model,UserMixin):

```

匿名用户尝试访问受保护的页面时，Flask-Login将重定向到登录页面

app/__init__.py

```
1 from flask_login import LoginManager
2 login_manager = LoginManager()
3 login_manager.login_view = 'auth.login' # login由蓝图auth定义，所以要加上蓝图的名称
4 def create_app(config_name):
5     # ...
6     login_manager.init_app(app)
7     # ...
```

Flask-Login需要指定一个函数用于获取用户信息

app/models.py

```
1 from . import login_manager
2 @login_manager.user_loader
3 def load_user(user_id):
4     return User.query.get(int(user_id))
```

保护路由

```
from flask_login import login_required

@app.route('/secret')
@login_required
def secret():
    return 'Only authenticated users are allowed!'
```

使用login_required装饰器的函数会判断current_user.is_authenticated表达式的结果是否为 True

添加登录表单

```
1 class LoginForm(FlaskForm):
2     email=StringField('Email',validators=[DataRequired(),Length(1,64),Email()])
3     password=PasswordField('Password',validators=[DataRequired()])
4     remeber_me=BooleanField('Keep me logged in ')
5     submit=SubmitField('Login in')
```

添加登入登出按钮， `current_user`是Flask-Login 定义的，模板中自动加载

```
<ul class="nav navbar-nav navbar-right">
    {% if current_user.is_authenticated %}
    <li><a href="{{ url_for('auth.logout') }}">Log Out</a></li>
    {% else %}
    <li><a href="{{ url_for('auth.login') }}">Log In</a></li>
    {% endif %}
</ul>
```

登录

Login

Email

Password

☐ Keep me logged in

Login in

```
1 @auth.route('/login',methods=['POST','GET'])
2 def login():
3     form=LoginForm()
```

```
4     # 是否有提交
5     if form.validate_on_submit():
6         # 查询
7         user=User.query.filter_by(email=form.email.data).first()
8         # 通过验证
9         if user and user.verify_password(form.password.data):
10             login_user(user) # 把用户信息储存在session,这样用户才能被识别登录状态
11             # 用户访问未授权的URL时会显示登录表单，原URL会保存在next参数中
12             next=request.args.get('next')
13             # 不存在则重定向到首页
14             if not next or next.startswith('/'):
15                 next=url_for('main.index')
16             return redirect(next)
17         flash('Invalid username or password')
18     return render_template('auth/login.html',form=form)
```

登录： 用户在成功登录后， `current_user` 会被赋值为对应的 `User` 实例。此时，`current_user.is_authenticated` 为 `True`，表示用户已经登录。登陆后的用户ID会被存储到会话中，然后每次请求时，Flask-login会根据会话中的ID查找User对象，将其赋予`current_user`对象。

登出： 当用户登出时，Flask-Login 会清除 `current_user`，并将其设置为匿名用户（即 `AnonymousUserMixin`）。此时，`current_user.is_authenticated` 为 `False`，表示用户未登录。

登出

`logout_user`会删除并重设用户会话

```

1 from flask_login import login_required,logout_user
2 @auth.route('/logout')
3 @login_required
4 def logout():
5     logout_user()
6     flash('you have logged out')
7     return redirect(url_for('.main.index'))

```

Flask-Login运行机制

提交表单调用Flask-Login的login_user()函数将用户ID以字符串形式写入到用户会话，之后会重定向到其他页面。

在其他页面渲染时会使用Flask-Login的current_user，首先会调用Flask-Login内部的_get_user() 函数找出用户，_get_user() 检查用户会话中有没有用户ID，如果没有返回一个Flask-Login的AnonymousUser 实例，反之使用user_loader装饰器注册的函数传入用户ID，加载用户对象并返回。

注册表单

```

1 from flask_wtf import FlaskForm
2 from wtforms import StringField,SubmitField,PasswordField,BooleanField
3 from wtforms.validators import DataRequired,Email,Length,Regexp,EqualTo
4 from wtforms import ValidationError
5 from ..models import User
6
7 class RegistrationForm(FlaskForm):
8     # 邮箱
9     email=StringField('Email',validators=[DataRequired(),Length(1,64),Email()])
10    # 用户名，规定了字符组成
11    username=StringField('Username',validators=[DataRequired(),Regexp('^[A-Za-z][A-
12                                     Za-z._]*$'),0,'Usernames must have only letters, numbers, \
13                                     dots or underscores'])
14    # 密码有二次确认

```

```
14     password=PasswordField('Password',validators=[DataRequired(),EqualTo('password2',\
15         message='Passwords must match'))])
16     password2=PasswordField('Password2',validators=[DataRequired(),])
17     submit=SubmitField('Register')
18
19     # 验证函数 验证密码和邮箱是否已经被使用过
20     def validate_email(self,field):
21         if User.query.filter_by(email=field.data).first():
22             raise ValidationError('Email already registered')
23
24     def validate_username(self,field):
25         if User.query.filter_by(username=field.data).first():
26             raise ValidationError('Username already in use')
```

代理对象

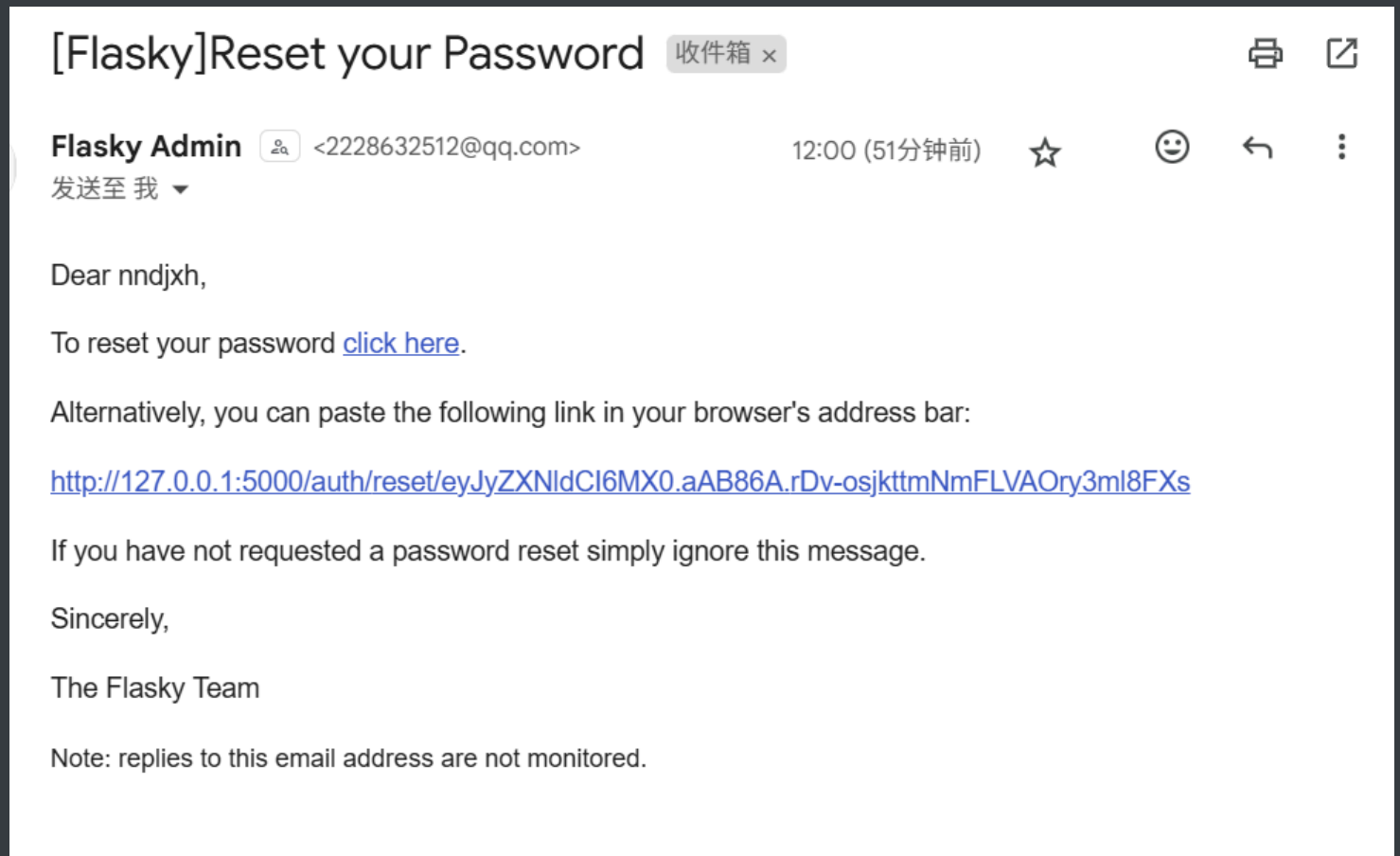
```
1 from flask_login import current_user
```

用于获取当前登录用户，如果用户已登录则返回User对象，反之返回匿名用户对象
AnonymousUserMixin

属性/方法	说明
current_user.is_authenticated	如果用户已登录返回 True ， 否则返回 False 。
current_user.is_active	如果用户账户是活跃的（未被禁用） 返回 True 。
current_user.is_anonymous	如果是匿名用户（未登录） 返回 True 。
current_user.get_id()	返回用户的唯一标识（通常是 id ），用于加载用户。

确认账户

通过邮件确认



在模型中添加一个验证字段，模型中添加生成和检验令牌的方法

```
1 from itsdangerous import URLSafeTimedSerializer as Serializer
2 class User(db.Model, UserMixin):
3     confirmed=db.Column(db.Boolean, default=False)
4
5     def generate_confirmation_token(self, expiration=3600):
6         '''
7         Serializer用于生成和验证加密令牌 secret_key密钥, salt盐值 防止生成相同的签名
8         '''
9         s=Serializer(current_app.config["SECRET_KEY"], 'confirmation')
10        # dumps接受要加密的数据, 把用户的ID作为加密数据, 返回字符串
11        return s.dumps({'confirm':self.id})
12
13    def confirm(self, token, max_age=3600):
14        # 解密时, 要用相同参数的Serializer
```

```

15         s=Serializer(current_app.config["SECRET_KEY"], 'confirmation')
16     try:
17         # loads接受令牌并检验令牌的有效性, max_age设置有效时间
18         data=s.loads(token.encode('utf-8'),max_age=max_age)
19     except Exception as e:
20         print(e)
21         return False
22     # 不匹配
23     print(data.get('confirm'))
24     if data.get('confirm')!=self.id:
25         return False
26     # 修改确认字段
27     self.confirmed=True
28     db.session.add(self)
29     return True

```

注册视图函数

```

1  @auth.route('/register',methods=['GET','POST'])
2  def register():
3      form=RegistrationForm()
4      if form.validate_on_submit():
5          # 创建用户
6
7          user=User(email=form.email.data,username=form.username.data,password=form.password.d
8          ata)
9
10         # 先提交, 因为token需要用户的id
11         db.session.add(user)
12         db.session.commit()
13         # 生成token
14         token=user.generate_confirmation_token()
15         # 发送确认邮件
16         send_email(user.email,'Confirm you
17         Accout','auth/email/confirm',user=user,token=token)
18         flash('A confirmation email has been sent to you by email.')
19         return redirect(url_for('auth.login'))

```

```
16     return render_template('auth/register.html', form=form)
```

模板

url_for需要完整，且带上token

```
1  <p>Dear {{ user.username }},</p>
2  <p>Welcome to <b>Flasky</b>!</p>
3  <p>To confirm your account please <a href="{{ url_for('auth.confirm', token=token,
    _external=True) }}">click here</a>.</p>
4  <p>Alternatively, you can paste the following link in your browser's address bar:</p>
5  <p>{{ url_for('auth.confirm', token=token, _external=True) }}</p>
6  <p>Sincerely,</p>
7  <p>The Flasky Team</p>
8  <p><small>Note: replies to this email address are not monitored.</small></p>
```

确认视图函数

```
1  @auth.route('/confirm/<token>')
2  @login_required # 需要先登录(因为允许未确认的用户访问不重要的页面)
3  def confirm(token):
4      # 如果已经确认过了，重定向
5      if current_user.confirmed:
6          return redirect(url_for('main.index'))
7      # 成功确认，重定向
8      if current_user.confirm(token):
9          db.session.commit()
10         flash('You have confirmed your account! Thanks')
11     else:
12         flash('The confirmation link is invlaid or has expired')
13     return redirect(url_for('main.index'))
```

允许未确认的用户登录但只显示一个页面(unconfirmed), 使用生命周期钩子

```
1 @auth.before_app_request
2 def before_request():
3     # 已登录未确认, 不在访问验证蓝图, 也不是对静态文件的请求, 则拦截
4     if current_user.is_authenticated \
5         and not current_user.confirmed \
6         and request.blueprint != 'auth' \
7         and request.endpoint != 'static':
8         return redirect(url_for('auth.unconfirmed'))
```

未确认用户只能访问主页

```
1 @auth.route('/unconfirmed')
2 def unconfirmed():
3     if current_user.is_anonymous or current_user.confirmed:
4         return redirect(url_for('main.index'))
5     return render_template('auth/unconfirmed.html')
```

重发确认邮件

```
1 @auth.route('/confirm')
2 def resend_confirmation():
3     token=current_user.generate_confirmation_token()
4     send_email(current_user.email, 'Confirm your
5     Account', 'auth/email/confirm', user=current_user, token=token)
6     flash('A confirmation email has been sent to you by email.')
7     return redirect(url_for('main.index'))
```

修改密码

Change Your Password

Old password

Password

Password2

Update password

在auth下进行

```
1 class ChangePasswordForm(FlaskForm):
2     old_password=PasswordField('Old password',validators=[DataRequired()])
3     password=PasswordField('Password',validators=[
4         DataRequired(),EqualTo('password2',\
5             message='Passwords must match')])
6     password2=PasswordField('Password2',validators=[DataRequired(),])
7     submit=SubmitField('Update password')
8
9 @auth.route('/change-password',methods=['POST','GET'])
10 @login_required
11 def change_password():
12     # 表单
13     form=ChangePasswordForm()
14     if form.validate_on_submit():
15         # 如果旧密码输入正确
16         if current_user.verify_password(form.old_password.data):
17             # 提交修改
18             current_user.password=form.password.data
19             db.session.add(current_user)
```

```
19         db.session.commit()
20         flash('Your password has been updated')
21         return redirect(url_for('main.index'))
22     else:
23         flash('Invalid password')
24     return render_template('auth/change_password.html', form=form)
```

重设密码

Login

Email

Password

☐ Keep me logged in

Login in

Forgot your password? [Click here to reset it.](#)

Reset Your Password

Email

Reset Password

在登陆页面选择重设密码->password_reset_request处提交email，如果email存在会发送携带token的邮件->点击邮件链接，验证token，如果正确则重设密码，返回登陆页面。


```

1 class PasswordResetRequestForm(FlaskForm):
2     email=StringField('Email',validators=[DataRequired(),Length(1,64),Email()])
3     submit = SubmitField('Reset Password')
4 class PasswordResetForm(FlaskForm):
5     password=PasswordField('Password',validators=[DataRequired(),EqualTo('password2',\
6         message='Passwords must match')])
7     password2=PasswordField('Password2',validators=[DataRequired(),])
8     submit=SubmitField('Reset Password')

```

视图函数

```

1 @auth.route('/reset/<token>',methods=['POST','GET'])
2 def password_reset(token):
3     # 已经登录不需要重设密码
4     if not current_user.is_anonymous:
5         return redirect(url_for('main.index'))
6     form=PasswordResetForm()
7     if form.validate_on_submit():
8         # token通过验证, 提交修改
9         if User.reset_password(token, form.password.data):
10             db.session.commit()
11             flash('Your password has been updated.')
12             return redirect(url_for('auth.login'))
13         else:
14             flash('Update fail ')
15             return redirect(url_for('main.index'))
16     return render_template('auth/reset_password.html',form=form)

```


邮箱修改

Change Your Email Address

New Email

Password

Update Email Addressrd

邮件修改在已经登陆的状态下进行，大致流程和修改密码差不多

对新地址要发送邮件确认，token中携带新邮件的地址，处于登录状态下(原地址)点击确认地址可以修改邮件地址

```
1 class User(db.Model,UserMixin):
2     def generate_email_change_token(self,new_email):
3         s=Serializer(current_app.config["SECRET_KEY"],'confirmation')
4         # 携带新邮件的地址
5         return s.dumps({'change_email':self.id,'new_email':new_email})
6     def change_email(self,token):
7         # 解密
8         s=Serializer(current_app.config["SECRET_KEY"],'confirmation')
9         try:
10             data=s.loads(token.encode('utf-8'))
11         except:
12             return False
13         #当前登录的用户不是要修改的用户
14         if data.get('change_email')!=self.id:
15             return False
16         # 没有携带
17         new_email=data.get('new_email')
18         if not new_email:return False
19         # 新邮件地址已经存在
```

```
20         if self.query.filter_by(email=new_email).first():return False
21
22         # 修改
23         self.email=new_email
24         db.session.add(self)
25         return True
```

添加表单

```
1 class ChangeEmailForm(FlaskForm):
2     email=StringField('New Email',validators=[DataRequired(),Length(1,64),Email()])
3     password=PasswordField('Password',validators=[DataRequired()])
4     submit = SubmitField('Update Email Addressrd')
5
6     # 自定义验证方式
7     def Validata_email(self,field):
8         if User.query.filter_by(email=field.email.data).first():
9             raise ValidationError('Email already registered.')
```

路由函数

```
1 @auth.route('/change_email',methods=['POST','GET'])
2 @login_required
3 def change_email_request():
4     form=ChangeEmailForm()
5     # 填写表单
6     if form.validate_on_submit():
7         # 验证密码
8         if current_user.verify_password(form.password.data):
9             new_email=form.email.data
10            # 生成token
11            token=current_user.generate_email_change_token(new_email)
12            # 发送邮件
```

```
13         send_email(new_email, 'Confirm your email
address', 'auth/email/change_email', user=current_user, token=token)
14         flash('An email with instructions to confirm your new email address has
been sent to you.')
15         return redirect(url_for('main.index'))
16     else:
17         flash('Invalid email or password.')
18         return render_template('auth/change_email.html', form=form)
19
20
21 @auth.route('/change_email/<token>', methods=['POST', 'GET'])
22 @login_required
23 def change_email(token):
24     # 验证并修改
25     if current_user.change_email(token):
26         db.session.commit()
27         flash('You email address has been updated')
28     else:
29         flash('Invalid request')
30     return redirect(url_for('main.index'))
```

用户角色

用户在数据库中需要分配不同的角色

模型

角色模型中用permission表示用户的权限

权限值设置2的幂次，这样权限在组合后的值仍是唯一的

操作	权限名	权限值
关注用户	FOLLOW	1
在他人的文章中发表评论	COMMENT	2
写文章	WRITE	4
管理他人发表的评论	MODERATE	8
管理员权限	ADMIN	16

```
1 class Permission:
2     Follow=1
3     COMMENT=2
4     WRITE=4
5     MODERATE=8
6     ADMIN=16
```

```
1 class Role(db.Model):
2     ...
3     # 是否是默认角色，只应有一个为True
4     default=db.Column(db.Boolean,default=False,index=True)
5     # 在定义时设置默认值，只有在添加到数据库后才会设置值，不方便做计算
6     permissions=db.Column(db.Integer)
7
8     def __init__(self,**kwargs):
9         # 父类的构造函数
10        super(Role,self).__init__(**kwargs)
11        # 将权限的初始话放到构造函数中
12        if not self.permissions:
13            self.permissions=0
14        ...
15
16    def has_permission(self,perm):
17        return self.permissions&perm==perm
18
19    def add_permission(self,perm):
20        if not self.has_permission(perm):
21            self.permissions+=perm
```

```

22     def remove_permission(self, perm):
23         if self.has_permission(perm):
24             self.permissions -= perm
25     def reset_permission(self):
26         self.permissions = 0
27
28     # 将预定义的角色及其权限写入到数据库，模型出现改动后，调用该方法更新橘色
29     @staticmethod
30     def insert_roles():
31         roles = { 'User': [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE ],
32                  'Moderator':
33 [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE, Permission.MODERATE],
34                  'Administrator':
35 [Permission.FOLLOW, Permission.COMMENT, Permission.WRITE, Permission.MODERATE, Permission
36 .ADMIN],
37                  }
38         default_role = 'User'
39         for r in roles:
40             role = Role.query.filter_by(name=r).first()
41             # 不存在则新建
42             if not role:
43                 role = Role(name=r)
44                 role.reset_permission()
45                 for perm in roles[r]:
46                     role.add_permission(perm)
47             # 将user设置为默认角色
48             role.default = (role.name == default_role)
49             db.session.add(role)
50         db.session.commit()

```

赋予角色

管理员用户在注册时应该被赋予管理员权限，初始化时判断注册的邮箱是否在FLASKY_ADMIN中

```
1 def __init__(self,**kwargs):
2     super(User,self).__init__(**kwargs)
3     if self.role is None:
4         if self.email in current_app.config['FLASKY_ADMIN']:
5             self.role=Role.query.filter_by(name='Administrator').first()
6         else:
7             self.role=Role.query.filter_by(default=True).first()
```

用户检验

需要经常检验用户是否拥有某项权限

```
1 class User(UserMixin, db.Model):
2     # ...
3     def can(self, perm):
4         return self.role is not None and self.role.has_permission(perm)
5     def is_administrator(self):
6         return self.can(Permission.ADMIN)
```

定义匿名用户类

```
1 from flask_login import AnonymousUserMixin
2 class AnonymousUser(AnonymousUserMixin):
3     def can(self, permissions):
4         return False
5     def is_administrator(self):
6         return False
7 login_manager.anonymous_user = AnonymousUser
```

想要让视图函数只对具有特定权限的用户开放，可以使用自定义的装饰器

app/decorators.py

```
1 from functools import wraps
2 from flask import abort
3 from flask_login import current_user
4 from .models import Permission
5
6 # 权限装饰器，带参数的装饰器要用装饰器的装饰器
7 def permission_required(permission):
8     def decorator(f):
9         @wraps(f)
10         def decorated_function(*args,**kwargs):
11             if not current_user.can(permission):
12                 abort(403)
13             return f(*args,**kwargs)
14         return decorated_function
15     return decorator
16
17 def admin_required(f):
18     return permission_required(Permission.ADMIN)(f)
```

在视图函数上使用多个装饰器时应把route放在首位，剩下的装饰器按照视图函数的执行顺序排列

```
1 @main.route('/moderate')
2 @login_required
3 @permission_required(Permission.MODERATE)
4 def for_moderators_only():
5     return "For comment moderators!"
```

用户资料

资料信息

```
1 class User(db.Model, UserMixin):
2     ...
3
4     location=db.Column(db.String(64))
5     about_me=db.Column(db.Text())
6     member_since=db.Column(db.DateTime(), default=datetime.now(timezone.utc))
7     last_seen=db.Column(db.DateTime(), default=datetime.now(timezone.utc))
8
9     # 刷新用户最后访问时间
10    def ping(self):
11        self.last_seen=datetime.now(timezone.utc)
12        db.session.add(self)
13        db.session.commit()
```



```
1 # 注意虽然是在auth蓝图下，但是会被全局触发
2 @auth.before_app_request
3 def before_request():
4     # 已登录未确认，不在访问验证蓝图，也不是对静态文件的请求，则拦截
5     if current_user.is_authenticated:
6         # 更新用户登录状态
7         current_user.ping()
8         if not current_user.confirmed \
9         and request.blueprint != 'auth' \
10        and request.endpoint != 'static':
11             return redirect(url_for('auth.unconfirmed'))
```

资料编辑器

资料编辑器分为用户级别编辑器和管理员级别编辑器

资料编辑表单

```
1 class EditProfileForm(FlaskForm):
2     name=StringField('Real name ',validators=[Length(0,64)])
3     location=StringField('Location',validators=[Length(0,64)])
4     about_me=TextAreaField('About me ')
5     submit=SubmitField('Submit')
6
7     def validate_name(self,field):
8         if User.query.filter_by(username=field).first():
9             raise ValidationError('Username already registered.')
```

Edit Your Profile

Real name

丁真

Location

中国四川

About me

世界上最纯真的男孩子

Submit

资料编辑路由

```
1 @main.route('/edit-profile',methods=['POST','GET'])
2 @login_required
3 def edit_profile():
4     form=EditProfileForm()
5     if form.validate_on_submit():
6         current_user.name=form.name.data
7         current_user.location=form.location.data
8         current_user.about_me=form.about_me.data
9         db.session.add(current_user)
10        db.session.commit()
11        flash('Your profile has been updated.')
12        # 返回到用户资料页面
13        return redirect(url_for('main.user',username=current_user.username))
14    form.name.data=current_user.name
15    form.location.data=current_user.location
16    form.about_me.data=current_user.about_me
17    return render_template('edit_profile.html',form=form)
```

管理员表单，管理员可以修改更多的内容

```
1 class EditProfileAdminForm(FlaskForm):
2     email=StringField('Email',validators=[DataRequired(),Length(1,64),Email()])
3     username=StringField('Username',validators=[DataRequired(),Length(1,64)\
4         ,Regex('^[A-Za-z][A-Za-z0-9_]*$',0,'Usernames must have only letters,
5         numbers, dots or underscores')])
6     confirmed=BooleanField('Confirmed')
7     # 修改权限，权限从模型中动态获取
8     role=SelectField('Role',coerce=int)
9     name=StringField('Real name ',validators=[Length(0,64)])
10    location=StringField('Location',validators=[Length(0,64)])
11    about_me=TextAreaField('About me ')
12    submit=SubmitField('Submit')
13
14    def __init__(self,user,*args,**kwargs):
15        super(EditProfileAdminForm,self).__init__(*args,**kwargs)
16        # 动态加载role，因为role模型可能会改变
17        self.role.choices=[(role.id,role.name) for role in
18            Role.query.order_by(Role.name).all()]
19        self.user=user
20
21    def validate_email(self, field):
22        if field.data != self.user.email and \
23            User.query.filter_by(email=field.data).first():
24            raise ValidationError('Email already registered.')
25
26    def validate_username(self, field):
27        if field.data != self.user.username and \
28            User.query.filter_by(username=field.data).first():
29            raise ValidationError('Username already in user')
```

Edit Your Profile

Email

mingyuhzou@gmail.com

Username

nndjxh

☒ Confirmed

Role

Administrator

Real name

Location

About me

Submit

管理员路由

```
1 @main.route('/edit-profile/<int:id>', methods=['POST', 'GET'])
2 @login_required
3 @admin_required
4 def edit_profile_admin(id):
5     # 首先要找到用户
6     user=User.query.get_or_404(id)
7     form=EditProfileAdminForm(user=user)
8     if form.validate_on_submit():
9         user.email=form.email.data
10        user.username=form.username.data
11        user.confirmed=form.confirmed.data
```

```
12         user.role=Role.query.get(form.role.data)
13         user.name=form.name.data
14         user.location=form.location.data
15         user.about_me=form.about_me.data
16         db.session.add(user)
17         db.session.commit()
18         flash('Your profile has been updated.')
19         # 返回到用户资料页面
20         return redirect(url_for('main.user',username=user.username))
21     form.name.data=user.name
22     form.location.data=user.location
23     form.about_me.data=user.about_me
24     form.email.data=user.email
25     form.username.data=user.username
26     form.confirmed.data=user.confirmed
27     form.role.data=user.role_id
28
29     return render_template('edit_profile.html',form=form,user=user)
```

用户头像

头像使用Gravatar（基于邮箱地址生成头像）提供的服务，用户在Gravatar注册账户上传头像，所有支持Gravatar的网站就可以通过用户的邮箱地址（经过MD5哈希处理）生成Gravatar的URL，指向用户的头像



ps

丁真

from [中国四川](#)

世界上最纯真的男孩子

Member since 04/18/2025. Last seen a few seconds ago.

Edit Profile

MD5哈希处理

```
1 # 字符串需先编码为字节，获取十六进制的哈希值
2 hash=hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
```

Gravatar的URL参数

参数名	说 明
s	图像尺寸，单位为像素
r	图像级别，可选值有 "g"、"pg"、"r" 和 "x"
d	尚未注册 Gravatar 服务的用户使用的默认图像生成方式，可选值有： "404"，返回 404 错误；一个 URL，指向默认图像；某种图像生成方式，包括 "mm"、"identicon"、"monsterid"、"wavatar"、"retro" 和 "blank"
fd	强制使用默认头像

计算md5值较为耗时，因此最好将其存储在User模型中

```
1 class User(db.Model,UserMixin):
2     avatar_hash=db.Column(db.String(32))
3     def __init__(self,**kwargs):
4         ....
5         if self.email and not self.avatar_hash:
6             self.avatar_hash=self.gravatar_hash()
7
8     def change_email(self,token):
9         ....
10        self.email=new_email
11        self.avatar_hash=self.gravatar_hash()
12        db.session.add(self)
13        return True
14
15    # 生成哈希值
16    def gravatar_hash(self):
17        return hashlib.md5(self.email.lower().encode('utf-8')).hexdigest()
18
19    # 构造请求网址
20    def gravatar(self,size=100,default='identicon',rating='g'):
21        # https和http请求不同的网址
22        if request.is_secure:
23            url='https://secure.gravatar.com/avatar'
24        else:
25            url='http://www.gravatar.com/avatar'
26
27        # 如果没有哈希值则生成
28        hash=self.avatar_hash or self.gravatar_hash()
29        return f'{url}/{hash}?s={size}&d={default}&r={rating}'
```

博客文章

显示博客

创建博客模型

```
1 class Post(db.model):
2     __tablename__='posts'
3     id=db.Column(db.Integer,primary_key=True) # 属性
4     body=db.Column(db.Text)
5     timestamp=db.Column(db.DateTime,index=True ,default=datetime.now(timezone.utc))
6     author_id=db.Column(db.Integer,db.ForeignKey('users.id'))
7
8 class User(UserMixin, db.Model):
9     # ...
10    posts = db.relationship('Post', backref='author', lazy='dynamic')
```

表单

```
1 class PostForm(FlaskForm):
2     body=TextAreaField("What's on your mind?",validators=[DataRequired()])
3     submit=SubmitField('Submit')
```

博客首页的路由


```

1 @main.route('/', methods=['GET', 'POST'])
2 def index():
3     form=PostForm()
4     if form.validate_on_submit():
5         # 使用_get_current_object, 是因为current_user只是个代理对象而不是实例
6         post=Post(body=form.body.data, author=current_user._get_current_object())
7         db.session.add(post)
8         db.session.commit()
9         return redirect(url_for('main.index'))
10    # 按照时间降序, 显示所有文章
11    posts=Post.query.order_by(Post.timestamp.desc()).all()
12    return render_template('index.html', form=form, posts=posts)

```



tonya89

Might land natural send figure majority history. When around success consumer sy
west hope staff large government. Defense rich campaign money.



fmartin

Face present notice others energy physical. Score anyone audience. Free than sir
arm stay enjoy. Up single better. Together act rise. Sing say whom message mode



james27

Knowledge just find value artist compare perhaps. Break skill worry price. Boy offic
involve company contain. Two Republican future money type.



brian39

Her know moment couple century say. International exactly plant poor. Past star fir
discuss pass group room either claim. Business same page bar.

分页显示

首先要创建一个包含大量数据的测试数据库, 使用faker生成随机数据填充到数据库中

```
1 from random import randint
2 from sqlalchemy.exc import IntegrityError
3 from faker import Faker
4 from . import db
5 from .models import User, Post
6
7 def user(count=100):
8     fake=Faker()
9     i=0
10    while i<count:
11        u=User(email=fake.email(),
12              username=fake.user_name(),
13              password='password',
14              confirmed=True,
15              name=fake.name(),
16              location=fake.city(),
17              about_me=fake.text(),
18              member_since=fake.past_date())
19        db.session.add(u)
20        try:
21            db.session.commit()
22            i+=1
23        except IntegrityError: # 违反了数据库的约束，因为邮件和用户名是随机生成的有可能的重
24                                # 撤销更改
25            db.session.rollback()
26
27 def posts(count=100):
28     fake=Faker()
29     user_count=User.query.count()
30     for i in range(count):
31         # offset跳过多少条记录，与randint一起使用相等于随机选择用户
32         u=User.query.offset(randint(0,user_count-1)).first()
33         p=Post(body=fake.text(),timestamp=fake.past_date(),author=u)
34         db.session.add(p)
35     db.session.commit()
```

执行shell命令创建数据

```
(venv) $ flask shell
>>> from app import fake
>>> fake.users(100)
>>> fake.posts(100)
```

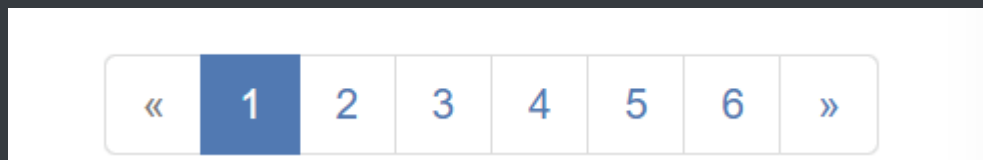
如果要分页的话就不能直接返回文章，而是要调用Flask-SQLAlchemy提供的paginate()方法

```
1 # 对查询的结果进行分页，paginate接受当前的页，per_page每页个数，error_out页数超出时只会返回空
   页面
2     pagination=Post.query.order_by(Post.timestamp.desc()).paginate(
3         page,per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
4         error_out=False
5     )
6
```

paginate() 方法的返回值是一个Pagination类对象

属 性	说 明
items	当前页面中的记录
query	分页的源查询
page	当前页数
prev_num	上一页的页数
next_num	下一页的页数
has_next	如果有下一页，值为 True
has_prev	如果有上一页，值为 True
pages	查询得到的总页数
per_page	每页显示的记录数量
total	查询返回的记录总数

方 法	说 明
<code>iter_pages(left_edge=2, left_current=2, right_current=5, right_edge=2)</code>	一个迭代器，返回一个在分页导航中显示的页数列表。这个列表的最左边显示 <code>left_edge</code> 页，当前页的左边显示 <code>left_current</code> 页，当前页的右边显示 <code>right_current</code> 页，最右边显示 <code>right_edge</code> 页。例如，在一个 100 页的列表中，当前页为第 50 页，使用默认配置，这个方法会返回以下页数：1、2、None、48、49、50、51、52、53、54、55、None、99、100。None 表示页数之间的间隔
<code>prev()</code>	上一页的分页对象
<code>next()</code>	下一页的分页对象



使用宏进行分页的渲染

```

1  {% macro pagination_widget(pagination, endpoint) %}
2  <ul class="pagination">
3      <li{% if not pagination.has_prev %} class="disabled"{% endif %}>
4          <a href="{% if pagination.has_prev %}{ url_for(endpoint,
page=pagination.prev_num, **kwargs) }{% else %}#{% endif %}">
5              &laquo;
6          </a>
7      </li>
8      {% for p in pagination.iter_pages() %}
9          {% if p %}
10             {% if p == pagination.page %}
11                 <li class="active">
12                     <a href="{ url_for(endpoint, page = p, **kwargs) }">{{ p }}</a>
13                 </li>
14             {% else %}
15                 <li>
16                     <a href="{ url_for(endpoint, page = p, **kwargs) }">{{ p }}</a>
17                 </li>
18             {% endif %}
19             {% else %}

```

```

20         <li class="disabled"><a href="#">&hellip;</a></li>
21     {% endif %}
22 {% endfor %}
23     <li{% if not pagination.has_next %} class="disabled"{% endif %}>
24         <a href="{% if pagination.has_next %}{ url_for(endpoint,
25             page=pagination.next_num, **kwargs) }}{% else %}#{% endif %}">
26             &raquo;
27         </a>
28     </li>
29 </ul>
30 {% endmacro %}
31

```

支持富文本

安装扩展

```
1 pip install flask-pagedown markdown bleach
```

Flask-PageDown为Flask应用提供了markdown编辑器支持，用户可以方便的写作Markdown格式的内容，然后在浏览器中呈现为HTML

首先要导入并初始化

```

1 from flask import Flask
2 from flask_pagedown import PageDown
3
4 app = Flask(__name__)
5 pagedown = PageDown(app)

```

将文本框换为PageDown提供的PagedownField

```
1 class PostForm(FlaskForm):
2     body=PageDownField("What's on your mind?",validators=[DataRequired()])
3     submit=SubmitField('Submit')
```

在用到PageDown的页面导入宏

```
1 {%block scripts %}
2 {{super()}}
3 {{pagedown.include_pagedown()}}
4 {%endblock%}
```

What's on your mind?

Title
asdad

Title

asdad

这样存入数据库的内容就是markdown格式的，但是发布的帖子的内容需要时html格式，为了避免每次都做转换，在模型中进行一次性转换。

```
1 class Post(db.Model):
2     body_html=db.Column(db.Text)
3
4     @staticmethod
5     def on_change_body(target,value,oldvalue,initiator):
6         # HTML中允许出现的标签
7         allowed_tags=['a', 'abbr', 'acronym', 'b', 'blockquote', 'code',
8                       'em', 'i', 'li', 'ol', 'pre', 'strong', 'ul',
9                       'h1', 'h2', 'h3', 'p']
```

```

10         '''markdown()将markdown内容转换为html
11         bleach.clean()清理HTML内容，过滤和清除不安全的HTML，strip控制是否完全删除不允许
    的标签和其内容
12         bleach.linkify()将文本中的URL自动转换为<a>标签
13         '''
14         target.body_html=bleach.linkify(bleach.clean(
15             markdown(value,output_format='html'),tags=allowed_tags,strip=True
16         ))
17
18     # 注册了一个事件监听器，监听set操作，如果修改了body内容则触发on_change_body
19     db.event.listen(Post.body,'set',Post.on_change_body)

```

在模板中渲染时，用|safe防止html被转义为文本

```

1  {% if post.body_html %}
2  {{ post.body_html | safe }}
3  {% else %}
4  {{ post.body }}
5  {% endif %}

```

修改文章

在渲染文章时带上文章ID，然后添加编辑按钮，在路由函数中处理

```

1  @main.route('/edit/<int:id>',methods=['GET','POST'])
2  @login_required
3  def edit(id):
4      post=Post.query.get_or_404(id)
5      # 只有管理员或文章作者才能修改文章
6      if current_user!=post.author and not current_user.can(Permission.ADMIN):
7          abort(403)

```

```
8     form=PostForm()
9     if form.validate_on_submit():
10         post.body=form.body.data
11         db.session.add(post)
12         db.session.commit()
13         flash('The post has been updated')
14         return redirect(url_for('mina.post',id=post.id))
15     form.body.data=post.body
16     return render_template('edit_post.html',form=form)
```



nndjxh

11 minutes ago

一个小测试

这是正文

Edit

Permalink



nndjxh

4 hours ago

fuck everything

Edit

Permalink



campbellpatrick

2 days ago

Yourself training company treat set. With unit adult low. Discover final social process. Or second same stand choose. Free dog on question opportunity ever miss. Seem next pretty.

Edit [Admin]

Permalink



mendezjoshua

3 days ago

Cold conference decade through. Field or to play beyond soldier. Husband next produce his memory. Be cover several painting. Set stage we. Baby hundred live herself lay traditional find.

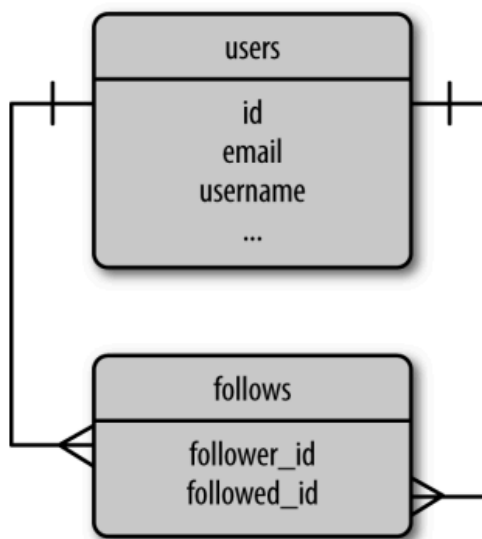
Edit [Admin]

Permalink

关注者

用户之间的关注，好友，联系人，伙伴等关系需要用多对多关系实现，原理见数据库笔记

这里的多对多关系较为特殊，因为只有一个模型，用户到用户，因此是自引用关系



使用一张关系表以及两张一对多的表

```
1 class Follow(db.Model):
2     __tablename__ = 'follows'
3     follower_id = db.Column(db.Integer, db.ForeignKey('Users.id'), primary_key=True) # 关
    注者
4     followed_id = db.Column(db.Integer, db.ForeignKey('Users.id'), primary_key=True) # 被
    关注的人
5     timestamp = db.Column(db.DateTime, default=datetime.now(timezone.utc))
6
7 class User(db.Model, UserMixin):
```

```
8
9     followed=db.relationship('Follow',foreign_key=
[Follow.follower_id],backref=db.backref('follower',lazy='joined'),
10                                     lazy='dynamic',cascade='all, delete-orphan')# 被我关注的人
    的ID 从User到关系表,就转换为了关注者(follower_id)
11     followers=db.relationship('Follow',foreign_key=
[Follow.followed_id],backref=db.backref('followed',lazy='joined'),
12                                     lazy='dynamic',cascade='all, delete-orphan')# 关注我的人的
    ID 从User到关系表,就转换为被关注者()
13
14     # 辅助方法,关注别人
15     def follow(self,user):
16         if not self.is_following(user):
17             f=Follow(follower=self,followed=user)
18             db.session.add(f)
19     # 取消关注
20     def unfollow(self,user):
21         f=self.followed.filter_by(followed_id=user.id).first()
22         if f:
23             db.session.delete(f)
24     # 是否关注
25     def is_following(self,user):
26         if user.id is None:
27             return False
28         return self.followed.filter_by(
29             followed_id=user.id).first() is not None
30     # 是否被关注
31     def is_followed_by(self,user):
32         if user.id is None:
33             return False
34         return self.followers.filter_by(
35             follower_id=user.id).first() is not None
36
```



campbellpatrick

Timothy Figueroa
from [Port Kellyland](#)

zjackson@example.net

Week time newspaper discussion too. Member allow side nation court send. Newspaper way debate need role him behavior.

Member since 04/11/2025. Last seen a day ago.

2 blog posts.

Unfollow

Followers: **1** Following: **0**

Edit Profile [Admin]

Followers of campbellpatrick

User

Since



nndjxh

04/19/2025

«

1

»

Followed by campbellpatrick

User

Since

«

»

联结操作

为了得到用户所关注的用户发表的文章，需要进行两次操作，一次查找用户关注的对象，第二次查找这些对象发表的文章，但是随着数据库不断变大，查找的工作量也不断增长，这种问题称为N+1，对于数据库查询最好的方法是一次完成，因此需要用到联结操作——查询结果中合并多个表的数据

```
1 class User(db.Model, UserMixin):
2     # 将方法定义为属性
3     @property
4     def followed_posts(self):
5         return
6         Post.query.join(Follow, Follow.followed_id==Post.author_id).filter(Follow.follower_id==
7                                     self.id)
```

显示关注的人


在渲染主页时，根据cookie的值显示不同的文章，主页建立两个选项卡分别调用路由来设置cookie

```
1 @main.route('/all')
2 @login_required
3 def show_all():
4     # cookie只能在响应对象中设置
5     resp=make_response(redirect(url_for('main.index')))
6     # set_cookie接收cookie的名称和值，以及过期时间
7     resp.set_cookie('show_followed', '', max_age=30*24*3600)
8     return resp
9
10 @main.route('followed')
11 @login_required
12 def show_followed():
13     resp=make_response(redirect(url_for('main.index')))
14     resp.set_cookie('show_followed', '1', max_age=30*24*3600)
15     return resp
16
17
```

```
18 @main.route('/', methods=['GET', 'POST'])
19 def index():
20     form = PostForm()
21     if current_user.can(Permission.WRITE) and form.validate_on_submit():
22         post = Post(body=form.body.data,
23                     author=current_user._get_current_object())
24         db.session.add(post)
25         db.session.commit()
26         return redirect(url_for('.index'))
27     page = request.args.get('page', 1, type=int)
28     show_followed = False
29     if current_user.is_authenticated:
30         show_followed = bool(request.cookies.get('show_followed', ''))
31     if show_followed:
32         query = current_user.followed_posts
33     else:
34         query = Post.query
35     pagination = query.order_by(Post.timestamp.desc()).paginate(
36         page=page, per_page=current_app.config['FLASKY_POSTS_PER_PAGE'],
37         error_out=False)
38     posts = pagination.items
39     return render_template('index.html', form=form, posts=posts,
40                           show_followed=show_followed, pagination=pagination)
```

All

Followed

 **nndjxh** 2 days ago


一个小测试

这是正文

Edit

Permalink

0 Comments


 **nndjxh** 2 days ago

fuck everything

Edit

Permalink

0 Comments


 **campbellpatrick** 4 days ago

Yourself training company treat set. With unit adult low. Discover final social process. Or second same stand choose. Free dog on question opportunity ever miss. Seem next pretty.

Edit [Admin]

Permalink

1 Comments

 **campbellpatrick** 9 days ago

Perhaps however same consider require understand apply resource. Direction general official audience. Recent color recognize.

Edit [Admin]

Permalink

0 Comments

«

1

»

用户评论

模型

```
1 class Comment(db.Model):
2     __tablename__ = 'comments'
3     id=db.Column(db.Integer,primary_key=True)
4     body=db.Column(db.Text)
5     timestamp=db.Column(db.DateTime,index=True ,default=datetime.now(timezone.utc))
6     author_id=db.Column(db.Integer,db.ForeignKey('Users.id'))
7     body_html=db.Column(db.Text)
```

```

8     disbale=db.Column(db.Boolean)
9     author_id=db.Column(db.Integer,db.ForeignKey('users.id'))
10    post_id=db.Column(db.Integer,db.ForeignKey('posts.id'))
11    @staticmethod
12    def on_change_body(target,value,oldvalue,initiator):
13        # HTML中允许出现的标签
14        allowed_tags=['a', 'abbr', 'acronym', 'b', 'code', 'em', 'i','strong']
15        '''markdown()将markdown内容转换为html
16        bleach.clean()清理HTML内容，过滤和清除不安全的HTML，strip控制是否完全删除不允许
    的标签和其内容
17        bleach.linkify()将文本中的URL自动转换为<a>标签
18        '''
19        target.body_html=bleach.linkify(bleach.clean(
20            markdown(value,output_format='html'),tags=allowed_tags,strip=True
21        ))
22    db.event.listen(Comment.body,'set',Comment.on_change_body)

```

同时还要在User和Post中添加一对多关系

修改路由函数

```

1  @main.route('/post/<int:id>',methods=['GET', 'POST'])
2  def post(id):
3      post=Post.query.get_or_404(id)
4      form=CommentForm()
5      if form.validate_on_submit():
6
7          comment=Comment(body=form.body.data,post=post,author=current_user._get_current_object())
8          db.session.add(comment)
9          db.session.commit()
10         flash('You commetnhas been published.')
11         return redirect(url_for('main.post',id=post.id,page=-1))
12     page=request.args.get('page',1,type=int)
13     if page== -1:
14         page=
15         (post.comments.count()-1)//current_app.config['FLASKY_COMMENTS_PER_PAGE']+1
16     pagination=post.comments.order_by(Comment.timestamp.asc()).paginate(

```

```
15     page=page,per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],error_out=False
16     )
17     comments=pagination.items
18     return render_template('post.html',posts=
19     [post],form=form,comments=comments,pagination=pagination)
```



campbellpatrick

4 days ago

Yourself training company treat set. With unit adult low. Discover final social process. Or second same stand choose. Free dog on question opportunity ever miss. Seem next pretty.

[Edit \[Admin\]](#)

[Permalink](#)

[1 Comments](#)



campbellpatrick

4 days ago

Yourself training company treat set. With unit adult low. Discover final social process. Or second same stand choose. Free dog on question opportunity ever miss. Seem next pretty.

[Edit \[Admin\]](#)

[Permalink](#)

[1 Comments](#)

Comments

Submit



nndjxh

1111

a few seconds ago

« 1 »

管理评论

```
1  @main.route('/moderate/enable/<int:id>')
2  @login_required
3  @permission_required(Permission.MODERATE)
4  def moderate_enable(id):
5      comment=Comment.query.get_or_404(id)
6      comment.disabled=False
7      db.session.add(comment)
8      db.session.commit()
9
10     return
11     redirect(url_for('main.moderate',page=request.args.get('page',1,type=int)))
12
13 @main.route('/moderate/disable/<int:id>')
14 @login_required
15 @permission_required(Permission.MODERATE)
16 def moderate_disable(id):
17     comment=Comment.query.get_or_404(id)
18     comment.disabled=True
19     db.session.add(comment)
20     db.session.commit()
21
22     return
23     redirect(url_for('main.moderate',page=request.args.get('page',1,type=int)))
24
25 @main.route('/moderate')
26 @login_required
27 @permission_required(Permission.MODERATE)
28 def moderate():
29     page=request.args.get('page',1,type=int)
30     pagination=Comment.query.order_by(Comment.timestamp.desc()).paginate(
31
32     page=page,per_page=current_app.config['FLASKY_COMMENTS_PER_PAGE'],error_out=False
33     )
34     comments=pagination.items
```

34

return

```
render_template('moderate.html', comments=comments, pagination=pagination, page=page)
```

Comment Moderation



nndjxh

2 minutes ago

This comment has been disabled by a moderator.

dsadad

Enable



nndjxh

38 minutes ago

This comment has been disabled by a moderator.

1111

Enable

« 1 »

应用编程接口

REST

REST——表现层状态转移，一种设计风格，看URL就知道请求什么，看http method就知道要做什么，看http status code就知道结果如何。

右边的是**REST** 风格

```
1 GET /rest/api/getDogs --> GET /rest/api/dogs 获取所有小狗狗
2 GET /rest/api/addDogs --> POST /rest/api/dogs 添加一个小狗狗
3 GET /rest/api/editDogs/:dog_id --> PUT /rest/api/dogs/:dog_id 修改一个小狗狗
4 GET /rest/api/deleteDogs/:dog_id --> DELETE /rest/api/dogs/:dog_id 删除一个小狗狗
```

比如酒店中对房间操作，各种操作对应的URL：开房/open/room/3242，退房/exit/3242/room，打理room/3242?method=clean，在REST架构下房间的URL就是/room/3242，而操作取决于http method

REST的特点如下：

- 一切都是资源
- URL中通常只有名词
- URL语义清晰明确
- 数据的载体是JSON
- 使用HTTP动词表示增删改查资源，GET：查询，POST：新增，PUT：更新，DELETE：删除
- 无状态，每一个请求都是独立的，完整的

如果想要手机上能浏览网页就需要REST架构

测试

代码覆盖度报告

可以检测单元测试覆盖了应用的多少功能，可以说明那些代码没有被检测到

```
1 pip install coverage
```

```
1 '''
2     开启代码覆盖统计，要在app创建之前，否则不会统计到模块
3 '''
4 COV=None
5 if os.environ.get('FLASK_COVERAGE'):
6     import coverage
7
8     # branch检测分支 include只对app/目录下的代码做覆盖率检测
9     COV=coverage.coverage(branch=True,include='app/*')
10    # 开始检测
11    COV.start()
12 ...
13
14 '''
15    注册flask test命令
16    --coverage 统计代码覆盖率并生成html报告
17 '''
18 @app.cli.command()
19 @click.option('--coverage/--no-coverage',default=False,help='Run tests under code
20 coverage.')
```

```
21 # 如果指定了coverage参数，但是没有环境变量中没有FLASK_COVERAGE就设置
22 if coverage and not os.environ.get('FLASK_COVERAGE'):
23     os.environ['FLASK_COVERAGE']='1'
24     # 需要重新运行程序否则不会统计到已经加载了的views, models,ma,pp模块
25     os.execvp(sys.executable, [sys.executable]+[sys.argv[0]+".exe"]+sys.argv[1:])
26
27 import unittest
28 # 自动加载tests/目录下所有以test*.py命名的测试模块
29 tests=unittest.TestLoader().discover('tests')
30 # 以详细模式运行测试
31 unittest.TextTestRunner(verbosity=2).run(tests)
32
33 # 停止统计，此时各个模块已经导入完毕
34 if COV:
35     COV.stop()
36     COV.save()
37     print('Coverage Summary:')
38     COV.report()
39     basedir=os.path.abspath(os.path.dirname(__file__))
40     covdir=os.path.join(basedir, 'tmp/coverage')
41
42     # 生成html报告
43     COV.html_report(directory=covdir)
44     print(f'HTML version: file:///{{covdir.replace(os.sep, "/")}}/index.html')
45     # 清除统计数据
46     COV.erase()
```

性能

记录影响性能的缓慢数据库查询

Flask-SQLAlchemy 提供了一个选项，可以记录一次请求中与数据库查询有关的统计数据

名 称	说 明
statement	SQL 语句
parameters	SQL 语句使用的参数
start_time	执行查询时的时间
end_time	返回查询结果时的时间
duration	查询持续的时间，单位为秒
context	表示查询在源码中所处位置的字符串

部署

首先编写dockerfile

```
1 # 指定基础镜像 即新镜像基于谁来构建
2 FROM python:3.11-alpine
3
4 ENV FLASK_APP=run.py
5 ENV FLASK_CONFIG=docker
6
7 # 创建新用户 -D默认选项——不设置密码不询问交互信息
8 RUN adduser -D flasky
```


指定脚本解释器

```
1  #!/bin/sh
```

激活python虚拟环境

```
1  . venv/bin/activate
```

执行自定义命令 数据库迁移，数据库初始化

```
1  flask deploy
```

run.py

```
1  @app.cli.command()
2  def deploy():
3      print("开始部署！")
4      # 执行数据库迁移
5      upgrade()
6      # 插入角色
7      print("插入角色...")
8      Role.insert_roles()
9      User.add_self_follows()
```

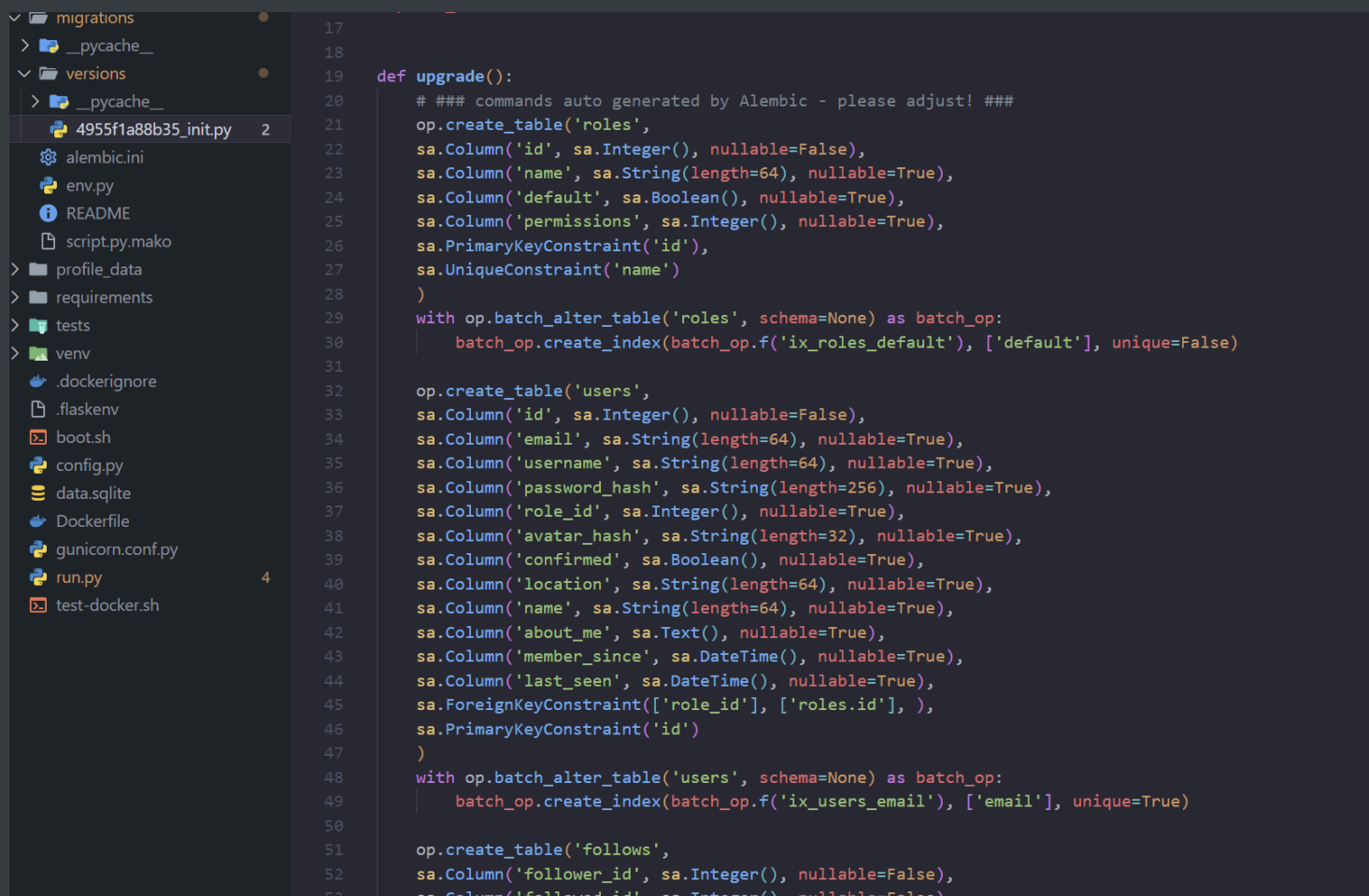
使用 Gunicorn 启动 Flask 应用，监听 5000 端口，`--access-logfile -` 访问日志输出到标准输出，`--error-logfile` 错误日志输出到标准错误，从 `run.py` 中加载 `app` 变量作为应用实例

```
1  exec gunicorn -b 0.0.0.0:5000 --access-logfile - --error-logfile - run:app
```


Gunicorn是一个python WSGI HTTP 服务器，用于生产环境部署Flask应用，flask run则是开发用的，相比于Gunicorn不稳定不安全。

注意想让数据库中能建立flask中的模型必须写对migrations，最好的方法是删除开发环境下的临时数据库以及migrations，在flask shell中db.create_all()，然后再初始化迁移文件flask db init

确保有如下所示的内容



```
17
18
19 def upgrade():
20     # ### commands auto generated by Alembic - please adjust! ###
21     op.create_table('roles',
22                     sa.Column('id', sa.Integer(), nullable=False),
23                     sa.Column('name', sa.String(length=64), nullable=True),
24                     sa.Column('default', sa.Boolean(), nullable=True),
25                     sa.Column('permissions', sa.Integer(), nullable=True),
26                     sa.PrimaryKeyConstraint('id'),
27                     sa.UniqueConstraint('name')
28     )
29     with op.batch_alter_table('roles', schema=None) as batch_op:
30         batch_op.create_index(batch_op.f('ix_roles_default'), ['default'], unique=False)
31
32     op.create_table('users',
33                     sa.Column('id', sa.Integer(), nullable=False),
34                     sa.Column('email', sa.String(length=64), nullable=True),
35                     sa.Column('username', sa.String(length=64), nullable=True),
36                     sa.Column('password_hash', sa.String(length=256), nullable=True),
37                     sa.Column('role_id', sa.Integer(), nullable=True),
38                     sa.Column('avatar_hash', sa.String(length=32), nullable=True),
39                     sa.Column('confirmed', sa.Boolean(), nullable=True),
40                     sa.Column('location', sa.String(length=64), nullable=True),
41                     sa.Column('name', sa.String(length=64), nullable=True),
42                     sa.Column('about_me', sa.Text(), nullable=True),
43                     sa.Column('member_since', sa.DateTime(), nullable=True),
44                     sa.Column('last_seen', sa.DateTime(), nullable=True),
45                     sa.ForeignKeyConstraint(['role_id'], ['roles.id'], ),
46                     sa.PrimaryKeyConstraint('id')
47     )
48     with op.batch_alter_table('users', schema=None) as batch_op:
49         batch_op.create_index(batch_op.f('ix_users_email'), ['email'], unique=True)
50
51     op.create_table('follows',
52                     sa.Column('follower_id', sa.Integer(), nullable=False),
53                     sa.Column('followed_id', sa.Integer(), nullable=False)
```

提交镜像

```
1 docker build -t "flasky" .
```

运行

```
1 docker run --name flasky -d -p 8000:5000 \  
2 -e SECRET_KEY=57d40f677aff4d8d96df97223c74d217 \  
3 -e MAIL_USERNAME=<your-gmail-username> \  
4 -e MAIL_PASSWORD=<your-gmail-password> flasky:latest
```

- --name 设置容器的名称
- -d 设置后台运行
- -p 设置端口映射，将运行机器的8000端口映射到5000，flask应用一般监听5000端口
- -e设置环境变量

在本地输入localhost:5000即可访问

使用sqlite数据库在容器停止时，数据就会丢失，因此在应用容器之外使用mysql数据库，通过docker不需下载mysql，只用拉去mysql镜像然后运行即可

使用mysql要安装依赖pymysql和cryptography，并更新依赖文件

运行

```
1 docker run --name mysql -d -e MYSQL_ROOT_PASSWORD=111111 -e MYSQL_DATABASE=flasky -e  
MYSQL_USER=flasky -e MYSQL_PASSWORD=111111 mysql:latest
```

- --name mysql 将mysql容器命名
- -d后台运行
- -e MYSQL_ROOT_PASSWORD 设置数据库root用户密码
- -e MYSQL_DATABASE
- -e MYSQL_DATABASE 创建一个名为flasky的数据库
- MYSQL_USER 设置数据库用户
- MYSQL_PASSWORD设置用户密码

- mysql:latest 使用镜像，如果没有拉取会自动拉取

flasky镜像

```
1 docker run -d -p 8000:5000 --link mysql:dbserver -e  
DATABASE_URL=mysql+pymysql://flasky:111111@dbserver/flasky -e  
MAIL_USERNAME=2228632512@qq.com -e MAIL_PASSWORD=jdilyngiczeadhha flasky:latest
```

- -p 8000:5000 将flasky镜像映射到运行机的8000端口
- --link mysql:dbserver 链接名为mysql的容器并重命名为dbserver
- -e DATABASE_URL 设置flasky中的环境变量