Flask 入门

后端技术栈概述

现代的 APP 普遍采用解耦的架构，在一个开发团队内部细分为前端和后端：前者负责界面设计、用户交互，后者（尤其是 Web 后端）负责数据管理、业务逻辑，来保证数据的安全可靠。

就拿用户登录的场景来说，从前端到后端的整个过程是这样的：

- 用户通过图形化的界面来登录
- 后端提供登录的**接口**，开发者（前端）根据这个接口来发送请求
- 后端提供登录**服务**，即，验证用户的用户名和密码是否匹配数据库
- 后端数据库执行查询指令，返回结果

从接口到服务，再到数据库——于是整个过程的后端技术栈就分成了：

- **服务器端框架**：负责提供接口和服务，也就是处理业务逻辑、与数据库交互
- **数据库**：用于数据的存储和管理

Flask 就是我们需要的一个轻量级的 Python 服务器端框架。

Flask 基于 WSGI（Web Server Gateway Interface）和 Jinja2 模板引擎，旨在帮助开发者快速、简便地创建 Web 应用。

Flask

为什么说 Flask 轻量？横向对比如 Django，它还提供 ORM，表单验证，后台管理等等其它功能。它的全面确实给开发者带来了很多便利；但同时，一个网络应用并不一定全要用上这些功能，正如很多人不喜欢大型 IDE 一样。Flask 在这个意义上更像是一个后端骨架，要什么往上加就好。

顺带一提，实际工作中最常用的是 Java 技术栈（Spring 系列），考虑到现代 Java 框架对于初学者并不友好（同时我们也不希望和大家的课程冲突），我们在本学期仅介绍 Python 的 Flask 框架：它足够简单，性能也足够支撑大型 Web 应用，同时拥有良好的社区生态，有大量的第三方扩展库可供选择；在合适的团队管理下，Flask 的代码也不会变成一团浆糊。

网络基础知识

对 Web 应用我们要了解的一些基本认知有：

一个网站只是一堆保存在硬盘上的文件，而网站包含一种称为 HTML 的代码。存放网站的这些文件的专用机器称为**服务器**。Web 的这种网络架构称为 Client/Server（客户端/服务器）架构。

互联网就是由互相连通的机器 (服务器) 组成的网络（海底电缆地图网站）

但是很明显，我们不太可能用电缆去连接任何两台上网的电脑。我们需要将请求通过很多很多不同的机器传递，以便连接一台机器。

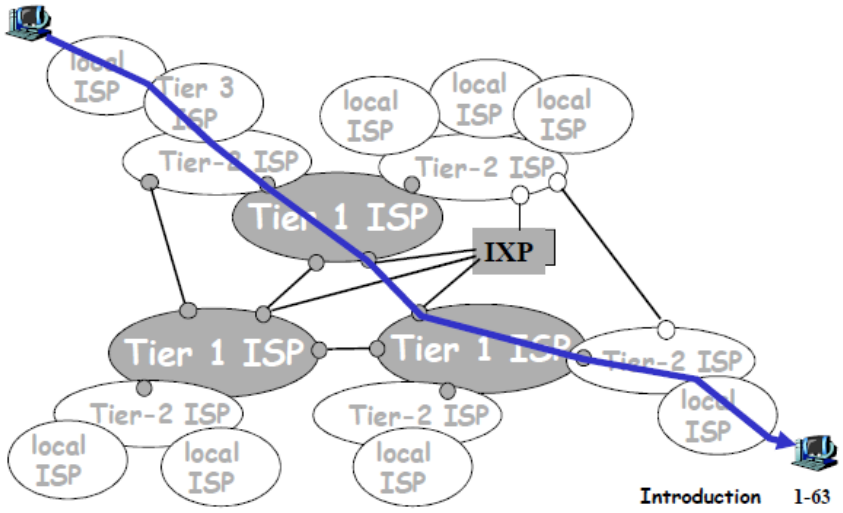


Figure 2: 互联网是一个“网络的网络”，一个分组会经过许多网络，不同范围的 ISP（互联网服务提供商），在 ISP 内部又会经过不同的分组

想象一下，当你键入 `https://www.swufe.edu.cn/`，你会发送一封“信”到学校的服务器——

你的信件去了离你最近的邮局。然后它去离你的收件人稍近一点的邮局，然后再去另一个，以此类推地到达它的目的地。唯一的事情是，如果你将许多信件 (**数据包**) 发送到同一个地方，他们可以通过完全不同邮政局 (**路由器**)。这取决于每个办公室的分布情况。

你发送信息，然后期望一些回复。当然，你使用的不是纸笔，而是比特数据（bits）。

我们使用的地址叫做 IP **地址**：what is my ip address

IP 地址用来标识互联网上的一个设备，而**端口号**则用来标识该设备上的一个特定服务或应用程序。

当然你记不住这一串串的数字，所以我们通常用**域名**来代替：当你访问 `https://www.swufe.edu.cn/`，会经过 DNS（**域名服务器**）来进行域名解析[1]，这个过程有点像老式的电话簿，在那里你可以找到你想联系的人的名字，并且找到他们的电话号码和地址。

相信一定有同学改过 Windows 的 hosts 文件 :)

```
# Added by Docker Desktop
10.60.139.208 host.docker.internal
10.60.139.208 gateway.docker.internal
```

[1] 当你无法访问 `www.github.com` 的时候，很有可能就是在 DNS 解析上出现了一些小问题。

当你发送一封信时，它需要确切的功能以便正确的传递：地址，邮票等等。你使用一种接收者同样能够理解的语言。你送到网站的数据包也类似如此。我们使用一个名为 HTTP (Hypertext Transfer Protocol：**超文本传输协议**) 的协议。

所以，当你有一个网站，你需要有一台服务器用于网站的托管。当服务器接收到一个访问**请求**（一封信），它把网站内容作为**响应**发送回去（回信）。

安装 Flask

创建虚拟环境

为什么要使用虚拟环境？随着你的 Python 项目越来越多，你会发现不同的项目会需要不同的版本的 Python 库。同一个 Python 库的不同版本可能不兼容。

举一个 Pandas 的例子，如果你克隆了一个上古项目，那么它读取 csv 的代码是这样的：

```
import pandas as pd

df = pd.read_csv('data.csv', na_values=['NA', 'NULL'])
```

但 `na_values` 这个参数名已经弃用(depreciated). 如果你的全局 Python 解释器安装的是版本 1.0+，那么代码会报错。当然，你也肯定不想手动降级至 pandas 0.24.0 版本以下，也懒得改代码发起一条 PR，这时候就可以使用虚拟环境。

虚拟环境可以为每一个项目安装独立的 Python 库，这样就可以隔离不同项目之间的 Python 库，也可以隔离项目与操作系统之间的 Python 库。

Python 内置了用于创建虚拟环境的 `venv` 工具：

```
python -m venv .venv
```

上面的指令会以系统指定的 python 解释器创建一个名为 `.venv` 的全新虚拟环境(这个名称可以自定义, 例如 `env` 也是可以的)。

激活虚拟环境：

```
.venv\Scripts\activate # Windows
. .venv/bin/activate   # MacOS/Linux
```

这时候命令行前缀会变成 `(.venv)`：

```
(.venv) $
```

安装 Flask：

```
pip install Flask
```

在 PyCharm 中，新建一个 Flask/Django 项目都会指定创建的虚拟环境(它还会要求你指定使用的包管理器)。

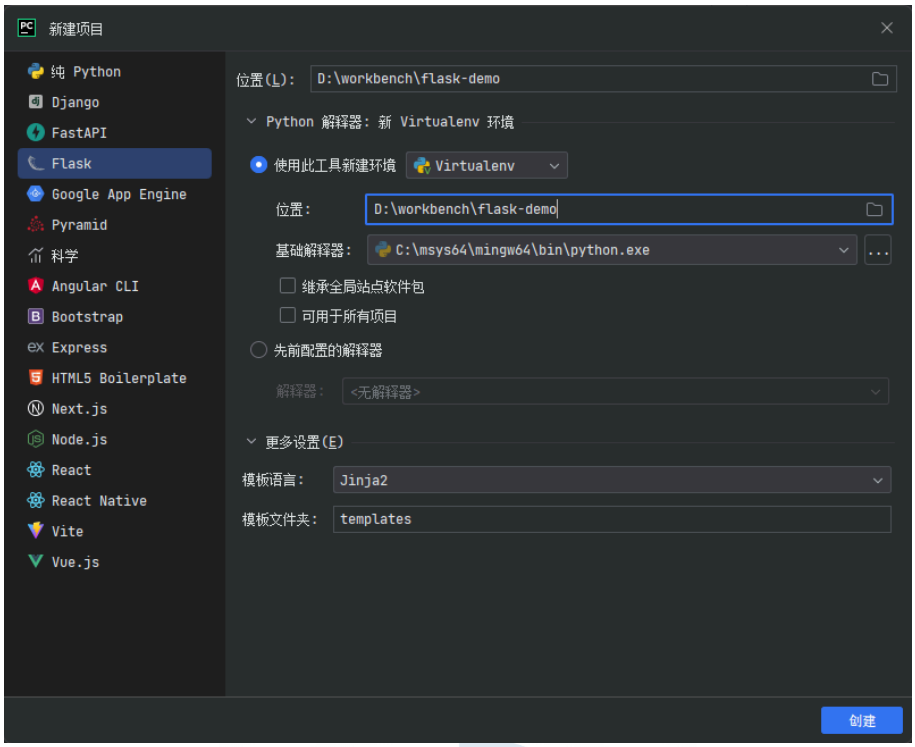


Figure 3: PyCharm 默认的包管理器是 `virtualenv`, 你还可以尝试 `pipenv`, `conda`, `poetry` 等

第一个 Flask 应用

运行应用

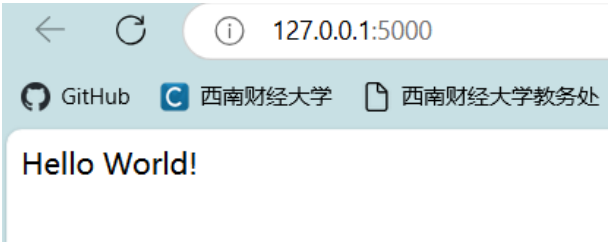
从 PyCharm 默认创建的 Flask 项目模板开始, 我们来了解 Flask 应用是怎么运行的:

```
from flask import Flask

app = Flask(__name__)


@app.route('/')
def hello_world():  # put application's code here
    return 'Hello World!'


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

在终端里运行

```
(.venv)$ flask run
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use
a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Flask 会使用内置的开发服务器来运行程序。这个服务器默认监听本地机的 `5000` 端口，也就是说，我们可以通过在地址栏输入 `http://127.0.0.1:5000` 或是 `http://localhost:5000` 访问程序。



创建应用实例

```
app = Flask(__name__)
```

- 用 `Flask` 创建一个应用实例.
- `__name__` 是 Python 的一个内置变量(bulitins)，它代表当前模块的名称.

Flask 默认会假设你把程序存储在名为 `app.py` 或 `wsgi.py` 的文件中, 如果你希望 Flask 的命令行工具能找到你更名后的程序(例如 `server.py`), 可以尝试更改系统变量:

```
export FLASK_APP=server.py # linux/MacOS
$env:FLASK_APP = "server.py" # Windows powershell
```

或者配置一个环境变量文件

```
pip install python-dotenv
```

```
touch .env | echo FLASK_APP=server.py > .env # Linux/MacOS
echo $env:FLASK_APP = "server.py" >> .env # Windows powershell
```

在 `server.py` 里注意修改 utf-8 编码来读取(中文编码系统的话):

```
from dotenv import load_dotenv
load_dotenv(encoding='utf8')
```

或者尝试带有 `--app` 参数的 `flask run` 命令:

```
flask --app server run
```

PyCharm 提供的程序模板中:

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

我们在 Python 的主函数中, 手动指定了启动的 Flask 应用实例, 这样你就可以直接执行这个 python 文件来启动 Flask 了:

```
python app.py
```

路由

接下来，我们要注册一个处理函数，这个函数是处理某个请求的处理函数，我们把它叫做**视图函数**（view funciton）：

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

所谓的“注册”，就是给这个函数戴上一个装饰器(什么是装饰器?).

我们使用 `app.route()` 装饰器来为这个函数绑定对应的 URL，当用户在浏览器访问这个 URL 的时候，就会触发这个函数，获取返回值，并把返回值显示到浏览器窗口.

Flask 内部会维护一个 URL 映射表

```
>>> from app import app
>>> app.url_map
Map([<Rule '/static/<filename>' (OPTIONS, HEAD, GET) -> static>,
 <Rule '/' (OPTIONS, HEAD, GET) -> hello_world>])
```

这里的 `/` 对应的是主机名后面的路径部分，完整 URL 就是 `http://localhost:5000/`。更改 URL 路径，如

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

现在访问 `http://localhost:5000/hello` 就可以看到你的消息了.

`app.route` 对这个函数实际做的事情是:

```
app.add_url_rule('/', view_func=hello_world, endpoint='hello_world')
```

在 Django 里, 你通常需要手动设定这样的 URL 映射, 而 Flask 可以简便地使用装饰器.

除此之外, 你可以使用 `url_for` 来生成视图函数的 URL

```
>>> from flask import url_for
>>> url_for('hello_world')
'/'
>>> url_for('hello_world', _external=True)
'http://localhost:5000/'
```

Flask 与 HTTP

当我们访问 `http://127.0.0.1:5000/` 时, Flask 的日记会输出:

```
127.0.0.1 - - [25/Oct/2024 16:10:06] "GET / HTTP/1.1" 200 -
```

- `"GET / HTTP/1.1"` 是用户发起的 HTTP 请求，其中 `GET` 是请求方法，`/` 是请求路径，`HTTP/1.1` 是 HTTP 协议版本.
- `200` 是 HTTP 状态码，表示请求成功.

每一个 Web 应用都包含这种处理模式，即 HTTP 的“**请求-响应循环**（Request-Response Cycle）”：客户端发出请求，服务器处理请求并返回响应.

在用户请求中可以包含很多东西: 当你访问一条网址时，在浏览器打开 F12 检查网络, 请求信息里会有:

- **请求方法**(method): GET, POST, PUT, DELETE, HEAD, OPTIONS...
- **请求路径**(url): 网址的路径部分
- **请求参数**(query string): 例如 `http://localhost:5000/search?q=flask` 中的 `q=flask`
- **请求体**(data): 通常会传入一个表单
- **请求头**(header): 例如 `User-Agent`, `Accept-Language`, `Accept-Encoding`
- ...

请求与上下文

回到我们先前的代码, 注意到一件事: 我们在视图函数中并没有手动声明需要这样的 `request`, 对于其它框架(如 Spring)来说这是不太一样的:

```
@Postmapping("/register")
public String register(@RequestBody User requestUser) {
    // ....
}
```

在 Flask 的设计中: 请求对象和用户请求是一一对应的, 且请求对象是**只读的**.

获取用户的这次请求只需要从**上下文**(contexts)中去取即可, 当你需要它时, import 一下就好:

```
from flask import request

@app.route('/')
def fetch_your_ip():
    return request.remote_addr
```

- 通过访问 `request` 的 `remote_addr` 获得了该条请求来自的 IP.
- 除此之外, 访问 `request` 的 `form` 可以获取请求的表单数据.

借助上下文管理器, 我们可以显式地从上下文中获取请求对象(在应用外部创建一个请求上下文):

```
from flask import request, Flask

app = Flask("my flask app")
print(repr(request))
# <LocalProxy unbound>

with app.test_request_context():
    print(repr(request))
# <Request 'http://localhost/' [GET]>
```

根据作用域的不同, Flask 把上下文分成**应用级上下文**(`flask._app_ctx_stack`)和**请求级上下文**(`flask._request_ctx_stack`), 每种上下文环境会维护单独的**栈**:

```
ctx = app.test_request_context()
ctx.push()
# ...
ctx.pop()
```

应用级上下文包含了应用级别的信息，比如配置和应用的全局状态(`flask.g`)。

请求级上下文中还包括

- `flask.session`: **会话对象**, 用来存储用户的会话信息. 例如用户登录后浏览器就会保留记录, 下次访问就不需要再登录了.

对 HTTP 请求的这种设计是**线程安全**(thread-safe)的, 保证在多线程任务下不会出现数据竞争等问题![2]

响应

Flask 中, HTTP 响应通常是**隐式**创建的, Flask 调用视图函数后，会将其返回值作为响应的内容。多数情况下，响应就是一个简单的字符串，作为 HTML 页面回送客户端。

但 HTTP 协议需要的不仅是作为请求响应的字符串。HTTP 响应中一个很重要的部分是**状态码**，Flask 默认设为 200，表明请求已被成功处理。如果视图函数返回的响应需要使用不同的状态码，可以把数字代码作为第二个返回值，添加到响应文本之后。

响应可以是文本, 是一个元组(tuple), 或者是渲染过的模板:

[2]操作系统小知识: 线程是可单独管理的最小指令集。进程经常使用多个活动线程，有时还会共享内存或文件句柄等资源。多线程 Web 服务器会创建一个线程池，再从线程池中选择一个线程处理接收到的请求。

```
@app.route('/')
def index():
    return '<h1>Bad Request</h1>', 400, {'Content-Type': 'text/html'}
@app.route('/game')
def game():
    return render_template('2048.html')
```

当然你也可以用 `make_response` 的接口来显示创建 HTTP 响应:

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

响应有个特殊的类型，称为**重定向**。这种响应没有页面文档，只会告诉浏览器一个新 URL，用以加载新页面。重定向的状态码通常是 302，在 Location 首部中提供目标 URL。Flask 提供一个简单的接口来实现重定向:

```
from flask import redirect

@app.route('/')
def index():
    return redirect('http://www.example.com')
```

还有一种特殊的响应由 `abort()` 函数生成，用于处理错误. `abort()` 不会把控制权交还给调用它的函数，而是抛出异常。

参考资料

Flask doc: Flask 官方文档

Flask 入门教程: 很好的中文在线教程

互联网是如何工作的? : 对这部分更感兴趣的同学可以参考一些计算机网络课程的导论部分.

Flask Web 开发: 基于 Python 的 Web 应用开发实战: 这是一本很好的 Flask 入门书籍, 由于 DMCA 版权问题大家可以通过某种方式(例如 zlib)来下载它.