第125天：Flask 项目结构

原创 太阳雪 Python技术 2月12日

前面我们了解了 Flask 框架的特性和一些用法，比如创建一个简单应用、做些页面，以及增加鉴权模块等，如果要将 Flask 用于实际项目开发，还需要了解一下 Flask 项目结构。

Flask 是一个轻量级的 Web 框架，扩展性强，灵活性高，容易上手，不过 Flask 没有给出明确的项目结构，而是让开发者根据实际需求，创建适合自己的项目结构。对于初学者来说，面临的困难可能是不知道如何组织代码，特别是在看一些别人的代码时，弄不清结构，对理解和学习造成一定障碍。

需要说明的是今天所介绍的结构并不是最好的，不同的项目，不同的团队，不同的理念，会有不同的结构，今天介绍的只是一个参考。

我们就那之前的 **Flask数据持久化** 章节的练习作为实践。

按功能组织

按功能，指的是将 Web 项目的不同职能划分开，比如路由部分、模型部分、业务逻辑部分等

目录结构

```
project/
  forms/
    myform.py
    ...
  models/
    __init__.py
    profile.py
    user.py
    ...
  routes/
    __init__.py
    home.py
    profile.py
    ...
  static/
    ...
  services/
    __init__.py
    ...
  templates/
    createprofile.html
    profile.html
```

```
    ...
  __init__.py
  config.py
```

可以看到，项目根目录下，分为:

- forms（表单）：存放表单对象
- models（模型）：存放数据模型，即库表在程序中的映射对象，以及对象之间的关系
- routes（路由）：存放请求路由以及处理逻辑
- static（静态文件）：Flask 约定存放静态文件的目录
- templates（模板）：Flask 约定存放页面模板的目录
- services（服务）：存放业务逻辑或者其他服务类功能
- `__init__.py`：Flask app 初始化方法
- `config.py`：项目配置文件

这样的分类，相当于将之前写到一个代码文件（app.py）中的逻辑，按功能划分开，当项目逐渐变大变复杂后，这样的划分有助于开发和维护

初始化

按功能划分开，比较容易理解，不过将分开的部分有机结合起来是个问题， 推荐的方式是，在每个目录下创建一个 `__init__.py` 文件，有两个作用:

1. 将目录变为包（package），方便其他地方引入
2. 做些初始化工作，例如将目录下的内容统一起来，提供一站式装载

初始化模型

数据模型放在 `models` 目录下，一般数据模块需要和数据库交互，另外每个模型需要有个数据库实例，来创建模型以及字段定义

首先，创建目录的 `__init__.py` 代码文件：

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

def init_app(app):
    db.init_app(app)
    return db
```

- 使用之前了解过的 `Sqlalchemy` 库，做数据映射（ORM）框架
- 定义一个数据库对象 `db`，注意构造方法没有传递 app 参数，是因为此时还得到 app

- 定义一个 `init_app` 函数，接收一个参数 `app`，就是 Flask app，用 `db.init_app` 方法初始化 Flask app

模型代码示例 `project/models/profile.py` ：

```
from . import db

class Profile(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(20))
    birthday = db.Column(db.Date())
    createtime = db.Column(db.DateTime())
    about = db.Column(db.Text())
```

- 因为是在 models 目录下的代码文件，所以通过当前目录引入在 `__init__.py` 中定义的 `db`，用来定义数据模型及其字段
- 字段在前面的数据持久化中有详细说明，这里省略解释

初始化路由

当路由处理代码被分开之后，在主程序代码中初始化会变得比较麻烦，幸好 Flask 有个 blueprint（蓝图）的概念，能很好的将分离出来的代码管理起来，确切的说 blueprint 的作用不止于此，这里只是需要用到它的部分功能

> Web 项目被分成多个部分之后，每个部分可以单独成为一个子应用，blueprint 的作用就是可以让子应用的编写方式用在主应用中一样，比如注册路由，处理请求等，使用前，先创建一个 blueprint 实例，然后再将实例注册到 Flask app 实例中就好了。

路由处理定义示例， `project/routes/home.py` :

```
from flask import Blueprint

home_bp = Blueprint('home_bp', __name__)

@home_bp.route('/', methods=['GET', 'POST'])
def index():
    return "Hello World!", 200
```

- 引入 Flask 的 blueprint 模块
- 初始化 blueprint 实例的实例 `home_bp`，第一个参数是 终端点（endpoint）的名称，用在 `url_for` 方法中，第二个参数是作为模块被引入时候的名称
- 使用和 blueprint 实例同名的装饰器定义路由，代替了之前 `@app.route` 的方式，相当于 blueprint 实例是一个 Flask app 实例
- 视图函数的定义之前 Flask app 实例中的没有区别，这里只是简单的返回文字和状态

再看看路由模块的初始化， `routes/__init__.py` :

```
from .home import home_bp
from .profile import profile_bp

def init_app(app):
    app.register_blueprint(home_bp)
    app.register_blueprint(profile_bp)
```

- 从当前目录下引入具体路由文件，从中引入 blueprint 实例
- 定义初始化方法，参数就是 Flask app 实例，用 `register_blueprint` 方法将 blueprint 注册到 Flask app 实例中

Flask app 工厂方法

在之前的介绍中，在 `app.py` 中编写所有的东西，并且通过 `app.run()` 来启动应用，在实际项目中，推荐用 app 工厂方法的方式来启动，好处是：

1. 便于测试，可以在不同的测试用例中创建特别的 app 实体
2. 多实例运行，如果需要一个应用的多个版本，可以在一个应用进程中运行多个实例，而不必部署多个 Web 服务器（将在 Flask 部署中介绍）

创建 Flask app 写在 `project/__init__.py` 中：

```
from flask import Flask
from .config import config
from . import models, routes

def create_app(config_name='default'):
    app = Flask(__name__)
    app.config.from_object(config[config_name])
    config[config_name].init_app(app)

    models.init_app(app)
    routes.init_app(app)

    return app
```

- 引入 flask 库
- 从当前目录下引入配置文件
- 引入上面定义的模型模块和路由模块
- 定义工厂方法，默认方法名是 `create_app`，如果其他名称，需要在启动应用时指定（随后介绍），工厂方法有个参数，用来指定需要加载的配置内容，且设定了默认值（随后介绍）
- 工厂方法中，先创建 Flask app 实例，然后加载配置，最后为模块初始化 模型和路由
- 最后返回 Flask app 实例，方便测试和应用启动时的调用

工厂方法比单个文件写法更清爽，修改起来也更简单，另外这样定义还可以避免循环依赖问题，

> 循环依赖：如果在工厂方法中直接定义数据库模块 db，在模型中需要引用 db，而工厂方法又需要用模型来初始化 Flask app，就会引起循环依赖问题

启动项目

通过工厂方法创建的应用，因为没有明确的 `app.run()` 调用，不能直接像在前一样直接运行文件，而是要用 flask 命令行方式来启动

正常启动

启动之前，需要先设置 FLASK_APP 环境变量，指定需要运行的 Flask 项目， 值为项目文件夹名，即项目名:

- Linux 或者 Mac

```
export FLASK_APP=project
```

- Windows 命令行

```
set FLASK_APP=project
```

- Powershell

```
$env:FLASK_APP="project"
```

然后在项目目录的上一层目录下执行命令，启动项目:

```
flask run
```

如果一切正常，就可以看到类似的结果:

```
 * Serving Flask app "project"
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

设置启动参数

前面工厂方法可以定义一些参数，如何来指定呢？其中一种方法是设置环境变量 `FLASK_APP`，例如将 congfig_name 参

数设置为 testing：

- Linux 或者 Mac

```
export FLASK_APP=project:create_app('testing')
```

- Windows 命令行

```
set FLASK_APP=project:create_app('testing')
```

- powershell

```
$env:FLASK_APP="project:create_app('testing')"
```

- project 是 Flask app 所在的模块名，与上面直接写 project 一样
- 冒号(:)后即为工厂方法的名，并且将参数带入

您可能已经想到，这种方式可以解决工厂方法名不是默认 `create_app` 的情况了，例如工厂方法名为 `myapp_factory`，以 Linux 环境为例:

```
export FLASK_APP=project:myapp_factory()
```

按业务组织

一个大型项目中，会包含很多子业务，比如人员管理、订单管理、统计报表等等，每一部分都可以是独立的项目，在 Flask 中，按照业务的方式将文件划分开，就是按业务方式来组织项目结构，这样的组织方式有助于并行开发和分而治之。

大体的结构是：

```
project/
  __init__.py
  config.py
  db.py
  auth/
    __init__.py
    route.py
    models.py
    templates/
  blog/
    __init__.py
    route.py
    models.py
    templates/
```

```
    ...
```

- 项目结构中，`__init__.py` 和 `config.py` 是整个应用的
- auth、blog 目录是子业务，可以看到每个内部有自己的路由、模型和模板文件夹
- `db.py` 是数据库连接模块，为各个子业务提供统一的数据库连接和模型支持，因为是单独的数据库模块，所以不会出现前面说的循环依赖问题，需要使用数据库模块的地方直接导入就行。

按业务组织方式中，每个子业务都是以 blueprint 的方式注册到 Flask app 上的，在按功能组织的方式中，我们只将路由用 blueprint 注册了，其实可以将 blueprint 看成一个独立的 Flask app，不过不用真的去部署，只需要注册到真正的 Flask app 上就行。

项目配置

之前的练习中都是将配置写到 `app.py` 中的，只是为了方便练习。实际项目中，配置是很重要的部分，系统研发过程中，需要经历不同的环境，比如开发环境，测试环境和生产环境，如果配置错了，开发环境连接上了生产环境的数据库，那可就糟糕了，相反开发环境为了方便查错，往往开启了 Debug 模式，如果放在生产环境中将会带来很大的性能损耗。

Python 中提供了丰富的配置处理方式，今天就 Flask 的配置利用实例简单介绍下，不过介绍的不见得就是最好的，只是一种解决思路，大家可以根据具体项目有所调整。

看一下代码：

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'hard to guess string'
    SQLALCHEMY_COMMIT_ON_TEARDOWN = True

    @staticmethod
    def init_app(app):
        pass

class DevelopmentConfig(Config):
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DEV_DATABASE_URL') or 'sqlite:///' + os.path.join(base

class TestingConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('TEST_DATABASE_URL') or 'sqlite:///' + os.path.join(bas

class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or 'sqlite:///' + os.path.join(basedir,

```

```
config = {
    'development': DevelopmentConfig,
    'testing': TestingConfig,
    'production': ProductionConfig,

    'default': DevelopmentConfig
}
```

- 实例项目中使用的数据库是 Sqlite，需要指定数据库文件目录，所以在定义了统一的目录 `basedir`，获取的是项目所在文件夹目录，因为 `config.py` 就在项目根目录中
- 定义了 Config 基类，用类属性设置了 Flask app 的一些配置，其中 init_app 是为了在初始化 app 是附加一些额外配置用的
- 定义了三个不同场景下的配置子类，分别是开发环境、测试环境和生产环境，大同小异，特别需要注意的是数据库文件名称，不同的场景有不同的名称
- 将不同的配置放在一个词典（dict）中，在 Flask app 工厂方法中，通过参数来选择具体的配置类，为了严谨，还设置的默认配置，显然默认为开发环境是很好的选择

有必要说明的是，Flask app 有多种设置配置的方式，常用的如:

- 属性指定，如 app.config["MYCONFIG"] = "xxxx"
- 对象获取，如 app.config.from_object(configObj)
- 配置文件，app.config.from_pyfile(pyfilepath)
- JSON，app.config.from_json(jsonfilename)

总结

今天介绍了一个简单的 Flask 项目化结构，并利用之前做过的数据持久化的练习，做了项目化后的代码说明，并且说明了按功能和按业务两种项目组织方式，最后介绍了 Flask 项目中配置的方式。总之 Flask 是一个宽松的，简单的 Web 框架，每个人都可打造自己的一套项目构建方式，不过对于刚开始接触的同学，或者为了交流方便，还是有必要了解一下一般的组织方式，希望对您有所启发。

最后，示例代码中提供了较为完整的例子，其中还有如何自定义数据库初始化命令的例子，请您参考。

参考

- https://flask.palletsprojects.com/en/1.1.x/tutorial/factory/
- https://lepture.com/en/2018/structure-of-a-flask-project
- https://exploreflask.com/en/latest/blueprints.html

> 示例代码：https://github.com/JustDoPython/python-100-day

系列文章

第124天：Web 开发 Django 模板
第123天：Web 开发 Django 管理工具
第122天：Flask 单元测试
第121天：机器学习之决策树
从 0 学习 Python 0 - 120 大合集总结

PS： 公号内回复：Python，即可进入Python 新手学习交流群，一起**100天计划！**

-END-

Python 技术

关于 Python 都在这里