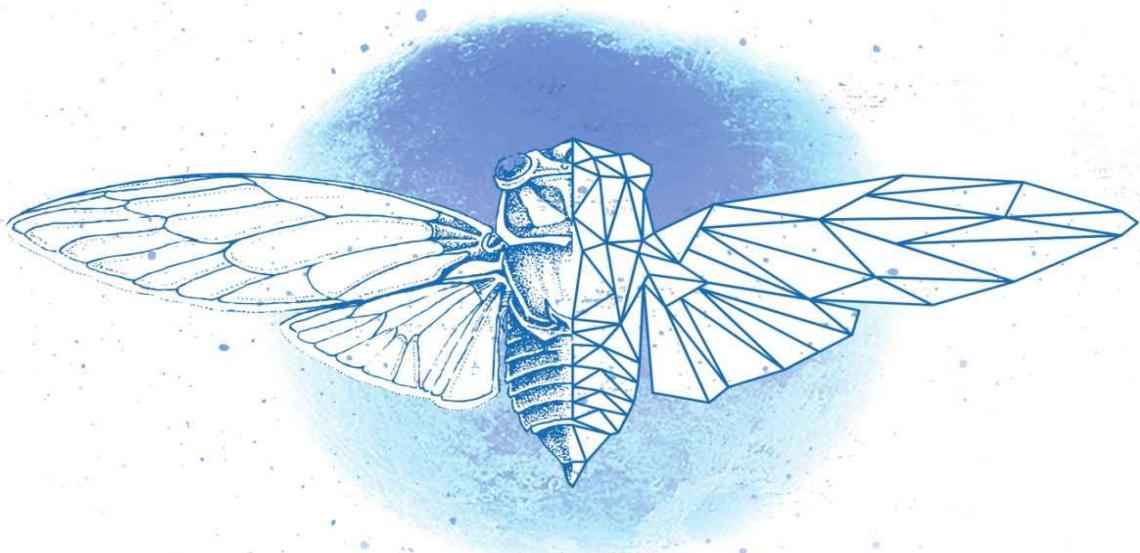




知了传课

ZHILIAO E-LEARNING

/ 匠心修炼 / 倾力打造 /



# Flask Web 全栈开发实战

黄 勇 ◎著

从Flask基础到Flask进阶，  
再到企业级论坛项目实战以及WebSocket在线聊天系统实战，  
囊括真实工作场景中绝大部分技术要点。

清华大学出版社

# Flask Web 全栈开发实战

黄 勇 著

清华大学出版社  
北京

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。举报：**010-62782989, beiqinquan@tup.tsinghua.edu.cn**。

图书在版编目（CIP）数据

Flask Web全栈开发实战 / 黄勇著. —北京：清华大学出版社，2022.6

ISBN 978-7-302-60928-5

I . ①F... II . ①黄... III. ①软件工具—程序设计 IV. ①TP311.561

中国版本图书馆CIP数据核字（2022）第088958号中国版本图书馆

责任编辑：贾旭龙 贾小红

封面设计：秦丽

版式设计：文森时代

责任校对：马军令

责任印制：丛怀宇

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦A座

邮 编：100084

社 总 机：010-83470000

邮 购：010-62786544

投稿与读者服务：010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质量反馈：010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 装 者：三河市东方印刷有限公司

经 销：全国新华书店

开 本：185mm×235mm

印 张：17.75

字 数：356千字

版 次：2022年7月第1版

印 次：2022年7月第1次印刷

定 价：79.00元

---

---

产品编号：094328-01

## 内容简介

本书围绕Flask框架，详细地讲解了使用Flask开发网站的各项技术要点。全书共11章，首先讲解了Flask项目开发中的环境搭建、项目配置、URL与视图、Jinja2模板、数据库、表单、Flask进阶、缓存系统等。然后拓展了知识面，在项目实战中分别介绍了RESTful API、邮箱验证码、Redis缓存、Celery异步任务、登录授权机制、角色权限管理、富文本编辑器、头像管理、文件上传以及Nginx、Gunicorn部署等技术要点；在WebSocket实战中讲解了Flask中使用WebSocket开发项目的全部过程。最后讲解了Flask异步编程。通过本书的学习，读者能够熟练掌握Flask Web开发技术。

本书适合没有Flask开发经验或者Flask基础比较薄弱、想要系统学习Flask Web开发技术的读者学习。

## 作者简介



黄勇

“知了传课”创始人，专注Python开发和培训工作。自2016年从事Python培训以来，全网学习人数超过20万，陆续发布过“Python框架Flask——全开发”“超详细讲解Django——打造大型企业官网”“Python数据分析入门到项目实战”“零基础：21天搞定Python分布式爬虫”等课程，获得一致好评。目前是网易云课堂、CSDN等在线教育平台签约讲师。

## 序 Foreword

最近几年，很多新的技术在企业已全面落地，让我们真真切切地感受到了“技术改变生活”，而且这将会越来越成为一种共识，“未来已来”，它从侧面告诉我们，科学技术重构的背后，本质上其实是社会整体往数字化转型的过程。互联网技术会重构所有的产业体系，不管我们是否承认，这一切都在悄然地进行中，而在过程中，我们每个人都无法置身事外，我们需要思考的是，如何让自己在这样一个充满不确定性的变化中去寻找到一份确定，以成就我们自己的事业和人生。我们需要让自己更轻、更灵活地应对在社会产业结构调整的过程中新型技术带来的挑战。在新的思维体系下，研发团队如何能够快速地应对市场变化和客户的诉求，支撑公司业务爆发式的增长并让自己服务的企业能够在市场竞争中赢得未来，这是我们重点考虑的问题。如此，我们需要轻量级的框架提升研发效率，而Flask正是一款轻量级的Web开发框架，其“微”小的设计理念，可以让我们轻松地使用它进行Web开发，一切都恰到好处。

我很庆幸能够认识黄勇老师，最初认识黄勇老师是通过网易云课堂平台，后来我们有过很多次的沟通交流以及课程方面的合作。黄勇老师是网易云课堂平台的优秀讲师，很多已工作的学生通过学习他的Django和Flask等课程，已经成功就职于一线的互联网公司；很多在校大学生也是通过学习他的课程在未毕业的时候就得到了Offer。在我看来，黄老师的课程不仅教会了别人知识，更多的是教会了别人科学体系化的学习方法，这点是难能可贵的。本书是黄老师在企业内训，以及网易云课堂几万学员的认可中打造出来的Flask Web开发精华，这些知识有利于或帮助学生系统全面地学习Flask框架下Web开发、前后端分离开发模式及微服务架构开发。本书结合了真实的企业案例，带领读者从Flask基础知识开始学习，随后一步步构建以Flask框架为核心的平台化产品和前后端分离模式下的微服务化产品。本书内容深入浅出，带领读者贯穿了整个Flask框架的开发流程，帮助读者从零到一开始构建一个符合企业级标准的平台。不管您是Python全栈开发者，还是Python语言爱好者，以及测试开发工程师，都能够从本书中收获良多。

最后，感谢黄勇老师的辛苦付出，也期待黄勇老师后期可以出版更多关于Python方面的书籍，希望本书能够帮助到更多Python全栈开发者和测试开发工程师。

——《Python自动化测试实战》作者 无涯

# 前言 Preface

## 创作背景

当前，Python的就业前景还是非常可观的，国内Python人才的需求呈大规模上升之势，薪资水平也是水涨船高。尤其在Linux运维、Python Web网站、Python自动化测试、数据分析和人工智能等诸多领域，对Python人才的需求非常旺盛。

Flask诞生于2010年，是作者Armin Ronacher用Python语言编写的一款轻量级Web开发框架。时至今日，使用Flask开发Web应用程序的人越来越多，Flask微框架也越来越受到关注。2021年5月，Flask 2.0版本发布，除了一些新增的特性，Flask 2.0实现了基本的异步支持。

使用Flask框架的优势：可以大大降低开发难度，提高开发效率，让快速、高效的Web开发成为可能；可以带来系统稳定性和可扩展性的提升；Flask自由、灵活、可扩展性强、第三方库的选择面广；对于初学者来说，入门门槛很低，简单易学，即便没有多少Web开发经验，也能很快做出网站，大大节约了初学者的学习成本。

本书围绕Flask框架展开讲解，从理论到实战，带领读者实现从零基础入门到动手开发项目的技术飞跃。书中贯穿了笔者总结的大量开发经验与实践思考，对开发人员有很大的借鉴意义。

## 目标读者

本书的目标读者是没有Flask开发经验或者有少量Flask使用经验的读者。通过学习本书可以熟练掌握Flask Web开发技术，包括但不限于以下岗位。

- Python全栈开发工程师：通过学习本书，可以掌握前后端开发的技术要点，能快速开发Web应用项目。
- 测试开发工程师：使用本书讲解的知识点，能有效提高自动化测试平台的开发能力。
- 运维开发工程师：使用本书讲解的知识点，能有效提高自动化运维平台的开发能力，以及阅读相关开源项目源码的能力。
- 数据 / 算法工程师：使用本书讲解的知识点，可以结合算法模型，将模型服务化，供普通用户使用。

## 内容提要

本书的内容由浅入深，从独立知识点的详细讲解，到项目实战的步步剖析，全面而具体。前面8章讲解了Flask的基础知识，第9章和第10章分别讲解了论坛项目和在线即时聊天项目实战，第11章则作为补充内容，讲解了Flask异步编程。下面分别介绍每章的知识点。

- 第1章：对Flask做了简要介绍，以及讲解了开发Flask项目的环境搭建，后续章节内容都是基于此章搭建的开发环境来讲解的。
- 第2章：详细讲解如何配置Flask项目，以及不同软件的配置方式。
- 第3章：对网站开发中最基本的URL与视图的绑定、URL传参、请求方法、页面重定向等进行详细讲解，学完本章内容读者会明白一个网站是如何与浏览器进行交互的。
- 第4章：主要讲解了Jinja2模板的使用。Jinja2作为Flask默认的模板引擎，有一套自己的渲染语法。Jinja2的功能非常强大，能够直接读取数据库数据，并使用函数对数据进行操作，学好Jinja2模板才能做出一个优美且实用的页面。
- 第5章：数据库是一个动态网站必备的模块。本章详细讲解Flask-SQLAlchemy使用ORM操作MySQL数据库的技术要点，实现不用写一行SQL代码就能操作数据库的需求，从而大大提高开发效率。
- 第6章：一个网站中经常需要提交数据到服务器，这时候就需要用到表单。Flask中的表单是传统HTML表单的加强版。本章除了讲解Flask表单的使用方式以外，还加入了作者的一些使用经验。
- 第7章：经过前面6章的学习后，读者基本可以独立使用Flask开发网站了，通过本章进阶内容的学习，可以学会Flask更高级的用法，以及对Flask原理有更深入的理解。
- 第8章：在网站的访问达到一定数量级后，需要使用缓存来提高网站的响应速度，本章将会讲解纯内存型的Memcached缓存系统，以及key-value带有同步机制的Redis缓存系统。
- 第9章：通过前面对Flask知识点的掌握，读者已经有能力开发一个完整的Flask项目了。本章从零开始讲解实现一个论坛项目的开发过程，包括注册、登录、邮箱验证码、头像、发帖、发布评论等功能。
- 第10章：为了适应市场需求，本章将通过项目实战案例介绍WebSocket在Flask中的应用。学完本章内容后，读者可以有能力开发即时聊天软件，或者将WebSocket功能集成到项目中，如客服系统、视频弹幕等。
- 第11章：对Flask异步编程进行了详细的讲解，首先讲解asyncio标准库、aiohttp库、

异步版Flask安装与异步编程性能，然后带领读者实战，即异步实现发送一些HTTP请求。

## 读者服务

示例代码。

学习视频。

读者可以通过扫码访问本书专享资源官网，获取示例代码、学习视频，加入读者群，下载最新学习资源或反馈书中的问题。



## 勘误和支持

由于笔者水平有限，书中难免会有疏漏和不妥之处，恳请广大读者批评指正。

## 致谢

首先感谢清华大学出版社的杜一诗编辑，感谢她这几个月以来对我的支持和鼓励，引导我完成了本书的编写工作。另外感谢所有支持我课程的粉丝和学员，是你们的支持才让我有动力和勇气完成此书。最后感谢我的家人对我的支持和陪伴，本书也是我送给女儿的出生礼物，希望她长大后有机会阅读到本书。

黄 勇

2021年10月于长沙

# 目 录

[内容简介](#)

[作者简介](#)

[序Foreword](#)

[前言Preface](#)

[chapter 1 第1章 Flask前奏](#)

[1.1 Flask简介](#)

[1.2 环境搭建](#)

[1.2.1 Python环境](#)

[1.2.2 Flask版本](#)

[1.2.3 开发软件](#)

[chapter 2 第2章 项目配置](#)

[2.1 Debug模式、Host、Port配置](#)

[2.1.1 Debug模式](#)

[2.1.2 设置Host和Port](#)

[2.2 在app.config中添加配置](#)

[2.2.1 使用app.config配置](#)

[2.2.2 使用Python配置文件](#)

[chapter 3 第3章 URL与视图](#)

[3.1 定义URL](#)

[3.1.1 定义无参数的URL](#)

[3.1.2 定义有参数的URL](#)

[3.2 HTTP请求方法](#)

[3.3 页面重定向](#)

[3.4 构造URL](#)

[chapter 4 第4章 Jinja2模板](#)

[4.1 模板的基本使用](#)

[4.1.1 渲染模板](#)

[4.1.2 渲染变量](#)

[4.2 过滤器和测试器](#)

[4.2.1 自定义过滤器](#)

[4.2.2 Jinja2内置过滤器](#)

[4.2.3 测试器](#)

[4.3 控制语句](#)

[4.3.1 if判断语句](#)

[4.3.2 for循环语句](#)

## 4.4 模板结构

4.4.1 宏和import语句

4.4.2 模板继承

4.4.3 引入模板

## 4.5 模板环境

4.5.1 模板上下文

4.5.2 全局函数

4.5.3 Flask模板环境

## 4.6 其他

4.6.1 转义

4.6.2 加载静态文件

4.6.3 闪现消息

# chapter 5 第5章 数据库

## 5.1 准备工作

5.1.1 MySQL软件

5.1.2 Python操作MySQL驱动

5.1.3 Flask-SQLAlchemy

## 5.2 Flask-SQLAlchemy的基本使用

5.2.1 连接MySQL

5.2.2 ORM模型

5.2.3 CRUD操作

## 5.3 表关系

5.3.1 外键

5.3.2 一对多关系

5.3.3 一对一关系

5.3.4 多对多关系

5.3.5 级联操作

## 5.4 ORM模型迁移

5.4.1 创建迁移对象

5.4.2 初始化迁移环境

5.4.3 生成迁移脚本

5.4.4 执行迁移脚本

# chapter 6 第6章 表单

## 6.1 表单验证

6.1.1 表单类编写

6.1.2 视图函数中使用表单

6.1.3 自定义验证字段

## 6.2 渲染表单模板

### 6.3 CSRF攻击

## chapter 7 第7章 Flask进阶

### 7.1 类视图

7.1.1 基本使用

7.1.2 方法限制

7.1.3 基于方法的类视图

7.1.4 添加装饰器

### 7.2 蓝图

7.2.1 基本使用

7.2.2 寻找模板

7.2.3 寻找静态文件

### 7.3 cookie和session

7.3.1 关于cookie和session的介绍

7.3.2 Flask中使用cookie和session

### 7.4 request对象

### 7.5 Flask信号机制

7.5.1 自定义信号

7.5.2 Flask内置信号

### 7.6 常用钩子函数

### 7.7 上下文

7.7.1 线程隔离对象

7.7.2 LocalStack类

7.7.3 LocalProxy类

## chapter 8 第8章 缓存系统

### 8.1 Memcached

8.1.1 安装Memcached

8.1.2 telnet操作Memcached

8.1.3 Python操作Memcached

8.1.4 Memcached的安全性

### 8.2 Redis

8.2.1 Redis使用场景

8.2.2 Redis和Memcached比较

8.2.3 Redis在Ubuntu中的安装与使用

8.2.4 Redis操作命令

8.2.5 同步数据到硬盘

8.2.6 设置密码

8.2.7 Python操作Redis

## chapter 9 第9章 项目实战

## 9.1 创建项目

[9.1.1 config.py文件](#)

[9.1.2 exts.py文件](#)

[9.1.3 blueprints模块](#)

[9.1.4 models模块](#)

## 9.2 创建用户相关模型

[9.2.1 创建权限和角色模型](#)

[9.2.2 创建权限和角色](#)

[9.2.3 创建用户模型](#)

[9.2.4 创建测试用户](#)

[9.2.5 创建管理员](#)

## 9.3 注册

[9.3.1 渲染注册模板](#)

[9.3.2 使用Flask-Mail发送邮箱验证码](#)

[9.3.3 使用Flask-Caching和Redis缓存验证码](#)

[9.3.4 使用Celery发送邮件](#)

[9.3.5 RESTful API](#)

[9.3.6 CSRF保护](#)

[9.3.7 使用AJAX获取邮箱验证码](#)

[9.3.8 实现注册功能](#)

## 9.4 登录

## 9.5 发布帖子

[9.5.1 添加帖子相关模型](#)

[9.5.2 初始化板块数据](#)

[9.5.3 渲染发布帖子模板](#)

[9.5.4 使用wangEditor富文本编辑器](#)

[9.5.5 未登录限制](#)

[9.5.6 服务端实现发帖功能](#)

[9.5.7 使用AJAX发布帖子](#)

## 9.6 首页

[9.6.1 生成帖子测试数据](#)

[9.6.2 使用Flask-Paginate实现分页](#)

[9.6.3 过滤帖子](#)

## 9.7 帖子详情

[9.7.1 动态加载帖子详情数据](#)

[9.7.2 发布评论](#)

## 9.8 个人中心

[9.8.1 使用Flask-Avatars生成随机头像](#)

[9.8.2 修改导航条上的登录状态](#)

[9.8.3 根据用户显示个人中心](#)

[9.8.4 修改用户信息](#)

## [9.9 CMS管理系统](#)

[9.9.1 CMS入口](#)

[9.9.2 权限管理](#)

[9.9.3 员工管理页面](#)

[9.9.4 添加员工](#)

[9.9.5 编辑员工](#)

[9.9.6 管理前台用户](#)

[9.9.7 帖子管理](#)

[9.9.8 评论管理](#)

[9.9.9 板块管理](#)

## [9.10 错误处理](#)

## [9.11 日志](#)

[9.11.1 loggers模块](#)

[9.11.2 handlers模块](#)

[9.11.3 filters模块](#)

[9.11.4 formatters模块](#)

## [9.12 部署](#)

[9.12.1 导出依赖包](#)

[9.12.2 使用Git上传代码](#)

[9.12.3 生产环境的配置](#)

[9.12.4 安装常用软件](#)

[9.12.5 配置网站](#)

[9.12.6 使用Gunicorn部署网站](#)

[9.12.7 使用Nginx部署网站](#)

[9.12.8 压力测试](#)

# [chapter 10 第10章 WebSocket实战](#)

[10.1 安装相应的包](#)

[10.2 创建SocketIO对象](#)

[10.3 实现登录](#)

[10.4 连接和取消连接](#)

[10.5 获取在线用户](#)

[10.6 实现单聊](#)

[10.7 实现群聊](#)

[10.8 部署项目](#)

# [chapter 11 第11章 Flask异步编程](#)

[11.1 asyncio标准库](#)

[11.2 aiohttp库](#)

[11.3 异步版Flask安装与异步编程性能](#)

[11.3.1 安装异步版Flask](#)

[11.3.2 Flask异步编程性能](#)

[11.3.3 实战——异步发送HTTP请求](#)

[11.3.4 使用异步SQLAlchemy](#)

[11.3.5 Jinja2开启异步支持](#)

# chapter 1 第1章 Flask前奏

## 1.1 Flask简介

Flask是一个基于Python的Web开发框架，它以灵活、微框架著称。Flask的出现也是一个偶然的机会，在2010年4月1日愚人节这天，作者Armin Ronacher开了个玩笑，在网上发表了一篇关于“下一代Python微框架”的文章，众开发者信以为真，并期待他能真正把文章中的想法实现出来。5天后，Armin Ronacher真的发布了一个“微”框架，就是Flask。Flask虽然是作者在愚人节开的一个玩笑，但是其框架设计却非常优秀，并且深受开发者喜爱，截至2021年6月，在Github上的Star（关注）数已经超过56000，仅次于2005年发布的Django的58000 Star数。相信在不久的将来，Flask的Star数一定会赶超Django。

Flask以微框架著称，本身不具备太多功能，但是通过丰富的第三方插件，可以轻松地应对现实开发中复杂的需求，并且有大量的企业正在使用Flask构建自己的产品。国内比较出名的如豆瓣、果壳网，国外的如Reddit、Netflix等，其稳定性和应对复杂业务需求的能力已经被大量企业所验证。因此读者无须担心Flask无法适应企业需求，放心大胆地去学好Flask，能够让你在Web开发工作中如虎添翼。

## 1.2 环境搭建

为了避免因为开发环境问题影响读者的学习，在阅读本书之前，我们先来了解本书知识点和案例所基于的开发环境，建议读者在阅读学习本书时，尽量保持跟本书一致的开发环境。

### 1.2.1 Python环境

本书使用的Python版本是Python 3.9，如果读者之前已经安装过其他版本，则必须保证是Python 3.6以上的版本。关于如何查看计算机中现有的Python版本，按照系统来分，可以用以下方式查看。

- Windows系统：按Win+R快捷键，输入cmd，按Enter键，在打开的命令行终端输入python，即可看到现有的Python版本，如图1-1所示。

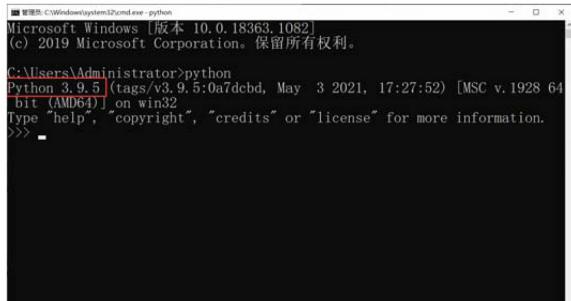


图1-1 Windows系统查看Python版本

从图1-1可以看到Windows系统安装的Python版本是3.9.5，读者也可以查看自己的计算机上安装的Python版本，如果不是3.6以上版本，那么可以到官网<https://www.python.org/>下载最新版本的Python，下载后直接安装即可。

- Mac系统：打开终端，输入python3，然后按Enter键，可以看到Mac系统安装的也是Python 3.9.5版本，如图1-2所示。

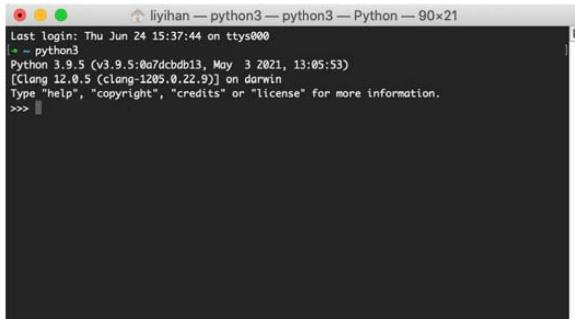


图1-2 Mac系统查看Python版本

### 注意

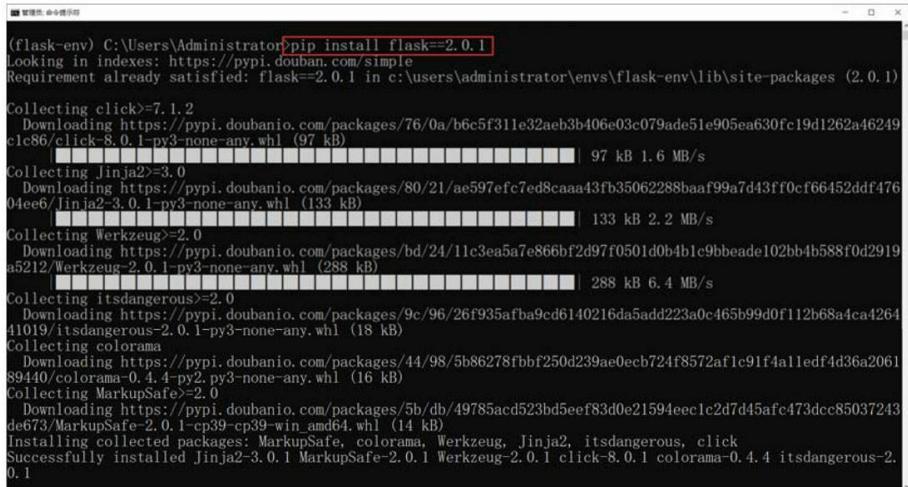
因为笔者的Mac系统装了两个Python版本，设置的python3命令指向的是Python 3.9，所以在此输入的是python3命令。

## 1.2.2 Flask版本

本书讲解的知识点和项目都是基于目前最新的Flask版本：2.0.1。Flask 2.0.1新增了许多新的特性，如增加了await/async异步支持、@post/@get快捷路由、嵌套蓝图等。如果使用旧版本的Flask，这些新的特性将无法学到。安装Flask 2.0.1版也非常简单，只要在系统的终端软件中输入以下命令，然后按Enter键即可安装。

```
$ pip install flask==2.0.1
```

安装效果如图1-3所示。



```
(flask-env) C:\Users\Administrator>pip install flask==2.0.1
Looking in indexes: https://pypi.douban.com/simple
Requirement already satisfied: flask==2.0.1 in c:\users\administrator\envs\flask-env\lib\site-packages (2.0.1)

Collecting click>=7.1.2
  Downloading https://pypi.douban.io/packages/76/0a/b6c5f311e32ae3b406e03c079ade51e905ea630fc19d1262a46249
click86/click-8.0.1-py3-none-any.whl (97 kB)
[██████████] 97 kB 1.6 MB/s
Collecting Jinja2>=3.0
  Downloading https://pypi.douban.io/packages/80/21/ae597efc7ed8caaa43fb35062288baaf99a7d43ff0cf66452ddf476
04ee6/Jinja2-3.0.1-py3-none-any.whl (133 kB)
[██████████] 133 kB 2.2 MB/s
Collecting Werkzeug>=2.0
  Downloading https://pypi.douban.io/packages/bd/24/11c3ea5a7e866bf2d97f0501d0b4b1c9bbeade102bb4b588f0d2919
a5212/Werkzeug-2.0.1-py3-none-any.whl (288 kB)
[██████████] 288 kB 6.4 MB/s
Collecting itsdangerous>=2.0
  Downloading https://pypi.douban.io/packages/9c/96/26f935afba9cd6140216da5add223a0c465b99d0f112b68a4ca4264
41019/itsdangerous-2.0.1-py3-none-any.whl (18 kB)
Collecting colorama
  Downloading https://pypi.douban.io/packages/44/98/5b86278fbff250d239ae0ecb724f8572af1c91f4a1edf4d36a2061
89440/colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Downloading https://pypi.douban.io/packages/5b/db/49785acd523bd5eef83d0e21594eec1c2d7d45afc473dcc85037243
de673/MarkupSafe-2.0.1-cp39-cp39-win_amd64.whl (14 kB)
Installing collected packages: MarkupSafe, colorama, Werkzeug, Jinja2, itsdangerous, click
Successfully installed Jinja2-3.0.1 MarkupSafe-2.0.1 Werkzeug-2.0.1 click-8.0.1 colorama-0.4.4 itsdangerous-2.0.1
```

图1-3 通过pip命令安装Flask

Flask还有许多第三方的插件，如提供数据库操作的Flask-SQLAlchemy，后续在讲到相关内容时再安装。

### 1.2.3 开发软件

许多软件都可以用来开发Flask项目，如Sublime Text、Visual Studio Code等，但是最专业的软件还是PyCharm。PyCharm是一个集成开发环境（integrated development environment，简称IDE），它提供了许多方便快捷的功能，如断点调试、版本控制等，对于企业级Python开发者而言，无疑是很好用的开发软件。

PyCharm是JetBrains公司出品的一款专门针对Python编程的软件，它有两大版本：一个

是PyCharm Professional，即专业版；另一个是PyCharm Community，即社区版，这两大版本的主要区别如下。

- PyCharm Professional：功能最全，适合开发任何类型的Python程序，包括做一些前端项目开发，但是需要收费。
- PyCharm Community：适合开发爬虫、数据分析、GUI等纯Python程序。对Python Web（如Flask和Django等）开发不够友好，没有足够的代码提示。好处是开源免费。

我们需要开发Flask项目，所以选择PyCharm Professional版本。关于它的收费问题，如果读者是学生，可以用学校提供的教育邮箱账号（一般以edu.cn结尾）去申请免费授权（申请网址<https://www.jetbrains.com/community/education/#students>）。如果读者是企业开发者，可以跟公司申请购买正版授权。如果您既不是学生又不想购买正版PyCharm Professional版本，则可以退而求其次选择PyCharm Community版本，也完全可以学习本书的内容，只是一些代码提示没有那么智能（PyCharm Professional有30天试用期）。

下面详细地讲解PyCharm的安装步骤及其使用方法。

#### （1）下载PyCharm。

首先到JetBrains官网<https://www.jetbrains.com/pycharm/download/>下载PyCharm，根据自己的情况，选择Professional版本或Community版本，如图1-4所示，然后单击Download按钮即可。

#### （2）安装PyCharm。

下载PyCharm后，双击pycharm-professional.exe文件即可打开安装界面。安装过程非常简单，全部使用默认选项，一直单击Next按钮即可。唯一需要注意的是，在安装过程中可以选择安装路径，如图1-5所示。

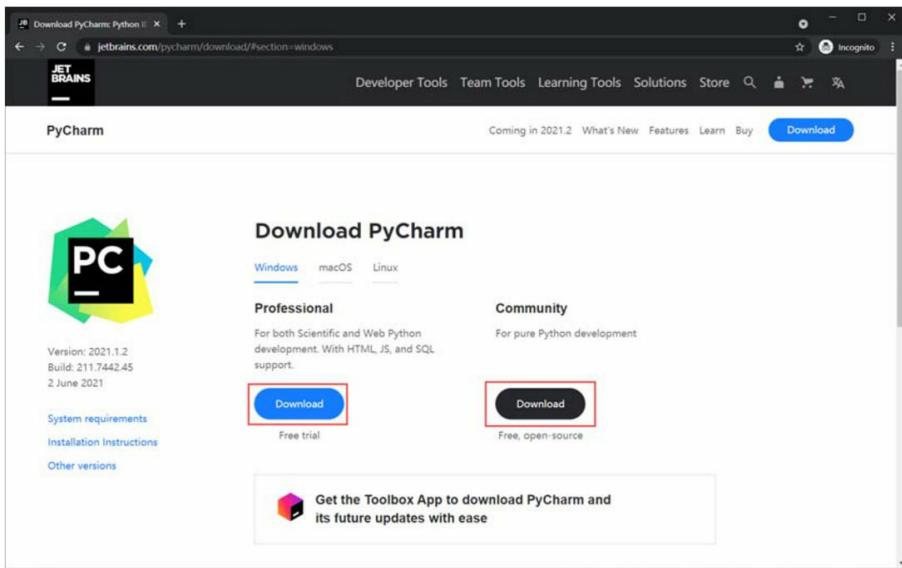


图1-4 下载PyCharm

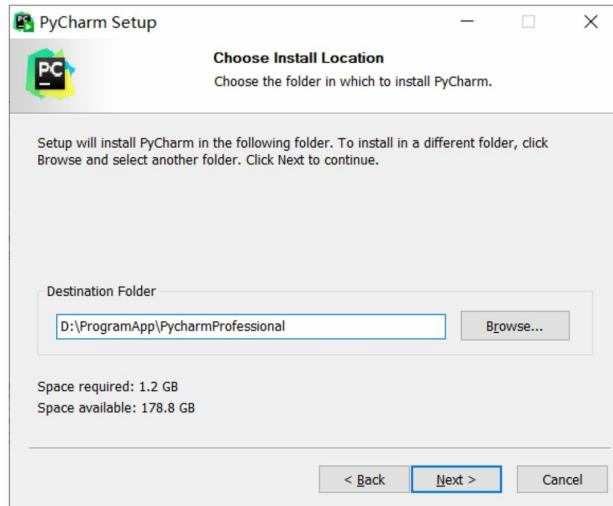


图1-5 安装PyCharm时可选择安装路径

(3) 创建项目，选择Python解释器。

打开PyCharm，然后单击New Project按钮创建一个项目，如图1-6所示。

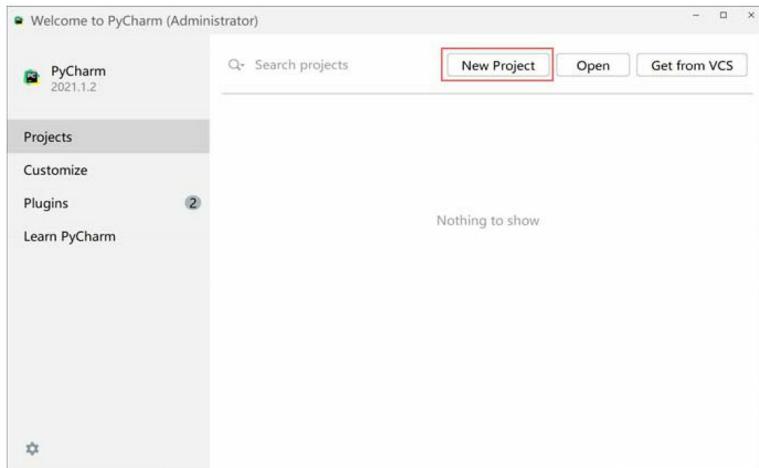


图1-6 创建新项目

再次单击New Project按钮后，进入下一个界面，在左侧选择Flask项目，然后设置项目的路径，接着设置Python解释器，如图1-7所示。

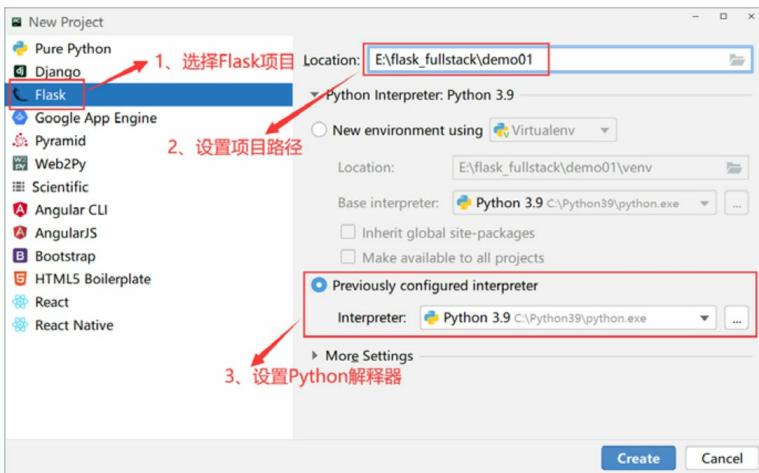


图1-7 使用PyCharm创建项目的选项

选择Flask项目以及设置项目路径的步骤都比较简单，最重要的是Python解释器的设置。系统默认选择New environment using [Virtualenv]选项，这个选项会为每个项目都创建一个虚拟环境，虚拟环境相当于一个独立的Python环境，之前通过pip命令安装的Python全都需要重新安装，对于我们学习而言，无疑是浪费时间，所以这里要选中Previously configured interpreter单选按钮。

### 注意

如果项目不是用来学习的，而是要上线到服务器使用的，则建议选择New environment using [Virtualenv]选项，可以避免和其他项目产生依赖包版本冲突，也方便在开发机和服务器上同步依赖包。

解释器选好后，单击Create按钮创建项目。

项目创建后，PyCharm默认会生成以下项目结构。

- app.py文件：是项目的入口文件，会默认生成一个主路由，并且视图函数名叫hello\_world，详情如图1-8所示。

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** demo01 - app.py
- Toolbar:** Includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help.
- Project Tool Window:** Shows the project structure with "demo01" selected. It also lists "External Libraries" and "Scratches and Consoles".
- Code Editor:** Displays the content of app.py:

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

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```
- Status Bar:** Shows file type recognized: File extension \*.xls' was reassig... (a minute ago), 8:26, CRLF, UTF-8, 4 spaces, Python 3.9, and other standard status bar information.

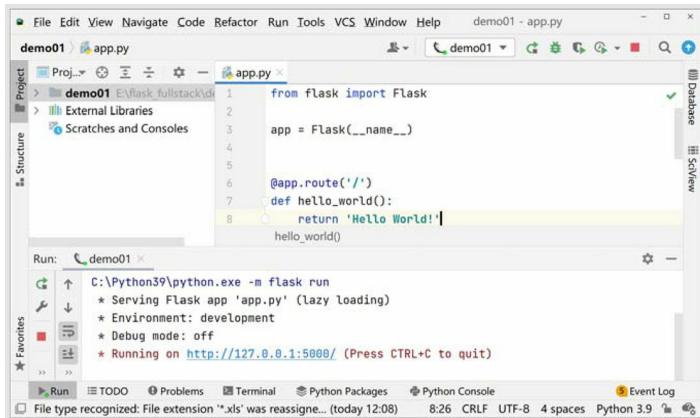
图1-8 Flask项目结构图

- templates文件夹：用于存放模板文件。

static文件夹：用于存放静态文件。

## 注意

如果读者用的不是PyCharm Professional版，那么将不会自动生成app.py文件，以及static和templates文件夹，这时读者可以自行创建这个项目结构，并在app.py文件中输入如图1-9所示的代码即可。



The screenshot shows the PyCharm Professional interface with the following details:

- Project Structure:** Shows a project named "demo01" containing an "app.py" file.
- Code Editor:** The "app.py" file contains the following code:

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

@app.route('/')
def hello_world():
    return 'Hello World!'
```
- Run Tab:** Displays the command "C:\Python39\python.exe -m flask run" and its output:

```
* Serving Flask app 'app.py' (lazy loading)
* Environment: development
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```
- Status Bar:** Shows the file type as "Python", encoding as "UTF-8", and Python version as "Python 3.9".

图1-9 app.py文件

最后，单击右上角的三角按钮运行项目。在浏览器中输入http://127.0.0.1:5000，可以看到浏览器网页中显示Hello World!（见图1-10）。至此，一个最简单的Flask项目就已经运行起来了。

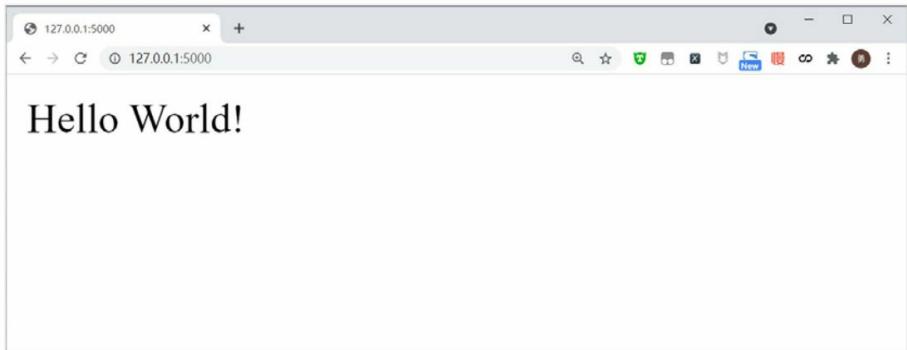


图1-10 浏览器中访问Flask项目

## chapter 2 第2章 项目配置

Flask项目在开发中，或部署到服务器上，都需要做一些配置。如是否开启Debug模式、连接数据库参数信息等操作都需要通过配置才能实现，本章将详细地讲解Flask项目中的几种配置方式，读者可以根据实际情况选择不同的配置方式。

### 2.1 Debug模式、Host、Port配置

Debug模式、Host、Port这3个配置项分别代表是否开启调试模式、项目运行使用的Host（可以先简单理解为访问项目的域名）、项目运行监听的端口号。这3个配置项单独拿出来讲，是因为它们在项目开发中使用的频率非常高，并且使用的是不同的开发工具，所以配置方式也不同。为了讲解方便，这里首先创建一个新的项目demo02，读者可以自行下载相关代码学习。下面分别学习这3个配置项的意义和配置方式。

#### 2.1.1 Debug模式

在使用Flask框架开发项目的过程中，会不断地添加新代码或者修改原代码，如果没有开启Debug模式，那么在修改代码后，必须要手动重新启动项目才能看到运行效果，这样会大大降低开发效率。所以一般在开发时，都会开启Debug模式，这样在代码修改完成后，只要单击“保存”按钮，或者按Ctrl+S快捷键，那么Flask将会自动重启项目。另外，如果程序出错了，在开启Debug模式下，在浏览器端会显示错误信息，并且标记错误行号，对于定位bug（故障）有非常大的帮助。那么Debug模式怎么打开呢？这要根据是否使用PyCharm Professional版来决定，以下分别进行讲解。

##### 1. 在PyCharm Professional版中开启Debug模式

如果你使用的是PyCharm Professional版，则需要单击右上角demo02（即项目名称）右侧下拉按钮，然后在弹出的下拉列表中选择Edit Configurations命令，如图2-1所示。

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

@app.route('/')
def hello_world():
    return 'Hello World!'

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

图2-1 编辑项目配置

在打开的本项目的编辑界面选中FLASK\_DEBUG复选框，然后单击OK按钮即可，如图2-2所示。

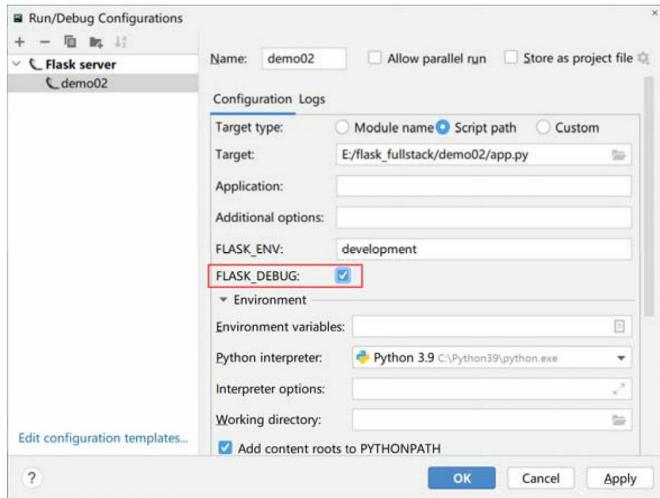


图2-2 选中FLASK\_DEBUG复选框

下面单击右上角的运行按钮运行项目，便可以看到在PyCharm控制台显示已经开启了Debug模式，如图2-3所示。

```
6     @app.route('/')
7     def hello_world():
8         return 'Hello World!'
9
10    if __name__ == '__main__':
11        hello_world()
```

Run: demo02

- \* Environment: development
- \* Debug mode: on 显示已经开启Debug模式
- \* Restarting with stat
- \* Debugger is active!
- \* Debugger PIN: 943-048-352
- \* Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit)

图2-3 开启Debug模式

接下来打开浏览器，在浏览器的地址栏中输入<http://127.0.0.1:5000>，可以看到网页中显示Hello World!。现在返回到PyCharm中，将字符串Hello World!修改成Hello Flask!，然后按Ctrl+S快捷键保存代码，可以看到PyCharm的控制台会自动地重新加载项目，如图2-4所示。

```
6     @app.route('/')
7     def hello_world():
8         return 'Hello Flask!'
```

Run: demo02

- \* Debugger is active!
- \* Debugger PIN: 943-048-352
- \* Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit)

Detected change in 'E:\\flask\_fullstack\\demo02\\app.py', reloading

- \* Restarting with stat
- \* Debugger is active!
- \* Debugger PIN: 943-048-352
- \* Running on <http://127.0.0.1:5000/> (Press CTRL+C to quit)

图2-4 重新加载项目生成的日志信息

在项目重新加载完成后，回到浏览器中，重新访问<http://127.0.0.1:5000>，可以看到网页上的显示信息已经变成了Hello Flask!。在这个过程中我们不需要手动地重启项目，这大大提高了开发效率。

## 2. 在非PyCharm Professional版中开启Debug模式

如果使用的是其他软件编写Flask项目，如PyCharm Community或Virtual Studio Code等，那么需要在app.run方法调用时，添加debug=True参数。在PyCharm Community版中打开项目并添加debug=True参数，如图2-5所示。

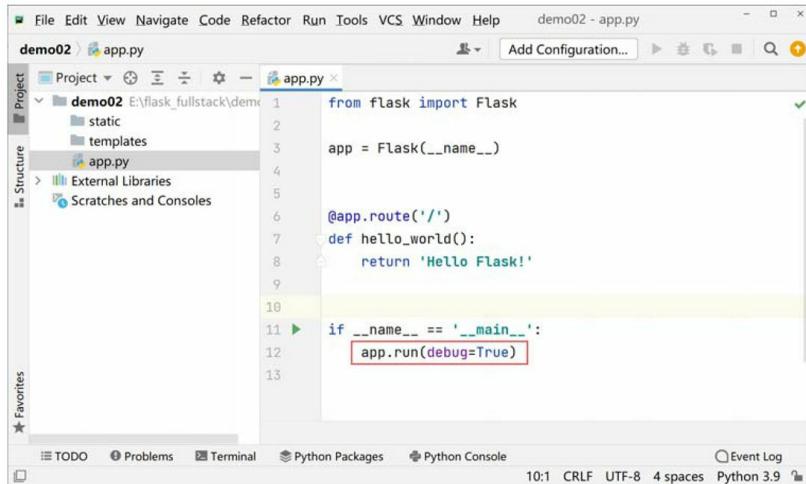


图2-5 在PyCharm Community中开启Debug模式

因为PyCharm Community版中没有集成Flask的运行模式，所以运行Flask项目时需要按照常规的Python程序来执行，也就是在app.py文件的任意空白处右击，然后在弹出的快捷菜单中选择Run 'app'命令，如图2-6所示。

开始运行app.py后，即可在PyCharm Community的控制台看到日志信息，也可以看到Debug模式已经被开启了，如图2-7所示。

以上便是Debug模式的开启方式，读者可以根据实际情况进行设置。Debug模式在开发过程中调试代码、定位bug非常方便，但是在项目部署上线后，记得一定要关闭Debug模式，否则项目一旦出现异常，相关代码就会显示在浏览器上，很容易被有心之人利用，从而威胁网站的安全。

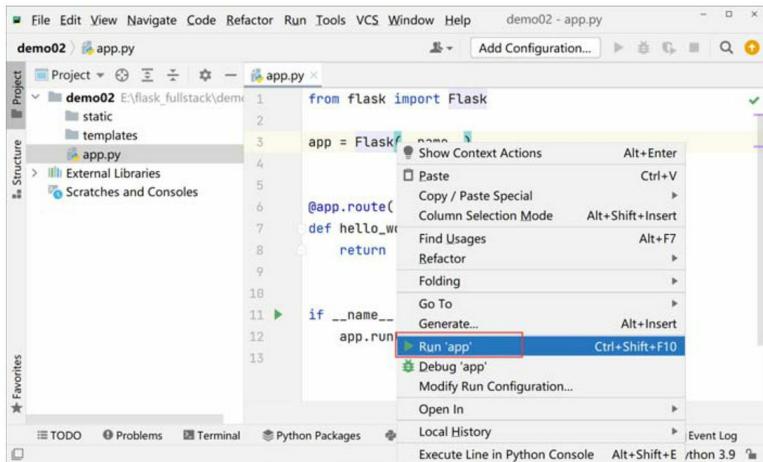


图2-6 在PyCharm Community中运行Flask项目

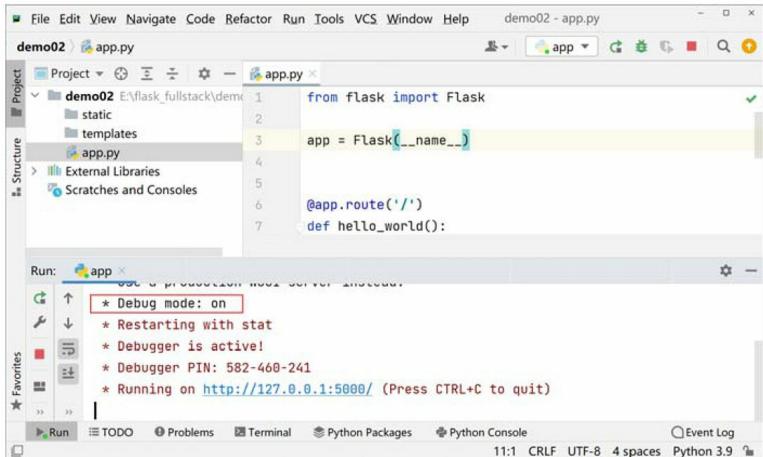


图2-7 PyCharm Community控制台显示开启了Debug模式

## 2.1.2 设置Host和Port

Host代表的是主机，Port代表的是端口号。下面举一个实际的例子来简单解释Host和Port。例如，百度首页网址为`https://www.baidu.com:443`，其中冒号前面的`www.baidu.com`即为Host，冒号后面的`443`即为Port。百度首页网址用的是https协议，因为https协议默认监听

的是443端口，所以在访问百度首页网址时，即使没有写明443端口，浏览器也会自动请求百度服务器的443端口，即通过https://www.baidu.com就可以访问到百度首页。

运行Flask项目后，如图2-7所示，可以看到控制台的打印信息Running on http://127.0.0.1:5000，此时的Host是127.0.0.1，Port是5000。如何修改Host和Port呢？这也要看是否使用的是PyCharm Professional版。下面分别进行讲解。

## 1. PyCharm Professional版修改Host和Port

首先，同修改Debug模式一样，先单击右上角项目名称旁的下拉按钮，然后在弹出的下拉列表中选择Edit Configurations命令，如图2-8所示。

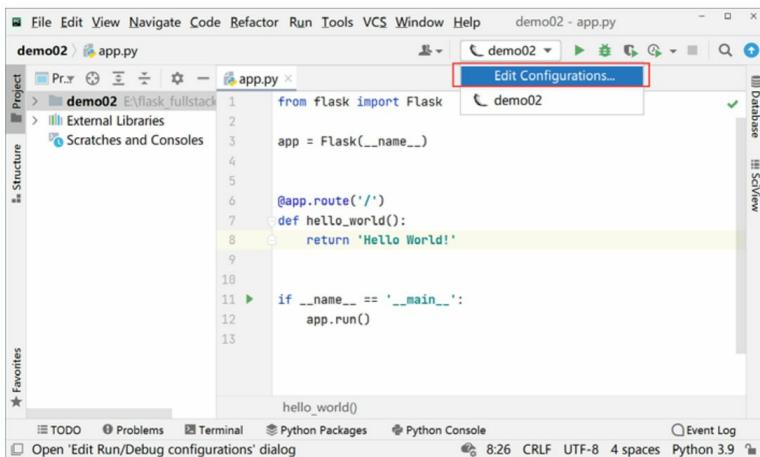


图2-8 PyCharm Professional编辑项目配置

打开编辑窗口后，找到Additional options，如图2-9所示。

首先来修改Port，在Additional options文本框中添加“--port=8000”（port前面两个“-”），然后单击OK按钮，如图2-10所示。回到PyCharm Professional主面板后，再单击运行按钮，即可在PyCharm Professional控制台看到项目监听的端口已经从之前的5000变成了8000，如图2-11所示。

以后我们再访问此项目时，就需要通过http://127.0.0.1:8000来访问了。

读者可能会好奇，在什么情况下需要修改Port呢？假设现在需要运行两个Flask项目A和B，如果不修改端口号，则A和B两个项目监听的都是5000端口，这样会导致其中一个项目不能被访问到。此时我们可以将B项目的端口号修改成8000，以后在浏览器中访问

`http://127.0.0.1:5000`就是A项目，访问`http://127.0.0.1:8000`就是B项目，这样就能非常明确地区分开来。总而言之，在5000端口被占用的情况下，都可以通过修改Port来让项目正常地运行起来。

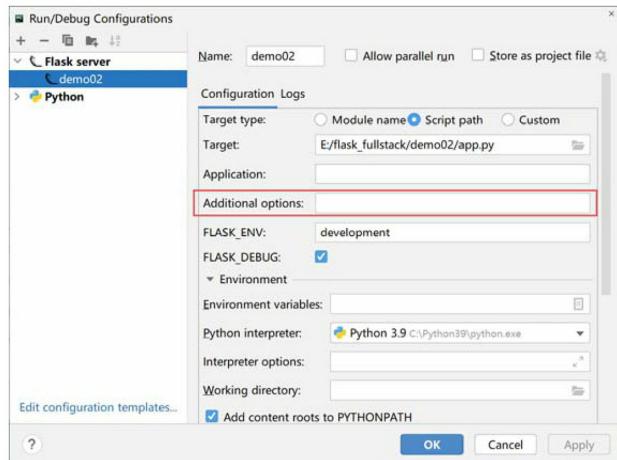


图2-9 添加额外参数

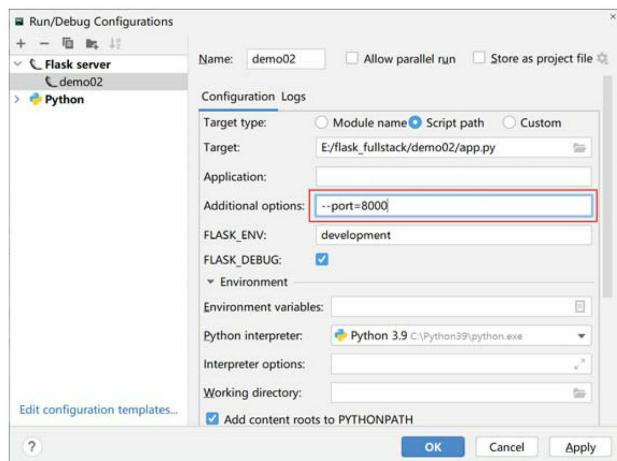


图2-10 设置Port参数

```
demo02 - app.py
def hello_world():
    return 'Hello Flask!'

hello_world()

Run: demo02
C:\Python39\python.exe -m flask run --port=8080
* Serving Flask app 'app.py' (lazy loading)
* Environment: development
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 943-048-352
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
```

图2-11 修改Port后的日志信息

接下来修改Host。修改Host的步骤与修改Port是一样的，在Additional options文本框中添加一个“--host=0.0.0.0”参数即可，如图2-12所示。

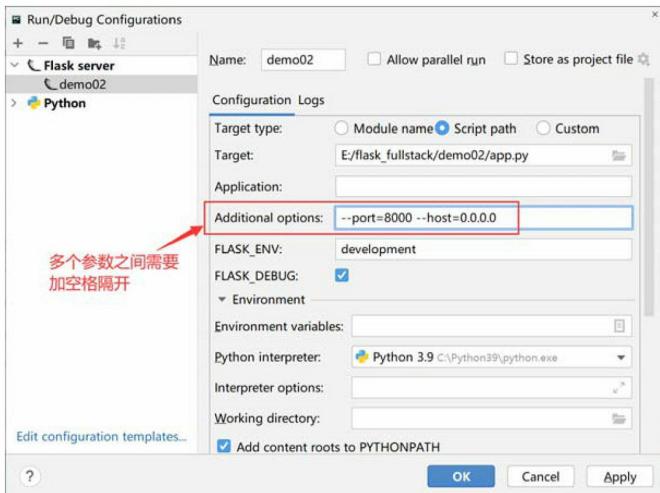


图2-12 添加Host配置

Host不是修改成什么都可以的，必须是以下三种之一。

- 本机的局域网IP地址。IP地址在Windows系统下可以在cmd命令行终端中输入ipconfig命令查看，在Mac或者Linux下系统可以通过ifconfig命令查看。如

查看笔者的局域网IP地址，如图2-13所示。

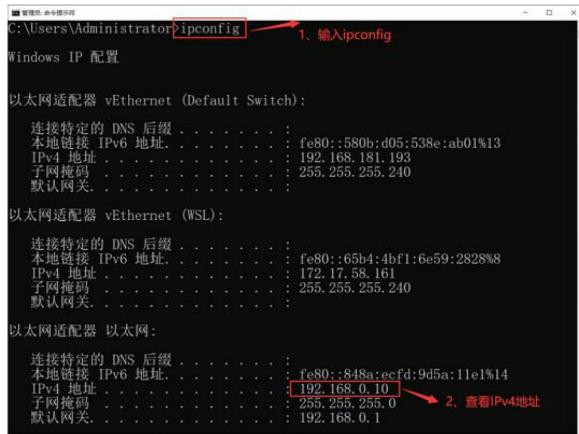


图2-13 查看笔者的局域网IP地址

如果设置成了局域网IP地址，那以后不管是自己的计算机访问，还是局域网中其他设备访问，都需要通过局域网IP地址才能访问到。

- 127.0.0.1：代表本机的IP地址。如果设置成本机的IP地址，则项目只能在自己的计算机上访问。局域网中其他用户不能访问。
- 0.0.0.0：代表既可以通过127.0.0.1访问，也可以通过局域网IP地址访问。

如果在项目中想让局域网中的其他用户访问，一般会把Host设置成0.0.0.0，这样别人能通过运行项目的计算机的局域网IP地址访问到项目，在本机上也可以通过127.0.0.1访问到项目。如图2-12所示设置，读者可以在家中同一个局域网下，用手机打开浏览器，输入http://局域网IP地址:8000，也可以访问到计算机上运行的Flask项目。

## 2. 非PyCharm Professional版修改Host和Port

在没有使用PyCharm Professional的情况下，只需要在app.run方法中传入host和port参数即可，如图2-14所示。

需要注意以下两点。

- port参数必须设置为整型，不能设置为字符串。
- host设置为0.0.0.0后，虽然控制台日志显示的是http://192.168.0.10:8000，但是在本机上也可以通过http://127.0.0.1:8000访问项目，其他设备则可以通过

http://192.168.0.10:8000访问到项目。

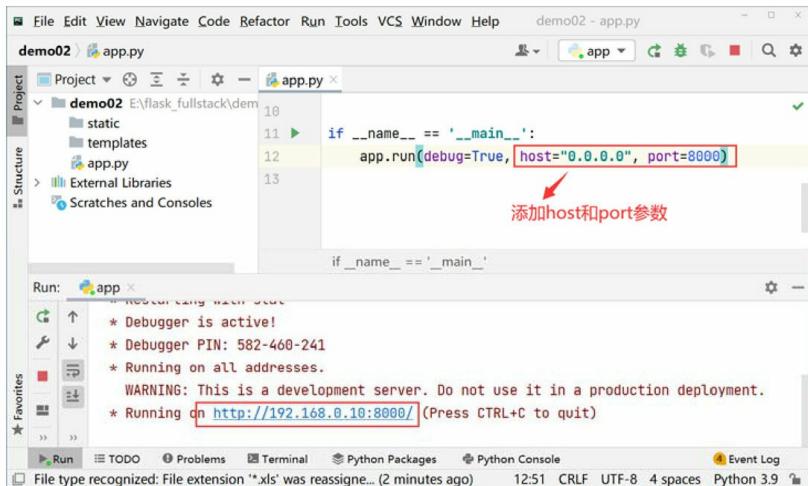


图2-14 非PyCharm Professional下修改host和port

## 2.2 在app.config中添加配置

除了Debug、Host、Port这3个配置项比较特殊外，其他的配置参数都需要配置到Flask对象的app.config属性中，在配置参数较多的情况下，还会放到配置文件中。以下分别来进行讲解。

### 2.2.1 使用app.config配置

app.config是Config的对象，Config是一个继承自字典的子类，所以可以像操作字典一样操作它。使用app.config必须要注意的一点是，所有配置项的名称都必须大写，否则不会被app.config读取到，示例代码如下。

```
app = Flask(__name__)
app.config["SECRET_KEY"] = "skhrek349Lx!@# "
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///test.db"
# 下面的配置项为小写，不会被读取
app.config["test"] = True
```

app.config中的配置项，可以设置Flask及其插件内置的一些配置项，也可以添加自定义的配置项。如果后续开发中需要用到app.config中提前配置好的选项，那么可以通过类似字典的方式获取，示例代码如下。

```
app = Flask(__name__)
app.config["TESTING"] = True
...
test = app.config["TESTING"]
```

使用app.config的方式配置项目，在项目体量较小的情况下比较方便，但是随着项目开发的复杂度越来越高，配置项也越来越多，使用app.config配置的方式就显得代码不够优雅，并且会让app.py文件越来越臃肿。因此企业开发中的项目都会使用配置文件。接下来讲解如何使用配置文件。

## 2.2.2 使用Python配置文件

首先，在当前项目（section01）文件夹下创建一个名为config.py的文件，这个文件专门用来存放配置选项。如在config.py中添加以下代码。

```
# config.py文件
TOKEN_KEY = "123456"
```

然后，在app.py中添加以下代码。

```
# app.py文件
import config
app = Flask(__name__)
app.config.from_object(config)
...
print(app.config["TOKEN_KEY"])
```

运行项目后可以看到控制台会打印123456，这说明使用Python配置文件也可以添加配置项。

app.config from\_object除了直接使用导入的Python模块以外，还可以通过字符串的形式加载，示例代码如下。

```
# app.py文件
app = Flask(__name__)
app.config.from_object("config")
```

Flask还有许多其他的方式来添加配置文件，如`app.config.from_file`和`app.config.from_json`，这里就不再一一展开讲解了，感兴趣的读者可以自行阅读Flask的官方文档<https://flask.palletsprojects.com/en/2.0.x/config>进行学习研究。

## chapter 3 第3章 URL与视图

在使用PyCharm Professional版创建一个Flask项目后，默认会生成app.py文件，文件中的默认代码如下。

```
from flask import Flask
import config

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

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

### 注意

如果读者用的不是PyCharm Professional版，那么可以手动创建app.py文件，并在app.py中手动写入以上代码。

我们把@app.route中的第一个字符串参数叫作URL，把被@app.route装饰的函数叫作视图函数。可以在代码中看到URL与视图函数的映射关系如下。

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

其中，@app.route装饰器中添加了访问URL的规则“/”，“/”代表网站的根路径，只要在浏览器中输入网站的域名即可访问到“/”。被@app.route装饰的hello\_world函数会在浏览器访问“/”时被执行，此时hello\_world函数没有做任何事，只是简单地返回了“Hello World!”字符串。因此在浏览器中访问http://127.0.0.1:5000时，我们就可以看到“Hello World!”，如图3-1所示。



图3-1 浏览器访问http://127.0.0.1:5000

本章首先用PyCharm Professional版创建一个新的项目demo03，后续的知识点讲解都将基于demo03项目，如图3-2所示。

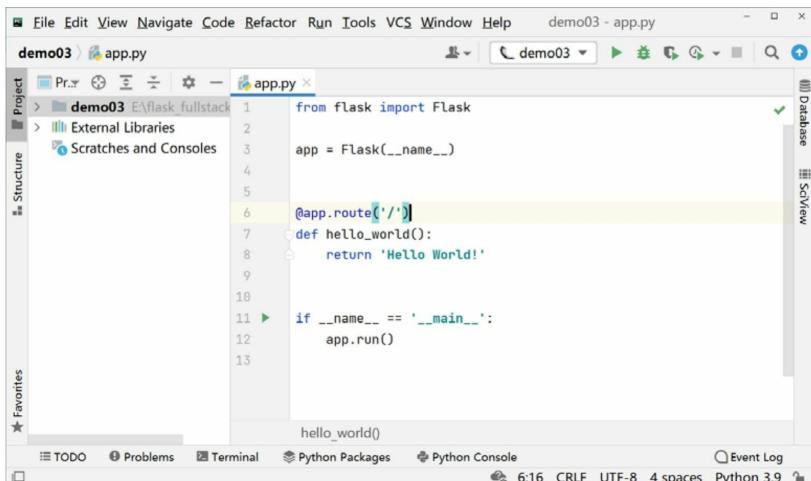


图3-2 demo03项目

### 3.1 定义URL

绝大部分网站都不可能只有首页一个页面，以一个最简单的博客网站为例，博客页面相关的有博客列表、博客详情等，用户页面相关的有注册、登录、个人中心等。所以在制作网站时，需要定义许多不同的URL来满足不同页面的访问需求，而URL总体上来说又分为两种，第一种是无参数的URL，第二种是有参数的URL，下面分别进行讲解这两种URL的定义。

### 3.1.1 定义无参数的URL

无参数URL是指在URL定义的过程中，不需要定义参数。这里以个人中心为例，如定义个人中心的URL为profile，可以使用以下代码实现。

```
@app.route('/profile')
def profile():
    return '这是个人中心'
```

这样在浏览器中访问http://127.0.0.1:5000/profile时，就可以看到“这是个人中心”的页面，如图3-3所示。



图3-3 在浏览器中访问http://127.0.0.1:5000/profile

#### 注意

我们说的访问/profile是不包含域名和端口号的，真正在浏览器中访问时应该在前面加上域名和端口号，如在本地开发应该为http://127.0.0.1:5000/profile，下文说的URL都是省略了域名和端口号的。

### 3.1.2 定义有参数的URL

很多时候，在访问某个URL时需要携带一些参数。如获取博客详情时，需要把博客的id传过去，那么博客详情的URL可能为/blog/13，其中13为博客的id。假如获取第10页的博客列表，那么博客列表的URL可能为/blog/list/10，其中10为页码。

在Flask中，如果URL中携带了参数，那么视图函数也必须定义相应的形参来接收URL中的参数。这里以博客详情的URL为例，示例代码如下。

```
@app.route("/blog/<blog_id>")  
def blog_detail(blog_id):  
    return "您查找的博客id为: "%blog_id
```

通过以上代码可以看到，URL中多了一对尖括号，并且尖括号中多了一个blog\_id，这个blog\_id就是参数。然后在视图函数blog\_detail中，也相应定义了一个blog\_id的形参，当浏览器访问这个URL时，Flask接收到请求后，会自动解析URL中的参数blog\_id，把它传给视图函数blog\_detail，在视图函数中，开发者就可以根据blog\_id从数据库中查找到具体的博客数据返回给浏览器。关于数据库操作，这里不做过多讲解，后续章节会详细讲到。现在在浏览器中输入http://127.0.0.1:5000/blog/1，可以看到如图3-4所示效果。



图3-4 在浏览器中访问http://127.0.0.1:5000/blog/1的效果

URL中的参数还可以指定其类型，指定参数类型有以下两点好处。

- 浏览器访问URL时，如果传的参数不能被转换为指定的参数类型，如定义URL时参数为整型，但是访问的时候传的是不能被转换为整型的参数，如hello，那么这个URL就不会被匹配，从而抛出404错误，保证网站的正常运行。
- URL本质上是一个字符串，如果没有指定参数类型，那么参数传进视图函数时默认也是字符串类型。如果指定了参数类型，那么在传给视图函数之前，会将参数转换为指定类型，这样视图函数拿到的参数就是经过转换后的，从而更加方便使用。

指定参数类型是通过语法: <类型: 参数名>实现的。这里以/blog/<blog\_id>这个URL为例，假如需要指定blog\_id为int类型，那么代码将修改成如下形式。

```
@app.route("/blog/<int:blog_id>")  
def blog_detail(blog_id):  
    return "您查找的博客id为: %s"%blog_id
```

如果在浏览器中访问/blog/hello，将显示Not Found，因为hello不能被转换为int类型，也就不会被匹配到这个URL了，所以会提示URL没有找到，如图3-5所示。

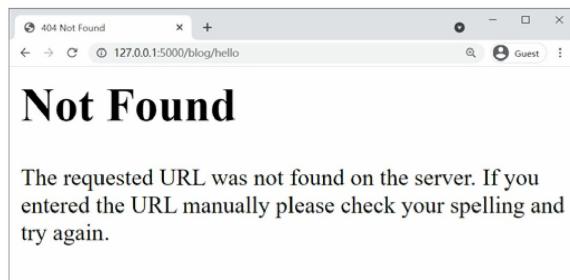


图3-5 访问/blog/hello失败

除了int类型以外，URL中的参数还可以指定其他类型，如表3-1所示。

表3-1 URL中的参数类型

参 数 类 型	描 述
string	字符串类型，可以接收除/以外的字符
int	整型，可以接收能通过int()方法转换的字符
float	浮点类型，可以接收能通过float()方法转换的字符
path	路径，类似string，但是中间可以添加/
uuid	UUID类型，由一组32位的十六进制数所构成
any	any类型，指备选值中的任何一个

表3-1中的参数类型除了any以外，其他的都好理解。这里着重讲解参数类型any的使用。例如，现在要实现一个获取某个分类的博客列表，但是博客分类只能是python、flask、django之一，用any就可以轻松实现。

```
@app.route("/blog/list/<any(python,flask,django):category>")
def blog_list_with_category(category):
    return "您获取的博客分类为: %s"%category
```

在浏览器中访问/blog/list/python，因为博客分类python被包含在了备选值中，所以可以正常显示内容，如图3-6所示。



图3-6 访问/blog/list/python

但是访问/blog/list/java，将会显示Not Found，如图3-7所示。

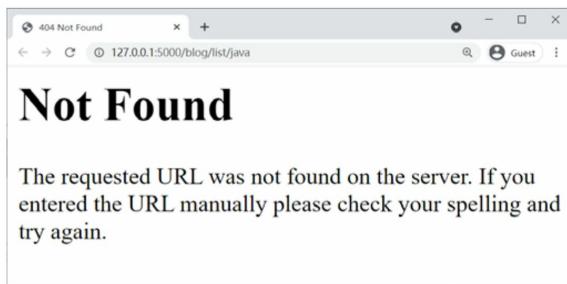


图3-7 访问/blog/list/java

参数选择什么类型，完全取决于视图函数对这个参数的期望，如果期望是整型，那就用int；如果期望是字符串类型，那就用string，其他亦然。

如果URL中需要传递多个参数，则只要用斜杠（/）分隔开来即可。如要获取一个某个用户的博客列表的URL，则需要传递用户id和分页页码两个参数，相关代码如下。

```
@app.route("/blog/list/<int:user_id>/<int:page>")
def blog_list(user_id,page):
    return "您查找的用户为: %s, 博客分页为: %s"%(user_id,page)
```

所以当需要获取用户id为10、页码为8的博客列表数据时，可以通过访问/blog/list/10/8来实现，如图3-8所示。

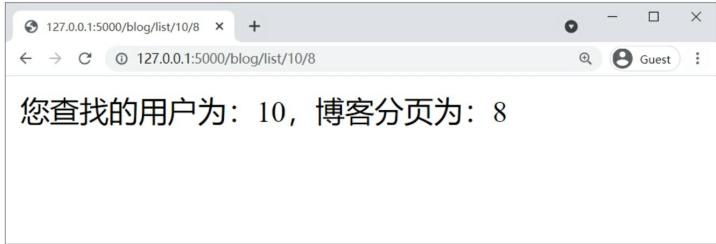


图3-8 访问/blog/list/10/8

在定义URL时，总是会力求简洁，如以上描述的获取某个用户博客列表的URL，默认情况下都是在第1页，这时如果能把page省略掉，不传这个参数，那么URL会变得更加简洁。代码可以修改为如下形式。

```
@app.route("/blog/list/<int:user_id>")
@app.route("/blog/list/<int:user_id>/<int:page>")
def blog_list(user_id,page=1):
    return "您查找的用户为: %s, 博客分页为: %s%(user_id,page)
```

通过以上代码可以看到，我们定义了两个URL，第一个URL中没有page参数，但是blog\_list视图函数的page形参有一个默认值为1，这样当访问不带page参数的URL时，默认的page就是1，从而简化了URL的使用，如图3-9所示。

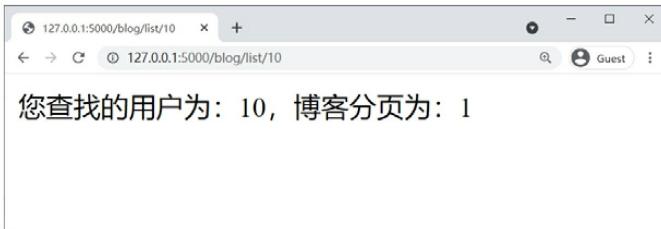


图3-9 访问/blog/list/10

关于在URL中传递参数，还可以通过查询字符串的方式来实现，即在URL后面通过?（英文问号）把参数添加上去，如果有多个参数，则通过&进行拼接，规则如下。

```
URL?参数名1=参数值1&参数名2=参数值2
```

通过查询字符串的方式传递参数，参数先不需要在URL中定义好，只需要在访问URL时将参数传进来即可。下面还是以获取某个用户的博客列表为例，用查询字符串的方式传递参数，则可以通过以下URL来访问。

```
/blog/list?user_id=10&page=8
```

通过查询字符串的方式传递参数，不需要在定义URL和视图函数时提前定义好参数，参数可以通过Flask中的`request.args`对象获取，如以上获取某个用户博客列表的URL，可以通过以下代码来实现。

```
from flask import Flask, request  
...  
@app.route("/blog/list")  
def blog_list_query_str():  
    user_id = request.args.get("user_id")  
    page = request.args.get("page")  
    return "您查找的用户为: %s, 博客分页为: %s" % (user_id, page)
```

其中，`request`是一个线程隔离的全局对象，`request.args`是一个继承自`dict`的`werkzeug.datastructures.MultiDict`对象，保存了当前请求的查询字符串参数，并且被解析成以键一值对的形式存在，后面就可以通过字典的方式获取参数。接下来在浏览器中访问`/blog/list?user_id=10&page=8`，效果如图3-10所示。



图3-10 访问`/blog/list?user_id=10&page=8`

通过查询字符串的方式传递参数，参数是视图函数先规定好的，然后浏览器再按照规定传递。如以上代码中的视图函数是通过`user_id`这个键来获取用户的id，如果前端传的键不是`user_id`，那么视图函数将获取不到用户id，从而导致数据获取失败。关于数据获取失败后如

何处理，后续章节会详细讲解。

在URL中定义参数和查询字符串虽然都能传递参数，但还是有些区别。在URL中定义参数是将参数嵌入URL，实际上已经成为URL的一部分；而查询字符串是HTTP协议层面用于传递参数的技术，后面学到获取POST请求参数时，读者会有更深的理解。这两种方式各有利弊，下面做了简单的总结，读者在开发网站时可自行衡量。

- 在URL中定义参数再传递参数比通过查询字符串的方式传递参数会更利于SEO优化，能更好地被搜索引擎收录和检索。如CSDN博客网站（blog.csdn.net）的博客详情页面的博客id就是通过在URL中定义参数再传递参数的。简书网站（www.jianshu.com）的文章详情，也同样用的是在URL中定义参数的方式传递文章id的。这两个网站的SEO优化做的都是非常不错的。
- 在URL中定义参数可以做好类型约束，不会让错误类型的数据匹配进URL，从而提高了程序的健壮性。
- 通过查询字符串的方式传递参数更加灵活，不需要修改URL，也方便随时添加或者修改参数。

## 3.2 HTTP请求方法

在HTTP协议中，请求URL有不同的方法（method），不同的请求方法有不同的应用场景，下面先来了解有哪些HTTP请求方法以及使用方法，如表3-2所示。

表3-2 HTTP请求方法及使用方法

请求方法	描述
GET	从服务器获取资源。在浏览器中输入网址访问默认用的GET请求
POST	提交资源到服务器。如提交表单或者上传文件，一般用于创建新资源或者修改已有的资源
HEAD	类似于GET请求。响应体中不包含具体的内容，用于获取消息头
DELETE	请求服务器删除资源
PUT	请求服务器替换或者修改已有的资源
OPTIONS	请求服务器返回某个资源所支持的所有HTTP请求方法。如AJAX跨域请求常用OPTIONS方法发送嗅探请求，来判断是否有对某个资源访问的权限
PATCH	与PUT方法类似，但是PATCH方法一般用于局部资源更新，PUT方法用于整个资源的替换

表3-2列举的HTTP请求方法，可以根据不同的场景选择不同的方法。如请求某个URL时，要获取数据，就用GET方法；要删除服务器数据，就用DELETE方法；要往服务器添加

数据，就用POST方法。其他亦然。

在Flask项目中使用app.route装饰器定义URL时，默认用的是GET请求，而在浏览器中，在地址栏中输入一个URL并进行访问，默认也是GET请求，所以可以正常访问。如果想更改URL的请求方法，可以在定义URL时，给app.route设置methods参数，示例代码如下。

```
@app.route("/blog/add",methods=['POST'])
def blog_add():
    return "使用POST方法添加博客"
```

通过以上代码可以看到，在app.route中通过给methods参数赋值一个列表，并且列表中只有一个POST参数，来限制/blog/add这个URL只能通过POST方法进行访问。如在浏览器中访问/blog/add，会显示错误信息“Method Not Allowed”，如图3-11所示。



图3-11 显示错误信息

如果需要一个URL既可以通过GET方法请求访问，也可以通过POST方法请求访问，那么可以给methods方法添加GET和POST参数，示例代码如下。

```
@app.route("/blog/add/post/get",methods=['POST','GET'])
def blog_add_post_get():
    if request.method == 'GET':
        return "使用GET方法添加博客"
    else:
        return "使用POST方法添加博客"
```

因为/blog/add/post/get同时支持GET和POST请求方法，所以在浏览器中访问/blog/add/post/get时也可以访问到页面，如图3-12所示。



图3-12 URL同时支持GET和POST

Flask从2.0版本开始，添加了5个快捷路由装饰器。如app.post表示定义的URL只接收POST请求。5个快捷路由装饰器如表3-3所示。

表3-3 快捷路由装饰器

快捷路由装饰器	描述
app.get("/login")	等价于 app.route("/login",methods=['GET'])
app.post("/login")	等价于 app.route("/login",methods=['POST'])
app.put("/login")	等价于 app.route("/login",methods=['PUT'])
app.delete("/login")	等价于 app.route("/login",methods=['DELETE'])
app.patch("/login")	等价于 app.route("/login",methods=['PATCH'])

### 3.3 页面重定向

页面重定向，下文简称重定向。重定向在页面中体现的操作是，浏览器会从一个页面自动跳转到另外一个页面。例如，用户访问了一个需要权限的页面，但是该用户当前并没有登录，因此重定向到登录页面。重定向分为永久性重定向和暂时性重定向，以下是相关介绍。

- 永久性重定向：HTTP的状态码是301，多用于旧网址已被废弃，要转到一个新的网址，确保用户正常的访问。最经典的案例就是京东网站的案例，在使用www.jd.com域名之前有过许多其他域名，如www.360buy.com、www.jingdong.com，在这两个域名没有被废弃之前，当用户在浏览器中输入这两个域名时，会自动跳转到www.jd.com，因为这两个域名以后要被废弃了，所以在这种情况下应该使用永久性重定向。
- 暂时性重定向：HTTP的状态码是302，表示页面的暂时性跳转。如访问一个需要权限的网址，但是当前用户没有登录，这时候就应该重定向到登录页面，并且是暂时性的重定向。

在Flask中，重定向是通过flask.redirect(location,code=302)函数来实现的，其中location表示需要重定向到哪个URL，code代表状态码，默认是302，即暂时性重定向。下面用一个简单的案例来说明这个函数的用法。

```
from flask import Flask,url_for,redirect

app = Flask(__name__)

@app.route('/login')
def login():
    return 'login page'

@app.route('/profile')
def profile():
    name = request.args.get('name')

    if not name:
        # 如果没有name, 说明没有登录, 重定向到登录页面
        return redirect("/login")
    else:
        return name
```

从以上代码可看出，在访问/profile时，如果没有通过查询字符串的方式传递name参数，那么就会被重定向到/login。如访问/profile?name=admin可以看到，浏览器中显示admin，但是如果直接访问/profile，就会被重定向到/login。读者可自行尝试。

### 3.4 构造URL

在3.3节中执行redirect("/login")函数，让页面跳转到登录页面，这里是直接把/login这个URL硬编码进去的，对于项目健壮性不太友好，更好的方式应该是通过url\_for函数来动态地构造URL。url\_for接收视图函数名作为第1个参数，以及其他URL定义时的参数，如果还添加了其他参数，则会添加到URL的后面作为查询字符串参数。这里以博客详情的URL为例来讲解url\_for函数的使用，示例代码如下。

```
@app.route("/blog/<int:blog_id>")
def blog_detail(blog_id):
    return "您查找的博客id为: %s"%blog_id

@app.route("/urlfor")
def get_url_for():
    url = url_for("blog_detail",blog_id=2,user="admin")
```

```
return url
```

在`get_url_for`视图函数中使用了`url_for`函数，把函数名`blog_detail`作为第1个参数，因为`blog_detail`的URL需要接收一个`blog_id`参数，因此把`blog_id`也传给了`url_for`函数。除此之外，还添加了一个`user`参数，因为`user`参数不是必需的，所以在构建成URL后，会把`user`作为查询字符串参数拼接上去。在浏览器中访问`/urlfor`，可以看到如图3-13所示的效果。



图3-13 访问/urlfor

相比在代码中硬编码URL，使用`url_for`函数来动态地构建URL有以下两点好处。

- URL是对外的，可能会经常变化，但是视图函数不会经常变化。如果直接把URL硬编码，若后期URL改变了，凡是硬编码了这个URL的代码都需要修改，费时费力。
- URL在网络之间通信的过程中，需要把一些特殊字符包括中文等进行编码，如URL中包含了特殊字符，用`url_for`函数会自动进行编码，省时省力。

## chapter 4 第4章 Jinja2模板

前面章节中的视图函数返回的都是一个字符串，而在实际网站开发中，为了让网页更加美观，需要渲染一个有富文本标签的页面，通常包含大量的HTML代码，如果把这些HTML代码用字符串的形式写在视图函数中，后期的代码维护将变成一场噩梦。因此，在Flask中，渲染HTML通常会交给模板引擎来实现，而Flask中默认配套的模板引擎是Jinja2，Jinja2是一个高效、可扩展的模板引擎。Jinja2可以独立于Flask使用，如被Django使用。

Jinja2目前最新版本是3.0.2，官方文档请参考<https://jinja.palletsprojects.com/en/3.0.x/>。

本章用PyCharm Professional版创建一个名叫chapter04的项目，后续的知识点讲解都是基于这个项目。

### 4.1 模板的基本使用

#### 4.1.1 渲染模板

在使用PyCharm Professional版创建完一个Flask项目后，默认会生成一个templates文件夹，如果没有修改模板查找路径，默认会在这个文件夹下寻找模板文件。模板文件可以是任意纯文本格式的文件，如TXT、HTML、XML等，但是为了让项目更规范，也为了与前端开发者更无缝地协作，一般都是用HTML文件来写模板代码。

#### 注意

如果读者用的是非PyCharm Professional版创建的Flask项目，则可以手动创建templates文件夹。

首先在templates文件夹下创建index.html文件，然后输入以下代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
</head>
<body>
```

```
<h1>这是首页</h1>
</body>
</html>
```

接下来在视图函数中使用`render_template`函数渲染`index.html`模板。在`app.py`中，将原来的`hello_world`视图函数修改为以下代码。

```
from flask import Flask, render_template
...
@app.route('/')
def index():
    return render_template("index.html")
```

`render_template`默认会从当前项目的`templates`文件夹下寻找`index.html`文件，读取后进行解析，再渲染成HTML代码返回给浏览器。在浏览器中访问`http://127.0.0.1:5000`，可以看到如图4-1所示的效果。



图4-1 首页渲染模板代码

从图4-1中可以看到，“这是首页”4个字已经是一级标题了，原因是模板中给“这是首页”4个字外面套了一个`h1`标签，至此我们就完成了一个最简单的模板渲染。

如果想修改模板文件的查找地址，可以在创建`app`时，给`Flask`类传递一个关键字参数`template_folder`指定具体路径，示例代码如下。

```
app=Flask(__name__,template_folder=r"E:\flask_fullstack\demo04\mytemplates")
```

如此操作以后，`Flask`在寻找模板文件时，就不再从当前项目下的`templates`文件夹寻找

了，而是从template\_folder指定的路径寻找。项目在Debug模式开启的前提下再访问http://127.0.0.1:5000，会出现如图4-2所示的错误。

The screenshot shows a browser window with the URL `127.0.0.1:5000`. The title bar says `jinja2.exceptions.TemplateNotfound`. The main content area displays the error message `jinja2.exceptions.TemplateNotFound: index.html` in a red-bordered box. Below this, a blue header bar says `Traceback (most recent call last)`. The stack trace lists several file paths and line numbers from the Flask framework's `app.py` and `templatting.py` files, ending with the line `return render_template("index.html")` which is also highlighted with a red box.

```
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 2088, in __call__
    return self.wsgi_app(environ, start_response)
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 2073, in wsgi_app
    response = self.handle_exception(e)
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 2070, in wsgi_app
    response = self.full_dispatch_request()
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 1515, in full_dispatch_request
    rv = self.handle_user_exception(e)
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 1513, in full_dispatch_request
    self.try_trigger_before_first_request_functions()
    try:
        request_started.send(self)
        rv = self.preprocess_request()
        if rv is None:
            rv = self.dispatch_request()
    except Exception as e:
        rv = self.handle_user_exception(e)
    return self.finalize_request(rv)

    def finalize_request():

File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\app.py", line 1499, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
File "E:\flask\fullstack\chapter04\app.py", line 8, in index
    return render_template("index.html")
File "C:\Users\Administrator\Envs\flask-env\lib\site-packages\flask\templatting.py", line 148, in render_template
    ctx.app.jinja_env.get_or_select_template(template_name_or_list),
```

图4-2 模板没有找到

模板没有找到的原因是，在template\_folder指定的文件夹下不存在一个叫作index.html的模板，如果想要解决此问题，只需要把templates文件夹下的index.html复制到template\_folder指定的文件夹下即可。

## 4.1.2 渲染变量

HTML文件中的有些数据是需要动态地从数据库中加载的，不能直接在HTML中写死。一般的做法是，在视图函数中把数据先提取好，然后使用`render_template`渲染模板时传给模

板，模板再读取并渲染出来。下面新建一个URL与视图函数映射，示例代码如下。

```
@app.route("/variable")
def variable():
    hobby = "游戏"
    return render_template("variable.html", hobby=hobby)
```

以上代码中渲染了一个variable.html模板，这个模板文件的创建接下来会具体讲解。除模板名称外，还给render\_template传递了一个hobby关键字参数，后续在模板中就可以使用这个变量了。

现在再在templates文件夹下创建一个variable.html模板文件（注意：要记得先删掉template\_folder参数），然后输入以下代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>变量使用</title>
</head>
<body>
    <h1>我的兴趣爱好是: {{ hobby }}</h1>
</body>
</html>
```

从以上代码中可以看到，把变量放到两对花括号中即可使用变量。项目运行起来后，在浏览器中访问http://127.0.0.1:5000/variable，效果如图4-3所示。



图4-3 变量使用

图4-3中的文字“游戏”是从视图函数中通过render\_template传过去的，并不是在HTML中

写死的，所以变量的使用可以让同一个HTML模板渲染无数个不同的页面。

字典的键和对象的属性在模板中都可以通过点(.)的形式访问。在variable这个视图函数中添加两个新的变量，分别是字典类型的person，以及类对象类型的user。示例代码如下。

```
class User:  
    def __init__(self,username,email):  
        self.username = username  
        self.email = email  
  
@app.route("/variable")  
def variable():  
    hobby = "游戏"  
    person = {  
        "name": "张三",  
        "age": 18  
    }  
    user = User("李四","xx@qq.com")  
    return  
    render_template("variable.html",hobby=hobby,person=person,user=user)
```

接下来，再在variable.html模板中通过点(.)的形式访问person的键和user属性。代码如下。

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>变量使用</title>  
</head>  
<body>  
    <h1>我的兴趣爱好是: {{ hobby }}</h1>  
    <p>person的姓名是: {{ person.name }}, person的年龄是: {{ person.age }}</p>  
    <p>user的用户名是: {{ user.name }}, user的邮箱是: {{ user.email }}</p>  
</body>  
</html>
```

在浏览器中重新访问http://127.0.0.1:5000/variable，效果如图4-4所示。

字典键和对象的属性也都可以通过中括号的形式获取，如以下代码实际上是等价的。

```
 {{ user.name }}  
 {{ user["name"] }}
```

读者可以自行修改variable.html中获取键和属性值的方式，最终效果是一样的。用点和中括号的形式访问，虽然效果一样，但是也存在以下不同。



图4-4 模板中通过点渲染字典和对象

(1) 在模板中有一个变量的使用方式为foo.bar，那么在Jinja2中则按以下方式进行访问。

- 通过getattr(foo, 'bar')访问，先访问这个对象的属性。
- 如果没有找到，就通过foo.\_\_getitem\_\_("bar")方式访问，即访问这个对象的键。
- 如果以上两种方式都没有找到，返回一个undefined对象。

(2) 在模板中有一个变量的使用方式为foo["bar"]，那么在Jinja2中则按以下方式进行访问。

- 通过foo.\_\_getitem\_\_("bar")方式访问，即先访问这个对象的键。
- 如果没有找到，就通过getattr(foo, "bar")方式访问，即访问这个对象的属性。
- 如果以上都没找到，则返回一个undefined对象。

以上案例中，传递了3个变量到模板中，在变量比较多的情况下，首先可以把所有的变量存放到字典中，然后在给render\_template传递参数时使用\*\*语法，将字典变成关键字参数，以上的variable视图函数代码可以改写为以下形式。

```
@app.route("/variable")  
def variable():  
    hobby = "游戏"
```

```
person = {
    "name": "张三",
    "age": 18
}
user = User("李四", "xx@qq.com")
context = {
    "hobby": hobby,
    "person": person,
    "user": user
}
return render_template("variable.html", **context)
```

以上代码的写法更加直观和简洁，在遇到需要传给模板的变量比较多的情况，都推荐使用这种方式。

## 4.2 过滤器和测试器

在Python中，如果需要对某个变量进行处理，可以通过函数来实现，而在模板中，则是通过过滤器来实现的。过滤器本质上也是函数，但是在模板中使用的方式是通过管道符号(|)来调用的。例如，有一个字符串类型的变量name，要获取它的长度，可以通过{{name|length}}来获取，Jinja2会把name当作第1个参数传给length过滤器底层对应的函数。length是Jinja2内置好的过滤器，Jinja2内置了许多好用的过滤器，如果内置的过滤器不能满足需求，还可以自定义过滤器。下面先来学习如何自定义过滤器，读者明白了过滤器的原理后，再去学习Jinja2内置的过滤器就会更加得心应手了。

### 4.2.1 自定义过滤器

过滤器本质上是Python的函数，它会把被过滤的值当作第1个参数传给这个函数，函数经过一些逻辑处理后，再返回新的值。过滤器函数写好后可以通过@app.template\_filter装饰器或者app.add\_template\_filter函数把函数注册成Jinja2能用的过滤器。这里以注册一个时间格式化的过滤器为例，来说明自定义过滤器的方法，示例代码如下。

```
def datetime_format(value, format="%Y-%d-%m %H:%M"):
    return value.strftime(format)

app.add_template_filter(datetime_format, "dformat")
```

在上面代码中定义了一个datetime\_format函数，第1个参数是需要被处理的值，第2个参

数是时间的格式，并且指定了一个默认值。下面通过app.add\_template\_filter将datetime\_format函数注册成了过滤器，过滤器的名字叫作dformat。那么以后在模板文件中，就可以按如下方式使用了。

```
 {{ article.pub_date|dformat }}  
 {{ article.pub_date|dformat("%B %Y") }}
```

如果app.add\_template\_filter没有传第2个参数，那么默认将使用函数的名称作为过滤器的名称。如以上注册过滤器代码可以改成以下代码。

```
 ...  
 app.add_template_filter(datetime_format)
```

在模板中则按以下方式使用。

```
 {{ article.pub_date|datetime_format }}  
 ...
```

当然，也可以通过@app.template\_filter装饰器在函数定义时，就将它注册成过滤器。如以上的datetime\_format函数，可以改写为如下形式。

```
@app.template_filter("dformat")  
def datetime_format(value, format="%Y-%d-%m %H:%M"):  
    return value.strftime(format)
```

datetime\_format被@app.template\_filter装饰后，就会自动被注册进Jinja2的过滤器中，并且@app.template\_filter中的参数即为自定义过滤器的名称，如果不传参数，也会自动使用函数名称作为过滤器的名称。

## 4.2.2 Jinja2内置过滤器

在理解了Jinja2过滤器的原理后，再来学习Jinja2中内置过滤器，读者无须全部记住这些过滤器，只需在使用的时候翻阅本书或者阅读Jinja2官方文档

<https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-filters>即可，用的次数多了自然会记住。

Jinja2中内置过滤器如下。

(1) `abs(value)`: 获取value的绝对值。  
(2) `default(value,default_value,boolean=False)`: 如果value没有定义，则返回第2个参数`default_value`。如果要让value在被判断为False的情况下使用`default_value`，则应该将后面的`boolean`参数设置为False。先看以下示例。

```
<div>default过滤器: {{ user|default('admin') }}</div>
```

如果`user`没有定义，那么将会使用`admin`作为默认的值。再看以下示例。

```
<div>default过滤器: {{ ""|default('admin',boolean=True) }}</div>
```

因为""（空字符串）在使用if判断时，返回的是False，这时如果要使用默认值`admin`，就必须加上`boolean=True`参数。

(3) `escape(value)`: 将一些特殊字符，如&、<、>、"、'进行转义。因为Jinja2默认开启了全局转义，所以在大部分情况下无须手动使用这个过滤器去转义，只有在关闭了转义的情况下，会需要使用到它。

(4) `filesizeformat(value,binary=False)`: 将值格式化成方便阅读的单位。如13KB、4.1MB、102Bytes等。默认是Mega、Giga，也就是每个相邻单位换算是1000倍。如果第2个参数设置为True，那么相邻单位换算是1024倍。

(5) `first(value)`: 返回value序列的第一个元素。

(6) `float(value,default=0.0)`: 将value转换为浮点类型，如果转换失败会返回0.0。

(7) `format(value,*args,**kwargs)`: 格式化字符串，示例代码如下。

```
{ { "%s,%s" | format (greeting, name) } }
```

(8) `groupby(value,attribute,default=None)`: value是一个序列，可以使用参数`attribute`进行分组。例如，有一个`users`列表，里面的`user`都有一个`city`属性，如果要按照`city`进行分组，

则可以使用以下代码实现。

```
<ul>
{% for group in users|groupby("city") %}
<li>{{ group.grouper }}: {{ group.list|join(", ") }}</li>
{% endfor %}
</ul>
```

(9) `int(value,default=0,base=10)`: 转换为整型，如果转换失败会返回0，并且默认按照十进制转换。

(10) `join(value,attribute)`: 使用`attribute`指定的元素，将一个序列拼接成一个字符串。与Python中的`join`方法类似。

(11) `last(value)`: 返回`value`序列的最后一个元素。

(12) `length(value)`: 返回`value`序列的长度。

(13) `list(value)`: 转换`value`为一个列表。

(14) `lower(value)`: 将`value`全部转换为小写。

如要将`titles`序列中每个元素的值都变成小写形式，那么可以使用以下代码实现。

```
{{ titles|map('lower') }}
```

(15) `map(value,*args,**kwargs)`: 将`value`这个序列都执行某个操作。如获取`users`这个序列中每个`user`的`username`字段。可以通过以下代码实现。

```
{{ users|map(attribute='username') }}
```

(16) `max(value)`: 求序列中的最大值。

(17) `min(value)`: 求序列中的最小值。

(18) `random(value)`: 返回`value`这个序列中的一个随机值。

(19) `reject(value,*args,**kwargs)`: 过滤`value`这个序列中的一些元素，过滤的条件通过后面的参数给定。如要过滤列表中所有的奇数，可以把Jinja2中内置的`odd`过滤器传给`reject`过滤器来实现，代码如下。

```
 {{ numbers|reject ('odd') }}
```

(20) `rejectattr(value,*args,**kwargs)`: 根据value序列中元素的某个属性进行过滤。只要这个属性满足条件，那么就会被过滤掉，示例代码如下。

```
 {{ users|rejectattr("is_active") }}  
 {{ users|rejectattr("email", "none") }}
```

上面第1行代码是过滤users中`is_active`为True的对象，第2行代码是过滤users中`email`为`none`的对象。

(21) `replace(value,old,new,count)`: 将字符串value中的old替换为new，并且可以通过count来确定替换多少个。与Python中字符串的`replace`方法用法一致。

(22) `reverse(value)`: 将value这个序列逆序。

(23) `safe(value)`: 在渲染value时，关闭自动转义。如以下代码所示。

```
<div>safe过滤器: {{ "<p style='background-color: red;'>中国</p>"|safe }}  
</div>
```

因为加了`safe`过滤器，就不会对前面的字符串进行转义，前面的字符串就会被当成HTML代码嵌入网页，从而看到一个红色的背景，背景中显示“中国”两个文字。

(24) `select(value,*args,**kwargs)`: 选择value序列中满足条件的元素，与`reject`正好相反。

(25) `selectattr(value,*args,**kwargs)`: 根据value序列中元素的某个属性进行过滤，留下满足条件的，过滤掉不满足条件的，与`rejectattr`正好相反。

(26) `sort(value,reverse=False,case_sensitive=False,attribute=None)`: 将value这个序列进行排序，底层用的是Python的`sorted`函数，`reverse`代表是否逆向排序，`case_sensitive`代表是否忽略大小写，`attribute`代表根据value序列中元素的某个属性排序。

(27) `string(value)`: 将value转换为字符串类型。

(28) `striptags(value)`: 将字符串value中的HTML标签去除，留下文本内容。

(29) `tojson(value)`: 将`value`转换为JSON格式的字符串。

(30) `trim(value)`: 删掉`value`前面和后面的空白字符（空格）。

(31) `truncate(value,length=255,killwords=False,end="...")`: 将字符串`value`进行截取，`length`代表保留多少个字符，`killwords`代表在截取字符串时是否要裁剪单词，`end`代表末尾的结束字符。这在文章简介、个人简介等只需要显示一部分字符的场景下非常有用。

(32) `unique(value,case_sensitive=False,attribute=None)`: 将`value`序列中的重复元素删除。`case_sensitive`代表是否忽略大小写，`attribute`代表使用`value`序列中元素的某个属性。

(33) `upper(value)`: 将`value`所有字符全部转换为大写。

(34) `urlencode(value)`: 如果`value`是字符串，那么底层会调用Python的`urllib.parse.quote`；如果`value`是字典，那么底层会调用Python的`urllib.parse.urlencode`。

(35)

`urlize(value,trim_url_limit=None,nofollow=False,target=None,rel=None,extra_schemes=None)`: 将`value`变成可以单击的链接，如URL和邮箱。注意：`value`必须是以`http://`、`https://`、`www.`、`mailto`开头的字符串。

(36) `wordcount(value)`: 统计`value`中共有多少个单词。

(37) `xmlattr(value,autospace=True)`: `value`为一个字典，根据这个字典创建一个xml格式的属性，示例代码如下。

```
<ul{{ {'class': 'my_list', 'missing': none,
       'id': 'list-%d'|format(variable)}|xmlattr }}>
...
</ul>
```

过滤器可以嵌套使用，如以下代码所示。

```
{{ titles|map("lower")|join(",") }}
```

在解析模板时，会先将`titles`传给`map`过滤器处理，得到结果后再传给`join`过滤器。

### 4.2.3 测试器

测试器用来测试某些元素是否满足某个条件，如测试一个变量是否是字符串、测试一个

变量能否被调用等。以下代码通过演示defined测试器，来讲解测试器的使用。

```
{% if user is defined %}  
    user定义了 : {{ user }}  
{% else %}  
    user没有定义  
{% endif %}
```

可以看到，测试器是通过if...is...来使用的，if后面是被测试的对象，is后面是测试器。除了defined测试器，Jinja2还提供了如表4-1所示的测试器。

表4-1 测试器

测试器	描述
boolean	是否为布尔类型
callable	是否能被调用
defined	是否定义
divisibleby	是否能被某个数整除
eq	是否和另外一个值相等
escaped	是否已经被转义
even	是否为偶数
false	是否为 False
filter	是否为过滤器
float	是否为浮点类型
ge	是否大于或等于某个数
gt	是否大于某个数
in	是否在某个序列中，与 Python 中的 in 语法类似
integer	是否为整型
iterable	是否为可迭代类型
le	是否小于或等于某个数
lower	是否全部为小写
lt	是否小于某个数
mapping	是否为一个 mapping 对象（如字典）
ne	是否不等于某个数
none	是否为 None
number	是否为数值类型
odd	是否为奇数
sameas	是否在内存中和另外一个对象是一样的
sequence	是否为序列（如列表、元组）
string	是否为字符串
test	是否为一个测试器
true	是否为 True
undefined	是否没有定义
upper	是否全部为大写

表4-1所示的测试器是Jinja2模板中内置的所有测试器，与过滤器的学习方式一样，读者可先简单阅读，无须强记，在需要使用时再翻阅Jinja2内置测试器的官方文档  
<https://jinja.palletsprojects.com/en/3.0.x/templates/#list-of-builtin-tests>即可。

## 4.3 控制语句

在模板中，也存在if判断和for循环等控制语句。所有的控制语句都是放在`{% %}`中间的，并且在控制语句结束后，要加入相应的结束语句。下面对if判断语句和for循环语句分别进行讲解。

### 4.3.1 if判断语句

Jinja2中的if判断语句和Python中的if判断语句非常类似，可以使用关系运算符`>、<、>=、<=、==、!=`来进行判断，也可以通过and、or、not来进行逻辑操作。我们首先创建一个视图函数`if_statement`，代码如下。

```
@app.route("/if")
def if_statement():
    age = 18
    return render_template("if.html", age=age)
```

以上代码定义了一个`age`变量，并且把这个`age`传给了`if.html`模板。在`if.html`模板中，可以根据`age`的大小判断是否成年。`if.html`模板的代码如下。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>if语句</title>
</head>
<body>
    {% if age > 18 %}
        <div>您已成年! </div>
    {% elif age < 18 %}
        <div>您未成年! </div>
    {% else %}
        <div>您刚成年! </div>
    {% endif %}
</body>
</html>
```

因为在视图函数中给`age`赋值为18，所以在模板中会匹配到以下代码。

```
<div>您刚成年! </div>
```

在浏览器中访问`http://127.0.0.1:5000/if`, 也可以看到显示的是“您刚成年！”，如图4-5所示。



图4-5 模板中使用if判断语句

仔细阅读`if.html`模板代码可以发现，在`if`语句结束后，需要添加`endif`关闭`if`代码块，这跟Python中的用法是有点不同。

### 注意

Jinja2中的代码缩进只是为了更加方便阅读，任何缩进都不是必需的。

## 4.3.2 for循环语句

Jinja2中的`for`循环与Python中的`for`循环也非常类似，Jinja2中的`for`循环只是比Python中的`for`循环多了一个`endfor`代码块。我们先来实现一下视图函数`for_statement`，代码如下。

```
@app.route("/for")
def for_statement():
    books = [
        {"name": "三国演义",
         "author": "罗贯中",
         "price": 100
        },
        {"name": "水浒传",
         "author": "施耐庵",
         "price": 99
        },
        {"name": "红楼梦",
```

```
        "author": "曹雪芹",
        "price": 101
    }, {
        "name": "西游记",
        "author": "吴承恩",
        "price": 102
    }]
return render_template("for.html", books=books)
```

在for\_statement视图函数中，首先创建了一个books变量，books是一个列表，里面存放的是图书信息的字典，然后渲染给for.html模板。接下来在模板文件中循环这个列表，代码如下。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>for 循环</title>
</head>
<body>
    <table>
        <thead>
            <tr>
                <th>书名</th>
                <th>作者</th>
                <th>价格</th>
            </tr>
        </thead>
        <tbody>
            {% for book in books %}
            <tr>
                <td>{{ book.name }}</td>
                <td>{{ book.author }}</td>
                <td>{{ book.price }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</body>
</html>
```

因为在table表格标签中，一个tr标签代表表格中的一行，所以在tr外面加了一个for循环，去循环这个books列表。在tr下面，一个td代表一列，每列从book中获取对应的数据，分别是书名、作者、价格。在浏览器中访问http://127.0.0.1:5000/for，显示结果如图4-6所示。

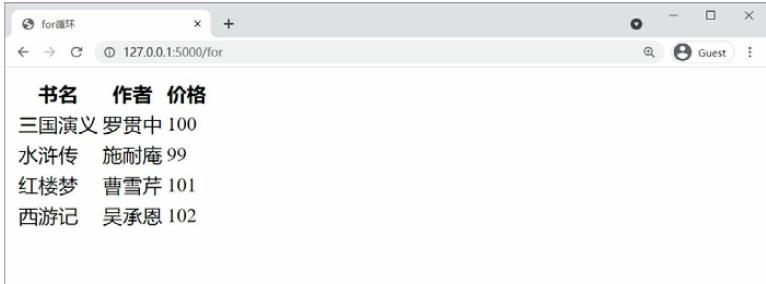


图4-6 Jinja2 for循环

如果被循环的序列（如以上代码中的books）中没有元素，那么可以使用else来处理。通常我们在浏览网页时，如果某个网页没有数据，则会显示“无数据”。我们在以上代码中的for.html模板加上else，来实现一个类似的需求，代码如下。

```
...
{%
    for book in books %}
    <tr>
        <td>{{ book.name }}</td>
        <td>{{ book.author }}</td>
        <td>{{ book.price }}</td>
    </tr>
{%
    else %}
    <tr>
        <td colspan="3" style="text-align: center;">无数据</td>
    </tr>
{%
    endfor %}
...

```

在books中无数据的情况下，会执行到else中，可以将for\_statement视图函数的books修改为一个空的列表，代码如下。

```
@app.route("/for")
def for_statement():
    books = []
    return render_template("for.html", books=books)
```

此时再在浏览器中访问http://127.0.0.1:5000/for，可以看到页面会显示“无数据”，如图4-7所示。

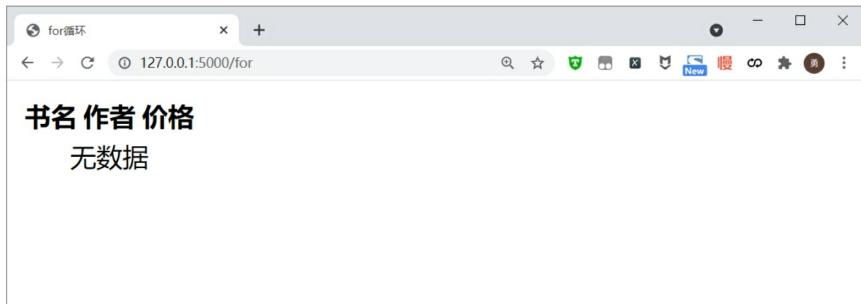


图4-7 显示页面无数据

Jinja2中的for循环中还内置了许多好用的变量。如要获取当前循环到第几次了，可以通过loop.index来实现。我们还是以for\_statement视图函数为例，首先将books变量恢复成以下形式。

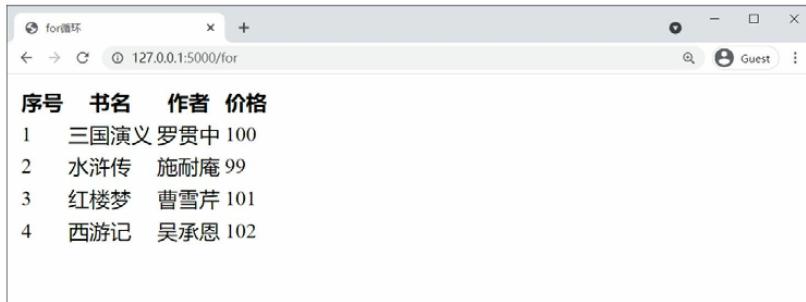
```
books = [{  
    "name": "三国演义",  
    "author": "罗贯中",  
    "price": 100  
, {  
    "name": "水浒传",  
    "author": "施耐庵",  
    "price": 99  
, {  
    "name": "红楼梦",  
    "author": "曹雪芹",  
    "price": 101  
, {  
    "name": "西游记",  
    "author": "吴承恩",  
    "price": 102  
}]
```

然后在for.html模板中给图书表格新增一个名为序号的列，用loop.index来显示每行的序号，修改后的for.html模板中的table代码如下。

```
<table>  
  <thead>  
    <tr>  
      <th>序号</th>  
      <th>书名</th>
```

```
<th>作者</th>
<th>价格</th>
</tr>
</thead>
<tbody>
{%
  for book in books %
}
<tr>
  <td>{{ loop.index }}</td>
  <td>{{ book.name }}</td>
  <td>{{ book.author }}</td>
  <td>{{ book.price }}</td>
</tr>
{%
  else %
}
<tr>
  <td colspan="3" style="text-align: center;">无数据</td>
</tr>
{%
  endfor %
}
</tbody>
</table>
```

通过以上代码可以看到，在thead标签中新增了一个叫作序号的表头，tbody中新增了一个叫作loop.index列，loop.index在每次循环时，会显示当前循环的次数，即代表第几行。在浏览器中访问http://127.0.0.1:5000/for，可以看到如图4-8所示的效果图。



The screenshot shows a simple web browser window titled "for循环". The address bar displays "127.0.0.1:5000/for". The main content is a table with the following data:

序号	书名	作者	价格
1	三国演义	罗贯中	100
2	水浒传	施耐庵	99
3	红楼梦	曹雪芹	101
4	西游记	吴承恩	102

图4-8 带有序号的循环

除loop.index外，Jinja2中的for循环中还提供了如表4-2所示的变量。

表4-2 for循环中的变量

变    量	描    述
loop.index	当前迭代的序号，从 1 开始
loop.index0	当前迭代的序号，从 0 开始
loop.revindex	当前迭代的逆向序号（最后一次为 1，倒数第二次为 2，以此类推），从 1 开始
loop.revindex0	当前迭代的逆向序号（最后一次为 0，倒数第二次为 1，以此类推），从 0 开始
loop.first	判断当前是否是第一次迭代
loop.last	判断当前是否是最后一次迭代
loop.length	序列的长度
loop.cycle	和外层的循环一起循环某个序列
loop.depth	在多层循环中，指示当前是在第几层循环，从 1 开始
loop.depth0	在多层循环中，指示当前是在第几层循环，从 0 开始
loop.previtem	当前迭代的上一个元素。如果当前迭代是第一次迭代，这个变量会返回 undefined 对象
loop.nextitem	当前迭代的下一个元素。如果当前迭代是最后一次迭代，这个变量会返回 undefined 对象
loop.changed	判断当前元素的某个值是否和上一次迭代一样，如果不一样，返回 True，否则返回 False

表4-2中的13个变量中，只有loop.cycle和loop.changed是函数，其余都是变量。这里再做两个案例来讲解loop.cycle和loop.changed的用法。

(1) loop.cycle: 假设现在有一个需求，需要针对table标签中行号为奇数的tr标签添加odd类，行号为偶数的tr标签添加even类，可以通过以下代码实现。

```
{% for book in books %}
  <tr class="{{ loop.cycle('odd', 'even') }}>
    <td>{{ loop.index }}</td>
    <td>{{ book.name }}</td>
    <td>{{ book.author }}</td>
    <td>{{ book.price }}</td>
  </tr>
{% endfor %}
```

在循环books的过程中，loop.cycle也会不断地在odd和even两个变量中循环，从而实现奇数使用odd类，偶数使用even类。

(2) loop.changed: 假设现在想要知道当前循环的book.name是否和上次循环的一致，可以通过loop.changed实现，代码如下。

```
{% for book in books %}
  <tr>
    <td>{{ loop.index }}</td>
    <td>{{ book.name }}</td>
    <td>{{ book.author }}</td>
    <td>{{ book.price }}</td>
    <td>{{ loop.changed(book.name) }}</td>
  </tr>
{% endfor %}
```

Jinja2模板的for循环不存在break和continue来中断循环的语句，这一点是和Python中的for循环最大的区别，另外Jinja2中只有for循环，不存在while循环，读者在使用时尤其要注意这两点。

## 4.4 模板结构

Jinja2比传统HTML拥有更加强大的功能，其中一个就表现在代码模块化上。HTML除了通过iframe标签在浏览器端动态加载其他网页，几乎不具备任何代码模块化的能力。而Jinja2则可以通过宏、模板继承、引入模板等方式实现代码模块化，下面分别进行讲解。

### 4.4.1 宏和import语句

模板中的宏与Python中的函数类似，可以传递参数，但是不能有返回值。可以将一些常用的代码片段放到宏中，然后把一些不固定的值抽取出来当成一个参数，下面用一个例子来阐述宏的用法。

```
{% macro input(name,value='', type='text') %}
  <input type="{{ type }}" value="{{ value|escape }}" name="{{ name }}">
{% endmacro %}
```

以上代码通过macro标签创建了一个叫作input的宏，这个宏接收两个参数，分别是name和type。以后在创建input标签时，可以通过以下代码快速创建。

```
<div>{{ input('username') }}</div>
<div>{{ input('password', type='password') }}</div>
```

在实际的开发工作中，经常会将一些常用的宏单独放到一个文件中，在需要使用时再从这个文件进行导入。导入使用的是import语句，import语句的用法与Python中的import类似，形式可以直接是import...as...，也可以是from...import...，或者是from...import...as...。下面先创建一个forms.html，然后添加以下代码。

```
{% macro input(name, value='', type='text') %}
<input type="{{ type }}" value="{{ value|escape }}" name="{{ name }}">
{% endmacro %}

{% macro textarea(name, value='', rows=10, cols=40) %}
<textarea name="{{ name }}" rows="{{ rows }}" cols="{{ cols }}"
}>{{ value|escape }}</textarea>
{% endmacro %}
```

在forms.html中添加了两个宏，分别是input和textarea。下面再在另外一个文件中通过import语句进行导入。

(1) 通过import...as...形式导入。

```
{% import 'forms.html' as forms %}
<dl>
<dt>Username</dt>
<dd>{{ forms.input('username') }}</dd>
<dt>Password</dt>
<dd>{{ forms.input('password', type='password') }}</dd>
</dl>
<p>{{ forms.textarea('comment') }}</p>
```

(2) 通过from...import...as...或者from...import...形式导入。

```
{% from 'forms.html' import input as input_field, textarea %}
<dl>
<dt>Username</dt>
<dd>{{ input_field('username') }}</dd>
<dt>Password</dt>
<dd>{{ input_field('password', type='password') }}</dd>
</dl>
<p>{{ textarea('comment') }}</p>
```

需要注意的是，通过import导入模板并不会把当前模板的变量添加到被导入的模板中，

如果要在被导入的模板中使用当前模板的变量，可以通过以下两种方式实现。

- (1) 显示地通过参数形式传递变量。
- (2) 使用with context的方式，示例代码如下。

```
{% from 'forms.html' import input with context %}
```

通过以上两种方式，在forms.html中的代码也可以使用当前模板的所有变量了。

#### 4.4.2 模板继承

一个网站中的大部分网页的模块是重复的，如顶部的导航栏、底部的备案信息。如果在每个页面中都重复地去写这些代码，会让项目变得臃肿，增加后期的维护成本。比较好的做法是，通过模板继承，把一些重复性的代码写在父模板中，子模板继承父模板后，再分别实现自己页面的代码。首先来看一个父模板base.html的例子。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="base.css" />
    <title>{% block title %}{% endblock %}</title>
    {% block head %}{% endblock %}
</head>
<body>
    <div id="body">{% block body %}{% endblock %}</div>
    <div id="footer">
        {% block footer %}
            &copy; Copyright 2008 by <a href="http://domain.invalid/">you</a>
        {% endblock %}
    </div>
</body>
</html>
```

在以上父模板中编写了网页的整体结构，并且把所有子模板都需要用到的样式文件base.css也提前做了引用。针对子模板需要重写的地方，则定义成了block，如title、head、body、footer，子模板在继承了父模板后，重写对应block的代码，即可完成子模板的渲染。下面以继承base.html的方式实现一个index.html文件，代码如下。

```
{% extends "base.html" %}  
{% block title %}首页{% endblock %}  
{% block head %}  
    <style type="text/css">  
        .detail{  
            color: red;  
        }  
    </style>  
{% endblock %}  
{% block content %}  
    <h1>这里是首页</h1>  
    <p class="detail">  
        首页的内容  
    </p>  
{% endblock %}
```

以上代码首先通过extends语法加载父模板，因为base.html和index.html是在同一级目录，所以直接写base.html。这里需要注意的是，extends必须出现在子模板所有代码的最前面。接下来分别实现了title、head、content这3个block，实现的block中的代码将会被自动填充到父模板指定的位置，并且最终会生成一个完整HTML结构的index.html文件。

模板中不能出现重名的block，如果一个地方需要用到另外一个block中的内容，可以使用self.blockname的方式进行引用，如下代码所示。

```
<title>  
    {% block title %}  
        这是标题  
    {% endblock %}  
</title>  
<h1>{{ self.title() }}</h1>
```

在以上示例代码中，h1标签中通过{{ self.title() }}把title这个block中的内容引用到了h1标签中。

如果子模板要继承父模板中某个block的内容，如以上案例中，如果要继承父模板footer这个block中已有的内容，则可以通过super()来实现，示例代码如下。

```
{% block footer %}  
    {{ super() }}  
    // 子模板自己的代码  
{% endblock %}
```

以上代码中，如果没有使用{{ super() }}，那么子模板将不能继承父模板footer这个block中的内容。

### 4.4.3 引入模板

有些HTML模块需要在几个页面中使用，如果用模板继承会在所有子模板中都加载，不太合适；如果在每个需要使用这个HTML结构的页面中都重复相同的代码，会增加后期项目的维护成本。这时就可以通过include语法引入模板。如在网站中推广客服二维码联系方式，在有些页面中需要使用，但是并不是所有页面都需要，因此可以把相关代码写成\_contact.html文件，然后在需要的位置进行引用即可，示例代码如下。

```
{% include "_contact.html" %}
```

因为\_contact.html是作为被引用的模板而存在的，所以一般在命名前加一个下画线，这样可以和普通的页面模板区分开来。

## 4.5 模板环境

### 4.5.1 模板上下文

通过render\_template传入的变量，实际上是保存到了模板的上下文中，当然Jinja2也有一些内置的上下文变量，可以通过app.context\_processor来添加全局上下文。所以简单地理解上下文就是模板中可以直接使用的变量。

#### 1. 自定义变量

变量除了通过render\_template渲染外，还可以在模板中通过set语法来定义新变量。示例代码如下。

```
{% set name='admin' %}
```

使用set赋值语句创建的变量在其之后都是有效的。如果不让一个变量污染全局环境，可以使用with语句来创建一个内部的作用域，将set语句放到其中，这样创建的变量就只能在with代码块中才有效，示例代码如下。

```
{% with %}
    {% set foo = 42 %}
    {{ foo }}
{% endwith %}
```

也可以在with后面直接添加变量，如以上写法可以简写成以下形式。

```
{% with foo = 42 %}
    {{ foo }}
{% endwith %}
```

以上两种写法是等价的，一旦超出with代码块，就不能再使用foo这个变量了。

## 2. Jinja2内置全局上下文变量

Jinja2为了方便开发者，已经提前内置了一些常用的全局上下文变量，如表4-3所示。

表4-3 Jinja2内置全局上下文变量

变 量	描 述
g	当前请求中的全局对象。一般会把当前请求中多个地方需要用到的变量绑定到上面
request	当前请求对象。通过它可以获取请求的信息
session	当前请求的 session 对象
config	项目的配置对象

表4-3所示的上下文变量可以在所有模板中直接使用，不需要额外传参。

## 3. 上下文处理器

Jinja2虽然内置了一些上下文变量，但有时候我们需要传递自定义的变量。如很多网站的导航条右上角会显示当前登录的用户名，这就需要把user变量传递到几乎所有模板中，如果通过render\_template传递，则会很麻烦。这时就可以通过上下文处理器装饰器@app.context\_processor来实现，示例代码如下。

```
@app.context_processor
def context_user():
    user = {"username": "admin", "level": 2}
    return {"user": user}
```

在自定义的上下文处理器函数中，需要把变量放到字典中才能在模板中被使用。另外，上下文中的变量，除了Jinja2内置的全局上下文变量以外，其余上下文变量不能再被import导入的模板中使用，如果需要使用，则需要使用with context语法，详情请参见4.4.1节“宏和import语句”。

## 4.5.2 全局函数

### 1. 内置全局函数

为了增强Jinja2模板的逻辑功能，Jinja2内置了一些全局函数，这些函数在所有模板中都可以使用，包括被导入的模板。内置的全局函数如表4-4所示。

表4-4 Jinja2内置全局函数

函数名	描述
range(start,stop,step):	获取一个等差级数的列表，与Python中的range一样
lipsum(n=5,html=True,min=20,max=100)	在模板中生成随机的文本，默认会生成5段HTML代码，每段是20~100个字符。如果html设置为False，那么会返回纯文本内容
dict(**items)	转换为字典，与Python中的dict一样

此外还有3个全局类，即cyclers、joiner、namespace，详细内容可参考Jinja2官方文档全局函数<https://jinja.palletsprojects.com/en/3.0.x/templates/#list-of-global-functions>。

除了Jinja2内置的全局函数外，Flask也提供了两个全局函数，如表4-5所示。

表4-5 Flask提供的全局函数

函数名	描述
url_for	用于加载静态文件，或者用于构建URL
get_flashed_message	用于获取闪现消息

url\_for函数可以用来构建URL和加载静态文件，构建URL与在Python脚本中的用法是一样的，示例代码如下。

```
 {{ url_for("book_detail",book_id=1) }} 
```

关于url\_for函数如何加载静态文件，以及get\_flashed\_message函数的使用，后续内容会详细讲解。

## 2. 自定义全局函数

如果要实现自定义的全局函数，可以通过app.template\_global装饰器来实现，示例代码如下。

```
@app.template_global()  
def greet(name):  
    return "欢迎! %s"%name
```

以上自定义的全局函数可以在模板中直接使用，示例代码如下。

```
<div>{{ greet("张三") }}</div>
```

### 4.5.3 Flask模板环境

在Flask中使用Jinja2，还可以使用app.jinja\_env属性来配置模板。app.jinja\_env是jinja2.Environment类的对象，下面讲解jinja2.Environment对象常用的属性。

#### 1. 设置autoescape

Jinja2默认是开启了全局转义的，如果要关闭全局转义，可以通过以下代码实现。

```
# 关闭全局转义  
app.jinja_env.autoescape = False
```

#### 2. 添加过滤器

添加过滤器可以通过app.jinja\_env.filters实现，代码如下。

```
def myadd(a,b):  
    return a + b
```

```
app.jinja_env.filters["myadd"] = myadd
```

### 3. 添加全局对象

app.template\_global()装饰器只能添加全局函数，如果需要其他Python对象，则可以通过app.jinja\_env.globals实现，可以为其添加任意类型的Python对象，代码如下。

```
app.jinja_env.globals["user"] = user
```

### 4. 添加测试器

添加测试器可以通过app.jinja\_env.tests实现，代码如下。

```
def is_admin(user):
    if user.role == "admin":
        return True
    else:
        return False

app.jinja_env.tests["is_admin"] = is_admin
```

关于jinja2.Environment的其他属性，读者可以自行阅读其官方文档  
<https://jinja.palletsprojects.com/en/3.0.x/api/?highlight=environment#jinja2.Environment>进行了了解。

## 4.6 其他

### 4.6.1 转义

在模板渲染字符串时，字符串中有可能包含一些危险的字符，如&、<、>、"、'等。这些字符会破坏原来HTML标签的结构，更严重的可能会发生XSS跨域脚本攻击。因此，遇到这些特殊字符时，应该将其转义成HTML能正确表示这些字符的写法，如<（小于号）在HTML中应该用<来表示。

在使用render\_template渲染模板时，Flask会针对以.html、.htm、.xml和.xhtml结尾的文件

进行全局转义，但是对于其他类型的文件，则不会开启全局转义。当然针对以.html、.htm、.xml、.xhtml结尾的文件，如果要关闭全局转义，通过设置app.jinja\_env.autoescape=False即可关闭。如果要渲染由用户提交上来的字符串，强烈建议开启全局转义。

在没有开启自动转义的情况下，对于一些不信任的字符串，可以通过{{ value|escape }}进行局部转义。在开启了自动转义的情况下，对于一些安全的字符串，可以通过{{ value|safe }}进行局部关闭转义。使用autoescape语法可以将一段代码块整体关闭或开启自动转义。示例代码如下。

```
{% autoescape false %}  
<p>这个里面的关闭了自动转义</p>  
<p>{{ will_not_be_escaped }}</p>  
{% endautoescape %}
```

如果将以上代码中的false改成true，将在autoescape代码块中开启自动转义。

## 4.6.2 加载静态文件

一个网页中除了HTML代码以外，还需要CSS、JavaScript和图片文件的综合应用才能更加美观和实用。静态文件默认存放到当前项目的static文件夹中，如果要修改静态文件存放的路径，可以在创建Flask对象时设置static\_folder参数，示例代码如下。

```
app = Flask(__name__, static_folder='C:\static')
```

在模板文件中，可以通过url\_for加载静态文件，示例代码如下。

```
<link href="{{ url_for('static', filename='about.css') }}">
```

url\_for的第1个参数static是固定的，表示生成一个静态文件的URL；第2个参数filename是可以传递的文件名或者文件路径，路径是相对于static或者static\_folder参数自定义的路径。以上代码在被模板渲染后，会被解析成如下形式。

```
<link href="/static/about.css">
```

### 4.6.3 闪现消息

用户在发送一个请求后，网站可能需要给这个用户一些提示，如登录成功提示、登录失败提示，这时可以用闪现消息解决。使用闪现消息，需要先在视图函数中通过flash函数提交消息内容，消息内容可以有多条，然后在模板中再使用get\_flashed\_messages函数获取视图函数中提交的消息内容。get\_flashed\_message函数返回的是一个列表，因此需要用for循环或者通过下标取出消息内容。闪现消息的视图函数部分示例代码如下。

```
@app.route("/flash")
def myflash():
    flash("我是消息内容1...")
    flash("我是消息内容2...")
    return render_template("flash.html")
```

闪现消息的模板部分示例代码如下。

```
<ul>
    {% for message in get_flashed_messages() %}
        <li>{{ message }}</li>
    {% endfor %}
</ul>
```

因为闪现消息是存储在session中的，使用session之前必须要在app.config中设置SECRET\_KEY，如果读者没有设置SECRET\_KEY，那么会出现如图4-9所示的错误信息页。

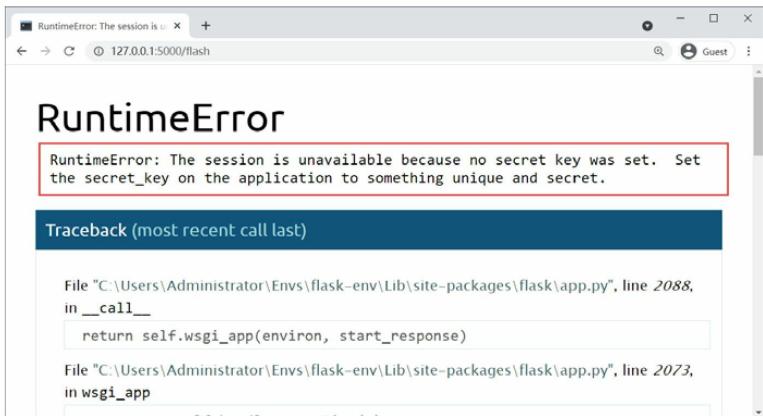


图4-9 错误信息页

我们可以在app.config中随便设置一个字符串，示例代码如下。

```
app.config['SECRET_KEY'] = "ewgnlwe&S;(12zd-+"
```

在刷新浏览器页面后，即可看到闪现消息内容，如图4-10所示。



图4-10 模板中显示闪现消息

## chapter 5 第5章 数据库

数据库是一个动态网站必备的基础功能。通过使用数据库，数据可以被动态地展示、修改、删除等，极大地提高了数据的管理能力，以及数据传递的效率。数据库有很多种，如SQL Server、Oracle、PostgreSQL、MySQL等。其中MySQL数据库由于拥有免费、不受平台限制、灵活度高、稳定性强等优点，已经成为最流行的关系型数据库之一。在本书中使用MySQL来演示数据库的操作。首先用PyCharm Professional创建一个database项目，后续的知识点讲解都将基于这个项目。

### 5.1 准备工作

#### 5.1.1 MySQL软件

本书使用的数据库是目前最新的MySQL 8.0版本，读者在学习后续内容之前，要先下载和安装好该软件。关于MySQL软件的使用方法不在本书的讲解范围内，如果读者对MySQL的下载和安装有疑问，可以参考以下两个链接。

- (1) 官方下载链接为<https://dev.mysql.com/downloads/mysql/>。读者可以根据自己的操作系统，下载不同的MySQL安装包。
- (2) MySQL安装图文教程链接为<https://zlkt.net/book/detail/10/306>。

#### 5.1.2 Python操作MySQL驱动

Flask要操作数据库，必须要先安装Python操作MySQL的驱动。在Python中，目前有以下MySQL驱动包。

- (1) MySQL-python：也就是MySQLdb，是对C语言操作MySQL数据库的一个简单封装，遵循了Python DB API v2，但是只支持Python2。
- (2) mysqlclient：是MySQL-python的另外一个分支。支持Python3并且修复了一些bug，是目前为止执行效率最高的驱动，但是安装的时候容易因为环境问题出错。
- (3) pymysql：纯Python实现的一个驱动。因为是纯Python编写的，因此执行效率不如mysqlclient高。也正因为是纯Python写的，所以可以和Python代码无缝衔接。
- (4) mysql-connector-python：MySQL官方推出的纯Python连接MySQL的驱动，执行效率比pymysql还低。

为了减少读者出错，提高学习效率，本书选择使用pymysql作为驱动程序。读者在学完本章内容后，如果有需要，可以自行考虑移植到mysqlclient。pymysql是一个第三方包，因此需要通过以下命令安装。

```
pip install pymysql
```

### 5.1.3 Flask-SQLAlchemy

在Flask中，我们很少会使用pymysql直接写原生SQL语句去操作数据库，更多的是通过SQLAlchemy提供的ORM技术，类似于操作普通Python对象一样，实现对数据库的增、删、改、查操作。而Flask-SQLAlchemy是对SQLAlchemy的一个封装，这使得在Flask中使用SQLAlchemy更加方便。Flask-SQLAlchemy需要单独安装，因为Flask-SQLAlchemy依赖SQLAlchemy，所以只要安装了Flask-SQLAlchemy， SQLAlchemy便会自动安装。安装命令如下。

```
pip install flask-sqlalchemy
```

SQLAlchemy类似于Jinja2，可以独立于Flask使用，而且完全可以在任何Python程序中使用。SQLAlchemy的功能非常强大，本书不能全部都讲到，读者如果有兴趣，可以在学完本章内容后阅读SQLAlchemy的官方文档<https://www.sqlalchemy.org/>进行深入研究。

## 5.2 Flask-SQLAlchemy的基本使用

### 5.2.1 连接MySQL

在使用Flask-SQLAlchemy操作数据库之前，要先创建一个由Flask-SQLAlchemy提供的SQLAlchemy类的对象。在创建这个类时，要传入当前的app，然后还需要在app.config中设置SQLALCHEMY\_DATABASE\_URI，来配置数据库的连接，示例代码如下。

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
```

```
# MySQL所在的主机名
HOSTNAME = "127.0.0.1"
# MySQL监听的端口号, 默认3306
PORT = 3306
# 连接MySQL的用户名, 读者用自己设置的
USERNAME = "root"
# 连接MySQL的密码, 读者用自己的
PASSWORD = "root"
# MySQL上创建的数据库名称
DATABASE = "database_learn"

app.config['SQLALCHEMY_DATABASE_URI'] = f"mysql+pymysql://{USERNAME}:{PASSWORD}@{HOSTNAME}:{PORT}/{DATABASE}?charset=utf8"

db = SQLAlchemy(app)

# 测试是否连接成功
with db.engine.connect() as conn:
    rs = conn.execute("select 1")
    print(rs.fetchone())
```

Flask-SQLAlchemy在连接数据库时，会从app.config中读取

SQLALCHEMY\_DATABASE\_URI参数，以上代码分别设置了MySQL主机名、端口号、用户名、密码及数据库名称，数据库应该提前在MySQL中创建好。

SQLALCHEMY\_DATABASE\_URI根据不同的数据库有不同的连接方式，MySQL的连接方式如下。

```
mysql+[driver]://[username]:[password]@[host]:[port]/[database]?charset=utf8
```

其中[]中是变量，需要配置时填充进去即可。如果单击运行后，在PyCharm的控制台中打印了(1,)，则说明已经连接成功。

## 5.2.2 ORM模型

对象关系映射（object relationship mapping，简称ORM）是一种可以用Python面向对象的方式来操作关系型数据库的技术，具有可以映射到数据库表能力的Python类我们称之为ORM模型。一个ORM模型与数据库中的一个表相对应，ORM模型中的每个类属性分别对应表的每个字段；ORM模型的每个实例对象对应表中的每条记录。ORM技术提供了面向对象与SQL交互的桥梁，让开发者用面向对象的方式操作数据库，使用ORM模型具有以下优势。

(1) 开发效率高：几乎不需要写原生SQL语句，使用纯Python的方式操作数据库，大大地提高了开发效率。

(2) 安全性高：ORM模型底层代码对一些常见的安全问题，如SQL注入做了防护，比直接使用SQL语句更加安全。

(3) 灵活性强：Flask-SQLAlchemy底层支持SQLite、MySQL、Oracle、PostgreSQL等关系型数据库，但针对不同的数据库，ORM模型的代码几乎一模一样，只需修改少量代码，即可完成底层数据库的更换。

下面用Flask-SQLAlchemy来创建一个User模型，示例代码如下。

```
class User(db.Model):
    __tablename__ = "user"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(100))
    password = db.Column(db.String(100))

db.create_all()
```

以上代码中，首先创建了一个User类，并使它继承自db.Model类，所有ORM模型必须是db.Model的直接或者间接子类。然后通过`__tablename__`属性，指定User模型映射到数据库中表的名称。接着定义了3个db.Column类型的类属性，分别是`id`、`username`、`password`，只有使用db.Column定义的类属性，才会被映射到数据库表中成为字段。在这个User模型中，`id`是db.Integer类型，在数据库中将表现为整型，并且传递`primary_key=True`参数指定`id`作为主键，传递`autoincrement=True`参数设置`id`为自增长。`username`和`password`属性分别指定其类型为db.String类型，在数据库中将表现为varchar类型，并且指定其最大长度为100。

最后通过`db.create_all()`把User模型映射成数据库中的表。我们可以通过navicat软件来查看数据库中的表，如图5-1所示。

The screenshot shows the 'user' table structure in Navicat. The table has three columns: 'id' (int, primary key, auto-increment), 'username' (varchar, length 100), and 'password' (varchar, length 100). The 'id' column is highlighted.

名	类型	长度	小数点	不是 null	虚拟	键	注释
<code>id</code>	int			<input checked="" type="checkbox"/>	<input type="checkbox"/>		<code>1</code>
<code>username</code>	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>		
<code>password</code>	varchar	100		<input type="checkbox"/>	<input type="checkbox"/>		

图5-1 navicat中查看user表

创建数据库的字段类型，除了以上的db.Integer和db.String，还有以下字段类型，如表5-1

所示。

表5-1 Flask-SQLAlchemy字段类型

类 型	描 述
db.Integer	整型。范围与数据库一致
db.SmallInteger	小整型。范围与数据库一致
db.BigInteger	长整型。范围与数据库一致
db.Decimal	定点类型。可以指定总长度和小数点后位数
db.Boolean	布尔类型
db.Date	日期类型。存储 Python 中的 datetime.date 对象
db.DateTime	日期时间类型。存储 Python 中的 datetime.datetime 对象
db.Time	时间类型。存储 Python 中的 datetime.time 对象
db.Interval	时间间隔。存储 Python 中的 datetime.timedelta 对象
db.String	字符串类型。使用时需要指定长度，不能超过 255 个字符
db.Text	文本类型。常用于字符串长度不可控的情况
db.Enum	枚举类型
db.PickleType	存储经过 Pickle 后的对象
db.LargeBinary	存储二进制数据

字段在数据库中的表现，都是通过 db.Column 上的参数实现的。db.Column 常用参数如表 5-2 所示。

表5-2 db.Column常用参数

参 数	描 述
name	字段在数据库表中的名称。如果没有设置，则使用此属性名作为字段名称
type_	字段类型
autoincrement	自动增长
default	默认值
index	如果设置为 True，则将此字段设置为索引
nullable	是否为空
onupdate	在修改对象时，会自动使用这个属性指定的值
primary_key	主键
unique	如果设置为 True，则此字段的值必须唯一
comment	在创建表时的注释

### 5.2.3 CRUD 操作

使用ORM进行CRUD（create、read、update、delete）操作，需要先把操作添加到会话中，通过db.session可以获取到会话对象。会话对象存在内存中，如果要把会话中的操作提取到数据库中，需要调用db.session.commit()操作；如果要把会话中的操作回滚，则需要调用db.session.rollback()实现。下面分别对CRUD操作进行讲解。

## 1. Create操作

使用ORM创建一条数据非常简单，先使用ORM模型创建一个对象，然后添加到会话中，再进行commit操作即可，示例代码如下。

```
@app.route('/user/add')
def user_add():
    user1 = User(username="张三",password="111111")
    user2 = User(username="李四",password="222222")
    user3 = User(username="王五",password="333333")
    db.session.add(user1)
    db.session.add(user2)
    db.session.add(user3)
    db.session.commit()
    return "用户添加成功!"
```

在以上代码中，首先用User类创建了3个对象，在创建对象时，必须通过关键字参数给字段赋值，否则SQLAlchemy将不知道是给哪个字段赋值，从而报错。由于id是作为一个自增长的主键，因此可以不需要赋值。然后再把3个对象添加到session中，最后再统一进行commit操作，即可把数据添加到数据库中。

## 2. Read操作

Read就是查询操作。ORM模型都是继承自db.Model，db.Model内置的query属性上有许多方法，可以实现对ORM模型的查询操作。query上的方法可以分为两大类，分别是提取方法和过滤方法。首先来看提取方法，示例代码如下。

```
@app.route('/user/fetch')
def user_fetch():
    # 1. 获取User中所有数据
    users = User.query.all()

    # 2. 获取主键为1的User对象
    user = User.query.get(1)
```

```
# 3. 获取第一条数据
user = User.query.first()

return "数据提取成功!"
```

在以上代码中，通过all()方法获取所有User对象，通过get方法获取指定主键的User对象，通过first()方法获取第一个User对象。提取数据的常用方法如表5-3所示。

表5-3 提取数据的常用方法

方 法 名	描 述
query.all()	获取查询结果集中的所有对象，是列表类型
query.first()	获取查询结果集中的第一个对象
query.one()	获取查询结果集中的一个对象，如果结果集中对象数量不等于 1，则会抛出异常
query.one_or_none()	与 query.one()方法类似，结果集中对象数量不等于 1 时，不是抛出异常，而是返回 None
query.get(pk)	根据主键获取当前 ORM 模型的第一条数据
query.exists()	判断数据是否存在
query.count()	获取查询结果集的个数

在查询数据时，经常需要做过滤操作。过滤最常用的两个方法是filter和filter\_by，filter方法传递查询条件，filter\_by方法传递关键字参数，示例代码如下。

```
@app.route('/user/filter')
def user_filter():
    # 1. filter方法:
    users = User.query.filter(User.username=="张三").all()

    # 2. filter_by方法:
    users = User.query.filter_by(username="张三").all()

    return "数据过滤成功!"
```

除了filter和filter\_by方法以外，Flask-SQLAlchemy还提供了以下过滤方法，如表5-4所示。

表5-4 常用过滤方法

方法名	描述
query.filter()	根据查询条件过滤
query.filter_by()	根据关键字参数过滤
query.slice(start,stop)	对结果进行切片操作
query.limit(limit)	对结果数量进行限制
query.offset(offset)	在查询时跳过前面 offset 条数据
query.order_by()	根据给定字段进行排序
query.group_by()	根据给定字段进行分组

在表5-4中，query.slice(start,stop)、query.limit(limit)、query.offset(offset)方法的使用比较简单，这里不做过多讲解。下面讲解query.order\_by()和query.group\_by()的用法。

(1) query.order\_by()的用法如以下代码所示。

```
@app.route('/user/filter')
def user_filter():
    ...
    # query.order_by方法
    # 正序排序
    users = User.query.order_by("id")
    users = User.query.order_by(User.id)

    # 倒序排序
    users = User.query.order_by(db.text("-id"))
    users = User.query.order_by(User.id.desc())

    from sqlalchemy import desc
    users = User.query.order_by(desc("id"))

    for user in users:
        print(user.id)

    return "数据过滤成功!"
```

以上代码中，以User的id字段为例（可以换成任何其他的字段），详细地罗列了正序和倒序排序的方法，读者在开发过程中可自行选择合适的方法实现排序。

(2) query.group\_by()方法是根据某个字段进行分组，分组的主要目的是获取分组后的数量、最大值、最小值、平均值、总和等。因为提取的数据不再是某个模型，所以不能通过<模型>.query的方式获取，而是通过db.session.query来提取，如要获取所有用户名在表中存在的个数，那么可以通过以下代码实现。

```
@app.route('/user/filter')
def user_filter():
    ...
    # query.group_by方法
    from sqlalchemy import func
    users = db.session.query(User.username,func.count(User.id)) .
group_by("username").all()

    return "数据过滤成功!"
```

过滤是数据提取的一个很重要的功能，除了直接使用==和!=关系运算符外，还可以使用以下常用的过滤条件进行过滤，但这些过滤条件只能通过filter方法实现。常用的过滤条件如下。

(1) like: 模糊查询，使用方式与SQL语句中的like类似，可以在搜索字符左右两边添加%来匹配任意字符。contains方法相当于like在搜索字符左右两边都添加%，例如，contains("张")与like("%张%")是一样的效果。示例代码如下。

```
users = User.query.filter(User.username.contains("张"))
users = User.query.filter(User.username.like("%张%"))
```

(2) in: 判断值是否在指定数据集中，如果是就提取，否则就不提取。为了不与Python中的in关键字混淆，在in后加下画线，实际的方法名为in\_，示例代码如下。

```
users = User.query.filter(User.username.in_(['张三','李四','王五']))
```

(3) not in: 作用与in相反。其使用方式是在in\_方法所在代码表达式前添加(~)，示例代码如下。

```
users = User.query.filter(~User.username.in_(['张三']))
```

(4) is null: 判断值是否为空，如果为空就提取，否则就不提取。可以通过判断值是否为none，或通过is\_方法实现，为了不与Python中的in关键字混淆，同样需要在is后加下画线，示例代码如下。

```
users = User.query.filter(User.username==None)
users = User.query.filter(User.username.is_(None))
```

(5) `is not null`: 作用与`is null`相反, 实际的方法名为`isnot`, 示例代码如下。

```
users = User.query.filter(User.username != None)
users = User.query.filter(User.username.isnot(None))
```

(6) `and`: 用于同时满足多条件的查询, 实际的方法名为`and_`, 示例代码如下。

```
from sqlalchemy import and_
users = User.query.filter(and_(User.username=="张三",User.id < 10))
```

(7) `or`: 用于满足一个或多个条件的查询, 实际的方法名为`or_`, 示例代码如下。

```
from sqlalchemy import or_
users = User.query.filter(or_(User.username=="张三",User.username=="李四"))
```

### 3. update操作

更新操作分为两种, 第一种针对一条数据, 第二种针对多条数据。针对一条数据, 可以直接修改对象的属性, 然后执行`commit`操作即可, 示例代码如下。

```
user = User.query.get(1)
user.username = "张三_重新修改的"
db.session.commit()
```

针对修改多条数据的情况, 则是通过调用`filter`或者`filter_by`方法获取`BaseQuery`对象, 然后再调用`update`方法, 实现批量修改的, 示例代码如下。

```
User.query.filter(User.username.like("%张三%")).update({"password":User.
password+"_被修改"},synchronize_session=False)
db.session.commit()
```

以上代码先通过filter方法过滤数据，然后再调用update方法，在所有的password后面都添加"\_被修改"字符串，并且因为使用了like方法作为过滤条件，所以需要指定 synchronize\_session参数为False，最后再调用commit()方法即可批量完成数据的修改。

## 4. delete操作

删除操作也是分成两种。第一种是删除一条数据，第二种是删除多条数据。删除单条数据的操作方式非常简单，直接调用db.session.delete方法即可，示例代码如下。

```
user = User.query.get(1)
db.session.delete(user)
db.session.commit()
```

删除多条数据的操作方式类似更新多条数据，通过BaseQuery的delete方法即可实现，示例代码如下。

```
User.query.filter(User.username.contains("张三")).delete(synchronize_
session=False)
db.session.commit()
```

## 5.3 表关系

关系型数据库的一个强大的功能，就是多张表之间可以建立关系。如文章表中，通常需要保存作者数据，但是我们不需要直接把作者数据放到文章表中，而是通过外键引用用户表。这种强大的表关系，可以存储非常复杂的数据，并且可以使查询非常迅速。在Flask-SQLAlchemy中，同样也支持表关系的建立，表关系建立的前提，是通过数据库的外键实现的。表关系总体来讲可以分为3种：一对多（多对一）、一对一、多对多。下面分别进行讲解。

### 5.3.1 外键

外键是数据库的技术，Flask-SQLAlchemy中支持在创建ORM模型时就指定外键，创建外键是通过db.ForeignKey实现的。如创建Article表，这张表有一个author\_id字段，通过外键

引用user表的id字段，用来保存文章的作者，那么Article的模型代码如下。

```
class Article(db.Model):
    __tablename__ = "article"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey("user.id"))
```

以上代码，除了添加常规的title、content属性外，还增加了一个author\_id，author\_id通过db.ForeignKey("user.id")引用了之前创建的user表的id字段。这里有个细节需要注意，author\_id因为引用user表的id字段，所以它的类型必须跟user表的id字段一致，否则会报错。

### 5.3.2 一对多关系

我们生活中有很多一对多的例子，如CSDN博客中的一篇文章只能属于一个作者，一个作者能发布多篇文章，作者和文章之间是一对多的关系，反过来文章和作者之间是多对一的关系。

#### 1. 建立关系

5.3.1节中通过外键，实际上已经建立起一对多的关系，即一篇文章只能引用一个作者，而一个作者可以被多篇文章引用。但是以上只是建立了一个外键，通过Article的对象还是无法直接获取到author\_id引用的那个User对象。为了使操作ORM对象与操作普通Python对象一样，Flask-SQLAlchemy提供了db.relationship来引用外键所指向的那个ORM模型。在以上的Article模型中添加db.relationship，示例代码如下。

```
class Article(db.Model):
    __tablename__ = "article"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey("user.id"))
    author = db.relationship("User")
```

我们添加了一个author属性，这个属性通过db.relationship与User模型建立了联系，以后

通过Article的实例对象访问author时，如article.author，那么Flask-SQLAlchemy会自动根据外键author\_id从user表中寻找数据，并形成User模型实例对象。下面通过创建Article对象，并通过访问Article实例对象的author属性来关联User对象，示例代码如下。

```
@app.route('/article/add')
def article_add():
    user = User.query.first()
    article = Article(title="aa", content="bb", author=user)
    db.session.add(article)
    db.session.commit()

    article = Article.query.filter_by(title="aa").first()
    print(article.author.username)
```

以上代码中，首先创建了一个article对象，并添加到数据库中，接下来再从数据库中提取，然后通过article.author.username访问到article对象的用户名。

## 2. 建立双向关系

现在的Article模型可以通过author属性访问到对应的User实例对象，但是User实例对象无法访问到和其关联的所有Article实例对象。因此为了实现双向关系绑定，还需要在User模型上添加一个db.relationship类型的articles属性，并且在User模型和Article模型双方的db.relationship上都需要添加一个back\_populates参数，用于绑定对方访问自己的属性，示例代码如下。

```
class User(db.Model):
    __tablename__ = "user"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(100))
    password = db.Column(db.String(100))

    articles = db.relationship("Article", back_populates="author")

class Article(db.Model):
    __tablename__ = "article"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey("user.id"))
    author = db.relationship("User", back_populates="articles")
```

在User端绑定了articles属性后，现在双方都能通过属性直接访问到对方了，示例代码如下。

```
user = User.query.first()
for article in user.articles:
    print(article.title)
```

### 3. 简化关系定义

以上User和Article模型中，通过在两边的db.relationship上传递back\_populates参数来实现双向绑定，这种方式有点烦琐，我们还可以通过只在一个模型上定义db.relationship类型属性，并且传递backref参数实现双向绑定，示例代码如下。

```
class User(db.Model):
    __tablename__ = "user"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(100))
    password = db.Column(db.String(100))

class Article(db.Model):
    __tablename__ = "article"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)

    author_id = db.Column(db.Integer, db.ForeignKey("user.id"))
    author = db.relationship("User", backref="articles")
```

以上代码中，我们删除了User模型上的articles属性，并且在Article模型上将author属性的db.relationship中的back\_populates修改为backref。backref参数的功能更加强大，其可以自动给对方添加db.relationship的属性。

这种方式虽然方便，但是在模型比较多、项目团队人数较多的情况下，也容易造成困扰。如User模型上根本没有看到定义的articles属性，但是却在Article模型上创建了，这着实会让人摸不着头脑。因此为了更加直观和方便团队协作，建议使用back\_populates来实现双向绑定。

#### 5.3.3 一对关系

要实现一对多关系，只需要在一对多的基础之上，将“多”的那一端设置为“一”即可，在Flask-SQLAlchemy中，通过给db.relationship传递uselist=False，即可将“多”设置为“一”，为了在数据库层面也实现一对一，还需要在外键上设置unique=True。这里以用户拓展表为例。在公司业务增长的情况下，需要存储用户的许多属性，但是有些属性是不常用的，为了提高网站的响应速度，我们会把那些不常用的属性放到拓展表中，只在需要的时候才访问。用户表和用户拓展表就是典型的一对一关系，一个用户只能有一条拓展数据，一条拓展数据只能属于一个用户。下面新增一个UserExtension模型，与原来的User模型建立一对一的关系，示例代码如下。

```
class User(db.Model):
    __tablename__ = "user"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(100))
    password = db.Column(db.String(100))

    extension = db.relationship("UserExtension", back_populates="user",
                               uselist=False)

class UserExtension(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    school = db.Column(db.String(100))
    user_id = db.Column(db.Integer, db.ForeignKey("user.id"), unique=True)
    user = db.relationship("User", back_populates="extension")
```

User和UserExtension的关系中，因为外键是添加到UserExtension模型上，因此User模型属于“多”的那一端，这时就设置uselist=False，即可将“多”转化为“一”。为了在数据库层面也实现一对一，将UserExtension模型上的user\_id属性设置为unique=True。此时如果要在同一个User对象上添加多个UserExtension对象，那么就会抛出异常，示例代码如下。

```
@app.route("/one2one")
def one2one():
    user = User.query.first()
    extension1 = UserExtension(school="清华大学", user=user)
    extension2 = UserExtension(school="北京大学", user=user)
    db.session.add(extension1)
    db.session.add(extension2)
    db.session.commit()
    return "一对一成功!"
```

上述代码中，我们试图在两个UserExtension实例对象上绑定同一个User对象，但是因为

设置了一对一的关系，因此以上代码将会抛出类似以下的异常。

```
sqlalchemy.exc.IntegrityError: (pymysql.err.IntegrityError) (1062,
"Duplicate entry '1' for key 'user_extension.user_id'")
[SQL: INSERT INTO user_extension (school, user_id) VALUES (%(school)s,
%(user_id)s)]
[parameters: {'school': '北京大学', 'user_id': 1}]
```

### 5.3.4 多对多关系

多对多关系在数据库层面是需要通过一张中间表来实现的，在Flask-SQLAlchemy中也是一样。这里以文章和标签为例，一篇文章可以添加多个标签，一个标签可以被多篇文章添加。我们创建标签Tag类，示例代码如下。

```
article_tag_table = db.Table(
    "article_tag_table",
    db.Column("article_id",db.Integer,db.ForeignKey("article.id"),primary_
key=True),
    db.Column("tag_id",db.Integer,db.ForeignKey("tag.id"),primary_key=True)
)

class Article(db.Model):
    __tablename__ = "article"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)
    author_id = db.Column(db.Integer, db.ForeignKey("user.id"))
    author = db.relationship("User", backref="articles")
    tags = db.relationship("Tag", secondary=article_tag_table,
    back_populates="articles")

class Tag(db.Model):
    __tablename__ = "tag"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(100))
    articles = db.relationship("Article", secondary=article_tag_table,
    back_populates="tags")
```

为了实现Article和Tag之间的多对多关系，我们使用db.Table创建了一张中间表

article\_tag\_table，并且添加了article\_id和tag\_id两个外键来分别与article和tag表进行关联。然后在Article和Tag类中分别添加了tags和articles属性，用来建立双向关系，并且在db.relationship中传递secondary=article\_tag\_table参数来绑定中间表。Article和Tag的多对多关系建立后，可以通过以下代码来添加数据。

```
@app.route('/many2many')
def many2many():
    article1 = Article(title="11",content="aa")
    article2 = Article(title="22", content="bb")

    tag1 = Tag(name="python")
    tag2 = Tag(name="flask")

    article1.tags.append(tag1)
    article1.tags.append(tag2)

    article2.tags.append(tag1)
    article2.tags.append(tag2)

    db.session.add_all([article1,article2])
    db.session.commit()
    return "多对多数据添加成功!"
```

上述代码中首先分别创建了两个Article对象和两个Tag对象，然后把两个Tag对象分别添加到article1和article2中，最后通过db.session.add\_all方法把article1和article2添加到会话中，然后执行commit操作。因为两个Tag对象都已经与article1和article2进行关联了，在article1和article2被添加到会话中后，两个Tag对象也会被添加到会话中。在多对多关系中，添加对象使用的是append方法，移除对象使用的是remove方法。

### 5.3.5 级联操作

级联操作（cascade）是在操作某个对象时，相关联的对象也会进行对应的操作。在数据库层面的级联操作包括级联删除、级联更新等。Flask-SQLAlchemy提供了比数据库更强大的级联操作，定义级联操作是通过对db.relationship传递cascade参数实现的，这个参数的值可以为all、save-update、merge、refresh-expire、expunge、delete中的一个或者多个，如果是多个，则通过英文逗号隔开，如“save-update, delete”。谁设置了cascade参数，谁就是父表，父表数据发生变化，相关联的从表也会执行相应操作。这里为了不影响之前的模型，我们创建两个新的ORM模型来讲解级联操作，示例代码如下。

```
class Category(db.Model):
    __tablename__ = "category"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(100))
    newses = db.relationship("News", back_populates="category")

class News(db.Model):
    __tablename__ = "news"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(100))
    content = db.Column(db.Text)
    category_id = db.Column(db.Integer, db.ForeignKey("category.id"))
    category = db.relationship("Category", back_populates="newses")
```

以上代码中创建了两个模型，分别是新闻分类模型Category和新闻模型News，并且在双方都建立了关系。下面分别来讲解级联操作常用的值。

## 1. save-update

save-update是默认选项，它的作用是当某个对象被添加到会话中时，与此对象相关的对象也会被添加进去，示例代码如下。

```
category = Category(name="军事")
news = News(title="新闻1", content="新闻内容1")
news.category = category
db.session.add(news)
db.session.commit()
```

以上代码中创建了category和news对象，因为让news和category进行了关联，因此只要添加news，那么category就会被自动添加进去。在News模型中创建category时设置cascade="",示例代码如下。

```
class News(db.Model):
    ...
    category = db.relationship("Category", back_populates="newses",
        cascade="")
```

重新执行上述添加news和category的代码，会在PyCharm控制台出现以下错误。

```
SAWarning: Object of type <Category> not in session, add operation along
'News.category' won't proceed
db.session.commit()
```

查看数据库后发现只有news数据被添加了，category并没有被添加进去。这说明一旦cascade没有设置save-update，那么被关联的对象就不会被添加到会话中。

## 2. delete

delete表示当删除某个对象时，被关联的所有对象都会被删除。这个值默认在cascade中是没有的。

```
news = News.query.first()
db.session.delete(news)
db.session.commit()
return "success"
```

## 3. delete-orphan

delete-orphan表示某个对象被父表解除关联时，此对象也会自动被删除。当然，如果父表中的数据被删除，此对象也会被删除。如某个News对象被从Category.news上删除，则这个News对象也会被删除。将Category的news属性修改为如下所示的代码。

```
class Category(db.Model):
    ...
    newses = db.relationship("News",back_populates="category",
                           cascade="delete,delete-orphan")
```

然后再执行删除操作，示例代码如下。

```
category = Category.query.first()
news = News(title="新闻2",content="新闻内容2")
category.newses.append(news)
db.session.commit()

# 将news从category中解除关联
category.newses.remove(news)
```

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

以上代码中，首先将news添加到category.newsес中，然后通过commit操作提交到数据库中。接着从category.newsес中移除，这样就把news从category上解除了关联，因为在Category中定义newsес属性时，设置了cascade为delete-orphan，那么一旦解除关联，news对象就成为孤儿（orphan）对象，即会自动从数据库中被删除。这个选项一般用在一对多关系上，不能用在多对多以及多对一关系上。如例子中的Category和News，Category属于“一”，News属于“多”。删除分类，该分类下的新闻也被删除了，这符合常理，但是如果新闻被删除了，分类也跟着删除，这就会造成数据混乱。

## 4. merge

merge是默认选项。在使用session.merge合并对象时，会将使用了db.relationship相关联的对象也进行merge操作。

这个参数几乎很少用到，读者作为了解即可。

## 5. expunge

进行移除操作时，会将相关联的对象也进行移除。这个操作只是将对象从session中移除，并不会真正地从数据库中删除。

我们首先将News模型中category属性的cascade参数修改为如下。

```
class News(db.Model):
    ...
    category = db.relationship("Category", back_populates="newses",
        cascade= "expunge")
```

然后执行以下操作。

```
news = News.query.first()
category = news.category
db.session.expunge(news)
category.name = '测试分类'
db.session.commit()
```

上述代码中，使用db.session.expunge方法将news对象从session中移除，因为news和category级联关系中设置了expunge选项，所以category对象也会跟着从session中移除，此时再去修改category.name的值，就不会同步到数据库中了。读者可以在执行上述代码前观察第一条新闻分类的名称，执行完上述代码后再观察，会发现分类名称没有发生变化。

## 6. all

all是对save-update、merge、expunge、delete的缩写，不包含delete-orphan。

## 7. 默认值

cascade在没有被修改时的默认值是save-update和merge。

## 5.4 ORM模型迁移

ORM模型定义好后，是通过db.create\_all将ORM模型映射到数据库中的。这种方式是有局限性的，它只能识别到新增了模型后映射到数据库中的对于模型中字段的修改，对于类型的修改，无法识别到。因此在实际开发中，都不会使用db.create\_all来做ORM模型迁移，而是借助一个第三方插件Flask-Migrate来实现。Flask-Migrate是基于alembic实现的，alembic是专门用来给SQLAlchemy的ORM模型做迁移的。要使用Flask-Migrate，首先需要通过pip命令安装。

```
pip install flask-migrate
```

alembic会随着flask-migrate安装而自动安装。在完成flask-migrate安装后，接下来讲解如何配置。

### 5.4.1 创建迁移对象

首先看创建迁移对象的代码，如下所示。

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
```

```
...  
db = SQLAlchemy(app)  
migrate = Migrate(app, db)
```

以上代码中，首先从flask\_migrate包中导入Migrate类，然后实例化这个类，在实例化时传入app和db对象，并且赋值给migrate变量。后续在执行迁移命令时，Flask-Migrate会自动读取app.py中的migrate变量，所以变量名必须为migrate。

## 5.4.2 初始化迁移环境

在创建完迁移对象后，需要初始化迁移环境。方法是在当前项目的根路径下执行如下命令。

```
flask db init
```

命令执行完成后，会在项目的根路径下生成一个migrations文件夹，在这个文件夹下有以下文件或文件夹。

- (1) **versions**: 文件夹，用于存放后面生成的迁移脚本文件。由于目前没有生成过任何迁移脚本，因此是一个空的文件夹。
- (2) **alembic.ini**: alembic的配置文件。
- (3) **env.py**: 配合Flask项目进行迁移的Python文件。
- (4) **script.py.mako**: 生成迁移脚本的模板文件。

以上4个文件或文件夹，除非你自己非常清楚要做什么，否则强烈建议不要自行修改里面的内容。到目前为止，初始化迁移环境的工作就已经完成。此工作只需做一次，后续只要不断生成迁移脚本和映射脚本即可，无须重复初始化。

## 5.4.3 生成迁移脚本

在初始化完迁移环境的前提下，无论是新增了ORM模型，或者是ORM模型中有任何字段信息发生改变，并且要将这些改变同步到数据库中，都要做的一件事情就是将当前的修改生成一个迁移脚本，生成迁移脚本的命令如下。

```
flask db migrate -m "备注信息"
```

以上命令中参数-m后面跟的是备注信息，通过添加备注信息，可方便以后查看当前迁移脚本做了哪些事情。当然，备注信息不是必需的，如果不想添加，则把-m参数以及后面的内容都删除即可。笔者强烈建议添加备注信息，特别是多人合作开发一个项目时，添加备注信息能让工作更加透明。在执行完以上命令后，可以看到versions文件夹中新增了一个Python脚本文件，这个脚本文件中记录了此次修改的变更内容。

#### 5.4.4 执行迁移脚本

迁移脚本只是写好了表变更的内容，但是并没有更新数据库。因此还需要执行迁移脚本将这些改变真正映射到数据库中，执行迁移脚本的命令如下。

```
flask db upgrade
```

以上命令会自动从versions文件夹中寻找最新的迁移脚本文件，然后执行迁移脚本文件中的upgrade函数。在这步工作完成后，模型的修改就能真正映射到数据库中了。

使用flask-migrate做ORM模型迁移时，有一点需要注意，被迁移的ORM模型必须被app.py直接或间接加载。如为了代码更加有序，我们一般会把ORM模型放到models.py文件中，如果这个models.py文件没有被app.py直接或间接加载，那么其中的ORM模型将不能被flask-migrate识别到，也就不会参与迁移。

## chapter 6 第6章 表单

表单是一个网站与用户交互必不可少的元素。表单中可以提供文本输入框、单选按钮、复选框、按钮等元素供用户提交数据。在Flask项目中，表单除了可以表示传统的HTML标签外，还有验证数据的作用。数据被发送到服务器后，服务器为了防止不法分子绕过前端限制提交一些非法数据，需要对提交上来的数据进行验证，验证合法后才进行后续的操作。要实现表单的验证功能，我们需要借助第三方插件Flask-WTF，Flask-WTF是对WTForms库的封装，让WTForms库在Flask项目中更方便地被使用，不过Flask-WTF提供的功能比较有限，大部分功能是直接从WTForms中直接导入的。WTForms的功能主要有两个，分别是验证数据和在模板中渲染表单HTML标签。当然，WTForms还包括一些其他功能，如CSRF保护、文件上传等。安装Flask-WTF的同时默认也会安装WTForms，安装命令如下。

```
pip install flask-wtf
```

安装完Flask-WTF后，我们用PyCharm Professional版创建一个名叫formlearn的项目，读者可以通过本项目查看到本章所讲内容的演示代码。

### 6.1 表单验证

这里以注册功能为例，讲解表单验证功能。注册时需要提交邮箱、用户名、密码、确认密码4个字段的数据。首先在templates文件夹中创建一个register.html文件，然后输入以下代码。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>注册</title>
</head>
<body>
    <form action="{{ url_for('register') }}" method="POST">
        <table>
            <tr>
                <td>用户名: </td>
                <td><input type="text" name="username"></td>
            </tr>
            <tr>
                <td>邮箱: </td>
                <td><input type="email" name="email"></td>
            </tr>
            <tr>
                <td>密码: </td>
                <td><input type="password" name="password"></td>
            </tr>
            <tr>
                <td>确认密码: </td>
                <td><input type="password" name="confirm_password"></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="submit" value="提交"></td>
            </tr>
        </table>
    </form>
</body>
</html>
```

以上代码中，首先创建了一个form标签，然后设置action为url\_for('register')，也就是将register视图函数反转为URL，在以后单击“提交”按钮时，会把所有form标签下输入框内的内容都提交给这个URL（看下文register视图函数）。接着还设置了method为POST，这意味着会以POST方式提交。然后在form标签下，分别添加了属性name为username、email、password以及confirm\_password的input标签。最后添加了一个type="submit"的input标签，被渲染出来的是一个按钮。

模板写好后，再用一个视图函数渲染，示例代码如下。

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

```
if request.method == 'GET':  
    return render_template("register.html")  
else:  
    pass
```

执行以上代码后，在浏览器中访问`http://127.0.0.1:5000/register`，即可看到如图6-1所示的效果。



图6-1 注册页面

### 6.1.1 表单类编写

到目前为止，我们完成了前端模板代码的编写，用户可以在此页面输入信息进行注册了。但是在此页面中，对每个字段是有一定要求的，具体要求如下。

- 用户名：为了防止重名，一般要求最少要输入3位以上字符。
- 邮箱：格式必须以@+域名结尾。
- 密码：要求最少输入6位以上字符。
- 确认密码：该字段内容必须和密码字段内容一致。

我们不能要求用户一次性就正确输入满足这些规则的内容，如果用户输入错误，应该在界面中及时给予提示。这个工作可以由前端通过JavaScript来完成，但是服务器端也要做好验证，因为对于有一定技术功底的用户来说，可以通过抓包的形式获取注册时的请求数据，然后通过代码或者工具来模拟注册，这就完全绕开了前端的JavaScript验证。服务器端的验证可以通过WTForms来实现。首先在项目根路径下创建一个forms.py文件，然后写入以下代码。

```
from wtforms import Form, StringField  
from wtforms.validators import length, email, equal_to
```

```

class RegisterForm(Form):
    username = StringField(validators=[length(min=3,max=20,message="请输入正确长度的用户名! ")])
    email = StringField(validators=[email(message="请输入正确格式的邮箱! ")])
    password = StringField(validators=[length(min=6,max=20,message="请输入正确长度的密码! ")])
    confirm_password = StringField(validators=[equal_to
    ("password",message="两次密码不一致! ")])

```

上述代码中，先从wtforms中导入Form基类，所有的表单类都必须继承自Form基类。然后在RegisterForm中分别添加了username、email、password以及confirm\_password这4个字段，这里字段的名称必须和HTML模板中表单元素的name的值一致。如在HTML模板中邮箱的input标签的name值为email，那么在RegisterForm中字段的名称也必须为email。

这4个属性现在都是字符串类型，因此使用StringField类型，除StringField外，还有以下类型的Field类，如表6-1所示。

表6-1 Field类的类型

字段类型	描述
StringField	字符串类型
IntegerField	整型类型
FloatField	浮点类型
DecimalField	定点类型
BooleanField	布尔类型
DateTimeField	日期时间类型
DateField	日期类型
TimeField	时间类型
FileField	文件类型

每个字段都传递了validators参数，这个参数是可以存储多个验证器的集合。不同的字段应根据实际需要设置不同的验证器。

- username:** 添加了length验证器，用来规定最短字符串长度为3，最长字符串长度为20，并且如果上传的值不在这个范围，会提示一个错误信息，错误信息的内容就是message指定的值。
- email:** 添加了email验证器，email验证器会自动验证上传的值是否满足邮箱的格式规则。如果不满足，同样也会提示message指定的错误信息。

- password: 同样用的是length验证器，指定字符长度为6~20。
- confirm\_password: 确认密码用的是equal\_to验证器，验证是否和password的值一致。

除了以上指定的length、email和equal\_to验证器，WTForms还提供了以下验证器，如表6-2所示。

表6-2 WTForms常用验证器

验证器	描述
length(min,max,message)	验证长度是否在区间内
email()	验证内容是否满足邮箱格式规则
equal_to(fieldname,message)	验证是否和另外一个字段的值相等
ip_address(ipv4,ipv6,message)	验证是否满足 IP 地址的规则
mac_address(message)	验证是否满足 mac 地址的规则
number_range(min,max,message)	验证数字是否在指定的区间内
optional(strip_whitespace)	设置数据可以为空，并停止其他验证器的验证
input_required(message)	验证是否为空
data_required(message)	验证是否有效
url(message)	验证是否满足 URL 规则
any_of(values,message,values_formatter)	验证是否是 values 中的一个
none_of(values,message,values_formatter)	验证是否不是 values 中的一个
regexp(regex,flags,message)	自己指定正则表达式验证

其中，input\_required和data\_required验证器用户经常会混淆，input\_required只是验证字段是否有值，而data\_required则是验证数据是否有效。如字段的类型是IntegerField，但是传入的数据不能被转换为整型时，则会验证失败。

### 注意

表6-2中的验证器都是小写形式，这个是为了方便使用，它们原始的名称遵循驼峰命名法，如length的原始名称为Length、input\_required的原始名称为InputRequired等。

## 6.1.2 视图函数中使用表单

表单写完后，就可以在视图函数中对数据进行验证了。这里以继续完善register视图函数

为例，来讲解表单在视图函数中的使用。

```
from flask import Flask, request, render_template, redirect, url_for, flash
from forms import RegisterForm
...
@app.route("/register", methods=['GET', 'POST'])
def register():
    if request.method == 'GET':
        return render_template("register.html")
    else:
        # request.form 是 html 模板提交上来的表单数据
        form = RegisterForm(request.form)
        # 如果表单验证通过
        if form.validate():
            email = form.email.data
            username = form.username.data
            password = form.password.data

            # 以下是可以把数据保存到数据库的操作
            print("email:", email)
            print("username:", username)
            print("password:", password)
            return "注册成功！"
        else:
            for errors in form.errors.values():
                for error in errors:
                    flash(error)
            return redirect(url_for("register"))
```

以上代码中，先使用RegisterForm创建了一个form对象，并且把request.form作为参数传给RegisterForm，request.form是一个类字典类型，以键一值对的形式保存了从浏览器中提交上来的表单数据。然后再调用form.validate()方法判断RegisterForm中定义的所有字段是否都验证通过，如果是，则通过form.<字段名>.data来获取对应字段的数据，这里是从form对象上获取数据而不是从request.form上获取数据的原因是，服务器获取从浏览器提交上的数据，其本质上都是字符串类型，所以request.form上所有数据都是字符串类型，但是通过form对象获取的数据则是经过处理后的，如某个字段是IntegerField，那么通过form对象获取到的则是整型。以上代码有个小细节，在视图中不需要获取confirm\_password的值，原因是confirm\_password字段存在的意义就是为了验证其值和password是否一致，如果表单验证通过，则意味着confirm\_password与password是相等的，所以没有必要再重新获取一次了。在所有数据都获取到后，就可以把数据存储到数据库中，或者再做其他操作。

如果表单验证失败，则可以通过form.errors获取错误信息。form.errors是字典类型，key

是字段名称，value是错误信息的列表。如在注册表单中什么都不输入，直接单击“提交”按钮，则form.errors的值如下所示。

```
{'username': ['请输入正确长度的用户名!'], 'email': ['请输入正确格式的邮箱!'],  
'password': ['请输入正确长度的密码!']}
```

所以在表单验证失败的情况下，首先通过循环form.errors.values()获取所有错误内容，并存储到flash中。然后在模板中把flash消息显示出来，register.html修改后的代码如下。

```
...  
    <ul>  
        { % for message in get_flashed_messages() %}  
            <li>{{ message }}</li>  
        { % endfor %}  
    </ul>  
  </form>  
</body>  
</html>
```

### 注意

使用flash消息，必须先在app上配置SECRET\_KEY，或直接通过app.secret\_key来设置秘钥。

如果在浏览器中访问http://127.0.0.1:5000/register，然后在表单不输入任何信息，直接单击“提交”按钮，那么网页将展现出如图6-2所示的效果。



图6-2 显示表单错误消息

### 6.1.3 自定义验证字段

虽然WTForms中提供了许多验证器，但有时候我们还是需要自定义验证逻辑。还是以RegisterForm为例，在验证email字段时，除了验证是否满足邮箱的格式规则，还需要验证邮箱是否已经被注册过，这时就必须查询数据库，判断邮箱是否存在。如果要自定义某个字段的验证逻辑，可以通过在表单类中自定义方法validate\_<字段名>来实现，这里以验证email为例，示例代码如下。

```
from wtforms import Form, StringField, ValidationError
...
registered_email = ['aa@example.com', 'bb@example.com']

class RegisterForm(Form):
    ...

    def validate_email(self, field):
        email = field.data
        if email in registered_email:
            raise ValidationError("邮箱已经被注册！")
        return True
```

这里为了模拟从数据库中判断邮箱是否已经被注册，定义了一个registered\_email变量，代表已经被注册的邮箱，读者可自行结合ORM知识实现真实的数据库查找。接着定义了一个validate\_email(self,field)方法，以后在视图函数中调用form.validate()方法时，RegisterForm底层会自动调用validate\_email方法，并且会传递一个field参数，这里因为验证的是email字段，所以这个field参数代表的是email字段，如果验证的是其他字段，则field会代表相应字段。然后通过field.data拿到对应的值，再进行逻辑判断，如果认为验证失败了，则可以抛出wtforms.ValidationError异常，并且指定一个错误消息，这个错误消息会出现在form.errors中，否则直接返回True即可。

## 6.2 渲染表单模板

WTForms和Flask-WTF提供了将Python表单对象渲染成HTML表单模板的功能。这里以登录为例，先来实现一个登录的表单类，代码如下。

```
from wtforms import Form, StringField, BooleanField, SubmitField,
```

```
ValidationError
from flask_wtf import FlaskForm

...
class LoginForm(FlaskForm):
    email = StringField(label="邮箱:", validators=[email(message="请输入正确的邮箱!")], render_kw={"placeholder": "请输入邮箱"})
    password = StringField(label="密码:", validators=[length(min=6, max=20, message="请输入正确长度的密码!")], render_kw={"placeholder": "请输入密码"})
    remember = BooleanField(label="记住我:")
    submit = SubmitField(label="提交")
```

这里定义了一个LoginForm类，并使其继承自FlaskForm类。FlaskForm的父类是wtforms.Form类，其在wtforms.Form类的基础上增加了一些方便的方法，以使在验证表单数据时，不再需要手动传入request.form。接着分别定义了email、password、remember和submit这4个字段，这4个字段都新增了一个label参数，这个参数在渲染表单模板时会为除submit以外的字段生成一个label标签，因为submit是一个提交按钮，在提交按钮中label参数会被设置成属性value的值。如果在表单标签上设置一些属性，如placeholder，可以通过参数render\_kw来实现。

表单类定义好后，就可以在视图函数中使用了，因为现在表单类跟模板深度结合，所以使用方式跟之前有所不同，先看如下示例代码。

```
from forms import RegisterForm, LoginForm
...
@app.route("/login", methods=['GET', 'POST'])
def login():
    form = LoginForm(meta={"csrf": False})
    if form.validate_on_submit():
        email = form.email.data
        password = form.password.data
        return redirect("/")
    return render_template("login.html", form=form)
```

上述代码中，先定义了一个login视图函数，并使得其同时支持GET和POST请求。然后创建了一个LoginForm对象，并且通过传递meta参数，关闭了CSRF验证（关于CSRF，6.3节会讲到，这里先关闭即可）。我们重点关注渲染模板的代码，这里把form对象传给了login.html模板，而在login.html模板中，现在就可以使用form对象来渲染表单元素了，代码如下。

```

<form action="" method="POST">
    <table>
        <tbody>
            <tr>
                <td>{{ form.email.label }}</td>
                <td>{{ form.email }}</td>
            </tr>
            {% for error in form.email.errors %}
            <tr>
                <td></td>
                <td>{{ error }}</td>
            </tr>
            {% endfor %}
            <tr>
                <td>{{ form.password.label }}</td>
                <td>{{ form.password }}</td>
            </tr>
            {% for error in form.password.errors %}
            <tr>
                <td></td>
                <td>{{ error }}</td>
            </tr>
            {% endfor %}
            <tr>
                <td>{{ form.remember.label }}</td>
                <td>{{ form.remember() }}</td>
            </tr>
            <tr>
                <td></td>
                <td>{{ form.submit }}</td>
            </tr>
        </tbody>
    </table>
</form>

```

上述代码中，首先通过form.<字段名>.label渲染了email、password和remember这3个字段的label标签，然后通过form.<字段名>渲染对应的input标签。在每个字段下面循环form.<字段名>.errors进行验证，如果验证出现错误，会把所有的错误信息展示出来。然后我们再回过头去看login视图函数，重点关注表单验证部分的代码。

```

...
def login():
    form = LoginForm(meta={"csrf":False})
    if form.validate_on_submit():
        email = form.email.data

```

```
password = form.password.data  
return redirect("/")  
return render_template("login.html", form=form)
```

以上代码中，通过调用form.validate\_on\_submit()方法判断是否通过POST请求使表单验证成功，如果验证通过，则进入下一步操作，否则依然渲染login.html模板。其实此时的form上已经保存了验证失败的错误信息，因为在模板中通过form.<字段名>.errors已经获取到了对应字段的错误信息。所以如果访问http://127.0.0.1:5000/login，不输入任何数据，并直接单击“提交”按钮，可以看到如图6-3所示的效果。

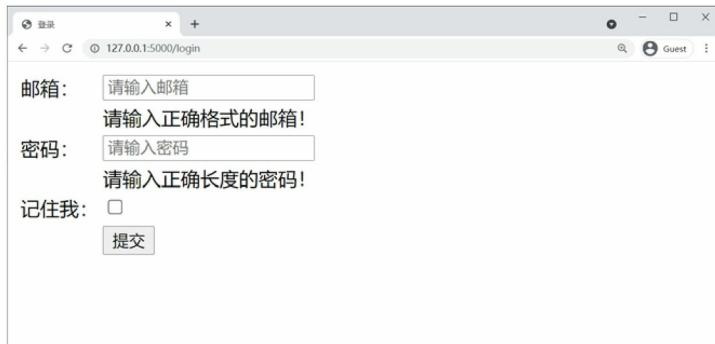


图6-3 不输入数据直接登录效果

如果输入正确格式的邮箱和密码，再单击“提交”按钮，则会跳转到首页。

### 注意

关于使用WTForms和Flask-WTF在模板中渲染表单，个人不太推荐。在Python类中定义HTML标签及其样式，会提高代码间的耦合度，造成本应在HTML模板中完成的工作，却要在Python文件中修改，特别是在前后端开发工程师共同开发项目的情况下，这种项目结构的缺点更是异常突出。

## 6.3 CSRF攻击

CSRF（cross site request forgery，跨站请求伪造）是一种网络攻击，这种攻击方式在

HTTP协议出现时即存在，令其名声大噪的事件，是在2007年，谷歌旗下的Gmail因为CSRF漏洞被黑客攻击而造成了巨大损失。先来了解一下CSRF攻击原理，如果读者能理解原理最好，如果不能理解，学完cookie和session部分的知识再回过头来理解也可以。

CSRF攻击原理：网站通常都是通过cookie来实现登录功能的，而浏览器在访问某个网站时，会自动把这个网站之前保存在浏览器中的cookie数据携带到服务器上去。这时候就存在一个漏洞，假设现在有一个病毒网站，这个网站在源代码中添加了恶意的JavaScript代码，在你访问这个网站时，会自动给具有CSRF漏洞的网站服务器发送请求（如给某个银行发起转账请求）。因为在发送请求时，浏览器会自动把这个网站的cookie数据发送给对应的服务器，而此时对应的服务器（如某银行网站）不知道这个请求是伪造的还是用户自己发起的，就被欺骗过去了，从而达到在用户不知情的情况下，给某个服务器发送了一个请求（如转账），从而造成了损失。

## 1. 防御CSRF攻击原理

CSRF攻击的要点，就是在向服务器发送请求时，相应的cookie会自动地被发送给对应的服务器，而服务器不知道这个请求是用户发起的还是伪造的。因此，可以在用户每次访问有表单的网页时，在表单中加入一个随机的字符串，如csrf\_token，同时在cookie中也加入一个具有相同值的csrf\_token键—值对。以后再给服务器发送请求时，必须在表单以及cookie中都携带csrf\_token。因为在不同域名下的JavaScript无法操作对方的cookie，所以服务器只有在检测到cookie中的csrf\_token和表单中的csrf\_token相同时，才认为这个请求是正常的，否则就认为请求是伪造的，服务器就会进行防御，那么黑客就没办法进行攻击了。

## 2. Flask-WTF防御CSRF攻击

使用Flask-WTF可以方便地实现CSRF防御。CSRF防御可以分成3种方式，第1种是全局防御，第2种是使用单表单防御，第3种是使用AJAX。这3种方式的前提都是已经在cookie中设置好了csrf，然后再从表单中获取csrf\_token，对比两者是否一致，一致则验证通过，否则即为验证失败。下面分别进行讲解。

### 1) 全局防御

CSRF全局防御是通过创建Flask-WTF中的CSRFProtect类对象实现的，这个类接收app作为参数，在使用CSRFProtect创建对象后，其会自动在Jinja2模板的上下文中添加csrf\_token函数，然后通过在模板中调用这个函数，就会自动生成csrf\_token。app.py中创建CSRFProtect对象的示例代码如下。

```
from flask_wtf import CSRFProtect

app = Flask(__name__)
app.secret_key = "自定义的app密钥"
CSRFProtect(app)
```

添加完以上代码后，我们就可以在模板中使用`csrf_token`函数来获取值了。如还是以`register.html`为例，添加完`csrf_token`后的`register.html`代码如下。

```
...
<form action="{{ url_for('register') }}" method="POST">
<table>
<tr>
<td></td>
<td><input type="hidden" name="csrf_token" value=
"{{ csrf_token() }}"></td>
</tr>
...
...
```

上述代码中，添加了一个`type`为`hidden`的`input`标签，这个标签不会被显示在浏览器上，但是单击“提交”按钮时，还是会把这个标签上的值发送给服务器。然后还给这个标签的`name`设置为`csrf_token`，除非通过配置文件已经修改了，否则必须将该`input`标签的`name`值设置为`csrf_token`，接着再调用`csrf_token`函数，设置`value`属性的值。这样渲染完模板后，在浏览器中访问`http://127.0.0.1:5000/register`，然后在页面右击，在弹出的快捷菜单中选择“查看网页源代码”命令，在源代码页面可以看到生成类似如图6-4所示的代码。

```
Line Wrap □
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>注册</title>
6 </head>
7 <body>
8   <form action="/register" method="POST">
9     <table>
10       <tr>
11         <td></td>
12         <td><input type="hidden" name="csrf_token" value="IjQ8YTUzzjcyfDRkHTJNh2Y2YQzZTV3OTNmjksNTQSHDZkHQNkZDmI.YQPT_g-Yggw8q259RXKM-dVyyz15Y1A49k"></td>
13       </tr>
14     </table>
15     <td>用户名: </td>
16     <td><input type="text" name="username"></td>
17   </tr>
18   <tr>
19     <td>邮箱: </td>
20     <td><input type="email" name="email"></td>
21   </tr>
```

图6-4 注册页面中生成的`csrf_token`标签代码

表单中有了`csrf_token`以后，在视图函数中调用`form.validate()`方法时，`RegisterForm`就会自动对比`request.form`中的`csrf_token`是否和`cookie`中的`csrf`值相等，相等就验证通过，否则就

验证失败。

## 2) 单表单防御

单表单防御是通过在模板中传递FlaskForm子类对象实现的，也就是6.2节中登录功能的实现。FlaskForm子类默认是开启了csrf防御，如果想要关闭，可以在创建表单对象时传递meta={"csrf":False}来实现，示例代码如下。

```
form = LoginForm(meta={"csrf":False})
```

如果要开启csrf防御，在创建表单时不传递meta参数即可。在login.html模板中，只要渲染form.csrf\_token，就会自动生成type为hidden类型的input标签，示例代码如下。

```
<form action="" method="POST">
<table>
  <tbody>
    <tr>
      <td></td>
      <td>{{ form.csrf_token }}</td>
    </tr>
```

我们在浏览器中访问http://127.0.0.1:5000/login，然后查看其网页源代码，可以看到如图6-5所示代码。



The screenshot shows the browser's developer tools with the 'Elements' tab selected. It displays the HTML source code of a login page. A red box highlights the line of code containing the csrf token:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>登录</title>
6 </head>
7 <body>
8   <form action="" method="POST">
9     <table>
10       <tbody>
11         <tr>
12           <td></td>
13           <td><input id="csrf_token" name="csrf_token" type="hidden" value="1jQ8YTU2ZjcyfDRkHTJNm2Y2YnQzTY3OTImNjkuHTQ8hD2kH2NkZDHl.yQPkgg.4T7hDCvBk9J0t9s1IxdU1xYkMvY"></td>
14         </tr>
15       </tbody>
16     </table>
17     <div>
18       <label for="email">邮箱: </label>
19       <input id="email" name="email" placeholder="请输入邮箱" type="text" value="">
20     </div>
21   </form>
22 </body>
23 </html>
```

图6-5 登录页面中生成的csrf\_token

同理，以后在视图函数中调用form.validate()方法时会自动和cookie中的csrf进行对比，判断是否验证通过。

## 3) 使用AJAX

使用AJAX提交表单数据，前提是要开启全局CSRF保护。为了方便，首先我们会在

HTML父模板的head标签内添加一个meta标签，然后把csrf\_token渲染到meta标签中。渲染到父模板中的原因是，为了让所有子模板都能获取到csrf\_token，示例代码如下。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <meta name="csrf_token" content="{{ csrf_token() }}">
</head>
...

```

在使用JavaScript发送请求之前，首先从模板中获取csrf\_token，然后再设置到请求头中。Flask-WTF验证csrf\_token，除了用cookie和表单中的值对比外，还会用cookie和请求头中的值对比，从请求头中获取，是通过X-CSRFToken参数获取的。这里以jQuery为例，设置在每次发送请求之前都添加好csrf\_token，示例代码如下。

```
var csrftoken = $('meta[name=csrf_token]').attr('content')
$.ajaxSetup({
    beforeSend: function(xhr, settings) {
        if (!/^(\GET|\HEAD|\OPTIONS|\TRACE)$/.test(settings.type) && !this.
crossDomain) {
            xhr.setRequestHeader("X-CSRFToken", csrftoken)
        }
    }
})

```

上述代码中，首先从模板文件中获取csrf\_token的值，然后通过设置beforeSend方法，使得在每次发送请求之前都把csrf\_token设置到请求头中，这样即可完成csrf的验证。

# chapter 7 第7章 Flask进阶

## 7.1 类视图

我们之前定义的视图都是用函数实现的，所以叫作函数视图。视图也可以使用类实现，即叫作类视图。类视图的好处是可以使用继承，把一些脚手架代码在父类中写好，子类只要聚焦到核心代码即可，从而为开发者节省时间。

### 7.1.1 基本使用

使用Flask类视图需要继承自flask.views.View类，然后在子类中实现dispatch\_request方法，这个方法类似于视图函数，可以进行逻辑处理，并且需要返回一个响应。这里以返回所有用户列表为例，用视图函数来实现，示例代码如下。

```
from flask.views import View

class ShowUsers(View):

    def dispatch_request(self):
        users = User.query.all()
        return render_template('users.html', objects=users)

    app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

以上代码中，首先从flask.views中导入View，然后定义类视图ShowUsers，让其继承自View。接着实现dispatch\_request方法，在这个方法中，我们从数据库中获取所有的用户，并且渲染模板，其操作方式跟视图函数一样。最后再把类视图，通过app.add\_url\_rule方法与路由进行绑定，在绑定的时候，必须通过as\_view()方法，把类转换为实际的视图函数，传给as\_view方法的字符串参数是视图函数的名称，以后通过url\_for进行反转时，需要使用到这个名称。此时用类视图并没有发现有什么优势，那么我们重构一下，把脚手架代码在父类中提前定义好，用子类去实现核心代码即可，示例代码如下。

```
from flask.views import View

class ListView(View):
```

```
def get_template_name(self):
    raise NotImplementedError()

def render_template(self, context):
    return render_template(self.get_template_name(), **context)

def dispatch_request(self):
    context = {'objects': self.get_objects()}
    return self.render_template(context)

class UserView(ListView):

    def get_template_name(self):
        return 'users.html'

    def get_objects(self):
        return User.query.all()
```

上述代码中，定义了一个ListView父视图，在这个父视图中提前定义好了用于获取模板的get\_template\_name方法，因为父视图无法知道具体的模板，所以抛出NotImplementedError异常，因此子视图必须要实现这个方法，并且返回模板路径。此外，还定义了用于渲染模板的render\_template方法，以及分发请求的dispatch\_request方法。子视图UserView分别实现了get\_template\_name和get\_objects方法，这样子视图就在代码量最小的情况下实现了列表渲染功能。

## 7.1.2 方法限制

在函数视图中，通过@app.route的methods参数即可限制请求的方法。类视图则通过定义methods类属性实现限制请求的功能，示例代码如下。

```
class MyView(View):
    methods = ['GET', 'POST']

    def dispatch_request(self):
        if request.method == 'POST':
            ...
        ...

app.add_url_rule('/myview', view_func=MyView.as_view('myview'))
```

上述代码中，在MyView中定义了methods属性，以后这个类视图就只能通过GET和POST方法进行访问了。

### 7.1.3 基于方法的类视图

现在的类视图是通过判断`request.method`来实现不同方法的逻辑代码，如果让子视图继承自`flask.views.MethodView`，则可以在类视图中重写对应小写形式的方法，如GET请求则实现`get`方法，POST请求则实现`post`方法，flask会自动根据浏览器请求的HTTP方法执行对应的方法，示例代码如下。

```
from flask.views import MethodView

class UserAPI(MethodView):

    def get(self):
        users = User.query.all()
        ...

    def post(self):
        user = User.from_form_data(request.form)
        ...
        app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))
```

以后在访问`/users/`这个URL时，如果用GET方法请求，那么就会执行`UserAPI`的`get`方法，用POST方法请求，就会执行`UserAPI`的`post`方法，示例代码如下。

```
class UserAPI(MethodView):

    def get(self):
        users = User.query.all()
        ...

    def post(self):
        user = User.from_form_data(request.form)
        ...
        app.add_url_rule('/users/', view_func=UserAPI.as_view('users'))
```

### 7.1.4 添加装饰器

在类视图中，如果想要添加装饰器，如某些视图需要登录才能访问，则可以在类视图中定义一个类属性`decorators`来实现，示例代码如下。

```
class UserAPI(MethodView):  
    decorators = [user_required]
```

以上代码中，`user_required`是自定义的装饰器，读者可以自行实现。

## 7.2 蓝图

现在所有的视图函数都是写在app.py文件中，随着项目越来越复杂，这种写法会导致app.py文件越来越臃肿，大幅地提高了后期项目维护的成本。对于一个商业项目而言，我们应该把代码进行模块化，蓝图就是为此而生的。我们以豆瓣网为例，豆瓣网目前有几个模块，分别为读书、电影、音乐、同城、小组、阅读等。每个模块都可以用一个蓝图来实现，最终在app中统一注册所有的蓝图，可以让项目结构更加清晰有序。下面对蓝图的使用进行讲解。

### 7.2.1 基本使用

这里以用户模块为例，注册一个蓝图，示例代码如下。

```
from flask import Blueprint  
bp = Blueprint('user', __name__, url_prefix='/user')  
  
@bp.route('/list')  
def user_list():  
    return "用户列表"  
  
@bp.route('/profile/<user_id>')  
def user_profile(user_id):  
    return "用户简介"
```

以上代码中，首先从flask中导入了Blueprint类，然后使用该类初始化一个对象，在初始化这个对象时传递了3个参数，3个参数的说明如下。

- (1) 蓝图名称：第1个参数是蓝图的名称，在使用url\_for反转蓝图中的某个视图时，需要用到蓝图名。视图名，如反转`user_list`，则代码为`url_for("user.user_list")`。
- (2) 模块名：第2个参数是模块名，一般设置为`__name__`，用于寻找模板文件和静态文件。

(3) URL前缀：第3个参数是URL前缀，以后访问这个蓝图的所有URL，都必须加上/user前缀，如获取用户列表的URL为/user/list。

蓝图注册完成后，还需要在Flask对象app中进行注册，示例代码如下。

```
from user import bp as user_bp  
...  
app.register_blueprint(user_bp)
```

## 7.2.2 寻找模板

在蓝图中渲染模板，默认会从项目根路径下的templates文件夹中寻找。如果想要更换寻找路径，可以在初始化Blueprint对象时，通过传递template\_folder参数实现，示例代码如下。

```
bp=Blueprint('user', __name__, url_prefix='/user', template_folder='templates')
```

这样以后在渲染模板文件时，如return render\_template("user.html")，默认就会从项目的根路径下的templates文件夹中寻找user.html，如果没有找到，则再从蓝图所在的文件夹下的templates文件夹中寻找。

## 7.2.3 寻找静态文件

默认是不设置任何静态文件路径的，Jinja2会在项目根路径下的static文件夹中寻找静态文件。在初始化Blueprint对象时，通过static\_folder参数可以指定静态文件的路径，示例代码如下。

```
bp=Blueprint('user', __name__, url_prefix='/user', static_folder='static')
```

static\_folder可以是相对路径（相对蓝图文件所在的目录），也可以是绝对路径。在配置完蓝图后，还需要注意如何在模板中引用静态文件。在模板中引用蓝图，应该使用蓝图名+.+static的格式来引用，示例代码如下。

```
<link href="{{ url_for('user.static',filename='about.css') }}">
```

## 7.3 cookie和session

cookie和session是Web开发中经常被使用的技术，因为HTTP请求是无状态的，也就是说每次请求间是相互独立的，如第一次请求成功后，接着发起第二次请求时，服务器依然不知道是谁发起的请求。而cookie和session就是为了解决这个问题而出现的。

### 7.3.1 关于cookie和session的介绍

#### 1. cookie

cookie的出现，就是为了解决HTTP请求无状态的问题。如果想要识别用户，可以在用户第一次请求后，在服务器端生成一段能识别用户的数据，存放到cookie中，然后返回给浏览器，浏览器会自动存储cookie数据。当该用户对同一网站发送第二次请求时，浏览器会自动把上次请求获取的cookie发送给服务器，服务器通过浏览器发送的cookie就能知道是哪个用户了。cookie存储的数据量有限，不同的浏览器存储容量也不同，但一般不超过4KB，因此cookie只适合存储少量的数据。

#### 2. session

session与cookie的作用类似，都是为了存储用户相关的信息。不同的是，cookie是存储在浏览器中，而session是一个概念，一个服务器存储授权信息的解决方案，不同的服务器，不同的框架，不同的编程语言都有不同的实现。Web开发发展至今，session的使用已经有了非常成熟的解决方案。在如今的市场环境中，一般session的存储有以下两种方式。

(1) 存储在服务器端：服务器在存储session时，首先生成session\_id，然后把session\_id和具体的session数据进行关联，并存储在服务器端，如数据库，或者是缓存中（如Memcached、Redis）。接着把session\_id存放到cookie中返回给浏览器，下次用户访问时，因为cookie会自动携带，所以服务器可以从cookie中获取session\_id，然后再从数据库或者缓存中获取具体的session数据。把session存储在服务器的好处是更加安全，不容易被窃取，坏处是会占用服务器资源，但是现在硬件条件已经非常先进了，存储一些session信息还是绰绰有余的。

(2) 存储在客户端：在存储session时，先将session进行加密，然后直接存储在cookie中返回给浏览器，下次用户访问时，则从cookie中获取session数据。Flask默认就是采用这种方法。

式，当然也可以替换为存储在服务器端的方式。

### 7.3.2 Flask中使用cookie和session

#### 1. Flask中操作cookie

在Flask中操作cookie是通过响应对象实现的，如Response或者其子类。Response及其子类有一个set\_cookie方法可以设置cookie，set\_cookie的参数如下。

- key: 设置cookie的key。
- value: 设置cookie的value。
- max\_age: 设置cookie距离现在多少秒后过期。
- expires: 设置cookie具体的过期时间，类型为datetime或者时间戳。
- domain: 设置cookie在哪个域名中有效。一般是设置子域名也能共享cookie，如 cms.example.com。
- path: 设置cookie在当前域名下的哪些path下有效，默认是在所有path下都有效。

在Flask中设置和获取cookie的示例代码如下。

```
from flask import make_response, request

# 设置cookie
@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp

# 获取cookie
@app.route('/user')
def user():
    username = request.cookies.get('username')
```

上述代码中，首先通过make\_response获取一个Response对象，然后调用set\_cookie方法设置cookie数据，最后在user视图函数中，通过request.cookies.get方法来获取cookie数据。

#### 2. Flask中操作session

在Flask中操作session是通过全局对象flask.session实现的，flask.session是一个类字典形

式，因此对session做增删改查操作时都可以使用字典相关的方法，示例代码如下。

```
from flask import session

# 设置session
@app.route('/')
def index():
    session["username"] = "the username"
    ...

# 获取session
@app.route('/')
def user():
    username = session['username']
    ...
```

另外，使用session的前提是，在app.config中配置好SECRET\_KEY。

## 7.4 request对象

在Flask项目中，如果要获取客户端提交上来的数据，可以通过全局线程安全对象flask.request实现。flask.request对象封装了许多属性和方法，常用的属性和方法如表7-1所示。

表7-1 常用的属性和方法

属性	描述
args	客户端通过查询字符串传过来的参数，其中查询字符串是经过解析的
form	客户端通过表单传过来的参数
url	当前请求的 URL
base_url	类似于 URL，但是会去掉查询字符串参数
cookies	客户端发送过来的 cookie 数据
files	客户端发送过来的文件数据
json	客户端通过 mime/type 为 application/json 的方式发送过来的 JSON 数据
headers	客户端发送过来的请求头数据
host	客户端请求当前服务器所用的域名
host_url	客户端请求当前服务器协议以及所用的域名
is_secure	是否用 HTTPS 或者 WSS 协议发送
method	当前请求所用的方法，如 GET 和 POST 等
path	客户端发送请求的 URL 中 path 部分
query_string	客户端发送的查询字符串，没有经过解析。args 属性是经过解析的
remote_addr	客户端发送请求所用的地址，可以是 IP 地址或者域名
user_agent	通过 user_agent.string 可以获取发送请求的浏览器类型

flask.request对象完整的属性和方法，请参考Flask官方文档

<https://flask.palletsprojects.com/en/2.0.x/api/#incoming-request-data>。

## 7.5 Flask信号机制

信号是Flask中一项非常强大的功能。从软件工程角度讲，可以让数据传递代码彼此解耦；从功能实现角度讲，可以在某个事件发生时，就自动执行一系列的配套操作。下面分别讲解自定义信号和Flask内置信号。

### 7.5.1 自定义信号

Flask从0.6版本开始就支持信号机制。信号的作用是在发生某个事情时，直接通知某个函数执行，可以达到代码间解耦的作用。Flask中的信号是通过第三方库blinker实现的。我们先来学习一下如何创建信号以及发送信号，创建信号的示例代码如下。

```
from blinker import Namespace
# 定义Namespace对象
my_signals = Namespace()

# 创建名称为model-saved的信号
```

```
model_saved = my_signals.signal('model-saved')
```

上述代码中，首先是创建了一个Namespace对象my\_signals，然后通过my\_signals.signal方法添加了一个名称为model-saved的信号。接着还需要订阅此信号，也就是在发生此信号时，需要执行什么代码，示例代码如下。

```
def log_model_saved(sender):
    print("捕获到信号，发送者为: {}".format(sender))

my_signals.connect(log_model_saved)
```

上述代码中，首先通过my\_signals.connect方法订阅信号，在信号产生后，会执行log\_model\_saved方法。然后在其他代码中如有需要的情况，就可以通过my\_signals.send来发送信号了，示例代码如下。

```
class Model(object):
    ...

    def save(self):
        model_saved.send(self)
```

上述代码中，model\_save.send方法中的第1个参数是发送者，在订阅的函数中可以通过sender参数获取发送者的信息。

## 7.5.2 Flask内置信号

flask中已经提前内置了许多信号，在相应的事件发生后，会发送信号，我们只需监听即可。在监听的时候，可以设置只接收哪个发送者发送的信号，一般设置为app。下面来介绍常用的信号。

### 1. flask.template\_rendered

模板渲染完毕后会发送此信号，示例代码如下。

```
from flask import template_rendered
def log_template_rendered(sender,template,context,*args):
```

```
print('sender:',sender)
print('template:',template)
print('context:',context)

template_rendered.connect(log_template_renders,app)
```

## 2. flask. request\_started

在接收到请求且到达视图函数之前会发送此信号，示例代码如下。

```
def log_request_started(sender,**extra):
    print 'sender:',sender
    print 'extra:',extra
request_started.connect(log_request_started,app)
```

## 3. flask. request\_finished

请求结束后，在响应发送到客户端之前会发送此信号，示例代码如下。

```
def log_request_finished(sender,response,*args):
    print 'response:',response
request_finished.connect(log_request_finished,app)
```

## 4. flask. got\_request\_exception

在请求过程中出现异常时会发送此信号，示例代码如下。

```
def log_exception_finished(sender,exception,*args):
    print 'sender:',sender
    print type(exception)
got_request_exception.connect(log_exception_finished,app)
```

## 5. flask. request\_tearing\_down

请求完成后，在对象被销毁之前会发送此信号。即使请求过程中发生异常，也会发送此信号，示例代码如下。

```
def log_request_tearing_down(sender, **kwargs):
    print 'coming...'
request_tearing_down.connect(log_request_tearing_down, app)
```

## 7.6 常用钩子函数

钩子函数是从收到请求到响应请求在整个链条中可以进行拦截的函数，钩子函数都是通过装饰器来注册的。常用的钩子函数如下。

### 1. before\_first\_request

在收到第一个请求之前执行，示例代码如下。

```
@app.before_first_request
def first_request():
    print 'first time request')
```

### 2. before\_request

在每次收到请求之前执行，示例代码如下。

```
@app.before_request
def before_request():
    if not hasattr(g, 'user'):
        setattr(g, 'user', 'xxxx')
```

### 3. teardown\_appcontext

不管是否有异常，都会在每次请求之后执行，示例代码如下。

```
@app.teardown_appcontext
def teardown(exc=None):
    if exc is None:
        db.session.commit()
    else:
        db.session.rollback()
```

## 4. context\_processor

上下文处理器在每次渲染模板时，会把这个钩子函数中返回的数据添加到模板中，示例代码如下。

```
@app.context_processor  
return {'current_user':'xxx'}
```

## 5. errorhandler

用于指定在出现非200状态码时的错误处理方法，示例代码如下。

```
@app.errorhandler(404)  
def page_not_found(error):  
    return 'This page does not exist',404
```

## 7.7 上下文

在使用Flask开发项目时，经常会使用到4个全局变量，即request、session、current\_app和g，这4个全局变量就是上下文对象。上下文对象是Flask中的一个非常优雅的设计，其作用是在一个请求到来之后，不需要再把一些常用的对象在函数间层层传递。Flask中的上下文对象相关说明如表7-2所示。

表7-2 Flask中的上下文对象

上下文对象	类 型	说 明
flask.request	请求上下文	保存了用户请求的信息
flask.session	请求上下文	用于记录多次请求之间的状态，默认加密后存储到 cookie 中，也可自定义存储方式
flask.current_app	应用上下文	获取当前的 app 对象，此对象并非真正的 app 对象，而是 app 对象的代理
flask.g	应用上下文	全局对象，常用于请求到来后函数间共享数据

### 7.7.1 线程隔离对象

在学习上下文对象的原理之前，我们首先需要理解线程隔离对象。线程隔离对象可以使数据在多个线程间拥有独自的备份，不会被其他线程影响。在Flask中，每收到一个请求则开启一个线程，而表7-2中的上下文对象因为是被存放到了线程隔离对象中，所以即使是定义成全局变量，在每个线程间都独有一份备份。

在Python内置的threading模块中，通过threading.local()即可创建一个用于保存线程隔离对象的变量，示例代码如下。

```
import threading

thread_local = threading.local()
thread_local.name = "我是主线程的"

def thread_func(index):
    thread_local.name = f"我是{index}线程的"
    print(thread_local.name)

if __name__ == '__main__':
    for x in range(1,3):
        th = threading.Thread(target=thread_func, kwargs={"index": x})
        th.start()
        th.join()
    print(thread_local.name)
```

上述代码中，首先创建了threading.local类的对象，然后在这个对象上绑定了name属性，之后在主线程和子线程中分别赋不同的值。执行上述代码会发现，每个线程的name值都是不一样的，并且一个线程修改了name的值，并不会影响到其他线程。线程隔离对象实现的原理并不复杂，我们只需根据线程id进行区分即可。

除了内置的threading模块提供的线程隔离对象外，werkzeug也单独定义了一个werkzeug.local.Local类，这个类的实现逻辑与threading.local大同小异，下面我们先来看werkzeug.local.Local源代码（为了让读者关注核心代码，笔者对源代码进行了删改）。

```

try:
    from greenlet import getcurrent as _get_ident
except ImportError:
    from threading import get_ident as _get_ident

class ContextVar:

    def __init__(self, _name: str) -> None:
        self.storage: t.Dict[int, t.Dict[str, t.Any]] = {}

    def get(self, default: t.Dict[str, t.Any]) -> t.Dict[str, t.Any]:
        return self.storage.get(_get_ident(), default)

    def set(self, value: t.Dict[str, t.Any]) -> None:
        self.storage[_get_ident()] = value


class Local:
    __slots__ = ("_storage",)

    def __init__(self):
        object.__setattr__(self, "_storage", ContextVar("local_storage"))

    def __getattr__(self, name):
        values = self._storage.get({})
        try:
            return values[name]
        except KeyError:
            raise AttributeError(name)

    def __setattr__(self, name, value):
        values = self._storage.get({}).copy()
        values[name] = value
        self._storage.set(values)
    ...

```

上述代码中，首先在Local类中创建(storage对象，其值为ContextVar对象，又在ContextVar中则定义了一个storage对象，此对象为一个字典类型，如果当前环境中安装了greenlet，则使用greenlet的协程id作为字典的键，否则使用threading模块的线程id作为键。storage字典的值也是一个字典，这个字典中存放的就是绑定到这个Local对象上的属性名和属性值。阅读Local的\_\_getattr\_\_和\_\_setattr\_\_方法后发现，绑定到Local对象上的属性，实际上是间接绑定到\_storage.storage上了。werkzeug.local.Local就是通过这种技术，实现了线程隔离的对象。

## 7.7.2 LocalStack类

werkzeug.local.LocalStack类是一个将对象存放到werkzeug.local.Local上的栈结构。flask.request、flask.app等都是存放在这个类的对象上的，以下为LocalStack的源代码（为了便于理解，笔者对源代码进行了删改）。

```
class LocalStack:

    def __init__(self) -> None:
        self._local = Local()

    def __call__(self) -> "LocalProxy":
        def _lookup() -> t.Any:
            rv = self.top
            if rv is None:
                raise RuntimeError("object unbound")
            return rv

        return LocalProxy(_lookup)

    def push(self, obj: t.Any) -> t.List[t.Any]:
        rv = getattr(self._local, "stack", []).copy()
        rv.append(obj)
        self._local.stack = rv
        return rv # type: ignore

    def pop(self) -> t.Any:
        stack = getattr(self._local, "stack", None)
        if stack is None:
            return None
        elif len(stack) == 1:
            release_local(self._local)
            return stack[-1]
        else:
            return stack.pop()

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

上述代码提供了非常方便的push方法，用于往栈中添加数据，通过pop方法从栈顶中删

除数据，通过top属性可以获取栈顶数据。并且所有数据都存放在Local对象上，因此保证了数据在多线程中的独有性。那么LocalStack在哪里用到了呢？我们进入flask.globals模块中，可以看到以下两行代码。

```
_request_ctx_stack = LocalStack()
_app_ctx_stack = LocalStack()
```

其中\_request\_ctx\_stack用来存放请求上下文的栈对象，\_app\_ctx\_stack用来存放应用上下文的栈对象。那么又在哪里使用到了这两个对象呢？在一个请求到达Flask项目之后，首先会执行flask.app.wsgi\_app方法，此方法相关源代码如下。

```
def wsgi_app(self, environ: dict, start_response: t.Callable) -> t.Any:
    ctx = self.request_context(environ)
    error: t.Optional[BaseException] = None
    try:
        try:
            ctx.push()
            response = self.full_dispatch_request()
        except Exception as e:
            error = e
            response = self.handle_exception(e)
        except: # noqa: B001
            error = sys.exc_info()[1]
            raise
        return response(environ, start_response)
    finally:
        if self.should_ignore_error(error):
            error = None
        ctx.auto_pop(error)
```

接收到请求后，wsgi\_app首先会通过self.request\_context创建一个RequestContext（请求上下文）对象ctx，然后调用ctx.push方法，我们再来看ctx.push方法的源代码。

```
def push(self) -> None:
    # 获取请求上下文栈顶元素
    top = _request_ctx_stack.top
    # 如果存在，则先删除
    if top is not None and top.preserved:
        top.pop(top._preserved_exc)
    # 获取应用上下文栈顶元素
```

```
app_ctx = _app_ctx_stack.top
# 如果栈中没有元素，则创建一个应用上下文，然后推送到栈中
if app_ctx is None or app_ctx.app != self.app:
    app_ctx = self.app.app_context()
    app_ctx.push()
    self._implicit_app_ctx_stack.append(app_ctx)
else:
    self._implicit_app_ctx_stack.append(None)

# 将请求上下文推到栈顶
_request_ctx_stack.push(self)
...
```

查看push方法代码后，我们看到了\_request\_ctx\_stack和\_app\_ctx\_stack两个对象。整体的逻辑是先将应用上下文推送到\_app\_ctx\_stack栈顶，然后再推送请求上下文到\_request\_ctx\_stack栈顶。这一步操作使我们明白了，请求上下文和应用上下文是一起创建并一起销毁的，但是我们会有如下几个疑问。

- 这里推送到请求上下文和应用上下文的对象是后面用到的request和current\_app吗？
- LocalStack中的LocalProxy又是用来做什么的？为什么需要它？

### 7.7.3 LocalProxy类

通过以下代码可以看到，current\_app、request、session、g这4个上下文对象全部是werkzeug.local.LocalProxy对象。

```
current_app: "Flask" = LocalProxy(_find_app)  # type: ignore
request: "Request" = LocalProxy(partial(_lookup_req_object, "request"))
session: "SessionMixin" = LocalProxy(  # type: ignore
    partial(_lookup_req_object, "session"))
)
g: "_AppCtxGlobals" = LocalProxy(partial(_lookup_app_object, "g"))
```

那么LocalProxy是怎么实现的呢？我们先来看LocalProxy的核心源代码。

```

class LocalProxy:
    __slots__ = ("__local", "__name", "__wrapped__")

    def __init__(
        self,
        local: t.Union["Local", t.Callable[[], t.Any]],
        name: t.Optional[str] = None,
    ) -> None:
        object.__setattr__(self, "__LocalProxy__local", local)
        object.__setattr__(self, "__LocalProxy__name", name)

        if callable(local) and not hasattr(local, "__release_local__"):
            object.__setattr__(self, "__wrapped__", local)

    def _get_current_object(self) -> t.Any:
        # 如果 self.__local 没有__release_local__属性（即是一个可以调用的对象）
        if not hasattr(self.__local, "__release_local__"):
            # 调用这个对象
            return self.__local()

    try:
        # 从 self.__local 上获取 self.__name 的值
        return getattr(self.__local, self.__name__)
    except AttributeError:
        raise RuntimeError(f"no object bound to {self.__name__}")
...

```

因为LocalProxy的源代码非常多，我们只截取其中的核心部分。LocalProxy是一个代理对象，会对创建LocalProxy时传进来的local对象进行代理。在使用LocalProxy对象的某个属性时，会自动执行\_get\_current\_object方法，此方法会判断self.\_\_local是否可以调用，如果可以调用，则执行调用，否则从self.\_\_local上获取self.\_\_name指定的值。我们再回过头看current\_app和request的实现，代码如下。

```

def _find_app():
    top = _app_ctx_stack.top
    if top is None:
        raise RuntimeError(_app_ctx_err_msg)
    return top.app

def _lookup_req_object(name):
    top = _request_ctx_stack.top
    if top is None:
        raise RuntimeError(_request_ctx_err_msg)
    return getattr(top, name)

```

```
current_app: "Flask" = LocalProxy(_find_app)
request: "Request" = LocalProxy(partial(_lookup_req_object, "request"))
```

以上代码中，`current_app`是将`_find_app`函数传给了`LocalProxy`，而`_find_app`做的事情非常简单，就是从`_app_ctx_stack`应用上下文上获取栈顶数据。所以在使用`current_app`时，实际上是先执行`LocalProxy._get_current_object`方法，然后再执行`_find_app`方法将`_app_ctx_stack`栈顶数据进行返回。

再看`request`，其在创建`LocalProxy`时，传入的是一个偏函数，这里用偏函数可以把`request`传入`_lookup_req_object`进行调用。执行此函数时，首先获取`_request_ctx_stack`的栈顶元素，然后再获取栈顶元素上的`request`属性。

有读者可能会有疑惑，为什么表7-2中的4个上下文对象需要使用`LocalProxy`进行代理？原因是Flask中的上下文都是动态推送和删除的，如果不用代理，表7-2中的4个上下文对象只会被赋值一次，不会随着栈元素的更新而更新。

# chapter 8 第8章 缓存系统

随着网站访问量越来越高，我们应该提高网站的性能和响应速度。其中一个优化点就是减少数据库查询操作，数据库操作是I/O操作，它的性能再高也无法和直接在内存中操作相比，因此可以把一些不是非常重要的数据存储到缓存中，如验证码、在线人数、session等。缓存系统目前用得最多的是Memcached和Redis。Memcached是一个纯内存的缓存软件，比较轻量级，不具有自动同步数据到硬盘的功能。Redis则比Memcached更强大、更重量级，其除了把数据存储到内存外，还可以自动定时把数据同步到硬盘中。下面将分别进行讲解。

## 8.1 Memcached

Memcached是一个高性能的分布式内存对象缓存系统，全世界有不少公司采用这个缓存系统来构建大负载的网站，以分担数据库的压力。Memcached是通过在内存里维护一张统一的、巨大的Hash表来存储各种各样的数据，包括图像、视频、文件以及数据库检索的结果等。简单地说，就是将数据调用到内存中，然后从内存中读取，从而大大提高读取速度。因为Memcached是把数据存储到内存中，并且不具有自动同步数据的功能，所以不建议存储一些非常重要的数据，以免因为机器故障导致数据丢失。Memcached的存储方式是以键—值对的方式存储。

### 8.1.1 安装Memcached

Memcached官方只提供了Linux系统的安装包，作为练习，我们可以使用社区版提供的Windows版本。这里以Windows系统和Ubuntu系统为例讲解Memcached的安装和基本使用。

#### 1. Windows系统

因为Memcached官方没有提供Windows版本，所以只能用社区版，读者可以到<https://www.runoob.com/Memcached/window-install-memcached.html>下载Windows版的Memcached，下载完成后解压可以得到以下文件，如图8-1所示。

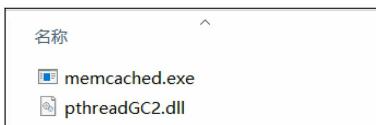


图8-1 Memcached解压后的文件

在cmd命令行终端输入以下命令完成安装和启动。

- 安装: memcached.exe -d install。
- 启动: memcached.exe -d start。

## 2. Linux系统（Ubuntu）

因为Memcached官方是支持Linux系统的，所以在Ubuntu系统中可以直接使用apt命令完成安装。在Ubuntu系统中安装和启动的命令如下。

- 安装: sudo apt install memcached。
- 启动: /usr/bin/memcached -d start。

在启动Memcached时，还可以传递以下参数配置Memcached的运行方式。

- d: 让Memcached在后台运行。
- m: 指定占用多少内存，以MB为单位，默认是64MB。
- p: 指定运行监听的端口号，默认的端口号是11211。
- l: 如设置-l0.0.0.0，表示允许其他机器通过本机IP地址连接到本机的Memcached， 默认只能本机连接。

### 8.1.2 telnet操作Memcached

Linux和Windows系统都有telnet命令，以下命令都是在Linux的终端和Windows的cmd终端中完成的。在Memcached启动的前提下，在终端或者cmd中，输入telnet 127.0.0.1 11211，即可进入Memcached命令界面。如果要连接其他机器的Memcached，则把127.0.0.1改成目标机器的IP地址即可，并且目标机器在运行时，需要通过-l参数设置允许其他机器连接的状态。下面对在Memcached中使用的命令进行讲解。

#### 1. set命令

使用set命令可以将数据添加到Memcached中，如果数据的键在Memcached中已存在，则会进行替换，语法如下。

```
set key zip timeout value_length
```

参数的说明如下。

- key：此数据的键。
- zip：是否压缩数据。
- timeout：过期时间，单位为s。
- value\_length：值的长度。

示例命令如下。

```
$ set username 0 60 7  
$ zhiliao
```

以上命令，我们设置了键为username的数据，过期时间为60s后，值的总长度为7。

## 2. add命令

add命令也是用于添加数据，如果数据的键在Memcached中已存在，则会添加失败，语法如下。

```
add key zip timeout value_length
```

示例命令如下。

```
$ add username 0 60 7  
$ zhiliao
```

## 3. get命令

get命令用于从Memcached中获取数据，如果数据过期，则会获取失败，语法如下。

```
get key
```

示例命令如下。

```
get username
```

## 4. delete命令

delete命令用于从Memcached中删除指定键一值对的数据，语法如下。

```
delete key
```

示例命令如下。

```
$ delete username
```

## 5. flush\_all命令

flush\_all命令用于删除Memcached中的所有数据。注意：这条命令要谨慎使用，语法如下。

```
flush_all
```

## 6. stats命令

如果要查看当前Memcached的运行状态，如总共存储了多少条数据、总共有多少个连接，则可以通过stats命令查看，语法如下，命令执行结果如图8-2所示。

```
stats
```

```
stats
STAT pid 6224 ① 进程ID
STAT uptime 3054548720 ② 总运行时间
STAT time 408290341
STAT version 1.4.4-14-g9c660c0 ③ memcached版本号
STAT pointer_size 64
STAT curr_connections 10 ④ 当前连接数
STAT total_connections 12 ⑤ 总连接数
STAT connection_structures 11
STAT cmd_get 0 ⑥ get命令执行的次数
STAT cmd_set 2 ⑦ set命令执行的次数
STAT cmd_flush 0
STAT get_hits 0 ⑧ get命令命中中的次数
STAT get_misses 0 ⑨ get命令未命中中的次数
STAT delete_misses 0
STAT delete_hits 0
STAT incr_misses 0
STAT incr_hits 0
STAT decr_misses 0
STAT decr_hits 0
STAT cas_misses 0
STAT cas_hits 0
STAT cas_badval 0
STAT auth_cmds 0
STAT auth_errors 0
STAT bytes_read 107 ⑩ memcached服务端通过网络读取的总字节数
STAT bytes_written 798 ⑪ memcached服务端发送到网络上的总字节数
STAT limit_maxbytes 67108864 ⑫ memcached分配的内存数
STAT accepting_conns 1 ⑬ 当前接收的连接数
STAT listen_disabled_num 0
STAT threads 4
STAT conn_yields 0
STAT bytes 80 ⑭ 存储数据总字节数
STAT curr_items 1 ⑮ 当前存储的item总数
STAT total_items 2 ⑯ memcached启动时候存储的item总数
STAT evictions 0
END
```

图8-2 命令执行结果

### 8.1.3 Python操作Memcached

使用Python操作Memcached需要先安装python-memcached，安装命令如下。

```
pip install python-memcached
```

安装完python-memcached包以后，就可以使用它来连接Memcached服务并操作Memcached了。下面介绍一下相关操作。

#### 1. 建立连接

建立连接，示例代码如下。

```
import memcache  
mc = memcache.Client(['127.0.0.1:11211'],debug=True)
```

上述代码中，首先导入了memcache模块，然后使用memcache.Client类创建了一个对象，并设置了连接到哪个Memcached服务器，而且为了更好地看到调试Memcached操作代码，设置debug=True。后续对Memcached执行的操作，都是通过memcache.Client对象mc来实现的。

## 2. 设置数据

通过mc.set可以设置一条数据，通过mc.set\_multi可以一次性设置多条数据，并且通过time参数可以设置过期时间，示例代码如下。

```
mc.set('username','hello world',time=60*5)  
mc.set_multi({'email':'xxx@qq.com','telephone':'111111'},time=60*5)
```

## 3. 获取数据

通过mc.get方法可以获取指定键的值，示例代码如下。

```
mc.get("username")
```

## 4. 删除数据

通过mc.delete方法可以删除指定键的数据，示例代码如下。

```
mc.delete("username")
```

## 5. 自增长

通过mc.incr可以对整型类型的值进行自增长，每执行一次会把值加1。如设置在线人数，示例代码如下。

```
mc.incr("online_count")
```

## 6. 自减少

通过mc.decr可以对整型类型的值进行自减少，每执行一次会把值减1。如还是以在线人数为例，示例代码如下。

```
mc.decr("online_count")
```

### 8.1.4 Memcached的安全性

Memcached的操作不需要任何用户名和密码，只需要知道Memcached服务器的IP地址和端口号即可。因此使用Memcached时尤其要注意安全性。这里提供了以下两种安全的解决方案。

- 使用-l参数设置只有本地可以连接：这种方式就只能通过本机连接，其他机器都不能访问，可以达到最好的安全性。
- 使用防火墙关闭对外的11211端口：外网无法访问到本机的11211端口，也就无法访问到Memcached服务。这里以Ubuntu系统为例，设置防火墙相关的命令如下。

```
ufw enable      # 开启防火墙  
ufw disable    # 关闭防火墙  
ufw default deny # 防火墙以禁止的方式打开，默认是关闭那些没有开启的端口  
ufw deny 端口号 # 关闭某个端口  
ufw allow 端口号 # 开启某个端口
```

## 8.2 Redis

Redis是一种NoSQL数据库，它的数据默认是存储在内存中的，同时Redis可以定时将内存中的数据同步到硬盘中。Redis比Memcached支持更多的数据结构，除了bool、int、float、string基本数据类型，还支持list（列表）、set（集合）、sorted set（有序集合）、hash（hash表）。

### 8.2.1 Redis使用场景

Redis的使用场景非常多，包括但不限于以下场景。

- 登录会话存储：可以把能识别客户端的数据存储在Redis中，如token、session等。
- 在线人数 / 计数器：如现在非常火的直播的实时变动的在线人数、新浪微博的点赞数等，为了提高响应速度，都可以把数据存储在Redis中。
- 作为消息队列：如在使用Celery实现异步操作时，Redis可以作为中间人。
- 常用的数据缓存：可以把一些网站中经常会用到的数据放到Redis中，以提高加载速度。
- 首页数据缓存：一个网站的首页访问频率是最高的，可以把首页中部分的数据放到Redis中。
- 好友关系：如微博的好友关系极其复杂，如果每次获取好友关系时都要查找数据库，那么响应速度将慢到无法忍受。这时就可以把好友关系存储到Redis中。
- 发布和订阅功能：Redis有发布和订阅功能，可以用来做聊天软件。

## 8.2.2 Redis和Memcached比较

Memcached和Redis都是非常优秀的缓存系统。Memcached的优点是轻量级和占用服务器资源少，缺点是功能少。而Redis的优点是功能更强大，配置项比Memcached更多一些，缺点是对服务器的要求比较高。Memcached和Redis的比较如表8-1所示。

表8-1 Memcached和Redis比较

属性	Memcached	Redis
类型	纯内存数据库	内存磁盘同步数据库
数据类型	在定义 value 时就要固定数据类型	不需要
虚拟内存	不支持	支持
过期策略	支持	支持
存储数据安全	不支持	可以将数据同步到 dump.db 中
灾难恢复	不支持	可以将磁盘中的数据恢复到内存中
分布式	支持	主从同步
订阅与发布	不支持	支持

## 8.2.3 Redis在Ubuntu中的安装与使用

Redis是没有Windows平台的官方支持版本的，如果要在生产环境中使用Redis，建议在生产环境上使用Linux系统。作为学习，可以使用非官方发布的Windows版本的Redis。下面分别讲解在Windows和Linux系统上安装和使用Redis。

## 1. 在Windows系统上安装和使用Redis

- 下载链接: <https://github.com/tporadowski/redis/releases>, 下载msi文件即可。
- 安装步骤: 首先双击msi文件, 在打开的Redis on Windows Setup对话框中选择安装路径并选中Add the Redis installation folder to the PATH environment variable复选框, 然后单击Next按钮, 如图8-3所示。

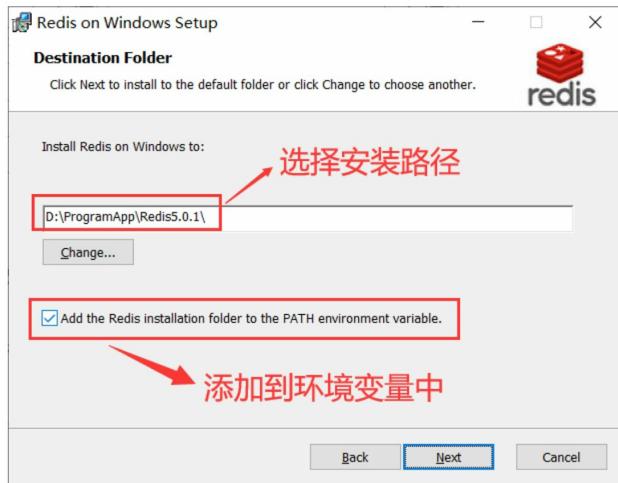


图8-3 选择安装路径并添加到环境变量中

进入设置端口号和防火墙的界面, Redis默认端口号是6379, 然后再把Redis服务的防火墙关掉(如果不关掉防火墙, 会导致其他计算机无法访问本机Redis服务), 如图8-4所示。

再次单击Next按钮, 会出现设置允许Redis占用最大内存的界面, 默认是无限, 这个对于我们只是用于学习来讲影响不大, 可设可不设。接着单击Next按钮, 然后单击Install按钮, 稍等片刻即可完成安装。

- 使用: 打开cmd命令行终端, 然后执行redis-cli命令, 即可进入Redis命令行, 然后按照8.2.4节中的命令操作Redis即可。

## 2. 在Linux系统上安装Redis

在Linux系统的版本中, Ubuntu版简单易用且长期支持。这里以Ubuntu系统为例进行讲解。



图8-4 设置端口号和关闭Redis防火墙

- 安装命令: apt install redis-server。
- 卸载命令: apt purge --auto-remove redis-server。
- 启动命令: Redis安装完成后会自动启动, 可以通过ps aux|grep redis命令进行查看。  
如果想手动启动, 可以通过service redis-server start命令启动。
- 停止命令: service redis-server stop。
- 使用: 在连接redis-server之前, 要先启动redis-server服务, 命令为service redis-server start。启动redis-server后, 再使用redis-cli命令进行连接, 语法如下。

```
redis-cli -h [ip] -p [端口]
```

redis-server默认的端口号是6379, 连接本机的redis-server命令如下。

```
redis-cli -h 127.0.0.1 -p 6379
```

在连接到redis-server后, 就可以使用8.2.4节中的命令对redis-server进行操作了。

## 8.2.4 Redis操作命令

Redis操作命令在Windows和Linux系统中都是一样的，相关命令介绍如下。

## 1. 添加数据

添加数据使用set命令，语法如下。

```
set key value EX timeout
```

如果key之前已经存在，则会进行覆盖。EX参数设置的是过期时间，如果没有EX参数，则意味着数据永远不会过期，示例命令如下。

```
set username zhiliao EX 60
```

或

```
set username zhiliao
```

或者可以在使用setex命令添加数据时就设置过期时间，示例命令如下。

```
setex username 60 zhiliao
```

## 2. 删除数据

删除数据使用delete命令，语法如下。

```
delete key
```

示例命令如下。

```
delete username
```

### 3. 设置过期时间

可以使用**expire**命令来设置数据的过期时间，语法如下。

```
expire key timeout(单位为秒)
```

示例命令如下。

```
expire username 60
```

### 4. 查看过期时间

如果忘记某个数据什么时候过期，可以通过**ttl**命令查看数据的过期时间，语法如下。

```
ttl key
```

示例命令如下。

```
ttl username
```

### 5. 查看所有key

如果想知道当前Redis中存储的所有key，则可以通过**keys**命令实现，命令如下。

```
keys *
```

### 6. 列表操作

在列表左边添加元素，语法如下。

```
lpush key value
```

将value插入key所指向的列表的最开始位置。如果key不存在，则会创建一个空的列表，并且执行lpush操作。当key存在但不是列表类型时，将会返回一个错误。

在列表右边添加元素，语法如下。

```
rpush key value
```

将value插入列表key的末尾。如果key不存在，则会创建一个空的列表，并且执行rpush操作。当key存在但不是列表类型时，将会返回一个错误。

查看列表中的元素，语法如下。

```
lrange key start stop
```

返回列表key中指定区间内的元素，区间以偏移量start和stop指定，如果要左边的第一个元素到最后一个元素，则语法如下。

```
lrange key 0 -1
```

移除列表左边的元素，语法如下。

```
lpop key
```

移除并返回列表右边的元素，语法如下。

```
rpop key
```

移除并返回列表中指定的元素，语法如下。

```
lrem key count value
```

以上命令将删除列表key中count个值为value的元素。

获取列表中指定位置的元素，语法如下。

```
lindex key index
```

以上命令将获取列表key中索引为index的元素。

获取列表中元素个数，语法如下。

```
llen key
```

删除指定元素，语法如下。

```
lrem key count value
```

以上命令将删除列表key中count个值为value的元素。如果count>0，则从左向右搜索，移除与value相等的元素，数量为count。如果count<0，则从右向左搜索，移除与value相等的元素，数量为count的绝对值。如果count=0，则移除列表中所有与value相等的值。

## 7. 集合操作

集合是无序的，并且里面的元素不能重复。下面来讲解集合的操作命令。

添加元素，语法如下。

```
sadd key value1 value2...
```

通过sadd命令可以一次性添加多个元素。

查看元素，语法如下。

```
smembers key
```

移除元素，语法如下。

```
srem key value1 value2
```

通过srem命令可以一次性删除多个元素。

查看集合中元素个数，语法如下。

```
scard key
```

获取多个集合的交集，语法如下。

```
sinter key1 key2...
```

获取多个集合的差集，语法如下。

```
sdiff key1 key2...
```

## 8. 哈希（hash）操作

哈希（hash）可以简单理解为Python中的字典，操作hash的相关命令如下。

添加新值，语法如下。

```
hset key field value
```

示例命令如下。

```
hset website baidu baidu.com
```

以上示例命令中，如果website这个key不存在，则会在redis-server中创建一张名为website的哈希表。

获取哈希表中field的值，语法如下。

```
hget key field
```

如果field已经在哈希表中存在，旧值将会被覆盖。示例命令如下。

```
hget website baidu
```

删除某个field，语法如下。

```
hdel key field
```

获取某张哈希表中所有的field，语法如下。

```
hkeys key
```

获取某张哈希表中所有的field和value，语法如下。

```
hgetall key
```

获取某张哈希表中所有的值，语法如下。

```
hvals key
```

判断哈希表中是否存在某个field，语法如下。

```
hexists key field
```

## 9. 事务操作

Redis事务可以一次性执行多个命令，具有隔离性和原子性。隔离性意味着在事务中执行的所有命令都会按顺序执行，不会被其他命令干扰。原子性意味着在事务中的命令，要么全部都执行，要么全部都不执行。下面来学习事务的相关命令。

开启一个事务，语法如下。

```
multi
```

执行以上命令后，接下来的所有命令都将在这个事务中执行。

执行事务，语法如下。

```
exec
```

exec命令会将multi和exec中的操作一并提交。

取消事务，语法如下。

```
discard
```

discard命令会将multi和discard中的操作全部取消。

监视key，语法如下。

```
watch key1 key2...
```

以上命令监视一个或者多个key，如果在事务执行之前，这些key被其他命令修改了，那么事务将中断执行。

取消所有key的监视，语法如下。

```
unwatch
```

## 10. 发布 / 订阅操作

Redis中的发布 / 订阅功能类似于Flask中的信号机制，可以先订阅某个频道，然后在某些事情发生的情况下给某个频道发送消息。下面来学习发布 / 订阅命令。

给某个频道发布消息，语法如下。

```
publish channel message
```

订阅某个频道，语法如下。

```
subscribe channel
```

### 8.2.5 同步数据到硬盘

Redis可以通过配置实现自动同步数据到硬盘。同步的策略有两种：第一种是RDB（Redis database），第二种是AOF（append only file），可以通过修改/etc/redis/redis.conf配置文件切换不同的同步策略。下面分别来讲解这两种策略的相关配置及其优缺点。

#### 1. RDB策略

RDB策略的相关配置内容如下。

- 开启和关闭：**RDB默认是开启模式的，如果想要关闭，则把配置文件中所有的save配置项都注释就可以关闭了。
- 同步机制：**可以指定某个时间内执行多少个命令进行同步。如1min内执行了两次命令，就做一次同步。
- 存储内容：**存储的是Redis里面具体的值。
- 存储文件路径：**根据dir和dbfilename配置项来指定路径和具体的文件名。

RDB策略的优点如下。

- 存储数据到文件中会进行压缩，文件体积比使用AOF策略小。
- 因为存储的是Redis中具体的值，所以在恢复的时候速度比使用AOF策略快。
- 非常适合用于备份。

RDB策略的缺点如下。

- RDB配置规则为多少时间内执行了多少条命令才进行同步数据，因为采用压缩机制，RDB在同步的时候要重新保存整个Redis中的数据，因此一般我们最少会设置5min以上才进行一次同步。在这种情况下，一旦服务器发生故障，就会造成5min的数据丢失。
- 在进行数据同步时，Redis会调用fork函数生成一个子进程来同步数据，在数据量比较大的情况下，会非常耗时。

## 2. AOF策略

AOF策略的相关配置内容如下。

- 开启和关闭：在/etc/redis/redis.conf配置文件中设置appendonly为yes，即可开启AOF，设置为no则可关闭AOF。
- 同步机制：每秒同步或者每次执行命令后同步数据。
- 存储内容：存储的不是真实的数据，而是每次执行的命令。
- 存储文件路径：根据dir以及appendfilename来指定具体的路径和文件名。

AOF策略有如下优点。

- AOF策略是每秒或者每次发生写操作时都会同步数据，因此即使服务器发生故障，最多只会损失1s的数据。
- AOF存储的是Redis命令，并且直接把新增的命令追加到AOF文件后面，因此备份的效率比使用RDB策略高。
- 如果AOF文件比较大，那么Redis会自动进行重写，只保留最小的命令集合。

AOF策略有如下缺点。

- AOF文件因为没有压缩，因此体积比使用RDB策略大。
- AOF是在每秒或者每次执行命令时都进行备份，因此如果并发量比较高，则效率会比较慢。

- ☑ 因为AOF备份的是命令，因此在灾难恢复时Redis会重新执行AOF中的命令，速度不及使用RDB策略。

## 8.2.6 设置密码

默认情况下，连接Redis是不需要密码的，这存在风险。我们可以通过配置/etc/redis/redis.conf文件中的requirepass实现加密访问，配置示例如下。

```
requirepass zhiliao
```

上述配置中，将密码设置为zhiliao，重启redis-server后通过redis-cli重新连接redis-server，然后使用命令auth password命令授权，示例命令如下。

```
auth zhiliao
```

只有成功授权后，才可以执行操作Redis的命令。

## 8.2.7 Python操作Redis

因为Redis是装在Linux系统上的，而我们经常会在Windows系统上编程，因此如果想要在其他机器上连接Redis，则必须先在/etc/redis/redis.conf文件中设置允许其他机器通过本机的IP地址连接。设置的配置项为bind，示例如下。

```
bind 0.0.0.0
```

通过以上配置，其他机器都可以通过redis-server所在机器的IP地址或域名进行连接了。

使用Python操作Redis，还需要通过一个第三方包python-redis，安装命令如下。

```
pip install redis
```

安装完python-redis后，就可以用Python来操作Redis了，相关操作方法如下。

## 1. 连接Redis

假设Redis所在机器的IP地址为192.168.174.130，并且监听默认的6379端口号，那么使用以下代码即可实现连接Redis。

```
# 从redis包中导入Redis类
from redis import Redis
# 初始化Redis实例变量
xtredis = Redis(host='192.168.174.130',port=6379)
```

## 2. 字符串操作

字符串操作示例代码如下。

```
# 添加一个值进去，并且设置过期时间为60s，如果不设置，则永远不会过期
xtredis.set('username','xiaotuo',ex=60)
# 获取一个值
xtredis.get('username')
# 删除一个值
xtredis.delete('username')
```

## 3. 列表操作

列表操作示例代码如下。

```
# 给languages列表往左边添加一个python
xtredis.lpush('languages','python')
# 给languages列表往左边添加一个php
xtredis.lpush('languages','php')
# 给languages列表往左边添加一个javascript
xtredis.lpush('languages','javascript')

# 获取languages列表中的所有值
print xtredis.lrange('languages',0,-1)
```

## 4. 集合操作

集合操作示例代码如下。

```
# 给集合team添加一个元素xiaotuo
xtredis.sadd('team','xiaotuo')
# 给集合team添加一个元素datuo
xtredis.sadd('team','datuo')
# 给集合team添加一个元素slice
xtredis.sadd('team','slice')

# 获取集合中的所有元素
xtredis.smembers('team')
> ['datuo','xiaotuo','slice'] # 无序的
```

## 5. 哈希操作

哈希操作示例代码如下。

```
# 给website哈希中添加baidu
xtredis.hset('website','baidu','baidu.com')
# 给website哈希中添加google
xtredis.hset('website','google','google.com')

# 获取website哈希中的所有值
print xtredis.hgetall('website')
> {"baidu":"baidu.com","google":"google.com"}
```

## 6. 事务操作

事务操作示例代码如下。

```
# 定义一个管道实例
pip = xtredis.pipeline()
# 做第一步操作，给BankA自增长1
pip.incr('BankA')
# 做第二步操作，给BankB自减少1
pip.desc('BankB')
# 执行事务
pip.execute()
```

## chapter 9 第9章 项目实战

从本章开始进入论坛项目实战阶段，将给大家讲到的是一个Python论坛项目的实现，项目的名称叫作pythonbbs，这会把之前的Flask基础知识融入进去，还会讲解一些真实企业项目功能的解决方案，如文件上传、邮件发送、头像处理、权限管理等。读者在学习完本项目实战后，可以把这些解决方案直接应用到实际开发中。

我们在开发一个产品之前，先要有项目需求文档。一个网站的大概的制作流程如下。

- (1) 公司相关领导提出产品制作计划，描述产品需求。
- (2) 产品经理整理需求，细化功能，制作产品原型图。
- (3) 设计师定主色调，按照原型图制作产品设计图。
- (4) 网站开发者按照原型图和设计图完成产品的制作。
- (5) 测试工程师对产品进行功能和性能测试，如有问题提交给网站开发者，开发者继续完善，直至测试通过。
- (6) 运维工程师部署产品上线，并负责保证产品正常运行。

为了节省时间，我们提前设定好了产品需求和HTML静态页面，所以无须经过前3个步骤的操作。在接下来的小节中将分别详细介绍后面3个步骤。

在学习开发pythonbbs项目之前，我们先来了解本项目的需求。本项目主要由两部分组成，分别为论坛前台和论坛后台。论坛前台功能包括登录、注册、查看帖子、过滤帖子、发布帖子、帖子详情、帖子评论、帖子点赞等功能。论坛后台功能包括论坛首页、帖子管理、评论管理、用户管理、权限管理、角色管理等。将以上项目需求绘制成思维导图，如图9-1所示。

在图9-1中，CMS是Content Management System的缩写，即内容管理系统，也就是论坛后台。了解了产品需求以后，下面就开始创建项目，实现产品功能。



图9-1 Python论坛项目思维导图

## 9.1 创建项目

首先打开PyCharm专业版，选择Flask项目，然后填写项目路径，并且选择Python Interpreter为New Virtualenv environment，也就是为本项目创建一个虚拟环境，这和第1章所提倡的用Previously configured interpreter（以下简称Previously）不同。之前提倡用Previously的原因是，避免每次创建学习项目时都要重新安装Flask。而现在创建的pythonbbs项目在开发完成后，需要部署到Linux服务器上运行，在部署前可以通过pip freeze→requirements.txt命令将pythonbbs项目依赖的包及其版本号全部导出来，方便后期部署，如果使用Previously，则会把一些不需要的包也导出来。下面使用PyCharm专业版创建pythonbbs项目，如图9-2所示。

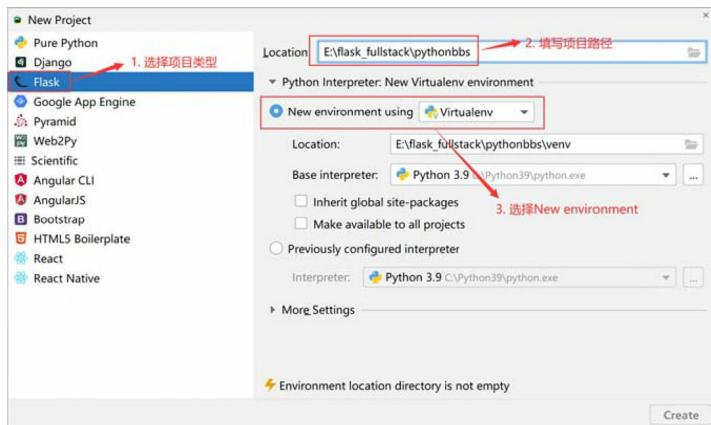


图9-2 PyCharm专业版创建pythonbbs项目

### 注意

虚拟环境是一个独立的Python环境，安装的包不会影响其他环境，其他环境安装的包也不会影响虚拟环境。

单击Create按钮后即可创建pythonbbs项目，pythonbbs项目的结构如图9-3所示。

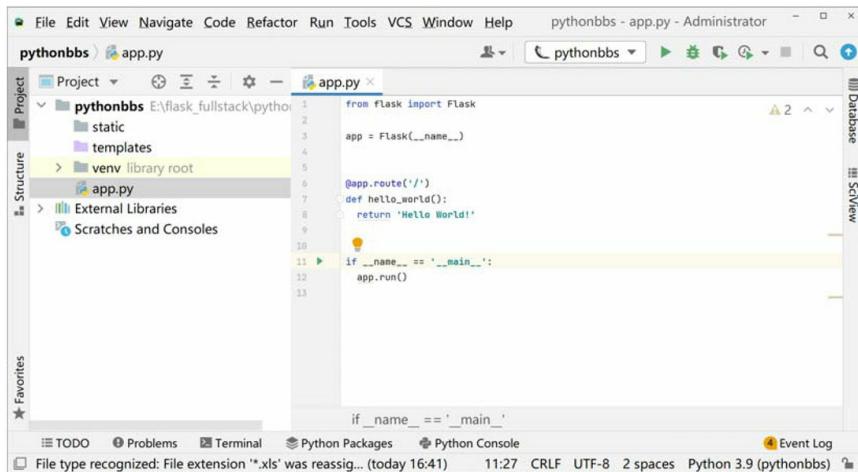


图9-3 pythonbbs项目结构

创建完项目后，PyCharm会默认在当前项目的虚拟环境中安装Flask，但是其他第三方插件是没有安装的，因此我们需要先把后期开发中会用到的包提前安装好。安装方法为打开PyCharm界面底部的Terminal，然后输入安装命令。以安装flask-sqlalchemy为例，操作步骤如图9-4所示。

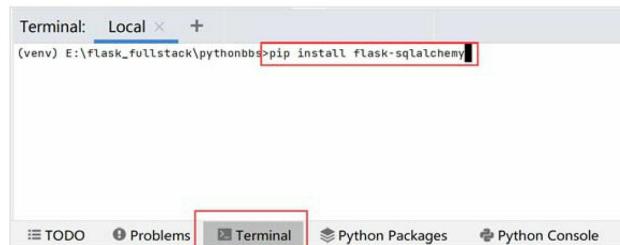


图9-4 PyCharm中安装flask-sqlalchemy的步骤

### 注意

之所以在PyCharm Terminal中安装包，是因为我们现在用了项目中的虚拟环境，直接打开PyCharm Terminal会进入虚拟环境中，此时通过pip install命令安装的包会安装在项目的虚拟环境中。如果在cmd（命令行终端）中安装，则会安装在系统的Python环境中，项

目的虚拟环境和系统的Python环境是两个独立的环境，互不影响。

项目需要提前安装的包，包括但不限于以下所示的包。

## 1. pymysql

- 安装命令：pip install pymysql。
- 作用：Python操作数据库的驱动程序。

## 2. Flask-SQLAlchemy

- 安装命令：pip install flask-sqlalchemy。
- 作用：用于在Flask中使用ORM模型操作数据库。

## 3. cryptography

- 安装命令：pip install cryptography。
- 作用：对密码加密和解密。

## 4. Flask-Migrate

- 安装命令：pip install flask-migrate。
- 作用：用于将ORM模型的变更同步到数据库中。

包安装完后，我们以大型项目为标准完善项目结构，把程序框架搭建起来。

### 9.1.1 config.py文件

在pythonbbs项目根路径下创建一个名叫config.py的Python文件，这个文件用来存放配置项。在项目运行过程中，会根据环境选择不同的配置。如以数据库连接配置为例，在开发时，可能连接的是开发环境的数据库；在测试时，可能连接的是测试服务器的数据库；而在上线后，则需要更换成线上服务器的数据库。为了满足不同环境下不同的配置，我们在config.py文件中根据环境创建不同的类，来分别实现具体的配置。如开发环境则创建DevelopmentConfig类，测试环境则创建TestingConfig类，线上环境则创建ProductionConfig类。还有一些在任何环境下都相同的配置项，我们再为其创建一个BaseConfig类，让以上3个类继承即可。config.py文件中的代码如下。

```
class BaseConfig:  
    SECRET_KEY = "your secret key"  
    SQLALCHEMY_TRACK_MODIFICATIONS = False  
  
class DevelopmentConfig(BaseConfig):  
    SQLALCHEMY_DATABASE_URI = "mysql+pymysql://root:root@127.0.0.1:3306/  
pythonbbs?charset=utf8mb4"  
  
class TestingConfig(BaseConfig):  
    SQLALCHEMY_DATABASE_URI = "mysql+pymysql://[测试服务器MySQL用户名]:[测试  
服务器MySQL密码]@[测试服务器MySQL域名]:[测试服务器MySQL端口号]/pythonbbs?  
charset=utf8mb4"  
  
class ProductionConfig(BaseConfig):  
    SQLALCHEMY_DATABASE_URI = "mysql+pymysql://[生产环境服务器  
名]:[生产环境服务器MySQL密码]@[生产环境服务器MySQL域名]:[生产环境服务器MySQL  
端口号]/pythonbbs?charset=utf8mb4"
```

接下来在app.py文件中，根据当前环境选择不同的配置类即可。这里以开发环境为例，app.py文件中绑定配置的代码如下。

```
from flask import Flask  
import config  
  
app = Flask(__name__)  
app.config.from_object(config.DevelopmentConfig)
```

在上述代码的DevelopmentConfig中，配置了数据库连接SQLALCHEMY\_DATABASE\_URI，读者可以根据自身情况选择MySQL服务器的域名、端口号、用户名、密码。此外，还需要先在MySQL数据库中创建pythonbbs数据库，在创建数据库时，选择字符集为utf8mb4，如图9-5所示。

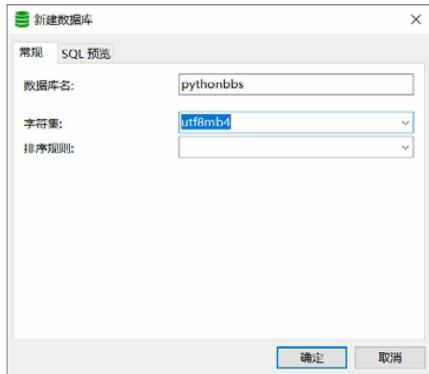


图9-5 创建pythonbbs数据库

### 9.1.2 exts.py文件

exts.py文件主要用来存放一些第三方插件的对象，如SQLAlchemy对象、Flask-Mail对象等。为什么要单独创建一个文件用来存放这些对象呢？这样做是为了防止循环引用。以 SQLAlchemy对象为例，一般会在app.py文件中通过以下代码创建一个db变量，用于创建ORM模型和进行ORM操作。

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy(app)
```

在项目越来越复杂的情况下，为了保持项目的可维护性，通常会把ORM模型放到其他文件中，而创建ORM模型又需要db变量，因此需要从app.py中导入db变量，而为了让ORM模型能被映射到数据库中，又需要把ORM模型直接或者间接导入app.py，这样就产生了循环引用，如图9-6所示。

循环引用会导致项目运行失败。为了打破循环引用，只要在两者中间加一个exts.py文件，把会引起循环引用的变量（如db变量）放到exts.py中，然后其他文件都从exts.py中导入变量。这里还是以db变量为例，在添加exts.py后，三者的关系如图9-7所示。

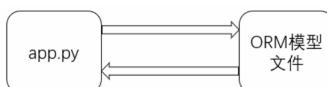


图9-6 循环引用

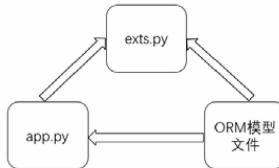


图9-7 使用exts.py打破循环引用

这里使用Flask-SQLAlchemy插件来创建一个SQLAlchemy对象，在exts.py中输入以下代码。

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

读者可以看到，上述代码在创建SQLAlchemy对象时，并没有传入app，原因是如果使用app.py中的app变量，那么又会产生循环引用。此时可以再回到app.py中，然后导入db变量，再通过db.init\_app(app)完成初始化，代码如下。

```
from flask import Flask
import config
from exts import db

app = Flask(__name__)
app.config.from_object(config.DevelopmentConfig)

db.init_app(app)
```

按以上代码一样可以对db变量完成初始化，并且还解决了循环引用的问题。后续其他第三方插件对象，如用于发送邮件的Flask-Mail插件的对象，都可以通过类似的方式实现。

### 9.1.3 blueprints模块

从图9-1所示的项目思维导图可知，项目被分成了许多模块。如果把这些模块的视图代码都放到app.py中，那么对后期维护将是一场灾难。为了让项目结构更加清晰，我们通常会使用蓝图来模块化，创建一个名叫blueprints的包，用于存放蓝图模块。首先在pythonbbs项目名称上右击，然后在弹出的快捷菜单中选择New→Python Package命令，如图9-8所示。

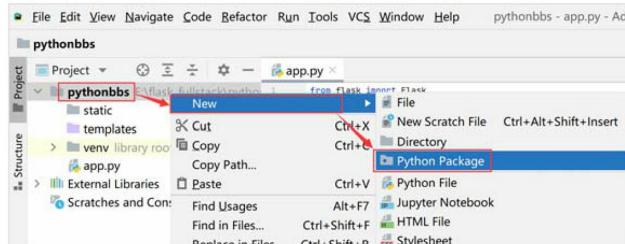


图9-8 创建blueprints包

输入blueprints，即可完成创建。然后在blueprints下分别创建名为cms、front和user的Python文件。创建完成后的项目结构如图9-9所示。

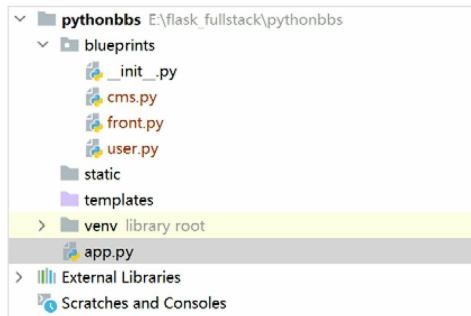


图9-9 创建blueprints后的项目结构图

接下来继续完善蓝图，分别在cms.py、front.py、user.py中创建蓝图对象，相关文件代码如下。

cms.py：

```
from flask import Blueprint  
  
bp = Blueprint("cms", __name__, url_prefix="/cms")
```

front.py：

```
from flask import Blueprint
```

```
bp = Blueprint("front",__name__,url_prefix="")
```

user.py:

```
from flask import Blueprint

bp = Blueprint("user",__name__,url_prefix="/user")
```

3个文件中都创建了蓝图对象，并且指定了url前缀，因为front是面向前台的，所以url前缀为空。在蓝图对象创建好之后，还需要在app.py中完成注册，否则是无法使用的。在app.py中注册蓝图的代码如下。

```
...
from blueprints.cms import bp as cms_bp
from blueprints.front import bp as front_bp
from blueprints.user import bp as user_bp

app = Flask(__name__)
app.config.from_object(config.DevelopmentConfig)

db.init_app(app)

# 注册蓝图
app.register_blueprint(cms_bp)
app.register_blueprint(front_bp)
app.register_blueprint(user_bp)
```

因为在后续开发中，所有前台的视图代码都会放到front蓝图中，所以在app.py中遗留的hello\_world相关代码（如下所示），可以直接删除。

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

## 9.1.4 models模块

为了让项目更加简洁，我们把所有的ORM模型也进行模块化，按照如图9-8所示的方

法，在pythonbbs根路径下创建一个名为models的Python Package，然后在models下分别创建user.py和post.py文件，这两个文件分别用来存放与用户和帖子相关的ORM模型。ORM模型后续再添加，models模块结构如图9-10所示。

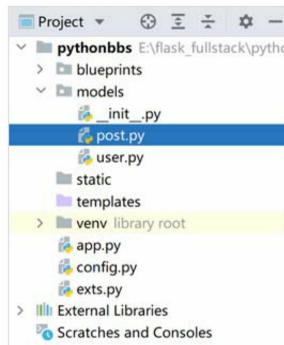


图9-10 models模块结构

到目前为止，pythonbbs项目结构就已经搭建好了。下面再讲解实现每个功能的技术细节。

## 9.2 创建用户相关模型

### 9.2.1 创建权限和角色模型

一个网站最开始做的功能应该是用户系统，因为后面许多功能都需要与用户系统交互，用户系统最核心的部分就是用户相关的ORM模型。该系统的前台和后台用的是同一个用户系统，而后台系统中需要角色和权限管理。首先来添加权限ORM模型，在models/user.py中添加以下代码。

```
class PermissionEnum(Enum):
    BOARD = "板块"
    POST = "帖子"
    COMMENT = "评论"
    FRONT_USER = "前台用户"
    CMS_USER = "后台用户"

class PermissionModel(db.Model):
    __tablename__ = "permission"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
```

```
name = db.Column(db.Enum(PermissionEnum), nullable=False, unique=True)
```

上述代码中，添加了PermissionEnum枚举类型和PermissionModel模型，为了在程序中更好地分辨普通类和ORM模型，在所有ORM模型的名称后面加上Model后缀。

PermissionEnum是存放权限类型的枚举，下面的PermissionModel中name的值就需要从这个枚举中获取。PermissionModel中有两个字段，分别是主键id以及权限名称，需要指定权限名称不能为空，且值也是唯一的。后期可以根据业务需求添加权限，如管理帖子的权限、管理评论的权限等，没有相应权限的用户则无法执行相关操作。PermissionModel不是直接和用户关联，而是先跟角色关联，角色再和用户关联。在执行某个操作时，会先判断用户所属的角色是否包含对应的权限。其中角色和权限属于多对多的关系，即一个权限可以被多个角色拥有，一个角色也可以拥有多个权限。下面再来实现角色模型，在models/user.py中添加以下代码。

```
from exts import db
from datetime import datetime
from enum import Enum

class PermissionEnum(Enum):
    BOARD = "板块"
    POST = "帖子"
    COMMENT = "评论"
    FRONT_USER = "前台用户"
    CMS_USER = "后台用户"

class PermissionModel(db.Model):
    __tablename__ = "permission"
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Enum(PermissionEnum), nullable=False, unique=True)

    role_permission_table = db.Table(
        "role_permission_table",
        db.Column("role_id", db.Integer, db.ForeignKey("role.id")),
        db.Column("permission_id", db.Integer, db.ForeignKey("permission.id"))
    )

class RoleModel(db.Model):
    __tablename__ = 'role'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(50), nullable=False)
    desc = db.Column(db.String(200), nullable=True)
    create_time = db.Column(db.DateTime, default=datetime.now)
```

```
permissions      = db.relationship("PermissionModel", secondary=permission_table, backref="roles")
```

上述代码中，添加了一个RoleModel模型，并且给该模型添加了4个常规字段，分别是主键id、角色名称name、角色描述desc，以及创建时间create\_time。除了这4个常规字段外，还添加了一个关系属性permissions，并与PermissionModel进行了关联，因为RoleModel和PermissionModel属于多对多的关系，所以在db.relationship中通过secondary参数设置中间表为role\_permission\_table，在role\_permission\_table中也分别添加了外键role\_id和permission\_id来引用role和permission表。另外，还在db.relationship中指定了backref参数值为roles，以后通过PermissionModel对象的roles属性即可访问到该权限下所有与其关联的角色。

在PermissionModel和RoleModel创建完成后，我们再把模型映射到数据库中。这里需要借助Flask-Migrate插件，下面返回app.py中，创建Migrate对象，代码如下。

```
...
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(config.DevelopmentConfig)
migrate = Migrate(app, db)
...
```

下面重新打开PyCharm Terminal，输入以下命令完成迁移环境的初始化。

```
$ flask db init
```

现在models/user.py模块并没有被直接或者间接地导入app.py，为了在迁移时能让程序识别到models/user.py中的ORM模型，先手动在app.py中导入models/user.py，代码如下。

```
...
from models import user
...
```

接着同样在PyCharm的Terminal中执行以下命令，以生成迁移脚本并完成迁移脚本的执行，执行migrate命令后的效果如图9-11所示。

```
flask db migrate -m "create permission and role model"
```

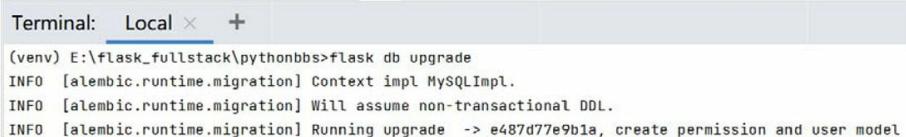


```
Terminal: Local +  
(venv) E:\flask_fullstack\pythonbbs>flask db migrate -m "create permission and role model"  
INFO [alembic.runtime.migration] Context impl MySQLImpl.  
INFO [alembic.runtime.migration] Will assume non-transactional DDL.  
INFO [alembic.autogenerate.compare] Detected added table 'permission'  
INFO [alembic.autogenerate.compare] Detected added table 'role'  
INFO [alembic.autogenerate.compare] Detected added table 'role_permission_table'  
Generating E:\flask_fullstack\pythonbbs\migrations\versions\e487d77e9b1a_create_permission_and_role_model.py ... done
```

图9-11 执行migrate命令后的效果图

执行完以上命令后，已经为ORM模型生成了迁移脚本，路径在图9-11中的最后一行。但是此时并没有真正同步到数据库中，因此还需要执行以下命令才会同步到数据库中，执行upgrade命令后的效果如图9-12所示。

```
flask db upgrade
```



```
Terminal: Local +  
(venv) E:\flask_fullstack\pythonbbs>flask db upgrade  
INFO [alembic.runtime.migration] Context impl MySQLImpl.  
INFO [alembic.runtime.migration] Will assume non-transactional DDL.  
INFO [alembic.runtime.migration] Running upgrade  -> e487d77e9b1a, create permission and role model
```

图9-12 执行upgrade命令后的效果图

## 9.2.2 创建权限和角色

权限和角色模型已经创建完成，为了方便后期开发，需要在这两张表中添加数据。一般情况下，权限和角色的数据在网站上线运营后便不会轻易更改。我们在实际开发中可以跟产品经理或者运营同事沟通，确定好权限规则以及角色安排，然后在项目中把这些数据写好，并且集成到命令中，项目上线时，只要执行这条命令即可完成数据的初始化。

我们首先来学习在Flask项目中如何创建命令。在安装Flask时，默认会安装click库，读者依次单击PyCharm的File→Settings→Project: pythonbbs→Python Interpreter，可以看到已经安装了click库，如图9-13所示。

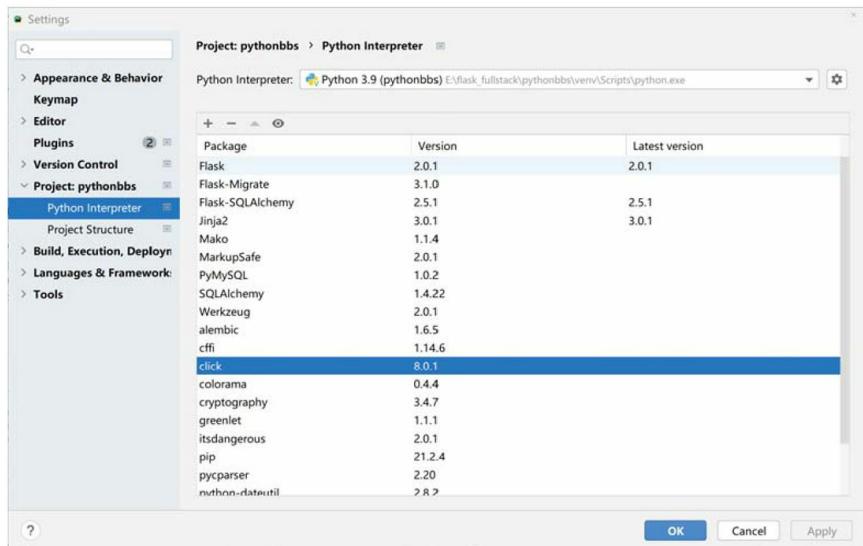


图9-13 查看是否安装click库

click库的主要作用就是用来实现命令，Flask中已经针对click库进行了集成，通过app.cli即可访问到click对象。我们先来实现一个简单的命令，在app.py中输入以下代码。

```
import click
...
@app.cli.command("my-command")
def my_command():
    click.echo("这是我自定义的命令")
...
```

上述代码中，通过@app.cli.command装饰器将my\_command函数添加到命令中，并且指定命令的名称为my-command，然后在PyCharm的Terminal中输入以下命令。

```
$ flask my-command
```

执行命令即可看到打印文字“这是我自定义的命令”，效果如图9-14所示。

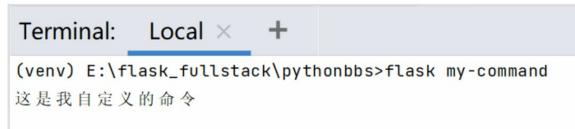


图9-14 执行my-command命令后的效果图

学会了如何在Flask中集成命令后，我们再来添加创建权限和角色的命令。按照项目需求，权限和角色是针对后台用户的，因此权限和角色都是针对后台数据管理的。后台需要管理的数据有板块、帖子、评论、前台用户、后台用户，我们针对每个模块分别添加一个权限。在app.py中，实现一个名叫create-permission的命令，代码如下。

```
from models.user import PermissionModel,RoleModel,PermissionEnum
import click
from exts import db

@app.cli.command("create-permission")
def create_permission():
    for permission_name in dir(PermissionEnum):
        if permission_name.startswith("__"):
            continue
        permission = PermissionModel(name=getattr(PermissionEnum,permission_name))
        db.session.add(permission)
    db.session.commit()
    click.echo("权限添加成功! ")
```

下面在PyCharm的Terminal中输入命令flask create-permission，看到输出文字“权限添加成功！”，说明权限数据已经创建成功，效果如图9-15所示。

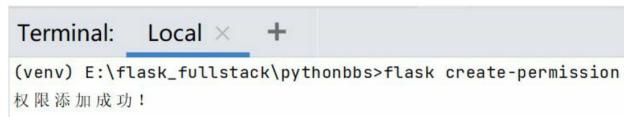


图9-15 执行添加权限命令后的效果图

权限数据创建成功后，我们再来添加角色。这里创建3个角色，分别为稽查、运营、管理员。这3个角色包含的权限如表9-1所示。

表9-1 角色包含的权限

角 色	权 限
稽查	帖子、评论
运营	板块、帖子、评论、前台用户
管理员	板块、帖子、评论、前台用户、后台用户

稽查角色主要是审核用户发布的帖子和评论是否存在违法或者违反社区正常运营的情况。如果存在，则进行处理。而运营角色则有较大权限，可以管理除后台用户以外的其他所有功能。管理员角色除拥有运营角色下所有权限以外，还拥有管理后台用户的权限，即可以设置谁为稽查、谁为运营等。根据以上需求，创建添加角色的命令create-role，代码如下。

```
@app.cli.command("create-role")
def create_role():
    # 稽查
    inspector = RoleModel(name="稽查", desc="负责审核帖子和评论是否合法合规!")
    inspector.permissions = PermissionModel.query.filter(PermissionModel.
name.in_([PermissionEnum.POST, PermissionEnum.COMMENT])).all()

    # 运营
    operator = RoleModel(name="运营", desc="负责网站持续正常运营!")
    operator.permissions = PermissionModel.query.filter(PermissionModel.
name.in_([
    PermissionEnum.POST,
    PermissionEnum.COMMENT,
    PermissionEnum.BOARD,
    PermissionEnum.FRONT_USER,
    PermissionEnum.CMS_USER
])).all()

    # 管理员
    administrator = RoleModel(name="管理员", desc="负责整个网站所有工作!")
    administrator.permissions = PermissionModel.query.all()

    db.session.add_all([inspector, operator, administrator])
    db.session.commit()
    click.echo("角色添加成功!")
```

上述代码中，创建了3个RoleModel对象，分别为稽查inspector、运营operator、管理员administrator，并且按照表9-1分别给3个对象设置了permissions属性。完成命令编写后，在PyCharm的Terminal下执行命令flask create-role，效果如图9-16所示。

```
Terminal: Local
(venv) E:\flask_fullstack\pythonbbs>flask create-role
角色添加成功！
```

图9-16 执行create-role命令后的效果图

现在虽然成功添加了命令，但是命令代码全部写在app.py中，如果以后还要增加其他命令，那么会导致app.py越来越臃肿。为了给app.py瘦身，我们把命令代码单独放到一个模块中，在pythonbbs项目根路径下创建一个commands.py文件，然后把create-permission和create-role函数及其装饰器代码剪切到commands.py文件中，并且再把命令代码中依赖的包也剪切过去，剪切后的commands.py文件内容如图9-17所示。

```
commands.py
1  from models.user import PermissionModel,RoleModel,PermissionEnum
2  import click
3  from exts import db
4
5
6  @app.cli.command("create-permission")
7  def create_permission():
8      for permission_name in dir(PermissionEnum):
9          if permission_name.startswith("__"):
10              continue
11          permission = PermissionModel(name=getattr(PermissionEnum,permission_name))
12          db.session.add(permission)
13          db.session.commit()
14          click.echo("权限添加成功! ")
15
16
17  @app.cli.command("create-role")
18  def create_role():
19      # 稿查员
20      inspector = RoleModel(name="稿查员",desc="负责审核帖子和评论是否合法合规！")
21      inspector.permissions = PermissionModel.query.filter(PermissionModel.name.in_([PermissionEnum.POST,PermissionEnum.COMMENT])).all()
22
23      # 运营
24      operator = RoleModel(name="运营",desc="负责网站持续正常运营！")
25      operator.permissions = PermissionModel.query.filter(PermissionModel.name.in_[
```

图9-17 命令代码剪切到commands.py文件中

从图9-17可以看到，commands.py中有两个错误，原因是create\_permission和create\_role的@app.cli.command装饰器中的app对象找不到。如果从app.py中导入app对象，那么会造成循环引用，只有把commands.py导入app.py才能把命令注册到项目中。为了解决这个问题，把create\_permission和create\_role函数上的@app.cli.command装饰器删除，然后在app.py中手动

添加，修改后的代码如下，效果如图9-18所示。

```
...
import commands
...
# 添加命令
app.cli.command("create-permission")(commands.create_permission)
app.cli.command("create-role")(commands.create_role)
```

The screenshot shows two code editors side-by-side. The left editor is titled 'commands.py' and contains Python code for creating permissions and roles. The right editor is titled 'app.py' and contains the main application setup. A red box highlights the import statement 'import commands' in app.py, and another red box highlights the command registrations '# 添加命令' in app.py.

```
commands.py
1  from models.user import PermissionModel, PermissionEnum
2  import click
3  from exts import db
4
5
6  def create_permission():
7      for permission_name in dir(PermissionEnum):
8          if permission_name.startswith("__"):
9              continue
10         permission = PermissionModel(name=getattr(
11             PermissionEnum, permission_name))
12         db.session.add(permission)
13         db.session.commit()
14         click.echo("权限添加成功！")
15
16  def create_role():
17      # 检查权限
18      inspector = RoleModel(name="稿森", desc="负责审核帖子和评论是否合法合规！")
19      inspector.permissions =
20      PermissionModel.query.filter(
21          PermissionModel.name.in_([PermissionEnum.POST,
22          PermissionEnum.COMMENT])).all()
23
24
25
26
27
28
```

```
app.py
1  from flask import Flask
2  from exts import db
3  import config
4  from flask_migrate import Migrate
5  import commands
6
7  app = Flask(__name__)
8  app.config.from_object(config.DevelopmentConfig)
9  db.init_app(app)
10
11 migrate = Migrate(app, db)
12
13 # 添加命令
14 app.cli.command("create-permission")(commands.create_permission)
15 app.cli.command("create-role")(commands.create_role)
16
17
18
19
20 if __name__ == '__main__':
21     app.run()
```

图9-18 重构命令后的效果图

在Python语法中，装饰器本质上是函数，所以可以把@app.cli.command装饰器直接当作函数来使用。通过以上代码重构，在不产生循环引用的前提下，就实现了将app.py瘦身的目的。

### 9.2.3 创建用户模型

接下来回到models/user.py文件中创建用户模型。用户数据是网站最重要的数据之一，如果把用户表主键存储为自增长的整型，则容易被竞争对手猜测出总共有多少用户，猜测方法非常简单，一般网站都有查看用户信息的个人中心页面，而个人中心页面的URL中必须携带用户ID参数，如果用户ID是自增长的整型，则竞争对手只要获取到用户ID的最大值，也就知道了此网站用户的数量，很有可能为公司带来巨大损失。因此在商业网站中，都用唯一的

字符串作为用户表的主键。产生唯一字符串方法的库有许多，最常用的是UUID（universally unique identifier），UUID会出现重复值的概率几乎为零，可以忽略不计。UUID的长度为32个字符，再加上4个横线，总共有36个字符，考虑到UUID太长会对数据库查询造成性能上的影响，我们使用shortuuid。shortuuid是一个第三方Python库，会对原始UUID进行base57编码，然后删除相似的字符，如I、1、L、o和0，最后生成默认22位长度的字符串。打开PyCharm的Terminal，输入以下命令安装shortuuid。

```
$ pip install shortuuid
```

在使用的时候，直接从shortuuid中导入uuid方法，然后调用该方法，便会自动生成一串唯一的字符串，如图9-19所示。



```
Terminal: Local × +
(venv) E:\flask_fullstack\pythonbbs>python
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from shortuuid import uuid
>>> uuid()
'fq6RZdFwUgHHF8VLeH6aZV'
>>> []
```

图9-19 python环境下使用uuid

安装完shortuuid后，就可以将其应用到用户模型中。我们再回到models/user.py中，添加以下代码。

```
...
from shortuuid import uuid
...

class UserModel(db.Model):
    __tablename__ = 'user'
    id = db.Column(db.String(100), primary_key=True, default=uuid)
    username = db.Column(db.String(50), nullable=False, unique=True)
    password = db.Column(db.String(200), nullable=False)
    email = db.Column(db.String(50), nullable=False, unique=True)
    avatar = db.Column(db.String(100))
    signature = db.Column(db.String(100))
    join_time = db.Column(db.DateTime, default=datetime.now)
    is_staff = db.Column(db.Boolean, default=False)
    is_active = db.Column(db.Boolean, default=True)
```

```
# 外键  
role_id = db.Column(db.Integer, db.ForeignKey("role.id"))  
role = db.relationship("RoleModel", backref="users")
```

上述代码中，我们添加了UserModel模型，关于UserModel的字段及相关介绍如下。

- id:** 主键，字符串类型，默认会使用shortuuid.uuid函数自动生成主键。
- username:** 用户名，不能为空，并且值必须唯一。
- password:** 密码，最大长度为200，应该先加密再存储。
- email:** 邮箱，不能为空，值唯一，作为登录的凭证。
- avatar:** 头像，存储图片在服务器中保存的路径，可以为空。
- signature:** 签名，可以为空。
- join\_time:** 加入时间，在第一次创建时会使用当前时间存储。
- is\_staff:** 是否是员工，只有员工才能进入后台系统，默认为False。
- is\_active:** 是否可用，默认情况下是可用的，如果不可用，则会限制其登录。
- role\_id:** 角色外键，引用role表的id字段。
- role:** 关系属性，引用RoleModel。反过来，也可以通过RoleModel对象的users属性获取此角色下的所有用户。

用户的密码数据，必须经过加密才能存储进去，这样做的目的是，即使服务器遭到黑客攻击，用户数据被泄露，黑客也只能获取到加密的密码，而不是原始密码。人们面对繁多的互联网产品，为了方便记忆，通常会设置同一个密码，如果黑客能获取到原始密码，并且通过某些手段获取到了用户在其他平台的账号，则大概率在其他平台也能登录成功，这对用户来讲无疑是灾难性的。在Flask项目中，可以通过werkzeug.security模块中的以下两个方法实现密码的加密和验证。

(1) `generate_password_hash(password)`: 对password进行加密，并返回加密后的密码。

(2) `check_password_hash(pwhash,password)`: pwhash是加密后的密码，password是原始密码，将password按照相同的加密方式加密，然后与pwhash进行对比，如果相等则认为密码正确，否则认为密码错误。

现在使用UserModel来添加一条数据，代码如下。

```
from werkzeug.security import generate_password_hash
```

```
user = UserModel(username='example', email="example@domain.com")
user.password = generate_password_hash("password")
```

为了简化创建用户时生成密码的代码，我们将UserModel进行重构，重构的思路是，在创建用户时，直接传password进去就会自动完成加密；或者如果用户通过user.password="password"的方式设置密码时，也要自动完成加密。重构后的UserModel代码如下。

```

class UserModel(db.Model):
    __tablename__ = 'user'
    id = db.Column(db.String(100), primary_key=True, default=uuid)
    username = db.Column(db.String(50), nullable=False)
    # 将 password 修改成 _password
    _password = db.Column(db.String(200), nullable=False)
    email = db.Column(db.String(50), nullable=False, unique=True)
    avatar = db.Column(db.String(100))
    signature = db.Column(db.String(100))
    join_time = db.Column(db.DateTime, default=datetime.now)
    is_admin = db.Column(db.Boolean, default=False)

    # 外键
    role_id = db.Column(db.Integer, db.ForeignKey("role.id"))
    role = db.relationship("RoleModel", backref="users")

    def __init__(self, *args, **kwargs):
        if "password" in kwargs:
            self._password = kwargs.get('password')
            kwargs.pop("password")
        super(UserModel, self).__init__(*args, **kwargs)

    @property
    def password(self):
        return self._password

    @password.setter
    def password(self, raw_password):
        self._password = generate_password_hash(raw_password)

    def check_password(self, raw_password):
        result = check_password_hash(self.password, raw_password)
        return result

    def has_permission(self, permission):
        return permission in [permission.name for permission in self.role.permissions]

```

上述代码中，为了实现通过password属性设置密码时能自动加密，把原来的password属性修改成了\_password。然后通过Python中的@property装饰器将password()方法定义成属性，以后通过user.password可以获取加密后的密码，通过user.password="password"会触发@password.setter下的password方法，在这个方法中把原始密码加密后赋值给了\_password属性。为了以后验证密码方便，实现了check\_password方法，该方法调用了check\_password\_hash，以后通过user.check\_password("password")即可返回密码是否正确。我

们还定义了has\_permission方法，用来快速判断当前用户是否拥有某个权限。至此，UserModel模型就创建成功了，打开PyCharm的Terminal，然后输入以下两条命令即可完成数据库的同步更新。

```
$ flask db migrate -m "create user model"
$ flask db upgrade
```

## 9.2.4 创建测试用户

为了方便后续开发，按照角色个数创建3个员工账号。在commands.py文件中，添加以下代码。

```
...
def create_test_user():
    admin_role = RoleModel.query.filter_by(name="管理员").first()
    zhangsan = UserModel(username="张三",email="zhangsan@zlkt.net",
password="111111",is_staff=True,role=admin_role)

    operator_role = RoleModel.query.filter_by(name="运营").first()
    lisi = UserModel(username="李四",email="lisi@zlkt.net",password=
"111111",is_staff=True,role=operator_role)

    inspector_role = RoleModel.query.filter_by(name="稽查").first()
    wangwu = UserModel(username="王五",email="wangwu@zlkt.net",password=
"111111",is_staff=True,role=inspector_role)

    db.session.add_all([zhangsan,lisi,wangwu])
    db.session.commit()
    click.echo("测试用户添加成功! ")
```

上述代码中，添加了张三、李四、王五3个员工账号，分别对应管理员、运营、稽查3种角色，方便后期进行测试使用。创建测试用户的代码写完后，再回到app.py中，把create\_test\_user集成到项目中，代码如下。

```
# 添加命令
app.cli.command("create-permission") (commands.create_permission)
app.cli.command("create-role") (commands.create_role)
# 添加创建测试用户命令
app.cli.command("create-test-user") (commands.create_test_user)
```

重新打开PyCharm的Terminal，然后输入以下命令。

```
$ flask create-test-user
```

执行以上命令即可把代码中的3个账号添加到数据库中。

### 9.2.5 创建管理员

项目后期在部署到服务器后，应该通过命令完成第一个管理员的初始化。在 commands.py 中添加以下命令。

```
@click.option("--username", '-u')
@click.option("--email", '-e')
@click.option("--password", '-p')
def create_admin(username, email, password):
    admin_role = RoleModel.query.filter_by(name="管理员").first()
    admin_user = UserModel(username=username, email=email, password=password,
                           is_staff=True, role=admin_role)
    db.session.add(admin_user)
    db.session.commit()
    click.echo("管理员创建成功！")
```

上述代码中，在命令函数create\_admin中，通过@click.option装饰器添加了3个参数。以后在命令行中即可使用--username、--email、--password将用户名、邮箱、密码当作参数传到函数中。在app.py中注册命令，代码如下。

```
# 添加创建管理员命令
app.cli.command("create-admin")(commands.create_admin)
```

## 9.3 注册

### 9.3.1 渲染注册模板

虽然通过命令能添加用户，但是网站上线运行后，必须要有界面能让普通用户注册。在讲解注册功能实现之前，先来说明一下我们提前准备好的模板。读者在获取到本项目后，可

以在pythonbbs项目根目录下看到awebiste文件夹，这个文件夹中的模板是笔者为方便读者学习提前准备的，所有模板都是纯静态的。首先执行以下两步操作。

- (1) 将pythonbbs/awebiste/static下的所有文件和文件夹全部复制到pythonbbs/static中。
- (2) 将pythonbbs/awebiste/templates下的所有文件和文件夹全部复制到pythonbbs/templates中。

在后续的章节中，会指导读者修改其中的部分代码。

以上步骤完成后，读者可以看到在pythonbbs/templates/front下有一个base.html文件，代码如下。

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <script src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
    <link href="https://cdn.bootcdn.net/ajax/libs/twitter-bootstrap/4.6.0/css/bootstrap.min.css" rel="stylesheet">
    <script src="https://cdn.bootcdn.net/ajax/libs/twitter-bootstrap/4.6.0/js/bootstrap.min.js"></script>
    <link rel="stylesheet" href="{{ url_for('static', filename='front/css/base.css') }}">
    <title>{% block title %}{% endblock %}</title>
    {% block head %}{% endblock %}
</head>
<body>
<nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="#">知了 Python 论坛</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav mr-auto">
            <li class="nav-item active">
                <a class="nav-link" href="/">首页 <span class="sr-only">(current)</span></a>
            </li>
        </ul>
        <form class="form-inline my-1q-0">
            <input class="form-control mr-sm-2" type="search" placeholder="请输入关键字" aria-label="Search">
            <button class="btn btn-outline-success my-2 my-sm-0" type="submit">搜索</button>
        </form>
        <ul class="navbar-nav ml-4">
            <li class="nav-item">
                <a class="nav-link" href="#">登录</a>
            </li>
            <li class="nav-item">
                <a class="nav-link" href="#">注册</a>
            </li>
        </ul>
    </div>
</nav>
<div class="main-container">
    {% block body %}{% endblock %}
</div>
</body>
</html>
```

base.html文件是所有前台页面的父模板，在<head></head>标签中加载了以下文件。

- jquery.min.js: 3.6.0版本的jQuery文件。jQuery文件可以快速寻找元素，发送AJAX请求。
- bootstrap.min.css: 4.6.0版本的Bootstrap样式文件。Bootstrap提供了丰富的样式，可以快速构建网页界面。
- bootstrap.min.js: 4.6.0版本的Bootstrap JavaScript文件。Bootstrap中的一些组件运行需要通过bootstrap.min.js来实现。
- base.css: 我们自己编写的样式文件，用于修改父模板的样式。

在head标签中还定义了两个block，分别为title、head。在body标签中定义了导航条，这样所有子模板不用重复写导航条代码即可拥有导航条。然后在main-container中定义了一个body的block，子模板页面的编写就是在这个block中实现。

接下来再看templates/front/register.html文件，此文件用于注册页面，代码如下。

```
{% extends 'front/base.html' %}

{% block title %}
    知了课堂注册
{% endblock %}

{% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='front/css/sign.css') }}"/>
{% endblock %}

{% block body %}
    <h1 class="page-title">注册</h1>
    <div class="sign-box">
        <form action="" id="register-form">
            <div class="form-group">
                <div class="input-group">
                    <input type="text" class="form-control" name="email" placeholder="邮箱">
                    <div class="input-group-append">
                        <button id="captcha-btn" class="btn btn-outline-secondary">发送验证码</button>
                    </div>
                </div>
                <div class="form-group">
                    <input type="text" class="form-control" name="captcha" placeholder="邮箱验证码">
                </div>
                <div class="form-group">
                    <input type="text" class="form-control" name="username" placeholder="用户名">
                </div>
                <div class="form-group">
                    <input type="password" class="form-control" name="password" placeholder="密码">
                </div>
                <div class="form-group">
                    <input type="password" class="form-control" name="confirm_password" placeholder="确认密码">
                </div>
                <div class="form-group">
                    <button class="btn btn-warning btn-block" id="submit-btn">立即注册</button>
                </div>
                <div class="form-group">
                    <a href="#" class="signup-link">返回登录</a>
                    <a href="#" class="resetpwd-link" style="float:right;">找回密码</a>
                </div>
            </div>
        </form>
    </div>
{% endblock %}
```

上述代码中，先是让register.html继承自front/base.html，然后分别实现了title、head和body这3个block。title用来设置页面标题，head中加载了一个自定义的sign.css文件，用来美化页面。body这个block中的代码，除了添加一个h1标签外，还添加了输入框的表单，表单中有5个输入框和1个提交按钮。输入框分别用来收集邮箱、邮箱验证码、用户名、密码、确认密码的信息。我们先来渲染模板，读者看到注册页面后会有更直观的感受。回到blueprints/user.py中，添加以下代码。

```
from flask import Blueprint, render_template

bp = Blueprint("user", __name__, url_prefix="/user")

@bp.route("/register")
def register():
    return render_template("front/register.html")
```

上述代码中，实现了register视图，用来渲染front/register.html模板。启动项目，在浏览器中访问http://127.0.0.1:5000/user/register，即可看到如图9-20所示的注册页面。

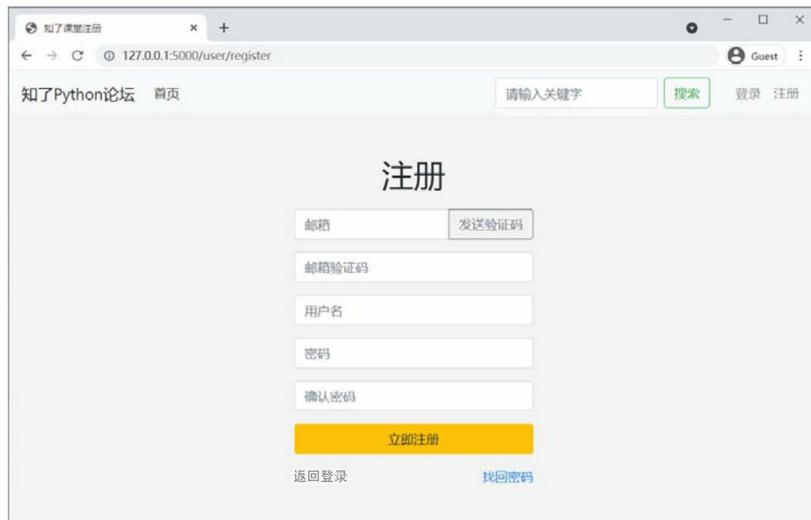


图9-20 注册页面

注册逻辑是，用户输入邮箱后单击“发送验证码”按钮，服务器就会给这个邮箱发送一个

验证码，用户收到邮箱验证码后填入输入框，并完成其他信息的输入，最后单击“立即注册”按钮实现注册。

### 9.3.2 使用Flask-Mail发送邮箱验证码

注册功能的第一步，就是给用户输入的邮箱发送验证码，下面讲解在Flask中发送邮件。在Flask中发送邮件非常简单，总共分为3步，第1步安装Flask-Mail，第2步配置邮箱参数，第3步发送邮件。

#### 1. 安装Flask-Mail

首先安装Flask-Mail，打开PyCharm的Terminal，然后输入并执行以下命令即可完成安装。

```
$ pip install flask-mail
```

Flask-Mail插件的详细使用说明，可参考其官方文档<https://pythonhosted.org/Flask-Mail/>。

#### 2. 配置邮箱参数

想要发送邮件，必须要有一个邮箱服务器。在本书中，为了方便读者学习，我们用个人版网易邮箱来讲解邮箱的配置和使用，读者也可以选择其他公司的邮箱服务，如QQ邮箱、新浪邮箱、Gmail邮箱等，使用方式都大同小异。有的公司会搭建自己的邮箱服务器，有的公司会向第三方提供商购买邮箱服务，如网易企业邮箱、腾讯企业邮箱、阿里邮箱企业版等，这些服务商都有非常详细的接入教程，遇到问题还可以咨询客服。

使用Flask-Mail发送邮件需要使用SMTP协议（simple mail transfer protocol）。首先在个人版网易邮箱中开启SMTP服务，登录个人版网易邮箱，单击顶部的“设置”按钮，然后在弹出的菜单中选择“POP3/SMTP/IMAP”，如图9-21所示。



图9-21 个人网易邮箱设置按钮

进入SMTP设置界面后，找到“IMAP/SMTP服务”，然后单击“开启”超链接，如图9-22所示。



图9-22 开启SMTP界面

单击“开启”超链接后会弹出一个“账号*(1)*安全提示”对话框，因为邮箱只是用于我们自己的项目，只要我们的代码不被公开，还是非常安全的，所以直接单击“继续开启”按钮即可，如图9-23所示。



图9-23 账号安全提示

单击“继续开启”按钮后，会弹出“账号安全验证”对话框，需要用手机发送短信验证码来验证是否是本人操作，按照提示信息进行操作即可，如图9-24所示。

图9-24中显示的二维码，可以使用网易邮箱大师App扫描并发送短信，或者单击“手动发送短信”，会提示你如何手动发送短信。在发送完短信验证码后，单击“我已发送”按钮会弹出“开启IMAP/SMTP”对话框，显示生成的授权密码，因为授权密码只会显示一次，所以读者要复制后保存下来，如图9-25所示。



图9-24 发送验证码开启邮箱



图9-25 显示授权密码

开启个人邮箱的SMTP服务后，再回到pythonbbs项目中，打开config.py文件，在DevelopmentConfig中添加如下配置。

```
class DevelopmentConfig(BaseConfig):
    ...
    # 邮箱配置
    MAIL_SERVER = "smtp.163.com"
    MAIL_USE_SSL = True
    MAIL_PORT = 465
    MAIL_USERNAME = "邮箱账号"
    MAIL_PASSWORD = "开启SMTP服务时生成的授权码"
    MAIL_DEFAULT_SENDER = "邮箱账号"
```

上述代码中关于邮箱配置参数的说明如下。

- MAIL\_SERVER:** 邮箱服务器，如网易是smtp.163.com、QQ邮箱是smtp.qq.com、新浪邮箱是smtp.sina.com。
- MAIL\_USE\_SSL:** 是否加密传输。设置MAIL\_USE\_SSL=True或MAIL\_USE\_TLS=True都可以实现加密传输，但是MAIL\_USE\_SSL用的是SSL/TLS协议，而MAIL\_USE\_TLS用的是STARTTLS协议，具体选择哪个，要根据邮箱服务商支持的协议来配置。网易邮箱不支持STARTTLS，因此使用MAIL\_USE\_SSL=True来配置加密。另外，如果配置MAIL\_USE\_SSL=True，那么MAIL\_PORT应该设置为465；如果配置MAIL\_USE\_TLS=True，那么MAIL\_PORT应该设置为587。

- MAIL\_USERNAME: 发送邮件所用的用户名，设置成邮箱账号即可。
- MAIL\_PASSWORD: 在开启邮箱SMTP服务时自动生成的授权密码。
- MAIL\_DEFAULT\_SENDER: 默认发送者，填邮箱账号即可。

执行以上操作后，即完成了邮箱的配置。

### 3. 发送邮件

发送邮件要先创建一个Flask-Mail对象，其使用方式与Flask-SQLAlchemy类似。先在exts.py中创建一个mail变量，代码如下。

```
from flask_sqlalchemy import SQLAlchemy
from flask_mail import Mail

db = SQLAlchemy()
mail = Mail()
```

下面再回到app.py中，从exts.py中导入mail变量，并进行初始化，代码如下。

```
...
from exts import db, mail

...
db.init_app(app)
mail.init_app(app)
```

以上代码完成了Flask-Mail对象的初始化，后续就可以使用mail变量发送邮件了。接着在blueprints/user.py中输入以下代码。

```
...
from flask_mail import Message
from exts import mail

...
@bp.route("/mail/captcha")
def mail_captcha():
    message = Message(subject="我是邮件主题", recipients=['目标邮箱地址'],
                      body="我是邮件内容")
```

```
mail.send(message)
return "success"
```

上述代码中，我们首先创建了一个mail\_captcha视图函数，然后创建了一个对象flask\_mail.Message, Message类传递的参数如下。

- subject: 邮件主题。
- recipients: 收件方，可以指定多个邮箱地址。
- body: 邮件内容。

创建Message对象后，再调用mail.send方法就可以把邮件发送出去了。

现在已经能正常发送邮件了，接下来再把验证码的逻辑加进去，验证码用四位随机的数字，需要用到Python中的random模块。由于接收验证码的邮箱地址是用户填写的，并不固定，所以我们通过查询字符串的形式接收目标用户的邮箱地址。在mail\_captcha视图函数中，将代码修改如下。

```
...
from flask import Blueprint, render_template, request
from flask_mail import Message
from exts import mail
import random

...
@bp.route("/mail/captcha")
def mail_captcha():
    email = request.args.get("mail")
    digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    captcha = "".join(random.sample(digits, 4))
    body = f"【知了Python论坛】您的注册验证码是: {captcha}，请勿告诉别人！"
    message = Message(subject="我是邮件主题", recipients=[email], body=body)
    mail.send(message)
    return "success"
```

上述代码中，首先通过在查询到的字符串中的mail参数中获取目标用户的邮箱地址。接下来定义了一个0~9的数字列表，类型都是字符串，然后调用random.sample方法随机取4个值，因为random.sample方法的返回结果是列表类型，所以需要使用字符串的join方法将random.sample方法返回的结果合并成字符串，最后再把验证码拼接到body中发给目标用户。

### 9.3.3 使用Flask-Caching和Redis缓存验证码

虽然现在能通过邮箱发送验证码，但是服务器并没有记录邮箱与验证码的映射关系。下次用户提交邮箱和验证码时，我们并不知道该验证码是否正确。邮箱验证码这类数据并不是非常重要，可以存储到缓存中，如Memcached或者Redis，考虑到后期会用Celery来异步发送邮件，Celery又需要Redis作为中间人，因此选择Redis作为缓存。关于Redis的使用，读者可以参考第8章。

要在Python中使用Redis，首先要安装redis包。打开PyCharm的Terminal，输入以下命令。

```
$ pip install redis
```

在Flask中使用Redis，可以借助第三方插件Flask-Caching实现。使用Flask-Caching操作Redis有以下好处。

- 缓存系统：Flask-Caching支持许多缓存系统，如纯内存、Memcached、Redis、FileSystem等。
- 上层接口：Flask-Caching对不同的缓存系统提供了统一的上层接口。以后如果不用Redis作为缓存了，只要修改配置即可，不需要修改上层操作缓存的代码。
- 缓存内容：Flask-Caching可以对Flask项目中的许多内容进行缓存，如视图缓存，模板缓存等。

打开PyCharm的Terminal，输入以下命令完成Flask-Caching的安装。

```
$ pip install flask-caching
```

Flask-Caching支持许多缓存系统，可通过在app.config中设置不同的CACHE\_TYPE选择具体的缓存系统。Flask-Caching支持的常见CACHE\_TYPE缓存配置如表9-2所示。

表9-2 CACHE\_TYPE配置

CACHE_TYPE	说 明
NullCache	空的缓存，不能使用缓存
SimpleCache	简单缓存，仅用于开发环境
CACHE_TYPE	说 明
FileSystemCache	文件系统缓存，将数据存储在文件中
RedisCache	Redis 缓存，需要安装 redis 包
MemcachedCache	Memcached 缓存，需要安装 memcached 包

完整的CACHE\_TYPE配置请参考官方文档<https://flask-caching.readthedocs.io/en/latest/index.html#configuring-flask-caching>。

我们的项目选择的是RedisCache，其参数配置如表9-3所示。

表9-3 RedisCache下可配参数

配 置 项	说 明
CACHE_DEFAULT_TIMEOUT	过期时间，默认是 300s
CACHE_KEY_PREFIX	键前缀，默认是 flask_cache_
CACHE_REDIS_HOST	Redis 服务器域名
CACHE_REDIS_PORT	Redis 服务器端口号
CACHE_REDIS_PASSWORD	Redis 服务器密码

回到pythonbbs项目中，打开config.py文件，然后在DevelopmentConfig中添加Flask-Caching的配置信息，代码如下。

```
class DevelopmentConfig(BaseConfig):
    ...
    # 缓存配置
    CACHE_TYPE = "RedisCache"
    CACHE_REDIS_HOST = "127.0.0.1"
    CACHE_REDIS_PORT = 6379
```

上述代码中，配置了缓存类型为Redis，以及Redis服务器的域名和端口号。

接下来创建一个Flask-Caching对象，打开exts.py文件，然后输入以下代码。

```
...
from flask_caching import Cache
...
cache = Cache()
```

再回到app.py文件中，从exts.py文件中导入cache变量，并且进行初始化，代码如下。

```
from exts import db,mail,cache  
...  
  
app = Flask(__name__)  
...  
cache.init_app(app)
```

Flask-Caching初始化完成后，就可以用它来缓存数据了。我们回到blueprints/user.py文件中，先从exts.py文件中导入cache对象，然后将email\_captcha代码修改如下。

```
...  
from exts import mail,cache  
...  
  
@bp.route("/mail/captcha")  
def mail_captcha():  
    email = request.args.get("mail")  
    digits = ["0","1","2","3","4","5","6","7","8","9"]  
    captcha = "".join(random.sample(digits,4))  
    body = f"【知了Python论坛】您的注册验证码是: {captcha}，请勿告诉别人！"  
    message = Message(subject="我是邮件主题",recipients=[email],body=body)  
    mail.send(message)  
    cache.set(email,captcha,timeout=100)  
    return "success"
```

这样就完成了邮箱和验证码的缓存，以后要验证的时候，通过cache.get(mail)方法从缓存中获取即可。

### 9.3.4 使用Celery发送邮件

现在的mail\_captcha视图函数虽然可以正常发送邮件，但是用户体验不好，一方面，发送邮件需要发起网络请求，必须要等邮件发送成功后浏览器才能收到响应，用户等待时间过长。另一方面，发送邮件这种耗时操作会导致线程被长时间占用，从而导致无法服务其他请求，造成服务器资源紧张。对于这些耗时的操作，我们一般通过异步的方式来实现，其中最常用、最稳定的方式就是通过Celery来实现。

Celery是一个任务调度框架，是用纯Python实现的，可以轻松地集成到Flask项目中。

Celery由五大模块组成，分别是Task（任务）、Broker（中间人）、Celery Beat（调度器）、Worker（消费者）、Backend（存储）。

步骤如下：首先在程序中定义好Task，然后启动Celery，Celery读取Task并存放到Broker中，Broker是具有存储能力的服务，如Redis、RabbitMQ、数据库等。Celery Beat会根据配置，或者由开发者手动控制，分配任务给Worker，Worker在完成任务后把结果存储到Backend中，Backend也是具有存储能力的服务，一般为了方便，会和Broker使用同一个服务，Backend不是必需的，如果没有设置Backend，将不会存储结果。Celery的工作原理如图9-26所示。

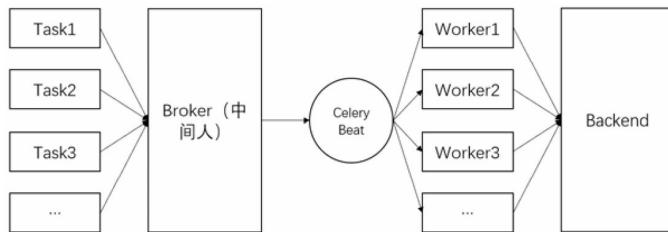


图9-26 Celery工作原理

Celery非常适用以下两种场景。

(1) 异步任务：对于一些会阻塞程序的耗时操作，如发送邮件、发送短信、视频转码等，非常适合放到Celery中执行。

(2) 定时任务：可以通过配置实现定时执行任务，如定时抓取数据、定时发送报告等。

Celery是一个第三方Python框架，需要单独安装。打开PyCharm的Terminal，输入以下命令即可完成安装。

```
$ pip install celery
```

Celery框架安装完成后，首先在config.py下的DevelopmentConfig中添加Broker和Backend配置信息，代码如下。

```
class DevelopmentConfig(BaseConfig):
    ...
    # Celery配置
```

```
# 格式: redis://:password@hostname:port/db_number
CELERY_BROKER_URL = "redis://127.0.0.1:6379/0"
CELERY_RESULT_BACKEND = "redis://127.0.0.1:6379/0"
```

再回到pythonbbs项目，在项目根路径下创建一个bbs\_celery.py文件，然后输入以下代码。

```
from flask_mail import Message
from exts import mail
from celery import Celery

# 定义任务函数
def send_mail(recipient, subject, body):
    message = Message(subject=subject, recipients=[recipient], body=body)
    mail.send(message)
    print("发送成功! ")

# 创建 Celery 对象
def make_celery(app):
    celery = Celery(app.import_name, backend=app.config['CELERY_RESULT_BACKEND'],
                   broker=app.config['CELERY_BROKER_URL'])
    TaskBase = celery.Task

    class ContextTask(TaskBase):
        abstract = True

        def __call__(self, *args, **kwargs):
            with app.app_context():
                return TaskBase.__call__(self, *args, **kwargs)

    celery.Task = ContextTask
    app.celery = celery

    # 添加任务
    celery.task(name="send_mail")(send_mail)

    return celery
```

上述代码中，定义了发送邮件的函数send\_mail，还定义了函数make\_celery。在make\_celery函数中，将send\_mail添加到celery任务中。

下面再回到app.py中，导入make\_celery函数，并创建一个Celery对象，代码如下。

```
...
```

```
from bbs_celery import make_celery  
...  
# 构建celery  
celery = make_celery(app)
```

下面回到blueprints/user.py的email\_captcha视图函数中，把之前发送邮件的代码删除，改成用celery任务的方式发送，代码如下。

```
...  
from flask import current_app  
...  
@bp.route("/mail/captcha")  
def mail_captcha():  
    email = request.args.get("mail")  
    digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]  
    captcha = "".join(random.sample(digits, 4))  
    subject="【知了Python论坛】注册验证码"  
    body = f"【知了Python论坛】您的注册验证码是: {captcha}，请勿告诉别人！"  
    current_app.celery.send_task("send_mail", (email,subject,body))  
    cache.set(email, captcha, timeout=100)  
    return "success"
```

接下来运行Celery，如果读者是在Windows系统下运行Celery，那么还需要安装一个第三方库：gevent，安装命令如下。

```
$ pip install gevent
```

在PyCharm的Terminal下，输入以下命令，启动Celery程序。

```
$ celery -A app.celery worker -P gevent -l info
```

如果是在Linux系统下运行，则无须“-P gevent”参数。更多关于Celery的使用请参阅官方文档[https://docs.celeryproject.org/en/stable/。](https://docs.celeryproject.org/en/stable/)

打开浏览器，输入http://127.0.0.1:5000/user/mail/captcha?mail=邮箱账号，浏览器可以立即收到响应，并且也能收到邮件。

### 9.3.5 RESTful API

RESTful也叫作REST（representational state transfer，表现层状态转换），是Roy Fielding博士在2000年提出的一种万维网架构风格，其主要作用是提供一种软件之间通过HTTP/HTTPS协议交互数据的风格。RESTful API有以下特点。

（1）直观简短的URL地址，如以下URL。

- /post/list：帖子列表。
- /posts/111：主键为111的帖子详情。
- /post/111/comments：主键为111的帖子下所有的评论。

（2）数据传输格式，如JSON、XML等。JSON因具有格式清晰、体积小等优点，已经取代XML，成为大多数互联网产品的首选格式了。

（3）操作资源的方法，根据操作资源的目的，选择不同的method。如获取资源用GET、创建资源用POST、替换资源用PUT、删除资源用DELETE。

（4）响应的状态码。服务器根据情况返回对应的状态码。2xx代表操作成功、3xx代表重定向、4xx代表客户端有错误、5xx代表服务器出现错误。

RESTful API是一种风格，不是规范，读者在产品开发中，不应拘泥于严格的RESTful API，而应该根据自身需求灵活调整。

在Flask中提供了jsonify方法，用于返回JSON格式的数据。下面使用jsonify方法封装一个用于返回RESTful API风格的模块。在pythonbbs根路径下，创建一个名叫utils的包，这个包主要用来存放一些工具类模块，接着在utils包下创建一个restful.py文件，然后输入以下代码。

```
from flask import jsonify

class HttpStatusCode(object):
    # 响应正常
    ok = 200
    # 没有登录错误
    unloginerror = 401
    # 没有权限错误
    permissionerror = 403
```

```

# 客户端参数错误
paramserror = 400
# 服务器错误
servererror = 500

def _restful_result(code, message, data):
    return jsonify({"message": message or "", "data": data or {}}), code

def ok(message=None, data=None):
    return _restful_result(code=HttpCode.ok, message=message, data=data)

def unlogin_error(message="没有登录！"):
    return _restful_result(code=HttpCode.unloginerror, message=message,
data=None)

def permission_error(message="没有权限访问！"):
    return _restful_result(code=HttpCode.paramserror, message=message,
data=None)

def params_error(message="参数错误！"):
    return _restful_result(code=HttpCode.paramserror, message=message,
data=None)

def server_error(message="服务器开小差啦！"):
    return _restful_result(code=HttpCode.servererror, message=message or
'服务器内部错误', data=None)

```

上述代码中，我们针对不同的状态码添加了对应的函数，状态码和对应函数说明如表9-4所示。

表9-4 状态码和对应函数说明

状态码	函数名	说 明
200	success	请求成功
401	unloginerror	请求需要登录的 API，但是用户并没有登录
403	permissionerror	请求需要指定权限的 API，但是用户并没有这项权限
400	params_error	请求 API 所提交的参数错误
500	server_error	请求 API 时服务器出现错误

表9-4中的函数，不管什么状态码，不管是否需要给客户端返回数据，最终调用的都是`_restful_result`函数，这个函数中封装了返回数据的格式，返回数据的格式类似如下。

```
{  
    "message": "...",  
    "data": ...  
}
```

保持格式一致有个好处，客户端在任何情况下都可以获取到这两个参数，不会因为某些API下没有某个参数而导致代码抛出异常。

在网站开发中，RESTful API一般用于AJAX（asynchronous JavaScript and XML）请求。AJAX是使用JavaScript语言异步发送请求的，根据响应对页面进行局部更新。发送邮箱验证码的请求非常适合使用AJAX方式，因为用传统表单的形式，会导致注册页面发生跳转，所以将blueprints/user.py中的email\_captcha的代码修改为如下形式。

```
...  
from utils import restful  
...  
  
@bp.route("/mail/captcha")  
def mail_captcha():  
    try:  
        email = request.args.get("mail")  
        digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]  
        captcha = "".join(random.sample(digits, 4))  
        subject="【知了Python论坛】注册验证码"  
        body = f"【知了Python论坛】您的注册验证码是: {captcha}，请勿告诉别人！"  
        current_app.celery.send_task("send_mail", (email,subject,body))  
        cache.set(email, captcha, timeout=100)  
        return restful.ok()  
    except Exception as e:  
        print(e)  
        return restful.server_error()
```

如果视图函数正常执行，并且邮件也能发送成功，那么返回状态码200，否则返回服务器错误状态码500。

### 9.3.6 CSRF保护

只要网页是通过模板渲染的，并且交互过程中存在非GET请求，那么就必须要开启CSRF保护。关于CSRF的详细介绍读者可以参考第6章。开启CSRF保护需要使用flask-wtf中的CSRFProtect，首先在PyCharm的Terminal下输入以下命令安装flask-wtf。

```
$ pip install flask-wtf
```

回到app.py中，添加以下代码。

```
from flask_wtf import CSRFProtect  
...  
# CSRF保护  
CSRFProtect(app)
```

这样以后所有的非GET请求都必须在请求头或者请求体中加上`csrf_token`，否则会出现状态码为400、提示信息为Bad Request的错误。注册功能用表单即可实现，因此需要在`templates/front/register.html`的form标签下添加以下代码。

```
...  
<input type="hidden" name="csrf_token" value="{{ csrf_token() }}>  
...
```

以后单击“立即注册”按钮，会自动把`csrf_token`所对应的值提交给视图函数。

### 9.3.7 使用AJAX获取邮箱验证码

在注册页面中，当单击“发送验证码”按钮后，这时用户一般不希望看到页面的刷新，所以要使用异步发送获取验证码的请求，也就是AJAX技术。我们使用jQuery提供的`$.ajax`方法来实现，考虑到项目已经启动了CSRF保护，每次发送请求都要在请求体或者请求头中添加`csrf_token`。为了方便，先在`templates/front/base.html`的head标签中添加以下代码。

```
<meta name="csrf-token" content="{{ csrf_token() }}>
```

读者按照9.3.1节的复制操作后，在`static/common`下有一个`zlajax.js`文件，这个文件是对`$.ajax`做了一层封装，在非GET请求之前，先从模板的`meta`标签中读取`csrf-token`的值，然后将读取到的值设置到请求头中，这样就不需要每次发送非GET请求前都手动设置`csrf_token`了。`zlajax.js`代码如下。

```
var zlajax = {
    'get': function (args) {
        args['method'] = "get"
        return this.ajax(args);
    },
    'post': function (args) {
        args['method'] = "post"
        return this.ajax(args);
    },
    'put': function(args){
        args['method'] = "put"
        return this.ajax(args)
    },
    'delete': function(args){
        args['method'] = 'delete'
        return this.ajax(args)
    },
    'ajax': function (args) {
        // 设置csrfToken
        this._ajaxSetup();
        return $.ajax(args);
    },
    '_ajaxSetup': function () {
        $.ajaxSetup({
            beforeSend: function (xhr, settings) {
                if (!/^((GET|HEAD|OPTIONS|TRACE))/i.test(settings.type) && !this.crossDomain) {
                    var csrfToken = $('meta[name=csrf-token]').attr('content');
                    xhr.setRequestHeader("X-CSRFToken", csrfToken)
                }
            }
        });
    }
};
```

考虑到很多页面都需要使用AJAX，所以把zlajax.js文件放到templates/front/base.html的head标签中，这样以后所有页面就都能使用这个文件了，代码如下。

```
...
<script
src="{{ url_for('static',filename='common/zlajax.js') }}"></script>
...
```

因为zlajax.js依赖jQuery，所以必须要把zlajax.js放到jQuery文件后面。

接下来在static/front/js下创建一个register.js文件，这个文件用于绑定“发送验证码”按钮的单击事件，并且执行AJAX请求，代码如下。

```
$(function () {
    $('#captcha-btn').on("click", function(event) {
        event.preventDefault();
        // 获取邮箱
        var email = $("input[name='email']").val();

        zlajax.get({
            url: "/user/mail/captcha?mail=" + email
        }).done(function (result) {
            alert("验证码发送成功! ");
        }).fail(function (error) {
            alert(error.message);
        })
    });
});
```

上述代码中，为id为captcha-btn的按钮绑定了单击事件。单击按钮后，先是获取用户输入的邮箱，然后通过zlajax.get方法发送请求，URL不需要带域名，向以 / 开头的URL发送请求，浏览器会自动使用当前域名。如果请求成功，会执行done中的函数，如果请求失败，则会执行fail中的函数。这里不管成功还是失败，我们都使用alert函数反馈结果。由于js文件必须要加载到模板中才能生效，所以打开/templates/front/register.html文件，然后将head这个block中的代码修改如下。

```
...
{% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='front/css/
sign.css') }}">
    <script src="{{ url_for('static', filename='front/js/register.js') }}">
</script>
{% endblock %}
...
```

### 9.3.8 实现注册功能

邮箱验证码可以正常获取后，我们再来完善注册功能。首先在templates/front/register.html中的form标签上添加action和method属性，代码如下。

```
...
<form action="{{ url_for('user.register') }}" method="post" id="register-form">
...

```

action属性的作用是，在单击“提交”按钮时，将数据发送到某个URL，这里是提交到user.register视图函数中。method属性用来表示用什么方法发送数据，表单数据通常采用POST方法。把表单数据提交到视图函数后，需要先对表单数据做验证，在pythonbbs项目根路径下创建一个名叫forms的Python Package，然后创建user.py文件，代码如下。

```
from wtforms import Form, StringField, ValidationError
from wtforms.validators import Email, EqualTo, Length
from exts import cache
from models.user import UserModel

class RegisterForm(Form):
    email = StringField(validators=[Email(message="请输入正确格式的邮箱! ")])
    captcha = StringField(validators=[Length(min=4, max=4, message="请输入正确格式的验证码! ")])
    username = StringField(validators=[Length(min=2, max=20, message="请输入正确长度的用户名! ")])
    password = StringField(validators=[Length(min=6, max=20, message="请输入正确长度的密码! ")])
    confirm_password = StringField(validators=[EqualTo("password", message="两次密码不一致! ")])

    def validate_email(self, field):
        email = field.data
        user = UserModel.query.filter_by(email=email).first()
        if user:
            raise ValidationError(message="邮箱已经存在")

    def validate_captcha(self, field):
        captcha = field.data
        email = self.email.data
        cache_captcha = cache.get(email)
        if not cache_captcha or captcha != cache_captcha:
            raise ValidationError(message="验证码错误!" )
```

上述代码中，创建了一个RegisterForm表单类，然后定义了email、captcha、username、password和confirm\_password这5个字段，并且分别指定了验证器，其中Email邮箱验证器必须要安装第三方库email\_validator，打开PyCharm的Terminal，输入以下命令完成安装。

```
$ pip install email_validator
```

除此之外，还对email和captcha单独做了验证。验证email的目的是判断邮箱是否已经被注册过，验证captcha的目的是判断验证码是否正确。

考虑到以后在视图函数中的表单验证失败后，需要把错误信息传到模板中。我们定义一个父类，用于从form.errors中提取所有字符串类型的错误信息。在pythonbbs/forms下新建一个baseform.py文件，然后输入以下代码。

```
from wtforms import Form
class BaseForm(Form):
    @property
    def messages(self):
        message_list = []
        if self.errors:
            for errors in self.errors.values():
                message_list.extend(errors)
        return message_list
```

将RegisterForm的继承关系修改为如下。

```
...
from .baseform import BaseForm

class RegisterForm(BaseForm):
    ...
```

以后可以通过form.messages获取所有字符串类型的错误信息，用以传给模板进行渲染，这样用户就能知道是哪里输入错了。

下面再把RegisterForm导入blueprints/user.py，完善register视图函数，代码如下。

```

from flask import Blueprint, render_template, request, current_app,
redirect, url_for
from exts import cache, db
import random
from utils import restful
from forms.user import RegisterForm
from models.user import UserModel

...
@register_bp.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'GET':
        return render_template("front/register.html")
    else:
        form = RegisterForm(request.form)
        if form.validate():
            email = form.email.data
            username = form.username.data
            password = form.password.data
            user = UserModel(email=email, username=username, password=password)
            db.session.add(user)
            db.session.commit()
            return redirect(url_for("user.login"))
        else:
            for message in form.messages:
                flash(message)
            return redirect(url_for("user.register"))

@register_bp.route('/login')
def login():
    return "login"

```

上述代码中，register视图函数同时支持GET和POST两种method的请求。因此需要设置@bp.route中的methods参数为['GET', 'POST']。在register视图函数中，通过判断request.method来执行对应操作。如果是GET请求，那么就返回模板；如果是POST请求，则构建RegisterForm对象，在表单验证通过的情况下，创建UserModel对象并保存到数据库中。注册完成后，需要跳转到登录页面，让用户自行登录，这里为了跳转登录页面时不报错，创建了一个非常简单的login视图函数。如果验证失败，则把表单的错误信息添加到flash中，重新加载注册页面。

在表单验证失败的情况下，由于视图函数已经把错误消息添加到flash中了，所以模板中可以通过get\_flashed\_messages获取所有的错误消息。在templates/front/register.html中的“立即

注册”按钮上添加以下代码。

```
{% with messages = get_flashed_messages() %}  
  {% if messages %}  
    <div class="form-group">  
      <ul>  
        {% for message in messages %}  
          <li class="text-danger">{{ message }}</li>  
        {% endfor %}  
      </ul>  
    </div>  
  {% endif %}  
{% endwith %}  
<div class="form-group">  
  <button class="btn btn-warning btn-block" id="submit-btn">立即注册  
</button>  
</div>
```

在浏览器中访问`http://127.0.0.1:5000/user/register`，然后按照规则输入数据，即可完成注册。如果数据格式输入错误，邮箱已存在，或者验证码错误等，都会在注册页面中显示错误信息，如图9-27所示。

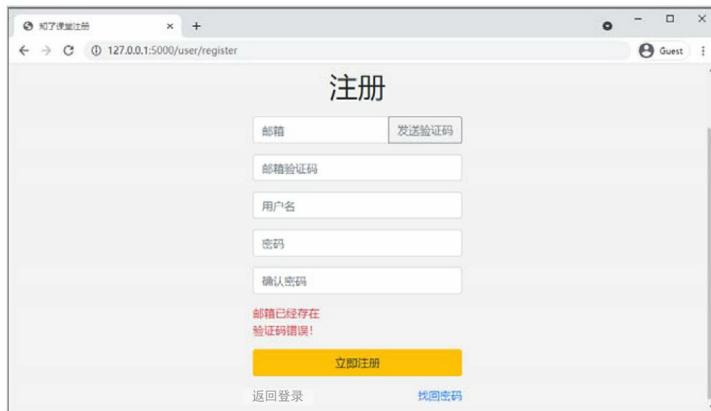


图9-27 注册页面显示错误信息

## 9.4 登录

登录的流程是这样的，用户在登录界面首先输入邮箱和密码，然后提交到视图函数中，视图函数验证邮箱和密码是否正确，如果正确，则把能识别用户的数据添加到cookie中，再

返回给浏览器。当浏览器下次再访问本网站下的其他页面时，会自动携带cookie，我们就能知道这个请求是哪个用户发出的。在blueprints/user.py中已经定义了一个非常简单的login视图函数，login视图函数与register一样，也是同时支持GET和POST方法，在选用GET时返回登录模板，我们将login视图函数代码修改如下。

```
@bp.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template("front/login.html")
    else:
        pass
```

下面在浏览器中访问`http://127.0.0.1:5000/user/login`，即可看到如图9-28所示的界面。

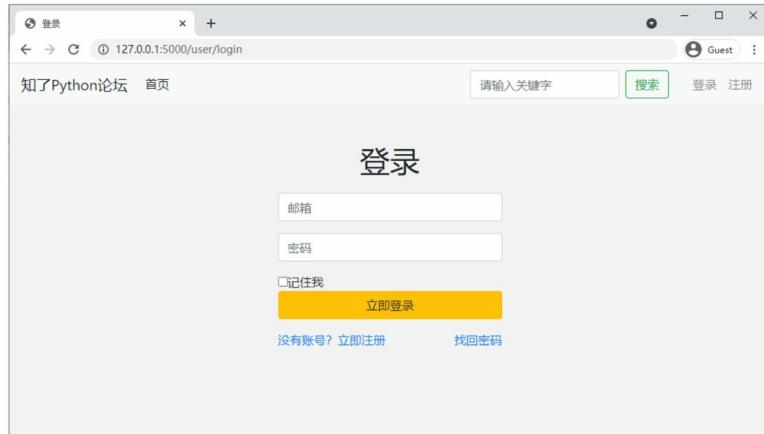


图9-28 登录界面

同样，在`templates/front/login.html`中的form标签下，添加`csrf_token`的`input`标签，以及渲染表单验证错误信息，代码如下。

```
...
<form action="{{ url_for('user.login') }}" method="post">
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
...
{% with messages = get_flashed_messages() %}
{%- if messages %}
    <div class="form-group">
        <ul>
            {% for message in messages %}
                <li class="text-danger">{{ message }}</li>
            {% endfor %}
        </ul>
    </div>
{%- endif %}
{%- endwith %}
<div class="form-group">
    <button class="btn btn-warning btn-block" id="submit-btn">立即登录
</button>
</div>
...

```

接着在forms/user.py中，添加一个登录表单LoginForm，代码如下。

```
class LoginForm(BaseForm):
    email = StringField(validators=[Email(message="请输入正确格式的邮箱! ")])
    password = StringField(validators=[Length(min=6, max=20, message="请输入正确长度的密码! ")])
    remember = BooleanField()
```

回到blueprint/user.py的login视图函数中，针对POST请求，添加以下代码。

```
from flask import session
from forms.user import LoginForm

@bp.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template("front/login.html")
    else:
        form = LoginForm(request.form)
        if form.validate():
            email = form.email.data
            password = form.password.data
            remember = form.remember.data
            user = UserModel.query.filter_by(email=email).first()
            if user and user.check_password(password):
                session['user_id'] = user.id
                if remember:
                    session.permanent = True
                return redirect("/")
            else:
                flash("邮箱或者密码错误！")
                return redirect(url_for("user.login"))
        else:
            for message in form.messages:
                flash(message)
    return render_template("front/login.html")
```

上述代码中，我们首先判断邮箱是否存在，然后判断密码是否正确。如果两个条件都满足，就认为是登录成功，然后把user.id存储在session中，因为Flask中的session默认是加密后存储在cookie中的，这也就意味着user.id会被加密存放到cookie中并返回给浏览器。

在登录页面中，我们添加了一个“记住我”复选框，如果用户选中“记住我”复选框，则应该让cookie过期时间长一点（默认情况是浏览器关闭时，cookie就会自动过期），通过设置session.permanent=True来实现，默认过期时间是31天，如果想自定义过期时间，可以在配置中添加参数PERMANENT\_SESSION\_LIFETIME，这个参数的类型为timedelta。如设置7天过期，那么在config.py的BaseConfig中添加如下代码。

```
from datetime import timedelta

class BaseConfig:
    ...

    PERMANENT_SESSION_LIFETIME = timedelta(days=7)
```

如果登录成功则让页面跳转到首页，为了确保登录成功后代码不报错，我们在blueprints/front.py中添加一个index视图函数，代码如下。

```
...
@bp.route("/")
def index():
    return "index"
```

在浏览器中访问http://127.0.0.1:5000/user/login，然后输入邮箱和密码，即可登录成功。

## 9.5 发布帖子

发布帖子模块与注册模块有些类似，都是用来收集用户输入数据并发送到服务器。与注册模块不同的是，发布帖子需要选择帖子所属板块，帖子内容是富文本内容，下面分别进行讲解。

### 9.5.1 添加帖子相关模型

在实现发布帖子功能之前，先创建与帖子相关的模型，包括板块模型、帖子模型、评论模型。打开models/post.py文件，添加以下代码。

```
from exts import db
from datetime import datetime

# 板块模型
class BoardModel(db.Model):
    __tablename__ = 'board'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(20), nullable=False)
    create_time = db.Column(db.DateTime, default=datetime.now)
    is_active = db.Column(db.Boolean, default=True)

# 帖子模型
class PostModel(db.Model):
    __tablename__ = 'post'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String(200), nullable=False)
    content = db.Column(db.Text, nullable=False)
    create_time = db.Column(db.DateTime, default=datetime.now)
    read_count = db.Column(db.Integer, default=0)
```

```
is_active = db.Column(db.Boolean, default=True)
board_id = db.Column(db.Integer, db.ForeignKey("board.id"))
author_id = db.Column(db.String(100), db.ForeignKey("user.id"),
nullable=False)

board = db.relationship("BoardModel", backref="posts")
author = db.relationship("UserModel", backref='posts')

# 评论模型
class CommentModel(db.Model):
    __tablename__ = 'comment'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    content = db.Column(db.Text, nullable=False)
    create_time = db.Column(db.DateTime, default=datetime.now)
    is_active = db.Column(db.Boolean, default=True)
    post_id = db.Column(db.Integer, db.ForeignKey("post.id"))
    author_id = db.Column(db.String(100), db.ForeignKey("user.id"),
nullable=False)

    post = db.relationship("PostModel", backref='comments')
    author = db.relationship("UserModel", backref='comments')
```

模型需要被直接或者间接地导入app.py文件才能被检索到，在blueprints/front.py文件中导入以上模型，然后在PyCharm的Terminal中执行以下两条命令，即可将模型同步到数据库中。

```
$ flask db migrate -m "add post comment board model"
$ flask db upgrade
```

## 9.5.2 初始化板块数据

在网站上线前，应该先在数据库中添加一些板块数据，而且板块的数据一旦确定了也不会轻易变化。我们通过命令初始化板块数据，在pythonbbs/comments.py文件中添加以下代码。

```
def create_board():
    board_names = ['Python语法', 'web开发', '数据分析', '测试开发', '运维开发']
    for board_name in board_names:
        board = BoardModel(name=board_name)
        db.session.add(board)
    db.session.commit()
    click.echo("板块添加成功! ")
```

下面再回到app.py文件中将create\_board添加到命令中，代码如下。

```
...  
app.cli.command("create-board") (commands.create_board)
```

打开PyCharm的Terminal，执行flask create-board命令，即可完成板块数据的初始化。

### 9.5.3 渲染发布帖子模板

在渲染模板之前，先来看下发布帖子页面的效果，如图9-29所示。

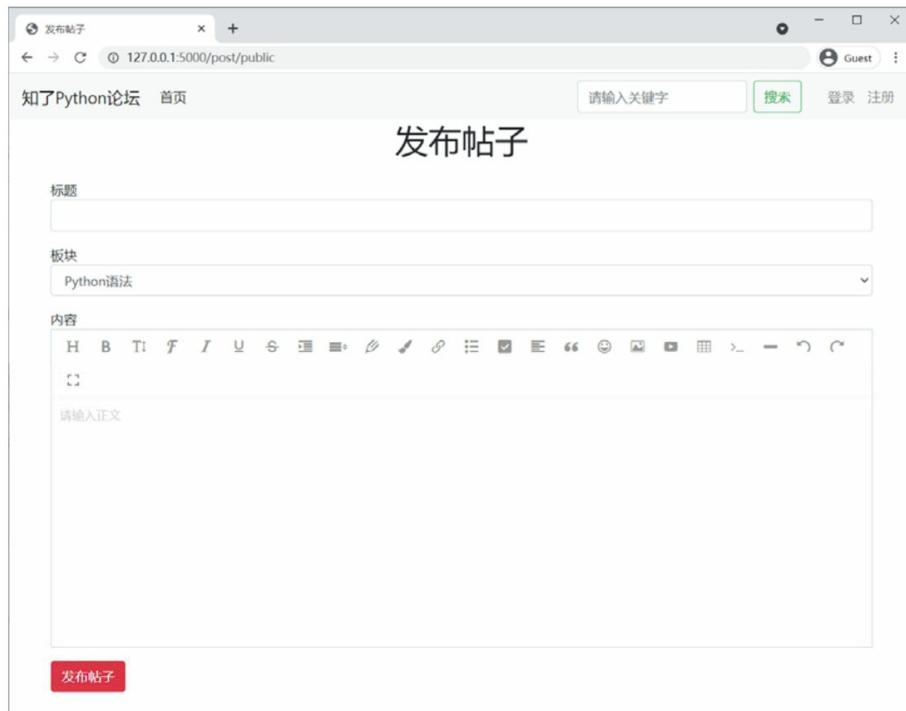


图9-29 发布帖子页面效果

用户需要输入的数据包括标题、板块、内容，我们先来完成模板的渲染。在 blueprints/post.py 文件中，添加 public\_post 视图函数，并且需要在模板中显示板块数据，所以在渲染模板时要把所有板块数据传给模板，代码如下。

```
@bp.route("/post/public", methods=['GET', 'POST'])
def public_post():
    if request.method == 'GET':
        boards = BoardModel.query.all()
        return render_template("front/public_post.html", boards=boards)
    else:
        pass
```

然后在 front/public\_post.html 文件中，将板块数据循环渲染到板块的 select 标签下，代码如下。

```
...
<div class="form-group">
    <label>板块</label>
    <select name="board_id" class="form-control">
        {% for board in boards %}
            <option value="{{ board.id }}>{{ board.name }}</option>
        {% endfor %}
    </select>
</div>
...
```

在浏览器中访问 <http://127.0.0.1:5000/post/public>，可以看到如图 9-30 所示的效果图。



图9-30 未加入富文本编辑器的发布帖子页面

现在模板虽然渲染了，但是还没有加入内容输入框，内容输入框需要加入富文本编辑器。下面讲解如何加入富文本编辑器。

#### 9.5.4 使用wangEditor富文本编辑器

内容部分的输入框，应该具有类似Word软件的功能，如可以设置字体大小、颜色等，还可以插入图片，并且所见即所得，也就是编辑的时候是什么效果，以后在网页中展示的也是什么效果，这类编辑器叫作富文本编辑器。目前互联网上有许多免费开源的富文本编辑器，如百度官方出品的UEditor、国外的CKEditor，以及国内以王福朋为首的前端团队开发的wangEditor。笔者尝试使用过多款富文本编辑器发现，wangEditor简单易用、功能稳定，且有详细的中文文档，本书使用wangEditor作为富文本编辑器。

##### 注意

wangEditor富文本编辑器的官方文档地址为<https://www.wangeditor.com/doc/>。

使用wangEditor分为4步，第1步在模板中引入wangEditor.js文件，第2步添加生成编辑器的占位标签，第3步初始化编辑器，第4步设置图片上传URL，下面分别来实现。

#### 1. 引入wangEditor.js文件

打开templates/front/public\_post.js文件，然后实现head这个block，并添加以下代码。

```
...
{%
    block head %}
    <script src="https://cdn.jsdelivr.net/npm/wangeditor@latest/dist/
wangEditor.min.js"></script>
{%
    endblock %}
...
```

上述代码中，我们使用jsdelivr服务器提供的cdn服务加载wangEditor.min.js脚本，读者如果要通过自己的服务器加载，可以把wangEditor.min.js保存下来，存放到自己的服务器加载即可。

## 2. 添加生成编辑器的占位标签

在需要生成wangEditor富文本编辑器的地方，添加一个占位标签，一般用div即可，为了后期方便寻找，在div标签上添加一个id属性，代码如下。

```
...
<div class="form-group">
    <label>内容</label>
    <div id="editor"></div>
</div>
...
```

## 3. 初始化编辑器

接下来将编辑器占位标签初始化成真正的编辑器，这里需要用到JavaScript代码，我们在static/front/js下创建一个public\_post.js文件，然后添加以下代码。

```
$(function () {
    var editor = new window.wangEditor("#editor");
    editor.create();
});
```

下面在templates/front/public\_post.html的head block中加载此js文件，代码如下。

```
...
{%
    block head %
}
<script src="https://cdn.jsdelivr.net/npm/wangeditor@latest/dist/
wangEditor.min.js"></script>
<script src="{{ url_for('static',filename='front/js/public_post.js') }}">
</script>
{%
    endblock %
}
...
```

执行以上代码后，访问`http://127.0.0.1:5000/post/public`，就可以看到wangEditor编辑器成功被渲染，效果如图9-29所示。

#### 4. 设置图片上传URL

如果不想在wangEditor中上传图片，这一步可以忽略，也不影响使用wangEditor。这里添加图片上传的功能。首先在blueprints/front.py文件中添加上传图片的视图函数，代码如下。

```
from flask import Blueprint, request, render_template, jsonify,
current_app, url_for
from flask import send_from_directory
from werkzeug.utils import secure_filename
import os

...
@bp.post("/upload/image")
def upload_image():
    f = request.files.get('image')
    extension = f.filename.split('.')[ -1].lower()
    if extension not in ['jpg', 'gif', 'png', 'jpeg']:
        return jsonify({
            "errno": 400,
            "data": []
        })
    filename = secure_filename(f.filename)
    f.save(os.path.join(current_app.config.get("UPLOAD_IMAGE_PATH"),
filename))
    url = url_for('media.media_file', filename=filename)
    return jsonify({
        "errno": 0,
        "data": [
            {
                "url": url,
                "alt": "",
                "href": ""
            }
        ]
    })
}
```

上述代码中，通过在视图函数upload\_image的request.files中获取image参数来获取图片，这也意味着前端在上传图片时需要用image作为参数名。通过判断文件的后缀名来判断文件是否是图片，如果不是就返回400错误。这里没有用restful模块中的函数返回JSON数据，而是用jsonify，这是因为wangEditor期望返回的结果为如下格式。

```
{
    "errno": 0,
    "data": [
        {
            "url": url,
            "alt": "",
            "href": ""
        }
    ]
}
```

返回结果中的参数说明如下。

- errno:** 为0代表正常，非0代表异常。
- data:** 图片上传后的信息数组，图片信息包括URL、提示信息alt以及跳转链接href。

为了防止黑客利用图片文件名攻击服务器，使用werkzeug.utils.secure\_filename产生一个安全的文件名来保存图片，并且把文件存储在配置项UPLOAD\_IMAGE\_PATH指定的路径下，这个配置项可以自行设置。

在图片保存完成后，再使用url\_for对media.media\_file进行反转，用来获取图片的URL。media.media\_file是新增的用于返回上传的文件的蓝图，为了后期方便，将上传的文件切换到Nginx服务器上部署。下面创建一个URL（以/media开头的蓝图），在开发阶段都会使用这个蓝图返回图片文件。首先在blueprints下创建media.py文件，然后输入以下代码。

```
from flask import Blueprint, current_app
import os

bp = Blueprint("media", __name__, url_prefix="/media")

@bp.get("/<path:filename>")
def media_file(filename):
    return
    os.path.join(current_app.config.get("UPLOAD_IMAGE_PATH"), filename)
```

接下来在app.py文件中对media蓝图进行注册，代码如下。

```
from blueprints.media import bp as media_bp
...
app.register_blueprint(media_bp)
...
```

图片上传的视图函数写完后，再来到底static/front/js/public\_post.js脚本中，添加设置上传图片URL的代码，代码如下。

```
var editor = new window.wangEditor("#editor");
editor.config.uploadImgServer = "/upload/image";
editor.config.uploadFileName = "image";
```

```
editor.create();
```

在浏览器中重新加载发布帖子的页面，然后在wangEditor富文本编辑器中单击“图片”按钮，可以看到上传图片的功能已经添加，如图9-31所示。

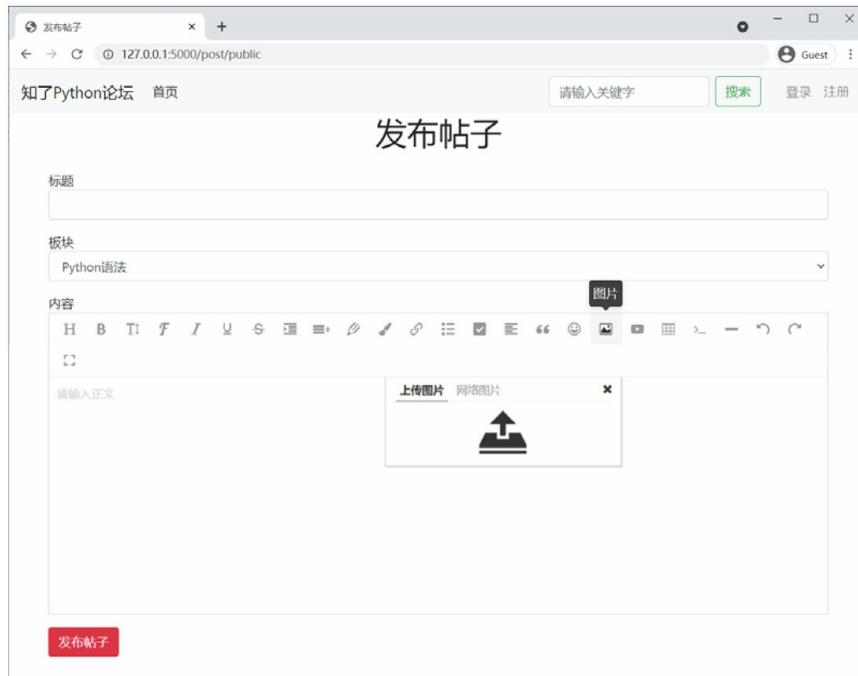


图9-31 wangEditor添加上传图片功能

但是现在还不能使用上传图片功能，因为wangEditor使用POST方法上传图片，而我们的项目对非GET请求做了CSRF防御，必须要提交`csrf_token`，这样上传图片才能保证安全。对此可以对`upload_image`视图函数关闭CSRF防御，在exts.py文件中添加以下代码。

```
...
from flask_wtf import CSRFProtect

...
csrf = CSRFProtect()
```

下面再回到app.py文件中，将原来CSRFProtect(app)的代码修改为以下代码。

```
...
from exts import csrf
...
csrf.init_app(app)
```

这样做是因为在blueprints/post.py文件中也需要使用csrf对象，如此可避免循环导入。然后在upload\_image函数定义中加上@csrf.exempt装饰器，代码如下。

```
@bp.post("/upload/image")
@csrf.exempt
def upload_image():
    ...
```

这样就针对upload\_image视图函数关闭了CSRF防御，我们在wangEditor中重新上传图片，可以看到上传图片功能已经可以正常使用了。

### 9.5.5 未登录限制

现在发布帖子的页面在用户未登录的情况下可以直接访问，这样是不严谨的。如果用户没有登录发布帖子页面，应该重定向到登录页面。在用户访问网站后，为了方便获取当前用户的信息，需要添加before\_request钩子函数。在pythonbbs项目的根路径下创建一个hooks.py文件，添加以下代码。

```
from flask import session, g
from models.user import UserModel

def bbs_before_request():
    if "user_id" in session:
        user_id = session.get("user_id")
        try:
            user = UserModel.query.get(user_id)
            setattr(g, "user", user)
        except Exception:
            pass
```

在pythonbbs/app.py文件中，将bbs\_before\_request钩子函数添加进去，代码如下。

```
import hooks

...
# 添加钩子函数
app.before_request(hooks.bbs_before_request)
```

这样用户在登录的情况下访问本网站，会在全局对象g上面添加一个user属性，以后通过g.user即可获取到当前登录用户的信息，如果g没有user属性，说明此用户没有登录。

我们把未登录限制做成装饰器，添加到需要登录才能访问的视图函数中。在pythonbbs根目录下创建一个decorators.py文件，然后添加以下代码。

```
from functools import wraps
from flask import redirect, url_for, g

def login_required(func):
    @wraps(func)
    def inner(*args, **kwargs):
        if not hasattr(g, "user"):
            return redirect(url_for("user.login"))
        else:
            return func(*args, **kwargs)

    return inner
```

上述代码中，添加了一个login\_required装饰器函数，login\_required接收一个函数作为参数，通过全局对象g有无user属性，判断用户是否登录，若未登录就重定向到登录页面，否则就按照正常流程执行被装饰的函数。然后在blueprints/post.py文件中的public\_post和upload\_image上添加login\_required装饰器，修改后代码如下。

```
from decorators import login_required

...
@bp.route("/post/public", methods=['GET', 'POST'])
@login_required
def public_post():
```

```
...
@bp.post("/upload/image")
@csrf.exempt
@login_required
def upload_image():
...
```

上述代码中，视图函数上的装饰器是有顺序的，在有多个装饰器的情况下，执行顺序是从里到外，开发者要充分考虑请求到达服务器后执行的过程，合理分配装饰器的位置。

此后用户在未登录的情况下重新访问发布帖子页面时，即会被重定向到登录页面了。

## 9.5.6 服务端实现发帖功能

在blueprints/post.py文件的public\_post视图函数中，GET请求是返回模板，POST请求则是发布帖子。先添加发布帖子的表单，用来验证客户端上传的数据是否正确。在pythonbbs/forms下创建一个post.py文件，然后添加以下代码。

```
from .baseform import BaseForm
from wtforms import StringField, IntegerField
from wtforms.validators import InputRequired, Length

class PublicPostForm(BaseForm):
    title = StringField(validators=[Length(min=2, max=100, message='请输入正确长度的标题! ')])
    content = StringField(validators=[Length(min=2, message="请输入正确长度的内容! ")])
    board_id = IntegerField(validators=[InputRequired(message='请输入板块id! ')])
```

上述代码中，添加了PublicPostForm类，在其中定义了title、content、board\_id这3个字段。然后把PublicPostForm导入blueprints/front.py文件，并在public\_post视图函数中添加如下代码。

```
@bp.route("/post/public", methods=['GET', 'POST'])
@login_required
def public_post():
    if request.method == 'GET':
        boards = BoardModel.query.all()
        return render_template("front/public_post.html", boards=boards)
```

```
else:
    form = PublicPostForm(request.form)
    if form.validate():
        title = form.title.data
        content = form.content.data
        board_id = form.board_id.data
        post = PostModel(title=title, content=content, board_id=board_id,
author=g.user)
        db.session.add(post)
        db.session.commit()
        return restful.ok()
    else:
        message = form.messages[0]
        return restful.params_error(message=message)
```

上述代码中，因为在before\_request钩子函数中已经把user绑定到g对象上，所以可以直接通过g.user获取用户信息，并在初始化PostModel时赋值给author属性。

### 9.5.7 使用AJAX发布帖子

现在的帖子内容部分是通过wangEditor进行编辑的，只有通过wangEditor提供的JavaScript接口才能获取用户输入的内容，因此发布帖子的请求需要用AJAX来实现。wangEditor是通过editor.txt.html()方法获取内容的，在static/front/js/public\_post.js脚本中添加以下代码。

```
$(function () {
    ...

    // 提交按钮单击事件
    $("#submit-btn").click(function (event) {
        event.preventDefault();

        var title = $("input[name='title']").val();
        var board_id = $("select[name='board_id']").val();
        var content = editor.txt.html();

        zlajax.post({
            url: "/post/public",
            data: {title, board_id, content}
        }).done(function (data) {
            setTimeout(function () {
                window.location = "/";
            }, 2000);
        }).fail(function (error) {
```

```
        alert(error.message);
    });
});
});
```

上述代码中，我们绑定了提交按钮事件，然后分别从HTML标签中获取用户输入的title、board\_id以及content数据，再用zlajax.post方法把数据发送到服务器上。在请求成功后跳转到首页，请求失败则弹出提示对话框。

## 9.6 首页

打开blueprints/front.py文件，然后找到index视图函数，将代码修改为返回index.html模板，代码如下。

```
@bp.route("/")
def index():
    return render_template("front/index.html")
```

在浏览器中访问http://127.0.0.1:5000，可以看到如图9-32所示的页面。

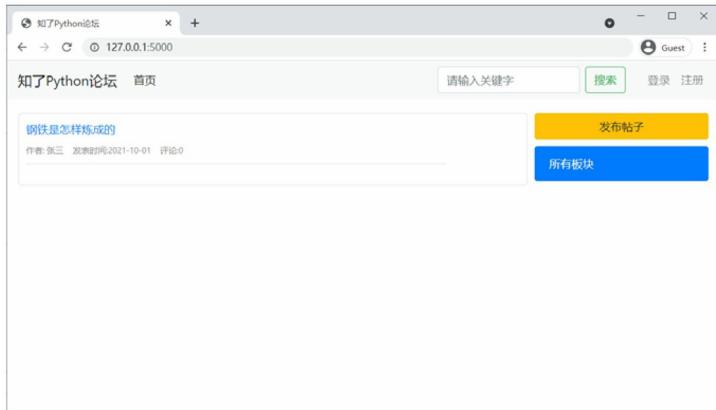


图9-32 渲染index.html模板后的效果

在首页中，左侧帖子列表和右侧板块列表都需要先从视图函数中提取数据，再传给index.html模板文件，我们将index视图函数代码修改为如下形式。

```
@bp.route("/")
def index():
    posts = PostModel.query.all()
    boards = BoardModel.query.all()
    context = {
        "posts": posts,
        "boards": boards
    }
    return render_template("front/index.html", **context)
```

接下来在templates/front/index.html文件中，循环帖子列表和板块列表。修改后的帖子列表和板块列表代码如下。

```

...
<ul class="post-list-group">
  {%
    for post in posts %
  <li>
    <div class="post-info-group">
      <p class="post-title">
        <a href="#">{{ post.title }}</a>
      </p>
      <p class="post-info">
        <span>作者: {{ post.author.username }}</span>
        <span>发表时间:{{ post.create_time }}</span>
        <span>评论:{{ post.comments|length }}</span>
      </p>
    </div>
  </li>
  {%
    endfor %
  </ul>

...
<div class="list-group">
  {%
    if current_board %
      <a href="/" class="list-group-item">所有板块</a>
    {%
      else %
        <a href="/" class="list-group-item active">所有板块</a>
    {%
      endif %
    {%
      for board in boards %
        {%
          if board.id == current_board %
            <a href="#" class="list-group-item active">{{ board.name }}</a>
          {%
            else %
              <a href="#" class="list-group-item">{{ board.name }}</a>
            {%
              endif %
            {%
              endfor %
          </div>

```

上述代码中，首先循环帖子列表，然后把帖子的标题、作者、发表时间以及该帖子评论的数量都显示出来。在板块列表，通过`current_board`参数来表示当前选中的是哪个板块，该参数后续在实现根据板块过滤帖子功能时再加进去。

## 9.6.1 生成帖子测试数据

在帖子超过一定数量时应该进行分页，但是现在数据库中帖子的数量还太少，我们来生

成一些测试数据。这里需要用到Faker库来生成随机文字，Faker库是一个用来生成随机数据的库，可以生成姓名、邮箱、地址以及段落等内容。在PyCharm的Terminal中通过以下命令安装Faker库。

```
$ pip install faker
```

### 注意

更多Faker的使用文档请参考<https://faker.readthedocs.io/en/master/index.html>。

我们把生成帖子测试数据写成命令，在commands.py文件中添加以下代码。

```
from faker import Faker
import random

def create_test_post():
    fake = Faker(locale="zh_CN")
    author = UserModel.query.first()
    boards = BoardModel.query.all()

    click.echo("开始生成测试帖子...")
    for x in range(98):
        title = fake.sentence()
        content = fake.paragraph(nb_sentences=10)
        random_index = random.randint(0, 4)
        board = boards[random_index]
        post = PostModel(title=title, content=content, board=board, author=author)
        db.session.add(post)
        db.session.commit()
        click.echo("测试帖子生成成功! ")
```

在app.py文件中注册命令，代码如下。

```
$ app.cli.command("create-test-post") (commands.create_test_post)
```

打开PyCharm的Terminal，执行flask create-test-post命令，即可随机生成98篇测试帖子。

## 9.6.2 使用Flask-Paginate实现分页

在Flask项目中使用Flask-Paginate插件可以轻松地实现分页，Flask-Paginate用的是Bootstrap样式，正好与现在的项目架构一样，如果要使用其他样式，可以修改CSS属性。在PyCharm的Terminal中输入以下命令安装Flask-Paginate。

```
$ pip install flask-paginate
```

在config.py文件的BaseConfig中添加PER\_PAGE\_COUNT配置，用来指定一页中展示多少数据，这里设置10条，代码如下。

```
class BaseConfig:  
    ...  
  
    PER_PAGE_COUNT = 10
```

在blueprints/front.py的index视图函数中，按照当前的页码提取对应的数据，代码如下。

```
@bp.route("/")  
def index():  
    boards = BoardModel.query.all()  
  
    # 获取页码参数  
    page = request.args.get("page", type=int, default=1)  
  
    # 当前page下的起始位置  
    start = (page - 1) * current_app.config.get("PER_PAGE_COUNT")  
    # 当前page下的结束位置  
    end = start + current_app.config.get("PER_PAGE_COUNT")  
  
    # 查询对象  
    query_obj = PostModel.query.order_by(PostModel.create_time.desc())  
    # 总共有多少帖子  
    total = query_obj.count()  
  
    # 当前page下的帖子列表  
    posts = query_obj.slice(start, end)  
  
    # 分页对象  
    pagination = Pagination(bs_version=4, page=page, total=total,  
                           outer_window=0, inner_window=2, alignment="center")
```

```
context = {
    "posts": posts,
    "boards": boards,
    "pagination": pagination
}
return render_template("front/index.html", **context)
```

上述代码中，首先从URL参数中提取page，然后计算当前page下提取帖子的起始位置和结束位置。接着按照帖子的发布时间进行排序，统计总共有多少帖子，然后使用start和end提取帖子列表，最后构建分页对象Pagination，Pagination中的参数说明如下。

- `bs_version`: Bootstrap版本，我们的项目中用的是4，所以这里写4。
- `page`: 当前的page页码，从1开始。
- `total`: 所有帖子的总数量。
- `outer_window`: 页码数量太多，出现省略号后，在省略号外层要显示多少个页码，如这里设置为0，那么就会显示1个页码，总是比设置的多1。
- `inner_window`: 出现省略号后，当前页码左右两边要显示几个页码。
- `alignment`: 分页盒子在父盒子内的对齐方式，默认是左对齐，这里修改为中间对齐。

下面在`templates/front/index.html`中的帖子列表的最底部添加`{{ pagination.links }}`，示例代码如下。

```
<div class="post-group">
    <ul class="post-list-group">
        ...
    </ul>
    {{ pagination.links }}
</div>
```

在浏览器中重新访问首页，可以看到如图9-33所示的帖子分页效果。

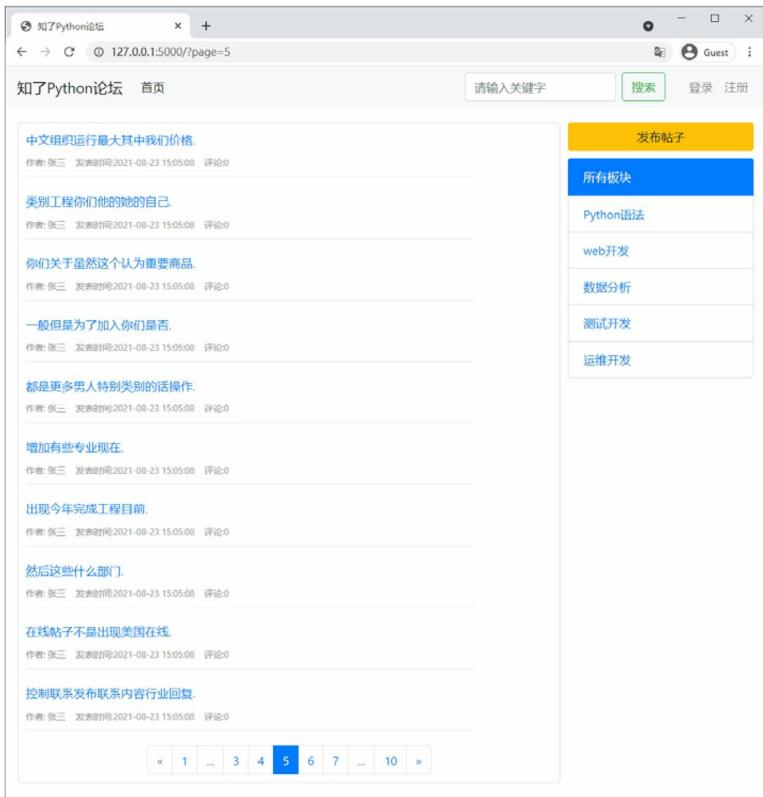


图9-33 帖子分页

### 9.6.3 过滤帖子

在图9-33中，实现了所有板块下的帖子分页功能，本节再来实现帖子按照板块过滤的功能。我们规定板块参数同样用查询字符串的方式传递，在blueprints/front.py的index视图函数中，将代码修改如下。

```
@bp.route("/")
def index():
    boards = BoardModel.query.all()

    # 获取页码参数
    page = request.args.get("page", type=int, default=1)
```

```

# 新增: 获取板块参数
board_id = request.args.get("board_id", type=int, default=0)

# 当前page下的起始位置
start = (page - 1) * current_app.config.get("PER_PAGE_COUNT")

# 当前page下的结束位置
end = start + current_app.config.get("PER_PAGE_COUNT")

# 查询对象
query_obj = PostModel.query.order_by(PostModel.create_time.desc())

# 新增: 过滤帖子
if board_id:
    query_obj = query_obj.filter_by(board_id=board_id)

# 总共有多少帖子
total = query_obj.count()

# 当前page下的帖子列表
posts = query_obj.slice(start, end)

# 分页对象
pagination = Pagination(bs_version=4, page=page, total=total,
outer_window=0, inner_window=2, alignment="center")

context = {
    "posts": posts,
    "boards": boards,
    "pagination": pagination,
    # 新增:
    "current_board": board_id
}
return render_template("front/index.html", **context)

```

上述代码中，注释中以“新增：”开头的是新增的代码。在上述代码中，我们先从request.args中获取board\_id参数，如果获取到了，则让query\_obj对board\_id进行过滤后重新赋值，这样得到的query\_obj就是该板块下所有的帖子。最后为了能在首页中显示当前的板块，将board\_id赋值给current\_board传给模板，为了在首页中实现板块过滤，给板块列表加上超链接，修改后的代码如下。

```

...
{% for board in boards %}
{% if board.id == current_board %}
<a href="{{ url_for('front.index', board_id=board.id) }}" class="list-
group-item active">{{ board.name }}</a>

```

```
% else %
<a href="{{ url_for('front.index', board_id=board.id) }}" class="list-
group-item">{{ board.name }}</a>
% endif %
% endfor %
...
```

加载以上代码后，在首页中单击右侧的某个板块，就可以实现根据板块过滤帖子的功能了。

## 9.7 帖子详情

### 9.7.1 动态加载帖子详情数据

在blueprints/post.py中，添加post\_detail视图函数，代码如下。

```
@bp.get("/post/detail/<int:post_id>")
def post_detail(post_id):
    post = PostModel.query.get(post_id)
    post.read_count += 1
    db.session.commit()
    return render_template("front/post_detail.html", post=post)
```

上述代码中，在URL中定义了一个post\_id参数，然后提取帖子对象，并且把帖子对象传到了post\_detail.html模板中。为了能访问帖子详情，首先要要在首页的帖子列表中补齐帖子标题的超链接，代码如下。

```
...
<p class="post-title">
    <a href="{{ url_for('front.post_detail', post_id=post.id) }}">
        {{ post.title }}
    </a>
</p>
...
```

在帖子详情页中，把帖子相关的数据填充进去，如发表时间、作者等，代码如下。

```
...
<div class="post-container">
```

```
<h2>{{ post.title }}</h2>
<p class="post-info-group">
    <span>发表时间: {{ post.create_time }}</span>
    <span>作者: {{ post.author.username }}</span>
    <span>所属板块: {{ post.board.name }}</span>
    <span>阅读数: {{ post.read_count }}</span>
    <span>评论数: {{ post.comments|length }}</span>
</p>
<article class="post-content" id="post-content">
    {{ post.content|safe }}
</article>
</div>
...
```

现在重新加载帖子详情页面，可以看到帖子数据都能正常显示在页面中了。

### 9.7.2 发布评论

在帖子详情页面底部有一个textarea文本框，用于发布帖子评论。在forms/post.py中实现一个评论表单功能，代码如下。

```
...
class PublicCommentForm(BaseForm):
    content = StringField(validators=[Length(min=2,max=200,message="请输入正确长度的评论! ")])
```

上述代码中，定义了content字段，发布评论只需要提交评论内容即可。然后在blueprints/front.py中，添加一个public\_comment视图函数，并添加以下代码。

```
@bp.post("/post/<int:post_id>/comment")
@login_required
def public_comment(post_id):
    form = PublicCommentForm(request.form)
    if form.validate():
        content = form.content.data
        comment = CommentModel(content=content, post_id=post_id, author=g.user)
        db.session.add(comment)
        db.session.commit()
    else:
        for message in form.messages:
            flash(message)
```

```
return redirect(url_for("front.post_detail", post_id=post_id))
```

上述代码中，我们把帖子id通过URL参数传过来，然后通过表单获取评论内容。发布评论的视图函数必须在登录的条件下才能访问，因此加了`login_required`装饰器。然后通过`g.user`即可获取到当前登录的用户作为该条评论的作者。评论发表成功，则重新加载帖子详情页面，这样用户就能看到最新的评论数据。评论发表失败，则把表单错误信息传到`flash`中，同时重新加载页面来显示错误信息。在`templates/front/post_detail.html`中添加显示错误信息和表单URL，代码如下。

```
...
<form action="{{ url_for('front.public_comment', post_id=post.id) }}" method="post">
    <textarea class="form-control" name="content" id="editor" cols="30" rows="5"></textarea>
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
    {% with messages = get_flashed_messages() %}
        {% if messages %}
            {% for message in messages %}
                <div class="text-danger mt-2">{{ message }}</div>
            {% endfor %}
        {% endif %}
        {% endwith %}
    <div class="comment-btn-group">
        <button class="btn btn-primary" id="comment-btn">发表评论</button>
    </div>
</form>
...
```

接着在`templates/front/post_detail.html`中，把帖子所有的评论循环显示出来，代码如下。

```
...
<ul class="comment-list-group">
    {% for comment in post.comments %}
        <li>
            <div class="comment-content">
                <p class="author-info">
                    <span>{{ comment.author.username }}</span>
                    <span>{{ comment.create_time }}</span>
                </p>
                <p class="comment-txt">
                    {{ comment.content }}
                </p>
            </div>
        </li>
    {% endfor %}
</ul>
...
```

现在重新访问任何一篇帖子的详情页，在登录的情况下都可以正常发表评论了。但是，新发表的评论会显示在帖子列表的后面，为了实现通过post.comments获取的评论列表能根据发表时间倒序排序，在CommentModel中将post的relationship代码修改如下。

```
...
post = db.relationship("PostModel", backref=db.backref('comments', order_by=create_time.desc()))
...
```

这样，就可以自动按照评论发表时间倒序排序了。

## 9.8 个人中心

在个人中心模块，可以修改用户头像、个性签名等。首先在blueprints/user.py中添加个人中心的视图函数，代码如下。

```
...
@bp.get("/profile/<string:user_id>")
def profile(user_id):
    user = UserModel.query.get(user_id)
    return render_template("front/profile.html", user=user)
...
```

上述代码中，把user\_id放到URL中当作参数传过来。访问当前用户的个人中心和其他用户的个人中心时都需要传递这个参数。下面添加限制，当访问当前用户的个人中心时可以进行编辑，访问其他用户的个人中心则不能编辑，代码如下。

```
...
@bp.get("/profile/<string:user_id>")
def profile(user_id):
    user = UserModel.query.get(user_id)
    is_mine = False
    if hasattr(g, "user") and g.user.id == user_id:
        is_mine = True
    context = {
        "user": user,
        "is_mine": is_mine
    }
    return render_template("front/profile.html", user=user)...
```

上述代码中，为了在模板中能区分当前是否是自己的个人中心，我们通过逻辑判断后把结果赋值给is\_mine，然后再把is\_mine传到模板中。

### 9.8.1 使用Flask-Avatars生成随机头像

用户没有上传头像时，如果显示空白会影响用户体验。可以使用Flask-Avatars插件来生成随机的头像。使用Flask-Avatars插件非常简单，先在exts.py中创建avatars对象，代码如下。

```
...
from flask_avatars import Avatars
...
avatars = Avatars()
```

在app.py中使用app对象进行初始化，代码如下。

```
...
from exts import avatars
...
avatars.init_app(app)
...
```

现在在模板中就可以使用avatars变量来随机生成头像了。Flask-Avatars提供了5种随机或默认头像的方案，下面分别进行讲解。

## 1. Gravatar

Gravatar（globally recognized avatar）是一种全球通用的头像服务。用户只需在Gravatar官网（<https://cn.gravatar.com/>）上使用邮箱注册一个账号，并且上传头像，那么在任何支持Gravatar的网站上使用相同的邮箱，都可以使用这个头像，当然如果用户没有上传过头像，则会生成一个随机的头像。在模板中使用avatars.gravatar函数，代码如下。

```

```

其中email\_hash是email的哈希值，可以通过以下代码获取邮箱哈希值。

```
import hashlib

def email_hash(email):
    return hashlib.md5(email.lower().encode("utf-8")).hexdigest()
```

Gravatar生成的头像效果图，如图9-34所示。

## 2. Robohash

Robohash（官网<https://robohash.org/>）是专门提供机器人随机头像的服务，通过avatars.robohash(text)获取头像，根据text随机生成头像，代码如下。

```

```

Robohash生成的头像效果图，如图9-35所示。

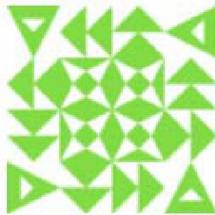


图9-34 Gravatar头像效果图

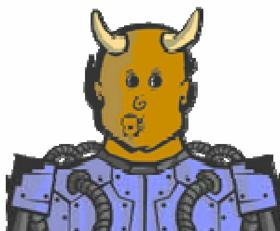


图9-35 Robohash头像效果图

### 3. 社交网站头像

针对全球比较流行的社交平台Twitter、Facebook、Instagram，可以使用`avatars.social_media`获取指定平台用户的头像。如使用Twitter上某人的头像，那么可以使用以下代码实现。

```

```

### 4. 默认头像

Flask-Avatars提供了一个默认的头像，可以通过设置`size`值为s、m、l来显示不同尺寸的头像，实现代码如下，效果如图9-36所示。

```

```

### 5. Identicon哈希头像

Identicon是一种基于用户信息的哈希值生成图像的技术，如根据IP地址、邮箱等，其雏形是9个方格的图案，现在已经发展到许多其他类型的生成模式了。Identicon生成的头像效果如图9-37所示。



图9-36 默认头像效果图

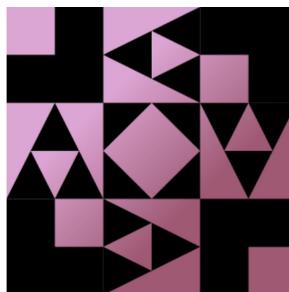


图9-37 Identicon头像效果图

Identicon会将生成的图片保存到指定位置。所以我们首先要在config.py中添加参数AVATARS\_SAVE\_PATH，如这里在DevelopmentConfig中添加如下代码。

```
AVATARS_SAVE_PATH = os.path.join(BaseConfig.UPLOAD_IMAGE_PATH, "avatars")
```

上述代码中，我们将头像的存储路径设置在BaseConfig UPLOAD\_IMAGE\_PATH的avatars文件夹下，然后在用户模型中实现一个生成头像的方法，示例代码如下。

```
class User(db.Model):
    avatar_s = db.Column(db.String(64))
    avatar_m = db.Column(db.String(64))
```

```
avatar_l = db.Column(db.String(64))

def __init__(self):
    generate_avatar()

def generate_avatar(self):
    avatar = Identicon()
    filenames = avatar.generate(text=self.email)
    self.avatar_s = filenames[0]
    self.avatar_m = filenames[1]
    self.avatar_l = filenames[2]
    db.session.commit()
```

以后在创建用户对象时，会自动调用generate\_avatar方法，并且将生成的头像分别存储到avatar\_s、avatar\_m和avatar\_l上。

现在采取上述第一种方案Gravatar来生成随机头像，avatars.gravatar需要使用邮箱的哈希值作为参数，需要在pythonbbs项目根路径下创建一个filters.py文件，用来存放过滤器。下面先创建一个email\_hash装饰器函数，代码如下。

```
import hashlib

def email_hash(email):
    return hashlib.md5(email.lower().encode("utf-8")).hexdigest()
```

下面在app.py中导入filters，通过app.template\_filter函数添加过滤器，代码如下。

```
# 添加模板过滤器
app.template_filter("email_hash")(filters.email_hash)
```

接着在templates/front/profile.html中，将头像部分的代码替换为如下代码。

```
...
<tr>
    <th>头像: </th>
    <td>
        
    </td>
</tr>
...
```

现在重新访问个人中心页面，即可以生成随机头像了。

### 9.8.2 修改导航条上的登录状态

现在再修改一下导航条右侧展示数据的逻辑。即在没有登录的情况下显示登录、注册链接；在登录的情况下显示用户名和退出登录链接；单击用户名链接时可以跳转到个人中心页面。下面在templates/front/base.html中，将导航条右侧部分的代码修改如下。

```
...
<ul class="navbar-nav ml-4">
    {% if g.user %}
        <li class="nav-item">
            <a href="{{ url_for('user.profile', user_id=g.user.id) }}" class="nav-link">{{ g.user.username }}</a>
        </li>
        <li class="nav-item">
            <a href="{{ url_for("user.logout") }}" class="nav-link">退出登录</a>
        </li>
    {% else %}
        <li class="nav-item">
            <a class="nav-link" href="{{ url_for('user.login') }}">登录</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="{{ url_for('user.register') }}">注册</a>
        </li>
    {% endif %}
</ul>
...
```

上述代码中，我们通过判断对象g是否有user属性，如果有，说明已经登录，则显示用户名和退出登录链接，如果没有，则显示登录和注册链接。上面我们还添加了退出登录的链接，接下来再来到blueprints/user.py中，添加logout视图函数代码。

```
...
@bp.get('/logout')
def logout():
    session.clear()
    return redirect("/")
...
```

通过在session中保存user\_id作为登录的标志，所以在上述代码中，清理session中的数据

即可完成退出登录，在退出登录后会跳转到首页。

### 9.8.3 根据用户显示个人中心

由于本人的个人中心和别人的个人中心用的是同一个URL和模板，因此需要在模板中进行区分。访问本人的个人中心时，数据可以编辑；访问别人的个人中心时，则仅展示数据。图9-38所示为本人的个人中心效果图，图9-39所示为别人的个人中心效果图。



图9-38 本人的个人中心



图9-39 别人的个人中心

图9-38和图9-39的区别在于，图9-38能对数据进行编辑，图9-39只能访问不能编辑。两个效果图的templates/front/profile.html模板代码如下。

```
...
<form action="" method="post" enctype="multipart/form-data">
  <table class="table table-bordered mt-5">
    <tbody>
      <tr>
        <th width="100px">用户名: </th>
        <td>
          {%- if is_mine %}
            <input type="text" name="username" value="{{ user.username }}"/>
          {%- else %}
            {{ user.username }}
          {%- endif %}
        </td>
      </tr>
      <tr>
        <th>头像: </th>
        <td>
          
          {%- if is_mine %}
            <input type="file" name="avatar" accept="image/jpeg, image/
png" value="上传头像">
          {%- endif %}
        </td>
      </tr>
      <tr>
        <th>签名: </th>
        <td>
          {%- if is_mine %}
            <input type="text" name="signature" value="{{ user.signature or
" " }}"/>
          {%- else %}
            {{ user.signature or " " }}
          {%- endif %}
        </td>
      </tr>
    </tbody>
  </table>
  {%- if is_mine %}
    <div style="text-align: center;">
      <button class="btn btn-primary">保存</button>
    </div>
  {%- endif %}
</form>
...
```

上述代码中，在渲染用户个人数据时，通过判断`is_mine`是否为`True`，用来决定是否需要渲染输入框。为了在`is_mine`为`True`的情况下，用户修改后的数据能进行保存，在`table`标签外套了一层`form`标签，并且因为涉及图片上传，所以又在`form`标签上添加了属性`enctype="multipart/form-data"`。由于现在还没有写好修改用户信息的URL和视图，因此`form`标签上的`action`暂时为空。另外，还有一些小细节，就是在使用`avatars.gravatar`渲染随机头像时，默认会使用Gravatar官网的服务器加载图片，因为Gravatar服务器在国外，可能导致加载太慢或者加载失败，所以将`https://gravatar.com/avatar/`替换为国内的镜像，如使用`https://gravatar.loli.net/avatar/`。

## 9.8.4 修改用户信息

在访问用户本人的个人中心，并且对数据进行修改后，单击“保存”按钮，把数据发送到视图函数进行修改。修改用户信息时，可以修改用户名、头像、签名，因此先在`forms/user.py`中添加一个`EditProfileForm`类，代码如下。

```
from wtforms import StringField
from flask_wtf import FileAllowed
...
class EditProfileForm(BaseForm):
    username = StringField(validators=[Length(min=2,max=20,message="请输入正确格式的用户名! ")])
    avatar = FileField(validators=[FileAllowed(['jpg','jpeg','png']),
        message="文件类型错误! "])
    signature = StringField()

    def validate_signature(self,field):
        signature = field.data
        if signature and len(signature) > 100:
            raise ValidationError(message="签名不能超过100个字符")
    ...

```

上述代码中添加了3个字段，分别为`username`、`avatar`和`signature`。其中`avatar`是文件类型，所以我们用`FileFiled`类型，为了限制文件后缀名，添加了`FileAllowed`验证器，指定只能上传`jpg`、`jpeg`以及`png`格式的文件。接着再把`EditProfileForm`类导入到`blueprints/user.py`，然后再实现一个视图函数`edit_profile`，代码如下。

```

...
from werkzeug.datastructures import CombinedMultiDict
from werkzeug.utils import secure_filename
from flask import send_from_directory
...
@bp.post("/profile/edit")
@login_required
def edit_profile():
    form = EditProfileForm(CombinedMultiDict([request.form, request.files]))
    if form.validate():
        username = form.username.data
        avatar = form.avatar.data
        signature = form.signature.data

        # 如果上传了头像
        if avatar:
            # 生成安全的文件名
            filename = secure_filename(avatar.filename)
            # 拼接头像存储路径
            avatar_path = os.path.join(current_app.config.get("AVATARS_SAVE_PATH"), filename)
            # 保存文件
            avatar.save(avatar_path)
            # 设置头像的URL
            g.user.avatar = url_for("media.media_file", filename=os.path.join("avatars", filename))

            g.user.username = username
            g.user.signature = signature
            db.session.commit()
            return redirect(url_for("user.profile", user_id=g.user.id))
        else:
            for message in form.messages:
                flash(message)
            return redirect(url_for("user.profile", user_id=g.user.id))

...

```

在Flask项目中，通过`request.files`可以获取前端上传的文件，通过`request.form`可以获取普通表单数据。为了对文件和普通表单数据都能做验证，使用`werkzeug.datastructures.CombinedMultiDict`方法将`request.files`和`request.form`合并成一个结构，再传给`EditProfileForm`进行验证。在上传了头像的情况下，把头像保存到`AVATARS_SAVE_PATH`参数配置项指定的路径下，然后使用`media.media_file`反转的URL，赋值给当前用户的`avatar`属性，并且在后面分别指定用户名和签名，即可完成用户信息的修改。

在个人中心模板中，再将头像渲染逻辑修改一下，首先判断user.avatar是否有值，如果有则渲染user.avatar，否则就渲染gravatar头像，代码修改后如下所示。

```
...
<th>头像: </th>
<td>
  {% if user.avatar %}
    
  {% else %}
    
  {% endif %}
  {% if is_mine %}
    <input type="file" name="avatar" accept="image/jpeg, image/png"
value="上传头像">
  {% endif %}
</td>
...
...
```

现在网站只有访问用户自己个人中心的入口，我们在帖子详情页中给帖子作者添加作者个人中心的入口，在`templates/front/post_detail.html`中，将显示作者部分的代码修改如下。

```
...
<span>作者: <a href="{{ url_for('user.profile',user_id=post.author.
id) }}>{{ post.author.username }}</a></span>
...
```

至此，实现了个人中心的所有功能。

## 9.9 CMS管理系统

一个能让用户发布内容的网站必须要有CMS管理系统，因为你不能确定用户发布的内容是否合法合规。`pythonbbs`网站的CMS系统包括帖子管理、评论管理、前台用户管理、后台用户管理等。其中许多模块的实现技术大同小异，我们会选择有代表性的模块进行讲解，其他模块读者可自行完成。

### 9.9.1 CMS入口

首先在`blueprints/cms.py`中实现一个CMS首页的视图函数，代码如下。

```
...
@bp.get("")
def index():
    return render_template("cms/index.html")
...
```

下面在前台页面导航条上判断是否是员工，如果是，则添加CMS入口链接。将 templates/front/base.html 的导航条部分代码修改如下。

```
...
{%
    if g.user.is_staff %}
    <li class="nav-item">
        <a href="{{ url_for('cms.index') }}" class="nav-link">管理系统</a>
    </li>
{% endif %}
<li class="nav-item">
    <a href="{{ url_for('user.profile', user_id=g.user.id) }}" class="nav-link">{{ g.user.username }}</a>
</li>
...

```

显示效果如图9-40所示。

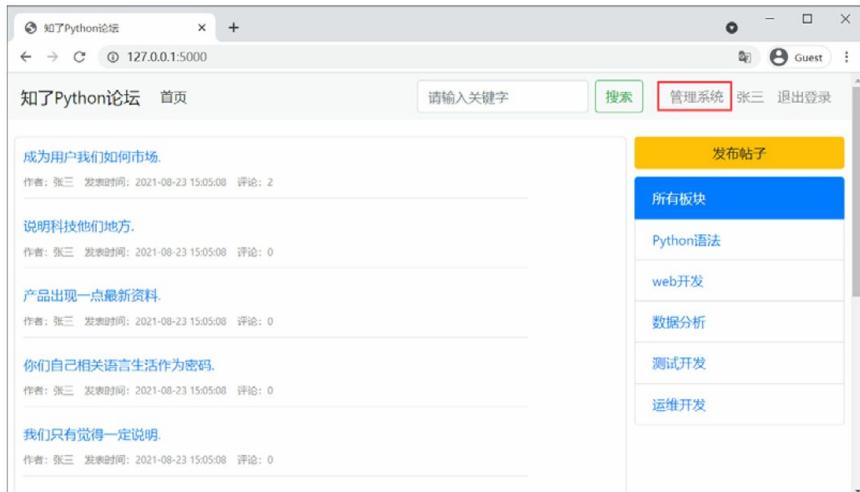


图9-40 有CMS入口的前台页面

单击右上角的“管理系统”，即可进入CMS系统的首页。

## 9.9.2 权限管理

我们在定义权限时，分别定义了板块、帖子、评论、前台用户、后台用户的管理权限，并且针对这些权限分别定义了稽查、运营、管理员3个角色，这3个角色拥有的权限请参考表9-1。另外，如果当前用户不是员工，则不允许访问CMS系统。我们先在blueprints/cms.py中添加before\_request钩子函数，并在访问该蓝图下的视图函数前，先判断user.is\_staff是否为True，如果不是，则不允许访问，代码如下。

```
...
@bp.before_request
def cms_before_request():
    if not hasattr(g, "user") or g.user.is_staff == False:
        return redirect("/")
...
```

之所以在cms的蓝图中添加before\_request钩子函数，而不是在app上添加，原因是这个判断只需针对cms的蓝图，而不需要进行全局判断。

我们再来针对角色做权限限制。先规定某个视图函数需要的权限，然后判断当前用户所属的角色有没有这个权限，如果有就能访问，否则就不能访问。我们可以使用装饰器来实现权限限制，在pythonbbs/decorators.py中添加以下代码。

```
...
from flask import abort
...
def permission_required(permission):
    def outer(func):
        @wraps(func)
        def inner(*args, **kwargs):
            if hasattr(g, "user") and g.user.has_permission(permission):
                return func(*args, **kwargs)
            else:
                return abort(403)
        return inner
    return outer
...
```

上述代码中，我们定义了一个装饰器permission\_required，这个装饰器接收一个权限作

为参数，在装饰器里面判断当前用户是否登录，并且是否拥有这个权限，如果有权限，则正常执行视图函数，否则抛出403错误。

如果用稽查用户访问CMS后台系统，依然可以看到，在左侧并没有权限的入口，如“用户管理”“员工管理”等，CMS管理系统首页如图9-41所示。



图9-41 CMS管理系统首页

这显然是不符合实际的。在侧边栏显示导航链接，应该先判断该用户是否有此项权限，有则渲染，无则不渲染。判断权限需要使用PermissionEnum，因此在blueprints/cms.py中添加context\_processor钩子函数，把PermissionEnum传给模板，代码如下。

```
...
from models.user import PermissionEnum

@bp.context_processor
def cms_context_processor():
    return {"PermissionEnum": PermissionEnum}
...
```

在templates/cms/base.html中，将侧边栏中的导航渲染功能修改为如下代码。

```
...
<ul class="nav-sidebar">
    <li class="unfold"><a href="{{ url_for('cms.index') }}>首页</a></li>
    {% set user = g.user %}
```

```
{% if user.has_permission(PermissionEnum.POST) %}
    <li class="nav-group post-manage"><a href="#">帖子管理</a></li>
{% endif %}
{% if user.has_permission(PermissionEnum.COMMENT) %}
    <li class="comments-manage"><a href="#">评论管理</a></li>
{% endif %}
{% if user.has_permission(PermissionEnum.BOARD) %}
    <li class="board-manage"><a href="#">板块管理</a></li>
{% endif %}
{% if user.has_permission(PermissionEnum.FRONT_USER) %}
    <li class="nav-group user-manage"><a href="#">前台用户管理</a></li>
{% endif %}
{% if user.has_permission(PermissionEnum.CMS_USER) %}
    <li class="nav-group UserModel-manage"><a href="#">员工管理</a></li>
{% endif %}
</ul>
...
```

此时我们再用稽查权限组下的用户访问CMS首页时，可以看到在侧边栏导航中只有“帖子管理”和“评论管理”了，如图9-42所示。

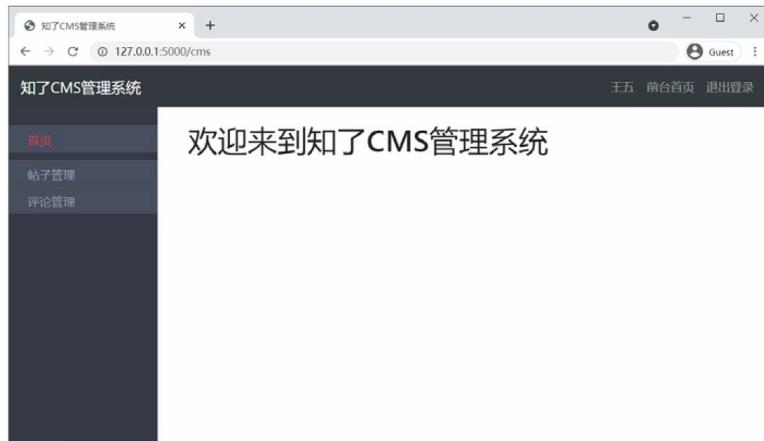


图9-42 有权限管理的CMS首页

### 9.9.3 员工管理页面

用户模型的属性`is_staff`为True的用户为员工。管理员角色下的用户可以对其他角色下的用户进行管理，如取消员工资格、修改分组等。管理员之间无法修改对方信息。我们首先在`blueprints/cms.py`中添加`staff_list`视图函数，代码如下。

```
@bp.get("/staff/list")
@permission_required(PermissionEnum.CMS_USER)
def staff_list():
    users = UserModel.query.filter_by(is_staff=True).all()
    return render_template("cms/staff_list.html",users=users)
```

因为员工管理必须要有PermissionEnum.CMS\_USER权限，所以要对视图函数添加@permission\_required(PermissionEnum.CMS\_USER)装饰器限制，包括后面的添加员工和编辑员工都属于员工管理，在编写相关视图时都要添加此装饰器限制。staff\_list视图函数中，首先提取了所有is\_staff为True的用户，没有做分页处理，分页逻辑与首页帖子列表分页是一样的，读者可以自行添加。然后将templates/cms/staff\_list.html的员工列表部分修改为如下代码。

```
<tbody>
  {% for user in users %}
    <tr>
      <td>{{ loop.index }}</td>
      <td>{{ user.email }}</td>
      <td>{{ user.username }}</td>
      <td>{{ user.join_time }}</td>
      <td>{{ user.role.name }}</td>
      <td>
        {% if not user.has_permission(PermissionEnum.CMS_USER) %}
          <a href="#" class="btn btn-info btn-sm">编辑</a>
        {% endif %}
      </td>
    </tr>
  {% endfor %}
</tbody>
```

上述代码中，我们循环员工列表，然后在表格的每列分别渲染对应的值。在渲染“编辑”按钮时，先判断该用户是否为管理员，如果是则不渲染，如果不是则渲染。在templates/cms/base.html中，在员工管理下添加超链接，实现单击即可跳转到员工管理页面的功能，代码如下。

```
...
  {% if user.has_permission(PermissionEnum.CMS_USER) %}
    <li class="nav-group UserModel-manage"><a href="{{ url_for('cms.staff_list') }}>员工管理</a></li>
  {% endif %}
...
```

访问员工管理页面，即可看到如图9-43所示的效果。

The screenshot shows a web browser window titled "员工管理" (Employee Management) with the URL "127.0.0.1:5000/cms/staff/list". The page header includes the "知了CMS管理系统" logo, a user session "张三", and navigation links "前台首页" and "退出登录". On the left, there is a vertical sidebar menu with items: 首页 (Home), 帖子管理 (Post Management), 评论管理 (Comment Management), 板块管理 (Category Management), 用户管理 (User Management), and 员工管理 (Employee Management). The main content area is titled "员工管理" and contains a blue "添加员工" (Add Employee) button. Below it is a table listing three employees:

#	邮箱	用户名	加入时间	角色	操作
1	zhangsan@zlt.net	张三	2021-08-17 20:28:07	管理员	<button>编辑</button>
2	wangwu@zlt.net	王五	2021-08-17 20:28:08	稽查	<button>编辑</button>
3	lisi@zlt.net	李四	2021-08-17 20:28:07	运营	<button>编辑</button>

图9-43 员工管理页面

#### 9.9.4 添加员工

在员工管理页面，有一个“添加员工”按钮，单击后可以跳转到“添加员工”页面。先在 blueprints/cms.py 中添加 add\_staff 视图函数，代码如下。

```
@bp.route("/staff/add", methods=['GET', 'POST'])
@permission_required(PermissionEnum.CMS_USER)
def add_staff():
    if request.method == "GET":
        roles = RoleModel.query.all()
        return render_template("cms/add_staff.html", roles=roles)
```

上述代码中，首先在GET请求中渲染了添加员工的模板。然后在员工管理页面的“添加员工”按钮上，添加跳转到“添加员工”页面的超链接，代码如下。

```
<a href="{{ url_for('cms.add_staff') }}" class="btn btn-primary mb-3">
添加员工</a>
```

单击“添加员工”按钮，即可看到如图9-44所示的效果。

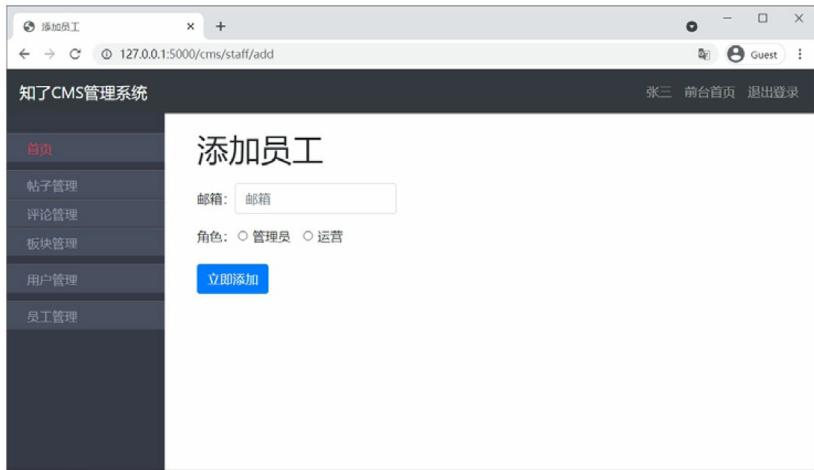


图9-44 “添加员工”页面效果

在图9-44中，角色只是静态数据，我们将角色部分代码修改成如下形式。

```
<div class="form-group">
    <label>角色: </label>
    {%
        for role in roles %
    %}
        <div class="form-check form-check-inline">
            <input class="form-check-input" type="radio" name="role" id="inlineRadio{{ loop.index }}"
            value="{{ role.id }}"/>
            <label class="form-check-label" for="inlineRadio{{ loop.index }}">
                {{ role.name }}
            </label>
        </div>
    {%
        endfor %
    </div>
```

我们再访问“添加员工”页面，角色部分就能渲染出真实数据了，如图9-45所示。

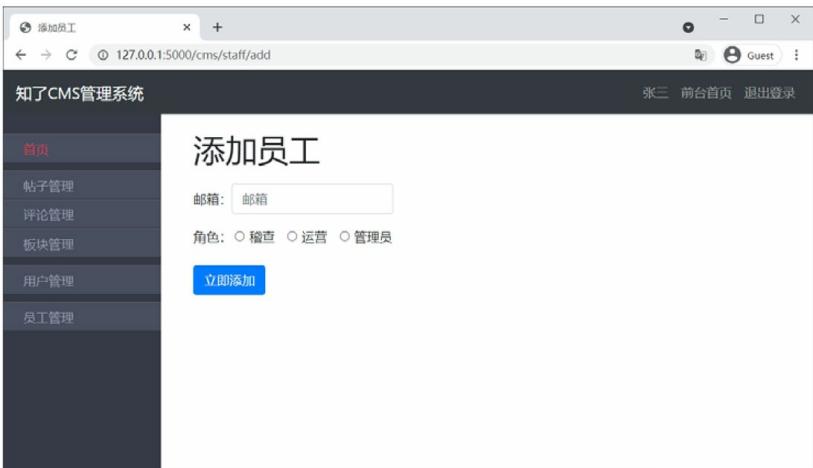


图9-45 渲染真实角色数据的“添加员工”页面

下面再在forms文件夹下创建cms.py文件，用来存放cms蓝图下的表单，并且创建AddStaffForm类，代码如下。

```
from .baseform import BaseForm
from wtforms import StringField, IntegerField
from wtforms.validators import Email, InputRequired

class AddStaffForm(BaseForm):
    email = StringField(validators=[Email(message="请输入正确格式的邮箱!")])
    role = IntegerField(validators=[InputRequired(message="请选择角色!")])
```

接着在blueprints/cms.py的add\_staff视图函数中，完成在POST请求情况下的员工数据验证与保存，代码如下。

```
@bp.route("/staff/add",methods=['GET','POST'])
@permission_required(PermissionEnum.CMS_USER)
def add_staff():
    if request.method == "GET":
        roles = RoleModel.query.all()
        return render_template("cms/add_staff.html",roles=roles)
    else:
        form = AddStaffForm(request.form)
        if form.validate():
            email = form.email.data
            role_id = form.role.data
            user = UserModel.query.filter_by(email=email).first()
            if not user:
                flash("没有此用户！")
                return redirect(url_for("cms.add_staff"))
            user.is_staff = True
            user.role = RoleModel.query.get(role_id)
            db.session.commit()
        return redirect(url_for("cms.staff_list"))
```

以后如果再添加员工，用户把注册网站时的邮箱发给管理员，即可选择角色添加为员工。

## 9.9.5 编辑员工

编辑员工要求只能编辑员工的分组、取消员工访问后台的权限，不能修改员工的邮箱、用户名、密码等信息。我们首先在blueprints/cms.py中添加一个名叫edit\_staff的视图函数，并添加以下代码。

```
@bp.route("/staff/edit/<string:user_id>",methods=['GET','POST'])
@permission_required(PermissionEnum.CMS_USER)
def edit_staff(user_id):
    user = UserModel.query.get(user_id)
    if request.method == 'GET':
        roles = RoleModel.query.all()
        return render_template("cms/edit_staff.html",user=user,roles=roles)
```

下面在员工管理的员工列表部分，为“编辑”按钮添加编辑员工的超链接，代码如下。

```
{% if not user.has_permission(PermissionEnum.CMS_USER) %}
<a href="{{ url_for('cms.edit_staff',user_id=user.id) }}" class="btn"
```

```
btn-info btn-sm">编辑</a>
{%
endif %}
```

重新访问员工管理页面，然后随机单击某个用户的“编辑”按钮，即可跳转到“编辑员工”页面，如图9-46所示。



图9-46 “编辑员工”页面

接着在forms/cms.py中添加一个EditStaffForm的表单，代码如下。

```
class EditStaffForm(BaseForm):
    is_staff = BooleanField(validators=[InputRequired(message="请选择是否为
    员工! ")])
    role = IntegerField(validators=[InputRequired(message="请选择分组! ")])
```

完善edit\_staff视图函数在POST请求情况下的逻辑处理，代码如下。

```
@bp.route("/staff/edit/<string:user_id>", methods=['GET', 'POST'])
@permission_required(PermissionEnum.CMS_USER)
def edit_staff(user_id):
    user = UserModel.query.get(user_id)
    if request.method == 'GET':
        roles = RoleModel.query.all()
        return render_template("cms/edit_staff.html", user=user, roles=roles)
    else:
        form = EditStaffForm(request.form)
        if form.validate():
            is_staff = form.is_staff.data
            role_id = form.role.data

            user.is_staff = is_staff
            if user.role.id != role_id:
                user.role = RoleModel.query.get(role_id)
            db.session.commit()
            return redirect(url_for("cms.edit_staff", user_id=user_id))
        else:
            for message in form.messages:
                flash(message)
            return redirect(url_for("cms.edit_staff", user_id=user_id))
```

加载以上代码，在浏览器中就可以修改非管理员角色的用户信息了。

## 9.9.6 管理前台用户

前台用户的管理工作主要包括禁用和取消禁用，如果业务复杂一些，还可以实现对某个用户禁言一段时间的功能。但是无法对用户的信息进行编辑，如修改邮箱、密码等。另外，考虑到数据的价值，一般不会轻易删除用户。下面在blueprints/cms.py中添加user\_list视图函数，用于返回用户列表。

```
@bp.route("/users")
@permission_required(PermissionEnum.FRONT_USER)
def user_list():
    users = UserModel.query.filter_by(is_staff=False).all()
    return render_template("cms/users.html", users=users)
```

再在templates/cms/base.html中，对侧边栏中的“用户管理”添加超链接，代码如下。

```
<li class="nav-group user-manage"><a href="{{ url_for('cms.user_list') }}>
    用户管理</a></li>
```

访问“用户管理”页面，即可看到如图9-47所示的效果。

在“用户管理”页面的用户列表中，每行都有“禁用”按钮，这种情况比较适合使用AJAX方式来实现，首先在blueprints/cms.py中实现active\_user视图函数，代码如下。

```
@bp.post("/users/active/<string:user_id>")
@permission_required(PermissionEnum.FRONT_USER)
def active_user(user_id):
    is_active = request.form.get("is_active", type=int)
    if is_active == None:
        return restful.params_error(message="请传入is_active参数！")
    user = UserModel.query.get(user_id)
    user.is_active = bool(is_active)
    db.session.commit()
    return restful.ok()
```

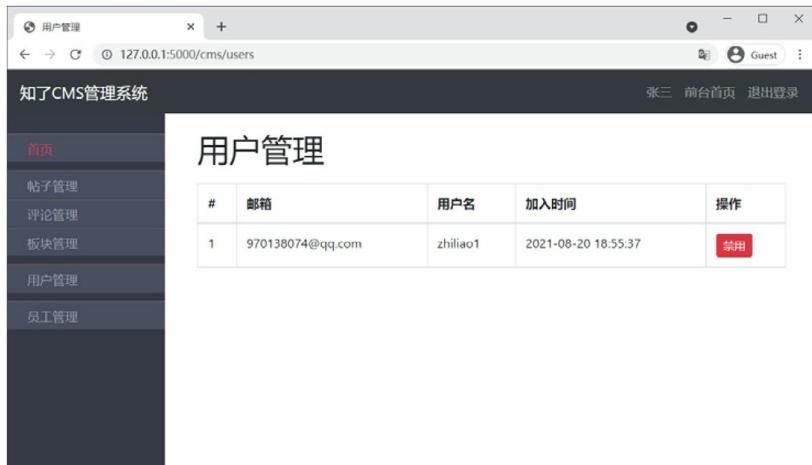


图9-47 “用户管理”页面效果

因为用AJAX来交互数据，所以视图函数中需要用restful模块返回JSON格式的响应，在实现JavaScript代码前，我们有必要先了解“禁用”和“取消禁用”按钮在模板templates/cms/users.html中的代码结构，代码如下。

```
...
```

```
<td>
  {%
    if user.is_active %}
      <button class="btn btn-danger btn-sm active-btn" data-active="1"
data-user-id="{{ user.id }}">禁用</button>
    {%
      else %}
        <button class="btn btn-info btn-sm active-btn" data-active="0"
data-user-id="{{ user.id }}">取消禁用</button>
    {%
      endif %}
  </td>
  ...

```

上述代码中，把用户的id和用户当前是否可用的值，通过data-\*属性绑定到了“禁用”和“取消禁用”按钮上，以方便后面在JavaScript代码中获取。接下来在static/cms/js文件夹下创建users.js文件，并且添加以下代码。

```
$(function () {
  $(".active-btn").click(function (event) {
    event.preventDefault();
    var $this = $(this);
    var is_active = parseInt($this.attr("data-active"));
    var message = is_active?"您确定要禁用此用户吗？ ":"您确定要取消禁用此用户吗？";
    var user_id = $this.attr("data-user-id");
    var result = confirm(message);
    if(!result){
      return;
    }
    var data = {
      is_active: is_active?0:1
    }
    zlajax.post({
      url: "/cms/users/active/" + user_id,
      data: data
    }).done(function (){
      window.location.reload();
    }).fail(function (error){
      alert(error.message);
    })
  });
});
```

上述代码中，因为所有的“禁用”和“取消禁用”按钮的类名都包含active-btn，所以通过寻找类名为active-btn的元素来绑定单击事件。然后通过获取is\_active来判断当前用户应“禁用”还是“取消禁用”。使用confirm来判断是否真的要执行下一步的操作，效果如图9-48所示。

图9-48中，如果单击OK按钮，则发送AJAX请求更改用户状态。

在用户被禁用后，应该限制用户登录，在blueprints/user.py的login视图函数中，将代码修改如下。

```
...  
if user and user.check_password(password):  
    if not user.is_active:  
        flash("该账户已被禁用！")  
        return redirect(url_for("user.login"))  
    session['user_id'] = user.id  
    if remember:  
        session.permanent = True  
    return redirect("/")  
...  
...
```

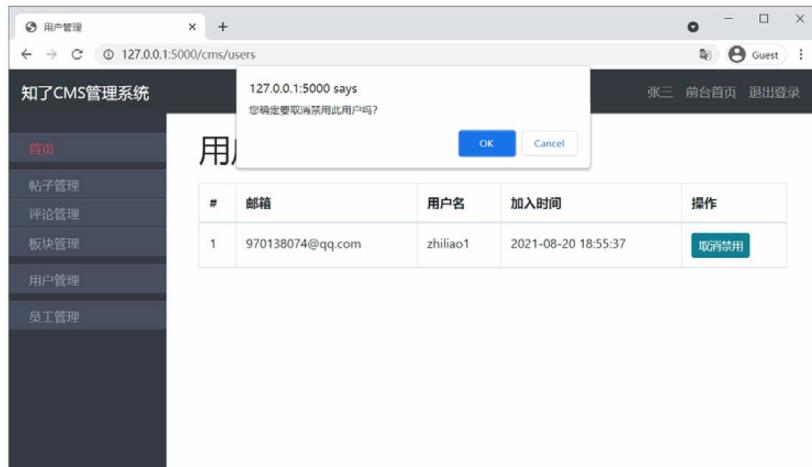


图9-48 确认是否取消禁用

除此之外，在decorators.py模块下的login\_required装饰器中，也应该添加对用户是否被禁用的验证。将login\_required代码修改如下。

```
...  
def inner(*args, **kwargs):  
    if not hasattr(g, "user"):  
        return redirect(url_for("user.login"))  
    elif not g.user.is_active:  
...
```

```
    flash("该用户已被禁用！")
    return redirect(url_for("user.login"))
else:
    return func(*args, **kwargs)
...
```

至此，在用户被禁用的状态下，被禁用的用户将无法访问所有需要登录权限的页面了。

### 9.9.7 帖子管理

帖子管理工作也仅仅是隐藏和显示，不能帮用户编辑帖子，隐藏帖子功能与禁用用户类似。我们首先实现帖子管理视图，用于渲染帖子列表。在blueprints/cms.py中添加post\_list和active\_post两个视图函数，代码如下。

```
...
@bp.get('/posts')
@permission_required(PermissionEnum.POST)
def post_list():
    posts = PostModel.query.all()
    return render_template("cms/posts.html", posts=posts)

@bp.post('/posts/active/<int:post_id>')
def active_post(post_id):
    is_active = request.form.get("is_active", type=int)
    if is_active == None:
        return restful.params_error(message="请传入is_active参数！")
    post = PostModel.query.get(post_id)
    post.is_active = bool(is_active)
    db.session.commit()
    return restful.ok()
...
```

上述代码中，post\_list视图函数没有帖子分页的功能，读者可以参照首页帖子列表的分页功能进行实现。在浏览器中访问“帖子管理”页面，效果如图9-49所示。

The screenshot shows a web browser window titled "帖子管理" (Post Management) with the URL "127.0.0.1:5000/cms/posts". The top navigation bar includes links for "张三" (Zhang San), "前台首页" (Frontend Home), and "退出登录" (Logout). On the left, there is a sidebar with a red header "知道了CMS管理系统" and a list of management modules: "首页" (Home), "帖子管理" (Post Management), "评论管理" (Comment Management), "板块管理" (Module Management), "用户管理" (User Management), and "员工管理" (Employee Management). The main content area is titled "帖子管理" and displays a table of posts. The table has columns: 标题 (Title), 发布时间 (Release Time), 板块 (Module), 作者 (Author), and 操作 (Operation). The data in the table is as follows:

标题	发布时间	板块	作者	操作
Python语法基础	2021-08-23 10:54:33	Python语法	zhiliao1	<button>隐藏</button>
说明科技他们地方.	2021-08-23 15:05:08	web开发	张三	<button>隐藏</button>
产品出现一点最新资料.	2021-08-23 15:05:08	测试开发	张三	<button>隐藏</button>
你们自己相关语言生活作为密码.	2021-08-23 15:05:08	web开发	张三	<button>隐藏</button>
我们只有觉得一定说明.	2021-08-23 15:05:08	Python语法	张三	<button>隐藏</button>
部门可能但是为了活动日本包括.	2021-08-23 15:05:08	运维开发	张三	<button>隐藏</button>

图9-49 “帖子管理”页面

在static/cms/js下创建一个posts.js文件，并添加以下代码。

```
$(function () {
    $(".active-btn").click(function (event) {
        event.preventDefault();
        var $this = $(this);
        var is_active = parseInt($this.attr("data-active"));
        var message = is_active?"您确定要隐藏此帖子吗？ ":"您确定要显示此帖子吗？ ";
        var post_id = $this.attr("data-post-id");
        var result = confirm(message);
        if(!result){
            return;
        }
        var data = {
            is_active: is_active?0:1
        }
        console.log(data);
        zlajax.post({
            url: "/cms/posts/active/" + post_id,
            data: data
        }).done(function () {
            window.location.reload();
        }).fail(function (error){
            alert(error.message);
        })
    });
});
```

---

上述代码的实现逻辑与禁用用户是一样的，仅需要修改相关参数和URL即可。接下来在 template/cms/posts.html 的 head block 中，通过 script 标签加载 posts.js 文件，代码如下。

```
{% block head %}  
  <script  
  src="{{ url_for('static', filename='cms/js/posts.js') }}"></script>  
{% endblock %}
```

这样在浏览器中就可以通过单击“隐藏”或者“显示”按钮来对帖子进行操作了。当帖子被隐藏后，在首页渲染帖子列表时，应该过滤掉被隐藏的帖子。在 blueprints/front.py 的 index 视图函数中，将提取帖子的代码修改如下。

```
...  
query_obj = PostModel.query.filter_by(is_active=True).order_by(PostModel.  
create_time.desc())  
...
```

## 9.9.8 评论管理

评论管理的实现逻辑与帖子管理类似，读者可以自行完成。但是有一点需要注意，评论被禁用后，在帖子详情页应该过滤被禁用的评论。现在我们是通过 post.comments 获取帖子下的评论，由于 post.comments 是一个列表，无法使用 filter\_by 方法进行过滤，所以我们将 models/post.py 中的 CommentModel 模型的 post 属性修改为如下代码。

```
...  
post = db.relationship("PostModel", backref=db.backref('comments', order_  
by=create_time.desc(), lazy="dynamic"))  
...
```

上述代码中，在 db.backref 函数中加入了 lazy="dynamic" 参数，这将使 post.comments 变成一个 AppenderQuery 对象，从而可以使用 filter 或者 filter\_by 方法进行过滤。我们在帖子详情页 templates/front/post\_detail.html 中将评论列表部分修改成如下代码。

```
...
```

```
{% for comment in post.comments.filter_by(is_active=True) %}  
...  
%
```

因为post.comments不是列表类型了，无法使用length过滤器，所以可以使用AppenderQuery的count方法，在首页模板templates/front/index.html和帖子详情模板templates/front/post\_detail.html的显示评论数量的地方，将代码修改如下。

```
...  
{% post.comments.count() %}  
...
```

## 9.9.9 板块管理

板块管理使用到的知识点与前面的用户管理类似，这里不再重复讲解，读者可以自行完成。板块管理要实现的功能如下。

- (1) “禁用”和“取消禁用”：禁用板块后，在首页中不应该再渲染该板块，并且该板块下的帖子不能访问，所以在帖子详情中要做好判断。
- (2) 编辑板块：仅可以对板块的名称进行修改。

## 9.10 错误处理

网站在处理用户的请求后，响应状态码不一定全部是200，有可能出现URL不存在的404错误，或者没有权限访问的403错误等。针对这些非200的错误状态码，可以为每个错误状态码实现一个页面，在出现类似错误时，Flask会自动返回一个包含错误状态码的页面。添加错误状态码的处理逻辑是通过@app.errorhandler装饰器实现的，也是属于钩子函数的一种，我们在hooks.py中添加以下代码。

```
...  
def bbs_404_error(error):  
    return render_template("errors/404.html"), 404  
  
def bbs_401_error(error):  
    return render_template("errors/401.html"), 401  
  
def bbs_500_error(error):  
    return render_template("errors/500.html"), 500
```

下面在app.py中，将以上3个处理错误的钩子函数注册到app中，代码如下。

```
...
app.errorhandler(401)(hooks.bbs_401_error)
app.errorhandler(404)(hooks.bbs_404_error)
app.errorhandler(500)(hooks.bbs_500_error)
```

在浏览器中访问一个不存在的URL，如http://127.0.0.1:5000/abc，可以看到会显示404错误页面，如图9-50所示。

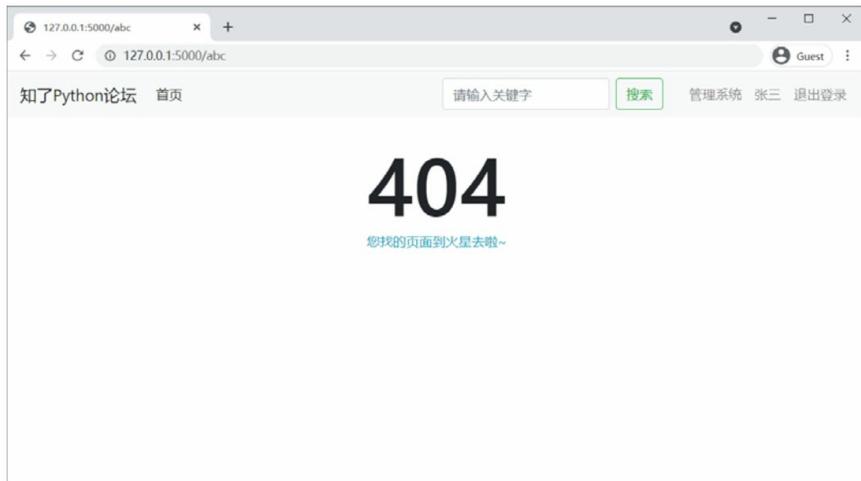


图9-50 404错误页面

## 9.11 日志

日志是一个商业网站必备的功能，用日志可以记录网站运行中产生的信息，这些信息包括网站运行时产生的异常或用户行为等，具体情况如下。

- 用户相关的行为：如登录、退出登录、注册、找回密码、不正确的密码尝试等。通过日志记录和分析用户的行为，可以提高网站用户体验以及发现非法登录等。
- 补充数据库记录：如后台用户谁禁用了帖子、谁添加了新员工。这些在数据库中没有记录的，可以用日志补充记录。

- ☑ 错误：如程序出现异常、数据库操作异常、响应了错误状态码等，都可以通过日志记录下来，方便后期优化程序。

Flask使用了Python中内置的logging模块。logging模块有4个子模块，分别为loggers、handlers、filters和formatters，下面分别进行讲解。

### 9.11.1 loggers模块

loggers模块是用来创建日志的，在Flask中通过app.logger可以获取当前的logger对象，然后使用logger.info、logger.debug等級別函数创建日志。在Python内置的logging模块中，日志分为6个级别，如表9-5所示。

表9-5 日志级别

级别名称	数值	描述
NOTSET	0	没有设置
DEBUG	10	调试级别
INFO	20	信息级别
WARNING/WARN	30	警告级别
ERROR	40	普通错误级别
CRITICAL/FATAL	50	致命错误级别

在记录日志时，只会记录比当前设置级别高的日志。如设置日志级别为INFO，那么logger.debug将不会产生日志记录。Flask中的默认级别为DEBUG，如果要修改默认级别，可以通过app.logger.setLevel来修改。如修改为INFO级别，代码如下。

```
app.logger.setLevel(logging.INFO)
```

我们在blueprints/front.py的index视图函数中，添加一条测试日志，代码如下。

```
current_app.logger.info("index页面被请求了")
```

在浏览器中访问首页后，在PyCharm的Run界面会显示如图9-51所示的日志。

```
[2021-08-26 16:22:57,656] INFO in front: index页面被请求了  
127.0.0.1 - - [26/Aug/2021 16:22:57] "GET / HTTP/1.1" 200 -
```

图9-51 PyCharm的Run界面显示的日志

### 9.11.2 handlers模块

handlers模块用来指定日志被定向到何处。在Flask中默认是定向到控制台打印。如果要定向到文件，那么就用FileHandler或RotatingFileHandler；如果要使用邮箱发送日志，那么就用SMTPHandler。

#### 注意

关于logging模块支持的所有handler，读者如果感兴趣可以参阅Python官方文档<https://docs.python.org/3/library/logging.handlers.html>中的logging模块。

如果要将日志定向到文件中，则可以给app.logger再添加一个handler，代码如下。

```
file_handler = logging.FileHandler("pythonbbs.log",encoding="utf-8")
app.logger.addHandler(file_handler)
```

我们在请求项目的首页可以看到，除了控制台显示日志信息，还会在pythonbbs项目的根路径下产生一个pythonbbs.log的文件，里面有日志信息。如果不想要打印Flask默认在控制台的日志信息，可以通过以下代码移除。

```
from flask.logging import default_handler
app.logger.removeHandler(default_handler)
```

FileHandler会把所有日志打印在一个文件中，在网站运行中这个文件大小将不断变大。在商业项目中可以使用RotatingFileHandler或TimeRotatingFileHandler，RotatingFileHandler可以指定文件大小，当超过最大文件限制时将自动开启一个新的日志文件。TimeRotatingFileHandler则是根据时间来判断是否要开启一个新的日志文件，下面分别进行讲解。

(1) RotatingFileHandler(filename,maxBytes=0,backupCount=0): 当maxBytes $\geq 0$ 时，如果文件大小超过maxBytes，则会创建一个新的文件，新的文件以filename为基本名称，并且在

名称后面加上“.1”“.2”等。如filename的值是app.log，那么产生的新文件为app.log.1、app.log.2等。backupCount指定最多创建多少个文件，只有backupCount≥1时，才会产生新的文件。RotatingFileHandler写入日志的逻辑是这样的，先写入filename指定的文件中，如app.log，如果app.log的大小接近maxBytes，则将app.log重命名为app.log.1，然后创建新的app.log用于写入日志。

(2) TimeRotatingFileHandler(filename,when,interval=1,backupCount=0): when参数用于指定按什么时间单位创建新的日志文件，when可以取以下值。

- S: 按照秒为单位。
- M: 按照分钟为单位。
- H: 按照小时为单位。
- D: 按照天为单位。
- midnight: 按照半夜12点为单位。
- W{0-6}: 按照星期为单位，W0为星期一。

interval表示等待多少个单位when的时间后创建新的日志文件。创建文件的规则与RotatingFileHandler类似，如果backupCount不为0，则最多保留backupCount个文件，如果创建超过backupCount个的文件，则最旧的文件会被删除。

### 9.11.3 filters模块

filters模块是用来给Logger和Handler提供过滤器的，只有过滤器返回True的日志才会被记录，先执行Logger的过滤器，再执行Handler的过滤器。这里以给Logger添加过滤器为例，代码如下。

```
class stringFilter(logging.Filter):
    def filter(self, record):
        if record.msg.startswith("abc"):
            return False
        return True

app.logger.addFilter(stringFilter())
app.logger.info("abc-test")
app.logger.info("123-test")
```

上述代码中，因为在app.logger中添加了stringFilter过滤器，会过滤掉以abc开头的日志，

因此只会记录123-test，而不会记录abc-test。其中filter方法中的record参数为logging.LogRecord对象。

#### 9.11.4 formatters模块

formatters模块用来指定日志记录的格式。logging模块内置了一些变量，我们在定义格式时可以直接使用。变量的说明如下。

- %(asctime)s: 日志被创建的时间。
- %(filename)s: 被创建的日志所在的文件。
- %(funcName)s: 被创建的日志所在的函数。
- %(levelname)s: 被创建的日志的级别。
- %(lineno)d: 被创建的日志所在文件的代码行号。
- %(message)s: 日志文本的内容。
- %(module)s: 被创建的日志所在的模块。

我们可以使用以上变量，灵活地组合自己想要的日志格式。完整的使用示例代码如下。

```
import logging
from flask.logging import default_handler
from logging.handlers import RotatingFileHandler

# 移除Flask自带的handler
app.logger.removeHandler(default_handler)

# 创建一个RotatingFileHandler对象
file_handler = RotatingFileHandler('pythonbbs.log', maxBytes=16384,
                                    backupCount=20)

# 设置handler级别为INFO
file_handler.setLevel(logging.INFO)

# 创建日志记录的格式
file_formatter = logging.Formatter('%(asctime)s %(levelname)s: %(message)s
                                    [in %(filename)s: %(lineno)d]')

# 将日志格式对象添加到handler中
file_handler.setFormatter(file_formatter)

# 将handler添加到app.logger中
app.logger.addHandler(file_handler)
```

## 9.12 部署

项目的所有功能都开发完成，并且经过测试没有bug后，就可以把项目部署到服务器上了，这里以虚拟机Ubuntu 20.04 LTS Server版的操作系统为例进行讲解。读者可以自行购买云服务器，如阿里云、腾讯云、华为云等进行部署，操作方式大同小异。

### 9.12.1 导出依赖包

在项目开发完成后，我们把项目使用的虚拟环境的依赖包导出，以方便在服务器上进行安装。在PyCharm的Terminal中输入以下命令完成依赖包的导出。

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

执行完上述命令后，会在项目的根路径下生成一个requirements.txt文件，这个文件记录了当前项目所依赖的包，当项目上传到服务器后，通过以下命令即可一次性完成所有依赖包的安装。

```
pip install -r requirements.txt
```

### 9.12.2 使用Git上传代码

本地开发好的项目代码，可以通过多种方式上传到服务器，如scp、Git等。Git更新代码更加方便，而且有版本管理功能，可以随时切换到之前的版本，本书选择Git来提交代码。首先在开发机上安装Git，只要从<https://git-scm.com/downloads>根据自己的操作系统下载最新的Git，然后安装即可。

Git安装完成后，还需要用到Git服务器。可以自己搭建Git服务器，也可以使用第三方Git服务。如国内的有Gitee（官网[www.gitee.com](http://www.gitee.com)）、Coding（官网[www.coding.net](http://www.coding.net)），国外的有Github（官网[www.github.com](http://www.github.com)）、Gitlab（官网[www.gitlab.com](http://www.gitlab.com)）等。Github是全球最大的代码托管网站，并且可以免费创建任意数量的私有项目，我们以Github为例来讲解。首先在Github上注册账号，然后在Github网站上创建一个名叫pythonbbs的仓库，为了使项目不被其他用户看到，选中Private单选按钮，如图9-52所示。

**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

---

Owner \*                          Repository name \*

 NunchakusHuang / pythonbbs

Great repository names are short and memorable. Need inspiration? How about [cuddly-rotary-phone?](#)

Description (optional)

Flask全栈开发书籍实战项目

---

 **Public**  
Anyone on the internet can see this repository. You choose who can commit.

 **Private**  
You choose who can see and commit to this repository.

---

**Initialize this repository with:**

Skip this step if you're importing an existing repository.

**Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

**Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

---

**Create repository**

图9-52 在Github上创建仓库

在图9-52中单击Create repository按钮后，出现如图9-53所示界面。

在图9-53中，可以看到新创建的pythonbbs项目的仓库地址，后续需要将本地的仓库地址和远程的仓库地址映射起来。

接下来，在本地pythonbbs项目的根路径下右击，在弹出的快捷菜单中选择Git Bash Here命令，就会打开Git的操作终端，如图9-54所示。

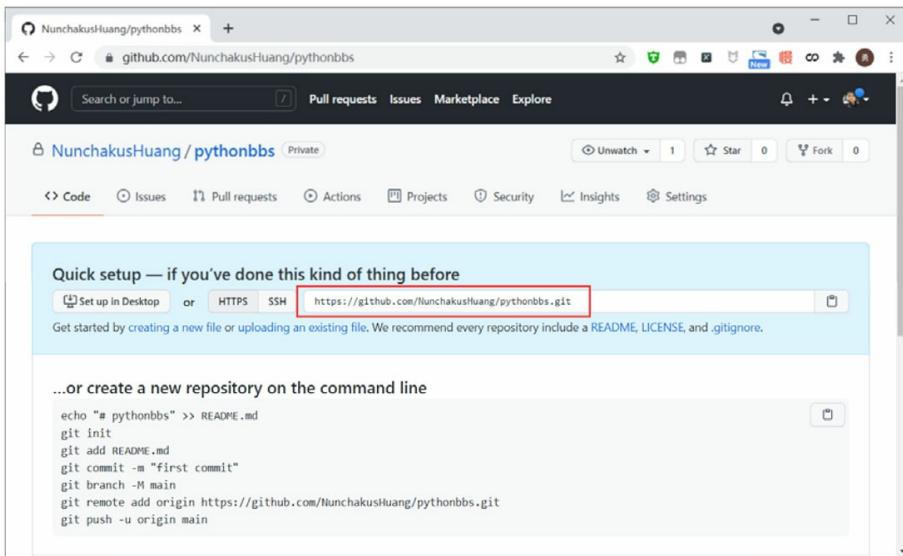


图9-53 创建仓库后的界面

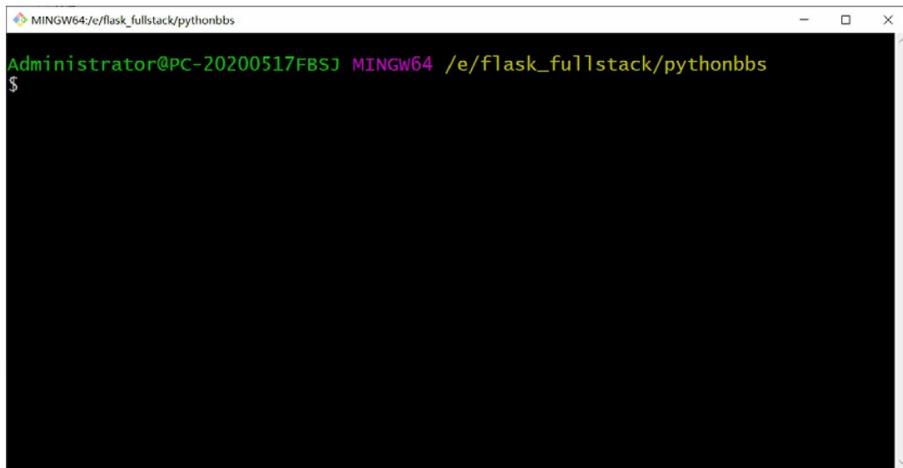


图9-54 Git操作终端

执行以下命令，将本地仓库的代码推送到Github服务器。

## 1. 初始化仓库

```
$ git init
```

执行以上命令后，会将pythonbbs项目初始化成git仓库，在项目的根路径下就会多出一个.git文件夹。

## 2. 添加远程仓库地址

```
$ git remote add [远程仓库地址]
```

以上命令的作用是在本地仓库添加远程仓库地址，后期可以把代码推送到这个地址。读者在操作时应该将“[远程仓库地址]”修改为自己的仓库地址。仓库地址可以通过图9-53中所示方式获取。

## 3. 添加所有代码到缓存区

```
$ git add .
```

将pythonbbs项目下的所有代码添加到缓存区。

## 4. 将代码提交到本地仓库

```
$ git commit -m "first commit"
```

将缓存区的代码添加到本地仓库中。

## 5. 将本地仓库代码推送到**Github**远程仓库

```
$ git push origin main
```

## 注意

git push origin main中的main代表分支，main分支一般用来表示项目最新的稳定版本。在Git中，main分支原先使用的名称是master，为了避免种族歧视问题，将master修改为main。

完成以上5个步骤，即可实现将本地仓库的代码推送到Github服务器上，以后我们在自己的服务器上，通过pull命令即可完成代码下载。现在机器有3种角色，分别是本地的开发机器、Git服务器以及运行网站的网站服务器，三者之间的角色关系和工作流程如图9-55所示。

我们在本地开发机器上开发的代码，经测试没有bug后推送到Git服务器上（现在用的是Github），然后在网站服务器上拉取代码，即可完成网站服务器代码的更新。

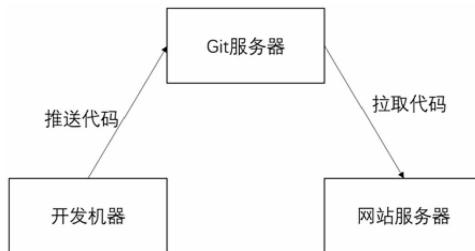


图9-55 3个机器的角色关系和工作流程

### 9.12.3 生产环境的配置

在Git中，分支是一个非常有用的功能。使用Git有一个基本的工作流程，就是尚未完成测试的代码，一般先放到dev或者development分支下。在测试没有bug后，再把代码合并到main分支下，并修改为生产环境下的配置信息，最后再把代码推送到网站服务器上，完成代码的更新。

按照Git的工作流程，在main分支下将pythonbbs/config.py中的ProductionConfig类根据实际情况修改配置信息，如数据库的域名和端口号等，并且设置DEBUG=False。然后再在pythonbbs/app.py中加载配置类，将之前的DevelopmentConfig修改为ProductionConfig，示例代码如下。

```
app.config.from_object(config.ProductionConfig)
```

完成以上操作后，在Git终端使用以下命令将代码推送到Git服务器。

```
$ git add .
$ git commit -m "production config"
$ git push origin main
```

## 9.12.4 安装常用软件

### 1. 安装OpenSSH

OpenSSH可实现远程控制，是一款能够在开发机上连接网站服务器的软件。OpenSSH是安装在网站服务器上的，通过以下命令即可完成安装。

```
$ sudo apt install openssh-server openssh-client
```

如果你的开发机是Windows系统，那么可以使用PuTTY（官网<https://www.putty.org>）或者Xshell（官网<https://www.xshell.com/zh/xshell/>）进行连接；如果是Mac系统，则可以直接在系统自带的终端软件下使用ssh命令进行连接。

### 2. 安装Vim

Vim是在Linux系统上使用的一款非常好用的文本编辑软件，几乎是Linux系统必备的软件，通过以下命令即可完成安装。

```
$ sudo apt install vim
```

### 3. 安装MySQL

这里我们是为了演示项目，把MySQL安装在了网站服务器上。在公司中如果有条件，建议购买单独的MySQL服务器，或者搭建独立的MySQL服务器，最大限度地保证数据的安全。在网站服务器上安装MySQL的命令如下。

```
$ sudo apt install mysql-server mysql-client  
$ sudo apt install libmysqld-dev
```

执行上述两条命令可以安装MySQL服务器和客户端。

## 4. 安装 Redis

我们的pythonbbs项目使用Redis，用来缓存邮箱验证码和Celery数据，通过以下命令可安装Redis。

```
$ sudo apt install redis
```

## 5. 安装 Python3

有的服务器包比较旧，可能没有Python3，可以通过以下命令安装。

```
# 安装Python3软件  
$ sudo apt install python3  
# 安装Python3下的pip工具  
$ sudo apt install python3-pip  
# 安装Python3依赖的头文件等的开发包  
$ sudo apt install python3-dev
```

## 6. 安装 Git

网站服务器需要通过Git工具从Git服务器上拉取代码，通过以下命令可以安装Git。

```
$ sudo apt install git
```

### 9.12.5 配置网站

在9.12.4节中的软件都安装完成后，我们就可以配置网站了，步骤如下。

#### 1. 创建新用户

默认的root账号权限过大，不建议使用root用户部署网站。接下来使用以下命令创建一个新的用户。

```
$ adduser zhiliao  
$ usermod -aG sudo zhiliao  
$ su zhiliao
```

上述命令首先创建了一个名叫zhiliao的用户，并且赋予了root权限，然后切换到了zhiliao用户。

## 2. 使用Git拉取代码

现在代码还是在Git服务器上，我们在/home/zhiliao下通过以下命令拉取代码到网站服务器。

```
$ cd /home/zhiliao  
$ mkdir pythonbbs  
$ git remote add origin https://github.com/NunchakusHuang/pythonbbs.git  
$ git pull origin main
```

在执行pull命令时，会提示输入Github网站的用户名和密码，输入之后，就可以将pythonbbs项目的代码拉取到网站服务器上了。

## 3. 安装Python依赖包

我们从开发机上导出了Python依赖包到requirements.txt中。在网站服务器上，可以通过以下命令完成依赖包的安装。

```
$ pip install -r requirements.txt
```

## 4. 创建数据库

登录MySQL后，使用以下命令创建名叫pythonbbs的数据库。

```
> create database pythonbbs charset utf8mb4;
```

## 5. 同步ORM模型到数据库中

我们在开发机上使用migrate生成的迁移脚本，可以直接在网站服务器上进行映射。通过以下命令即可完成。

```
$ flask db upgrade
```

以上命令会执行pythonbbs/migrations/versions下的所有迁移脚本，将ORM模型同步到数据库中生成表。

## 6. 创建初始数据

网站运营前应该把初始数据创建好，包括角色、权限、板块以及初始的管理员用户，相关命令格式如下。

创建角色。

```
flask create-role
```

创建权限。

```
flask create-permission
```

创建板块。

```
flask create-board
```

创建管理员。

```
flask create-admin --username 张三 --email xx@qq.com --password 111111
```

## 7. 运行Celery

pythonbbs项目的运行依赖于Celery，而Celery又依赖于Redis，因此在运行Celery之前，先运行Redis。命令如下。

```
$ sudo service redis start
```

### 9.12.6 使用Gunicorn部署网站

在学习使用Gunicorn（是Green Unicorn的简称）部署网站之前，先来厘清Web服务器、应用服务器和Web应用框架3个概念。

- Web服务器：**负责处理HTTP请求，并响应静态文件。常见的有Apache、Nginx以及微软的IIS等。
- 应用服务器：**负责处理逻辑的服务器。如Java、Python的代码是不能直接通过Nginx这种Web服务器来处理的，只能通过应用服务器来处理，常见的应用服务器有uWSGI、Gunicorn以及Tomcat等。
- Web应用框架：**使用某种编程语言、封装了常用的Web功能的框架就是Web应用框架。Flask、Django以及Java中的SSH（Structs+Spring+Hibernate）等都是Web应用框架。

WSGI（Web server gateway interface，Web服务网关接口）是Python中定义的Web服务器和Web应用框架之间的一种简单而通用的接口。我们在开发阶段使用werkzeug作为WSGI应用服务器，但是werkzeug仅用于开发，不能用于生产环境。在生产环境中应该选择性能强悍、稳定性高的WSGI应用服务器，如uWSGI和Gunicorn。因为Gunicorn性能强、配置简单，所以选择用Gunicorn作为pythonbbs项目的应用服务器。

首先通过以下命令安装Gunicorn。

```
$ sudo pip install gunicorn
```

接下来在项目的根路径下，创建gunicorn.conf.py文件，作为gunicorn的运行配置文件，代码如下。

```
$ import multiprocessing  
  
bind = "127.0.0.1:8000"  
workers = multiprocessing.cpu_count()*2 + 1  
accesslog = "/var/log/pythonbbs/access.log"  
daemon = True
```

上述代码中每个配置项的意义如下。

- bind: 监听的IP地址和端口号，语法为IP:PORT。如果想让其他机器能够访问，则设置成0.0.0.0或者留空，然后再让Gunicorn监听8000端口即可。后面会用Nginx做Web服务器，Nginx和Gunicorn之间要通过8000端口进行通信，所以IP设置为127.0.0.1。
- workers: workers的数量。Gunicorn官方推荐的配置是CPU数量×2+1。
- accesslog: 连接Gunicorn的日志文件。
- daemon: 是否作为守护进程执行，设置为True。

最后在pythonbbs项目的根路径下，执行以下命令启动pythonbbs项目。

```
$ gunicorn app:app
```

上述命令会自动在当前路径下寻找gunicorn.config.py文件，然后执行app.py模块下的app对象。至此，Gunicorn就完成了对pythonbbs项目的部署。

读者将bind的IP地址修改为0.0.0.0，执行gunicorn reload命令重新加载配置文件，然后就可以通过http://[机器的IP地址]:8000访问该网站了。

### 9.12.7 使用Nginx部署网站

虽然Gunicorn可以让网站正常运行，但这不是最佳选择。下面使用Gunicorn推荐的Nginx来作为Web服务器。Nginx、Gunicorn和pythonbbs三者之间的关系如图9-56所示。

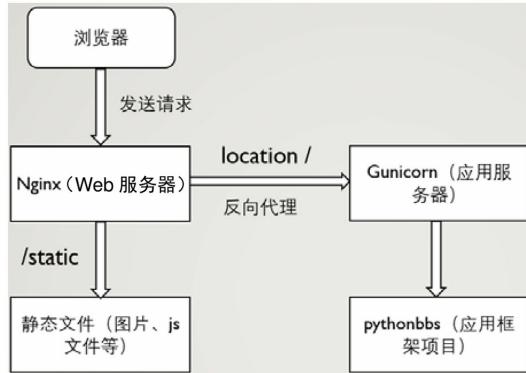


图9-56 Nginx、Gunicorn、pythonbbs三者关系图

通过图9-56可以发现，在浏览器向服务器发送请求后，先是由Nginx来处理，在遇到以/static开头的静态文件URL后，则直接返回静态文件；在遇到非静态路由后，则通过反向代理给Gunicorn服务器，Gunicorn再传递给pythonbbs项目进行逻辑处理。使用Nginx作为Web服务器有以下好处。

- Gunicorn对静态文件处理效率并不好，包括响应速度和缓存等，Nginx则不然。
- Nginx作为专业的Web服务器，可以对外界隐藏应用服务器，更加安全。
- Nginx运维起来更加方便。如设置负载均衡、配置IP黑名单等都非常方便，如果用Gunicorn实现会很麻烦。

要使用Nginx，首先需要通过以下命令安装Nginx。

```
$ sudo apt install nginx
```

Nginx的常用命令如下。

- 启动。

```
sudo service nginx start
```

- 关闭。

```
sudo service nginx stop
```

重启。

```
sudo service nginx restart
```

测试配置文件。

```
sudo service nginx configtest
```

启动Nginx后，在浏览器中访问http://[服务器IP]，即可看到如图9-57所示的界面。

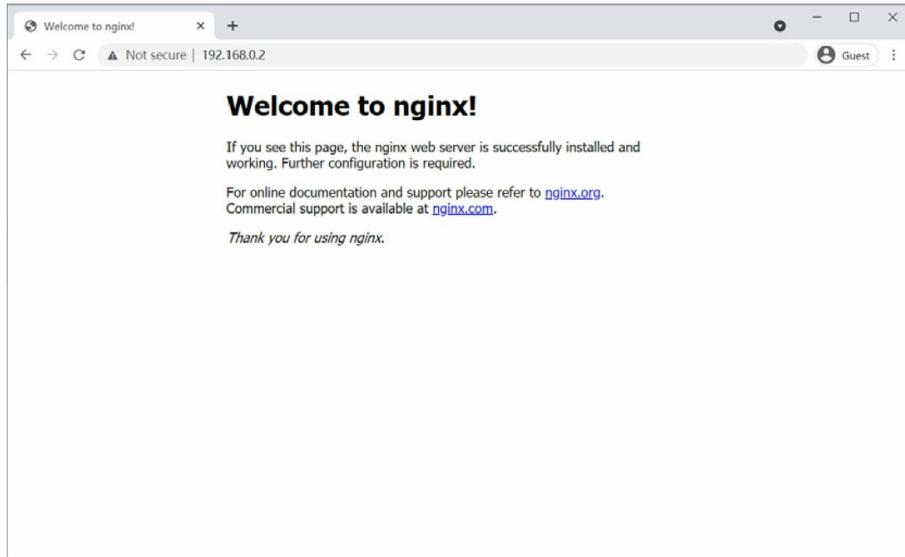


图9-57 Nginx默认界面

图9-57是Nginx的默认界面，看到此界面说明Nginx已经运行成功。接下来再添加Nginx配置文件，在/etc/nginx/conf.d目录下创建pythonbbs.conf文件，然后添加以下代码。

```

# 负载均衡
upstream pythonbbs{
    server 127.0.0.1:8000;
}

# 配置服务器
server {
    # 监听的端口号
    listen      80;
    # 域名
    server_name 192.168.0.2;
    charset     utf-8;

    access_log  /var/log/nginx/pythonbbs_access.log;
    error_log   /var/log/nginx/pythonbbs_error.log;

    # 最大的文件上传尺寸
    client_max_body_size 75M;

    # 静态文件访问的URL
    location /static {
        # 静态文件地址
        alias /home/zhiliao/pythonbbs/static;
        # 静态文件过期时间
        expires 60d;
    }

    location /media {
        # 文件上传地址
        alias /home/zhiliao/pythonbbs/media;
        expires 100d;
    }
    # 最后, 发送所有非静态文件请求到Gunicorn
    location / {
        proxy_pass http://pythonbbs;
        # uwsgi_params文件地址
        include      /etc/nginx/uwsgi_params;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}

```

上述配置代码中, `upstream`是用来设置负载均衡的, 里面可以设置多个server, Nginx会根据策略选择不同的后端服务器来实现负载均衡。下面的`server`部分用来配置Nginx服务器, 这里监听80端口, 并且指定服务器名称为192.168.0.2, 以后通过这个服务器名称即可访问到Nginx服务器。读者在这里可以修改为自己真实服务器的IP地址, 或者如果有域名, 并且做

好了域名和服务器的映射，则可以修改为域名。`access_log`和`error_log`是配置客户端连接Nginx服务器的日志。`location/static`用于指定以`/static`开头的静态文件的处理逻辑，使用`alias`指定静态文件寻找的路径，并且过期时间为60天。另外，我们还设置了`/media`，用来获取用户上传后的图片。最后是非静态文件和非`media`文件，通过反向代理传给Gunicorn服务器，下面设置的`proxy_set_header`是为了能准确获取客户端的信息。

现在通过80端口和8000端口都可以访问到网站，原因是Nginx监听80端口，Gunicorn监听8000端口。我们的预想是客户端仅通过80端口访问，8000端口用于Nginx和Gunicorn通信。可以通过防火墙关闭8000端口来实现这个需求，执行命令如下。

```
$ sudo ufw enable  
$ sudo ufw allow 80  
$ sudo ufw deny 8000  
$ sudo ufw allow 22
```

上述命令的作用分别是开启防火墙、打开80端口、关闭8000端口，打开22端口并且为了能在开发机上通过远程连接服务器，我们开启了22端口，也就是SSH服务。

### 9.12.8 压力测试

项目部署完成后，我们来进行压力测试，看看Nginx+Gunicorn部署的网站效率如何。这里要使用的是`apache2-utils`中的`ab`命令，首先通过以下命令安装`apache2-utils`。

```
$ sudo apt install apache2-utils
```

接着执行以下命令。

```
$ ab -n 50000 -c 1000 http://127.0.0.1/
```

`ab`是压力测试命令，其中参数意义如下。

- `-n`: 总共发起请求的数量，这里总共发起50000个请求。
- `-c`: 并发的数量，这里设置并发数为1000个请求。
- `http://127.0.0.1/`: 压力测试的URL。

执行完上述命令后，可以看到如下输出。

```
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 127.0.0.1 (be patient)
Completed 5000 requests                      # 请求完成的数量
Completed 10000 requests
Completed 15000 requests
```

```
Completed 20000 requests
Completed 25000 requests
Completed 30000 requests
Completed 35000 requests
Completed 40000 requests
Completed 45000 requests
Completed 50000 requests
Finished 50000 requests

Server Software:      nginx/1.18.0      # 服务器软件名称
Server Hostname:     127.0.0.1        # 服务器主机名
Server Port:         80                # 服务器端口号

Document Path:       /                  # 请求的 URL PATH
Document Length:    612 bytes        # HTTP 响应数据的正文长度

Concurrency Level:   1000              # 并发量
Time taken for tests: 3.762 seconds  # 总共耗费的时间
Complete requests:  50000              # 完成的请求数
Failed requests:    512               # 失败的数量
                           (Connect: 0, Receive: 0, Length: 256, Exceptions: 256)
Total transferred:   42481376 bytes  # 所有请求响应数据的长度
HTML transferred:   30443328 bytes
Requests per second: 13291.68 [#/sec] (mean)
Time per request:   75.235 [ms] (mean)
Time per request:   0.075 [ms] (mean, across all concurrent requests)
Transfer rate:      11028.30 [Kbytes/sec] received

Connection Times (ms)
                  Min  mean   [+/-sd] median   max
Connect:          0    36    103.1    29    1061
Processing:       11   37    10.6     38     71
Waiting:          0    29     9.6     29     63
Total:            32   74    104.3    70    1107

Percentage of the requests served within a certain time (ms)
 50%    70
 66%    73
 75%    75
 80%    76
 90%    79
 95%    83
 98%    87
 99%   1054
100%   1107 (longest request)
```

以上每项指标代表的意义如下。

- Completed xxx requests: 请求完成的数量。
- Server Software: 服务器软件名称。
- Server Hostname: 服务器主机名。
- Server Port: 服务器端口号。
- Document Path: 请求的URL PATH。
- Document Length: HTTP响应数据的正文长度。
- Concurrency Level: 并发量。
- Time taken for tests: 总共耗费的时间。
- Complete requests: 完成的请求数。
- Failed requests: 失败的请求数。
- Total transferred: 所有请求响应数据的长度。
- HTML transferred: 所有请求响应数据中HTML内容的长度。
- Requests per second: 每秒完成的请求数，也叫RPS（吞吐率）。
- Time per request(mean): 一个请求用户平均等待的时间。
- Time per request(mean,across all concurrent requests): 服务器完成一个请求平均的时间。
- Transfer rate: 网络传输速率。
- Connection Times (ms): 从请求到响应的整个过程消耗的时间。从请求到响应包含 Connect（网络连接）、Processing（程序处理）、Waiting（等待）3个过程。其中 min 表示最小值、mean 表示平均值、[+/-sd] 表示标准差（反应数据的离散程度，数值越大则数据越离散，也就是某个过程越不稳定）、median 表示中位数、max 表示最大值。
- Percentage of the requests served within a certain time (ms): 在50000个请求中的不同完成率阶段进行采样，请求所消耗的时间。如50%，也就是在第25000个请求时，花费了70ms的时间。在第50000个请求消耗了1107ms的时间。

以上压力测试的硬件环境配置为2GB内存+单核CPU 3600Hz+机械硬盘。其中QPS (queries per second, 每秒查询率) 达到了13000以上，说明网站运行的效率还是不错的。

---

(1)文中的“账号”同图中的“帐号”为同一内容，后文不再赘述。

## chapter 10 第10章 WebSocket实战

在实际开发中，经常会有许多场景需要让服务器主动发送消息给客户端，如视频弹幕、在线客服、协同编辑等。如果用传统的HTTP协议来实现，则需要每隔一段时间主动请求服务器才能获取到最新数据，使用这种方式的好处是开发简单、不会长时间占用服务器资源，缺点是效率低下、数据获取不及时。为了解决在Web开发中客户端和服务端双向即时通信的问题，IETF（Internet Engineering Task Force，互联网工程任务组）标准化了WebSocket协议，使用WebSocket协议相关的API，在客户端和服务端之间只需要完成一次握手，即可进行双向即时通信，使得视频弹幕、在线客服等类似场景变得易于实现。

在Flask中使用WebSocket有多种解决方案，最常用的两个框架为Flask-Sockets和Flask-SocketIO。其中Flask-Sockets仅对WebSocket协议进行包装，因此只适合支持WebSocket协议的浏览器。Flask-SocketIO则是基于JavaScript SocketIO开源库的消息协议的Flask实现，而SocketIO库对那些不支持WebSocket协议的浏览器也能实现同样的效果，因此使用Flask-SocketIO对客户端来说兼容性更好。

本章以实现一个在线即时聊天网站为例，让读者熟练掌握Flask-SocketIO的使用。在学完本章内容后，读者可以发散思维，将本章的知识点应用到其他业务场景，如在线客服、消息推送等。我们先来了解本项目要实现的功能，如表10-1所示。

表10-1 在线即时聊天网站功能表

功 能	相关技术点
登录	(1) session 的使用 (2) 权限验证
聊天页面	(1) WebSocket 连接 (2) 广播所有在线用户
单聊	(1) 获取对方信息 (2) 客户端发送单聊信息 (3) 服务端发送单聊信息
群聊	(1) 获取群聊信息 (2) 加入群聊 (3) 客户端发送群聊信息 (4) 服务端发送群聊信息

### 10.1 安装相应的包

在服务端，项目使用的是Flask-SocketIO库，输入以下命令即可完成安装。

```
$ pip install flask-socketio
```

在客户端，项目需要使用socketio库，在HTML文件中使用以下代码加载socket.io.min.js文件即可。

```
<script src="https://cdn.bootcdn.net/ajax/libs/socket.io/4.1.3/socket.io.min.js"></script>
```

## 10.2 创建SocketIO对象

首先使用PyCharm Professional版创建一个Flask项目，命名为im\_demo，然后在app.py文件中添加以下代码。

```
from flask import Flask, render_template, request, redirect, session
from flask_socketio import SocketIO

app = Flask(__name__)
app.config['SECRET_KEY'] = "fasdfsdfasdkj"
socketio = SocketIO(app)

@app.route('/')
def hello_world():
    return "Hello World!"

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

上述代码中，除了使用PyCharm Professional版自动生成的Flask项目源代码，我们还从flask\_socketio中导入了SocketIO类，并且把app作为参数，创建了一个SocketIO类的对象socketio。最后在运行时，将app.py的运行方式从之前的app.run改成了socketio.run。socketio.run会自动根据以下环境选择底层运行服务器。

(1) 如果开启了Debug模式，将使用werkzeug开发服务器。

- (2) 在生产环境中，如果安装了eventlet，会优先使用eventlet作为服务器。
- (3) 判断是否安装了gevent，如果安装了，会使用gevent作为服务器。gevent默认不带有WebSocket功能，如果使用gevent，则还需要安装gevent-websocket。
- (4) 如果eventlet和gevent都没有安装，则会使用werkzeug开发服务器运行项目。

## 10.3 实现登录

虽然app.py运行方式从之前的app.run改成了socketio.run，但是不影响Flask传统的开发模式，如渲染模板、处理cookie和session等。socketio.run在保留了Flask传统功能以外，还增加了WebSocket支持。因此，登录功能我们依然可以使用HTTP协议的方式。在app.py文件中添加以下代码。

```
class ResultCode:  
    OK = 200  
    ERROR_PARAMS = 400  
    ERROR_SERVER = 500  
  
def result(code=ResultCode.OK, data=None, message=""):  
    return {"code": code, "data": data or {}, "message": message}  
  
@app.route("/login", methods=['GET', 'POST'])  
def login():  
    if request.method == 'GET':  
        return render_template("login.html")  
    else:  
        username = request.form.get('username')  
        if not username:  
            return result(ResultCode.ERROR_PARAMS, message="请输入用户名")  
        elif UserManager.has_user(username):  
            return result(ResultCode.ERROR_PARAMS, message="此用户名已存在")  
        session['username'] = username  
    return result()
```

上述代码中，我们实现了登录URL与视图。如果用GET请求，则返回登录模板；如果用POST请求，则执行登录操作。这里我们简化了登录过程，用户只要输入用户名即可。如果条件都符合，则把用户名存储在session中。为了让代码更加规范，我们添加了返回状态码的ResultCode类，以及返回JSON格式的result函数。并且我们用了UserManager类来管理当前登录的用户，UserManager类的代码如下。

```
class UserManager:
    # 单例实例对象
    __instance = None
    # 所有用户，里面存储的是字典类型，字典中分别为 sid 和 username
    # 如{"sid": "assdfsgsd", "username":"张三"}
    _users = []

    # 设置单例设计模式
    def __new__(cls, *args, **kwargs):
        if not cls.__instance:
            cls.__instance = super(UserManager, cls).__new__(cls)
        return cls.__instance

    # 添加用户
    @classmethod
    def add_user(cls, username, sid):
        for user in cls._users:
            if user['sid'] == sid or user['username'] == username:
                return False
        cls._users.append({"sid": sid, "username": username})

    # 移除用户
    @classmethod
    def remove_user(cls, username):
        for user in cls._users:
            if user['username'] == username:
                cls._users.remove(user)
                return True
        return False

    # 根据 key 获取用户，key 可以为 sid 或 username
    @classmethod
    def get_user(cls, key):
        for user in cls._users:
```

```

        if user['sid'] == key or user['username'] == key:
            return user
        return None

    # 根据 key 判断是否有某个用户, key 可以为 sid 或 username
    @classmethod
    def has_user(cls, key):
        if cls.get_user(key):
            return True
        else:
            return False

    # 获取当前用户
    @classmethod
    def get_current_user(cls):
        username = session.get("username")
        for user in cls._users:
            if user['username'] == username:
                return user
        return None

    # 获取所有用户的用户名
    @classmethod
    def all_username(cls):
        return [user['username'] for user in cls._users]

```

上述代码中，定义了两个类属性，分别为`_instance`和`_users`，其中`_instance`是用于保存单例对象的，`_users`用来保存所有用户信息。通过重写`__new__`方法使得`UserManager`仅能被创建一次，也就是单例对象。接着定义了一系列的类方法，分别为添加用户（`add_user`）、移除用户（`remove_user`）、获取用户（`get_user`）、判断用户是否存在（`has_user`）、获取当前用户（`get_current_user`）以及获取所有用户的用户名（`all_username`）。这些类方法可使后续处理即时消息变得更加简单。

此时在浏览器中访问`http://127.0.0.1:5000/login`，可以看到如图10-1所示的效果。

在后续的请求中，为了保证用户必须在登录后才能进行访问，我们实现一个`login_required`装饰器，代码如下。

```

def login_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not session.get("username"):
            return redirect("/login")
        else:

```

```
    return func(*args, **kwargs)
return wrapper
```

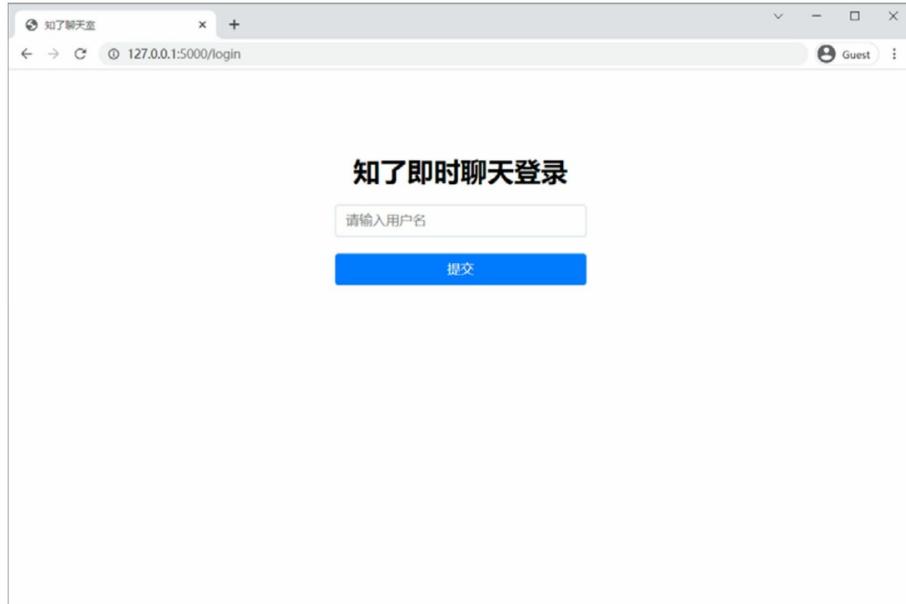


图10-1 在线即时聊天项目登录界面

上述装饰器代码中，我们判断session中是否包含username，如果没有，则重定向到登录页面，否则就正常执行被装饰的函数。

## 10.4 连接和取消连接

在进入首页后，我们要做的第一件事就是连接上WebSocket。Flask-SocketIO是通过事件的形式进行通信的，Flask-SocketIO默认内置了许多事件，其中最常用的就是connect和disconnect。我们先在服务端实现connect事件，代码如下。

```
from flask_socketio import SocketIO, emit
@socketio.on('connect')
@login_required
def connect():
    print("连接成功")
```

```
UserManager.add_user(session.get("username"), request.sid)
emit("users", {"users": UserManager.all_username()}, broadcast=True)
```

上述代码中，我们通过`@socketio.on`来设置事件发生后的执行函数，这里定义的是`connect`事件。客户端在连接`connect`事件后，先把当前用户存储在`UserManager`中，在存储用户信息时，把用户名以及当前用户的Session ID也就是`request.sid`都存放进去，然后再通过`emit`函数发送一个`users`事件，有新用户登录后，我们要通知所有已连接用户，所以设置`broadcast=True`，也就是采用广播的方式进行发送。接下来，在客户端中使用JavaScript SocketIO库发送`connect`事件，代码如下。

```
const socket = io();
socket.on("connect", function(){
    console.log("连接成功");
});
```

在浏览器中访问`http://127.0.0.1:5000 /`后，如果控制台打印了“连接成功”，则说明客户端和服务端已经完成握手，实现了长连接，后续可以通过其他事件来实现客户端和服务端的双向通信。

在浏览器被关闭后，应该主动断开连接。服务端在收到断开连接事件后，应把当前用户从`UserManager`中移除，并广播给其他用户，代码如下。

```
@socketio.on("disconnect")
@login_required
def disconnect():
    UserManager.remove_user(session.get('username'))
    emit("users", {"users": UserManager.all_username()}, broadcast=True)
```

接下来，在浏览器端监听页面的关闭事件，在关闭之前先发送`disconnect`事件，代码如下。

```
$(window).bind("beforeunload", function () {
    socket.on("disconnect");
});
```

上述代码中，我们使用了jQuery的事件绑定方式，对`beforeunload`事件进行了绑定，在发

生此事件后，主动向服务器发送disconnect事件。

## 10.5 获取在线用户

在客户端，我们要先获取当前有哪些用户在线，才方便选择相应用户进行单聊。在每个客户端成功连接后，服务端都会广播users事件，并且把当前登录的所有用户名都发送过去。因此在客户端想要获取在线用户，只需监听users事件即可，代码如下。

```
socket.on("users", function (result){
    let users = result.users;
    for(let index=0; index<users.length; index++){
        let username = users[index];
        console.log(username);
    }
});
```

users事件虽然不是socketio内置的，但是没有关系，服务端通过emit发送的任何事件，客户端都可以通过socket.on进行监听，无论事件是不是内置的。

## 10.6 实现单聊

socketio中并没有提供单聊的事件，因此需要自定义，这里定义personal事件。首先在服务端实现如下代码。

```
@socketio.on("personal")
def send_personal(data):
    to_username = data.get('to_user')
    message = data.get('message')
    if not to_username or not UserManager.has_user(to_username):
        return result(ResultCode.ERROR_PARAMS, message="请输入正确的目标用户")
    to_user = UserManager.get_user(to_username)
    emit("personal", {"message": message, "from_user": session.get
    ("username")}, room=[to_user.get("sid")])
```

上述代码中，通过@socketio.on("personal")自定义了personal事件及send\_personal处理函数，并且为了能接收客户端传过来的参数，我们还定义了一个data参数。在send\_personal处理函数中，先是获取此消息要发送的目标用户以及消息内容，在条件都满足后，通过emit函数发送personal事件到room参数指定的客户端，并且携带了消息内容以及消息发送者。客户

端则通过监听和发送personal事件来实现与服务端的交互，客户端监听personal事件的代码如下。

```
// 监听personal事件
socket.on("personal", function (data){
    let message = data.message;
    let from_user = data.from_user;
    console.log("message:"+message+",from_user:"+from_user);
});
```

客户端主动向服务端发送personal事件的代码如下。

```
// 客户端向服务端发送personal事件
socket.emit("personal", {"to_user": to_user, "message": message},
function () {
    console.log("消息发送成功");
});
```

上述监听和发送personal事件代码中，我们实现了回调函数，在personal消息被成功接收和发送后，都会执行对应的回调函数。在回调函数中，我们也可以执行一些其他操作，如添加消息体到对话框中。

## 10.7 实现群聊

在Flask-SocketIO中对群聊功能做了非常好的封装，如加入群聊有join\_room函数、离开群聊有leave\_room函数。服务端加入群聊和离开群聊的代码如下。

```
# 加入群聊
@socketio.on("join")
@login_required
def join(data):
    room = data.get("room")
    join_room(room)
    username = session.get("username")
    send(username+"加入群聊", to=room)

# 离开群聊
@socketio.on("leave")
@login_required
def leave(data):
```

```
room = data.get("room")
leave_room(room)
username = session.get("username")
send(username+"离开群聊", to=room)
```

客户端在某些情况下（如单击加入群聊），通过使用emit函数发送join事件，即可加入房间，代码如下。

```
socket.emit("join", {"room": "Flask交流群"});
```

通过以上代码就加入“Flask交流群”房间了。也可以在某些情况下发送leave事件，即可离开房间，代码如下。

```
socket.emit("leave", {"room": "Flask交流群"});
```

房间名称如果之前不存在，无须执行创建操作，Flask-SocketIO会自动创建并让此用户加入房间。加入房间后，就可以在房间里聊天了。我们先在服务端实现群聊功能，代码如下。

```
@socketio.on("room_chat")
@login_required
def room_chat(data):
    room = data.get("room")
    message = data.get("message")
    from_user = UserManager.get_current_user().get("username")
    send({"message": message, "from_user": from_user, "room": room},
          to=room)
```

上述代码中，先从data中获取房间名称和消息内容，然后使用flask\_socketio.send函数将消息内容和消息发送者发送到to参数指定的房间中。这样所有加入了此房间的用户都可以收到此消息，客户端监听房间消息的代码如下。

```
socket.on("room_chat", function (result) {
  let from_user = result.from_user;
  let message = result.message;
  let room = result.room;
```

```
        console.log("房间号: "+room+"收到来自"+from_user+"的消息: "+message);
    });
}
```

在其他客户端发送了room\_chat事件后，就能通过回调函数获取到消息的相关信息了。当然，客户端在某些情况下想要发送房间消息，直接调用emit函数即可，代码如下。

```
socket.emit("room_chat", {"room": "Flask交流群", "message": "大家好"});
```

## 10.8 部署项目

我们依然可以使用gunicorn来部署Flask-SocketIO项目，另外还需要使用eventlet或者gevent来运行Flask-SocketIO项目。因为eventlet的效率比gevent高，并且eventlet也是官方推荐的方式，安装eventlet的命令如下。

```
pip install eventlet
```

下面创建一个gunicorn.conf.py文件来配置gunicorn的运行参数，这里简化一下，直接使用以下命令部署项目。

```
gunicorn --worker-class eventlet app:app
```

上述命令中，使用--worker-class来指定工作进程由eventlet来运行即可。

与第9章中论坛项目一样，我们选择Nginx来作为项目的Web服务器。Nginx的配置与论坛项目一样，不同的是，Nginx需要增加JavaScript SocketIO默认的/socket.io转发。Nginx的配置如下。

```
server {
    listen 80;
    server_name _;

    location / {
        include proxy_params;
        proxy_pass http://127.0.0.1:5000;
    }
}
```

```
location /static {
    alias <path-to-your-application>/static;
    expires 30d;
}

location /socket.io {
    include proxy_params;
    proxy_http_version 1.1;
    proxy_buffering off;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
    proxy_pass http://127.0.0.1:5000/socket.io;
}
}
```

上述配置中，除了location/和location/static外，还增加了location/socket.io，这个URL是使用SocketIO创建对象时默认使用的URL。当然也可以修改为自定义的URL，如以下配置。

```
const socket = io({path: "/my-socketio-path"})
```

在监听到location/socket.io后，会把请求代理给Flask-SocketIO项目的location/socket.io路径，这样即可完成Nginx配置。

注意，JavaScript SocketIO的官方文档地址为<https://socketio.bootcss.com/docs/>。Flask-SocketIO的官方文档地址为<https://flask-socketio.readthedocs.io/en/latest/intro.html>。

## chapter 11 第11章 Flask异步编程

自从Python 3.6官方将异步编程标准库[asyncio](#)转正后，异步编程开始大放异彩。[aio-libs](#)组织（<https://github.com/aio-libs>）发布了一系列的异步编程库，如网络请求的[aiohttp](#)库（<https://docs.aiohttp.org/>）、连接Redis的[aioredis](#)-py库等。Flask在2.0版本宣布支持异步编程，同时配套的SQLAlchemy在1.4版本也增加了异步API，Jinja2在2.9版本开始支持异步模式，其他框架如Django在3.0版本也加入了异步支持。随着技术不断地更新，相信异步编程在未来会有更大的发展。

### 11.1 **asyncio**标准库

Flask异步编程是基于Python内置的[asyncio](#)模块的，因此我们只有学懂了[asyncio](#)的相关概念和基本用法后，才能更好地理解Flask中的异步编程。

下面围绕事件循环、协程、Task对象、Future对象等概念讲解[asyncio](#)的工作机制。

#### 注意

Python 3.8版本的[asyncio](#)库在Windows系统上有一个bug，如果在运行时出现 ValueError: set\_wakeup\_fd only works in main thread错误，升级到Python 3.9版本即可解决。

## 1. 事件循环

在非阻塞代码执行过程中，为了在事件发生时能准确地执行回调，需要不断轮询是否有事件发生。这个轮询等待事件的过程，被称为事件循环。在[asyncio](#)库中，一般情况下不需要自己手动创建事件循环，通过[asyncio.run](#)运行协程便会自动创建事件循环。

## 2. 协程

从Python 3.5版本添加[async](#)和[await](#)关键字以来，只要以[async def](#)开头定义的函数，都叫作协程。协程在执行中可以被挂起和恢复，因此可以用于构建异步程序。驱动协程执行必须使用[await](#)关键字，如以下代码。

```
import asyncio

async def nested():
    await asyncio.sleep(1)
    return 42

async def main():
    # 仅创建了一个协程对象，但是并没有执行
    nested()

    # 使用await关键字，驱动并等待协程对象执行
    result = await nested()
    print(result)

    # 创建一个事件循环并执行main协程
    asyncio.run(main())
```

在上述代码中，`nested()`函数仅是创建一个协程，`await nested()`函数才是驱动并等待协程的执行。最外层的`main`协程不能直接执行，需要通过`asyncio.run`方法执行，该方法会创建一个事件循环并执行协程。

### 3. Task（任务）对象

Task是用来调度协程挂起和恢复的。协程本身只能通过同步的方式执行，如果要并行执行多个协程，则必须将协程封装为Task对象。在`asyncio`库中，可以通过`asyncio.gather`方法将多个协程并行执行，`asyncio.gather`方法内部会自动将协程封装为Task，无须手动封装。并行执行任务的示例代码如下。

```
async def my_worker(index):
    await asyncio.sleep(1)
    print("worker %d"%index)

async def main():
    tasks = []
    for x in range(1,6):
        tasks.append(my_worker(x))

    await asyncio.gather(*tasks)
```

上述代码中，在`main`协程中创建了5个`my_worker`协程对象并添加到列表中，然后统一传给`asyncio.gather`方法并行执行。上述`main`协程只需1s即可完成5个`my_worker`协程的执行。

## 4. Future对象

Future对象有如下两个作用。

- (1) 用来保存协程执行后的结果。
- (2) 用来和Task对象配合，调度执行协程。

Future在asyncio库中相对来讲是比较低层级的对象，通常没有必要在应用层级代码中创建Future对象。

## 11.2 aiohttp库

aio-libs组织发布了一系列的异步库，其中就包含异步网络库aiohttp，aiohttp库有以下特点。

- (1) aiohttp既是HTTP客户端，也是HTTP服务端。
- (2) aiohttp同时支持服务端的WebSocket和客户端的WebSocket。
- (3) aiohttp作为服务端来说，有中间件、信号和可插拔的路由。

因为aiohttp库简单易用、功能强大，已经成为发送异步网络请求的首选框架。通过以下命令即可安装aiohttp库。

```
$ pip install aiohttp[speedups]
```

上述命令除了安装aiohttp库，还会安装cchardet和aiodns库，cchardet库是为了替换默认的chardet库，从而达到更高的效率。aiodns库可以提高DNS域名的解析速度。我们首先来看aiohttp的使用例子，代码如下。

```
import aiohttp
import asyncio

async def main():
    async with aiohttp.ClientSession() as session:
        async with session.get('https://python.org') as response:
            print("状态码:", response.status)
            html = await response.text()
            print("响应体: ", html)
```

```
if __name__ == '__main__':
    asyncio.run(main())
```

上述代码中，首先使用`async with`创建了一个会话对象`session`，其中`async with`是异步上下文，与普通的`with`语句用法类似。然后使用`session.get`方法获取数据，并将响应赋值给`response`对象。因为状态码可以直接获取，所以不需要执行`await`，而响应体则需要经过读取和解码，因此使用了`await`等待协程的方式获取。这样就使用`aiohttp`库完成了一个基本的网络请求功能。

## 11.3 异步版Flask安装与异步编程性能

Flask在2.0版本实现了异步功能，接下来讲解异步版Flask的安装、Flask异步编程性能，以及异步编程在Flask项目中的实战应用场景。

### 11.3.1 安装异步版Flask

要在Flask中执行异步编程，首先必须使用Python 3.7以上的版本，其次要在安装Flask时选择异步拓展。通过以下命令即可安装异步版Flask。

```
$ pip install flask[async]
```

以上命令除了安装异步版Flask，还会安装`asgiref`，从而可以方便地将同步代码和异步代码相互转换。安装了`asgiref`后，在Flask中，视图函数、钩子函数（如`error_handlers`、`before_request`等）都可以定义成异步的，示例代码如下。

```
@app.route("/get-data")
async def get_data():
    data = await async_db_query(...)
    return jsonify(data)
```

### 11.3.2 Flask异步编程性能

Flask是一个符合WSGI接口的框架，WSGI本身是同步的。Flask中的异步运行过程如图11-1所示。

可以看到，在请求到达WSGI服务器后，WSGI服务器会启动一个新的线程，然后在线程中创建一个事件循环来执行异步视图函数，每来一个请求就创建一个新线程，因此异步与同步的并发量其实是一致的。异步代码不一定比同步代码执行效率更高，只有并发执行I/O操作才能体现并发的优势，当满足以下两个条件的I/O操作时可以考虑使用异步代码。

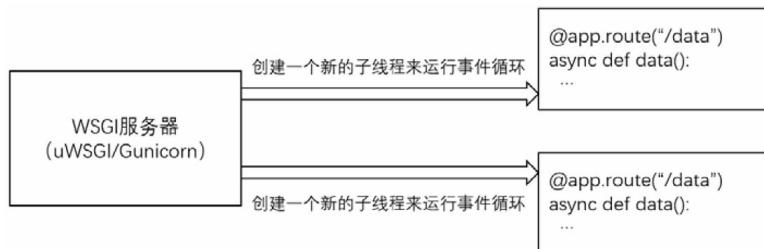


图11-1 Flask异步运行图

- (1) 有一些I/O操作。
- (2) 每个I/O操作不需要花费太长时间。

例如以下情形。

- (1) 向站外发送一些HTTP请求。
- (2) 和数据库有一些交互操作。
- (3) 有一些文件操作。

对于一些长时间任务和CPU密集型任务，使用异步是不合适的，如以下情形。

- (1) 运行机器学习模型。
- (2) 处理图片、转码视频或者生成PDF文件等。
- (3) 执行备份。

以上这类长时间或者CPU密集型任务，建议使用Celery异步框架。

### 11.3.3 实战——异步发送HTTP请求

发送一些HTTP请求是异步最适合的场景之一。按照同步的方式，必须要等第一个请求响应后才能执行第二个请求，以此类推，如果每个请求响应时间是1s，有10个请求则必须要10s才能完成。我们使用requests库发送同步请求，首先通过pip install requests命令安装requests库。然后发送5个网络请求，并计算时间，代码如下。

```
@app.route("/website-sync")
def website_sync():
    start_time = time.time()
    urls = [
        "https://www.python.org/",
        "https://www.php.net/",
        "https://www.java.com/",
        "https://dotnet.microsoft.com/",
        "https://www.javascript.com/"
    ]
    sites = []
    for url in urls:
        response = requests.get(url)
        sites.append({'url': response.url, 'status': response.status_code})

    response = '<h1>URLs: </h1>'
    for site in sites:
        response += f"<p>URL: {site['url']}, Status Code: {site['status']}</p>"

    end_time = time.time()
    print("time: {:.2f}{}".format(end_time - start_time))
    return response
```

上述代码向5个不同的URL发送请求，执行所消耗的时间为10.28s（不同速率的网络和配置的计算机会有所区别）。

异步则不同，它可以发送完一个请求后，不需要等待即可发送第二个请求，以此类推，如果每个响应时间是1s，则10个请求可以在1s多一点的时间即可完成。使用异步完成以上5个请求的示例代码如下。

```
async def fetch_url(session,url):
    response = await session.get(url)
    return {'url': response.url, 'status': response.status}

@app.route("/website/async")
async def website_async():
    start_time = time.time()
    urls = [
        "https://www.python.org/",
        "https://www.php.net/",
        "https://www.java.com/",
        "https://dotnet.microsoft.com/",
        "https://www.javascript.com/"
    ]
    async with aiohttp.ClientSession() as session:
```

```
tasks = []
for url in urls:
    tasks.append(fetch_url(session, url))
sites = await asyncio.gather(*tasks)

response = '<h1>URLs: </h1>'
for site in sites:
    response += f"<p>URL: {site['url']}, Status Code: {site['status']} </p>"

end_time = time.time()
print("time: %.2f" % (end_time - start_time))
return response
```

上述代码中，使用了`asyncio`将5个发送HTTP请求的协程并行运行。执行上述代码所消耗的时间为2.57s（不同速率的网络和配置的计算机会有所区别），执行效率是同步的4倍左右。如果上述代码不使用`asyncio.gather`并行运行协程，则执行过程依然为同步。

#### 11.3.4 使用异步SQLAlchemy

SQLAlchemy在1.4版本就添加了异步API，可以方便地被集成到Flask和其他Web框架中，如FastAPI。在Flask中使用异步SQLAlchemy不能选择Flask-SQLAlchemy，Flask-SQLAlchemy没有提供用来替换`AsyncEngine`和`AsyncSession`的接口，因此需要手动创建连接对象。这里为了简单，我们使用`sqlite`作为数据库。首先安装异步连接`sqlite`数据库的驱动包，命令如下。

```
$ pip install aiosqlite
```

#### 注意

如果读者想要异步操作MySQL数据库，则需要安装`pymysql`和`aiomysql`两个驱动程序，并在数据库连接URL上将`mysql+pymysql`修改为`mysql+aiomysql`。

使用SQLAlchemy创建异步连接`sqlite`对象的代码如下。

```
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import declarative_base, sessionmaker
```

```
from sqlalchemy import Column, Integer, String, Float

DATABASE_URL = "sqlite+aiosqlite:///./book.db"
engine = create_async_engine(DATABASE_URL, echo=True)
async_session = sessionmaker(bind=engine, expire_on_commit=False,
                             class_=AsyncSession)

Base = declarative_base()
```

上述代码中，首先创建了连接数据库配置的URL，然后使用`create_async_engine`方法创建了一个异步引擎，接着使用`sessionmaker`方法创建了一个异步`session`对象，并且指定创建的父类是`AsyncSession`。最后使用`declarative_base`方法创建了一个ORM模型的基类`Base`，这样以后所有ORM模型都需要继承自`Base`。下面创建一个`Book`模型，代码如下。

```
class Book(Base):
    __tablename__ = "books"
    id = Column(Integer, primary_key=True)
    name = Column(String(200), nullable=False)
    author = Column(String(200), nullable=False)
    price = Column(Float, default=0)
```

在项目第一次请求之前，我们把ORM模型创建到数据库中。添加以下钩子函数。

```
@app.before_first_request
async def before_first_request():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)
```

上述代码中，首先将`before_first_request`钩子函数定义成了协程，然后使用`async with`异步上下文创建了一个事务，以同步的方式将原来的表删除，并创建新的表。这里之所以用同步的方式，是因为`Base.metadata.drop_all`和`Base.metadata.create_all`两个方法都没有被定义成异步。

## 注意

如果想要使用类似flask-migrate的方式同步ORM模型到数据库中，可以使用alembic来

实现，官网为[https://alembic.sqlalchemy.org/en/latest/。](https://alembic.sqlalchemy.org/en/latest/)

接着添加一个创建图书的异步视图，代码如下。

```
@app.post('/books/add')
async def add_books():
    name = request.form.get('name')
    author = request.form.get("author")
    price = request.form.get('price')
    async with async_session() as session:
        async with session.begin():
            book = Book(name=name, author=author, price=price)
            session.add(book)
            await session.flush()
    return "success"
```

上述代码中，首先使用异步上下文创建了一个session对象，然后使用`session.begin()`创建了一个事务，又在事务中添加图书到数据库中。本例中没有涉及并行I/O的操作，与同步的方式效率相同。这也从侧面说明了，同步编程方式仍然是大部分场景的首选方案。

如果读者需要了解更多有关异步SQLAlchemy的知识，请参考其官方文档，网址为<https://docs.sqlalchemy.org/en/14/orm/extensions/asyncio.html>。

### 11.3.5 Jinja2开启异步支持

Jinja2从2.9版本开始增加了异步支持，在创建Environment时，通过设置`enable_async`参数即可开启异步模式。但是Flask对象默认已经创建了Environment对象，我们只需要通过设置`app.jinja_env.is_async=True`即可开启异步模式。Jinja2开启异步模式后，就可以在模板中使用协程。下面通过`app.jinja_env.globals`添加协程，代码如下。

```
app = Flask(__name__)
app.jinja_env.is_async = True

async def get_all_books():
    async with async_session() as session:
        stmt = select(Book)
        result = await session.execute(stmt)
        books = result.scalars().all()
    return books

app.jinja_env.globals["books"] = get_all_books
```

上述代码中，我们首先通过app.jinja\_env.is\_async开启了异步模式，然后定义了一个get\_all\_books的协程，取名为books，再将这个协程添加到了模板全局环境中。这样在模板中，就可以直接执行books协程了。

在templates下创建index.html模板，并在首页视图函数中通过flask.render\_template渲染模板，代码如下。

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

以上代码中的index.html模板的代码如下。

```
<table>
<thead>
<tr>
    <th>书名</th>
    <th>作者</th>
    <th>价格</th>
</tr>
</thead>
<tbody>
    {% for book in books() %}
        <tr>
            <td>{{ book.name }}</td>
            <td>{{ book.author }}</td>
            <td>{{ book.price }}</td>
        </tr>
    {% endfor %}
</tbody>
```

上述代码中，books协程被当作普通函数执行，即可得到books协程返回的结果。