第 3 章

AIOHTTP

AIOHTTP 是一个基于 asyncio 开发的 Python Web 开发框架，其稳定、强大、支持高并发。目前已实现的功能有 HTTP 服务器、HTTP 客户端、WebSocket 服务器，以及 WebSocket 客户端，服务功能还实现了路由、模板、会话(Session)等所有 Web 开发必备的功能模块。

3.1 创建异步 Web 服务器

AIOHTTP 的使用非常简单，只需创建一个 aiohttp. web. Application 对象，并启动它便可完成一个 Web 服务器的最基本功能，代码如下：

```
from aiohttp import web

#创建一个服务器应用
app = web.Application()
#启动服务器应用
web.run_app(app)
```

启动前需要使用命令 pip install aiohttp 安装第三方依赖项 aiohttp。因为该服务器暂未编写任何页面请求处理逻辑，所以对其任何页面的访问都将出现 404 代码(找不到资源)，如图 3-1 所示。

图 3-1　空服务器示例

若要让其可以处理请求，则需要创建一个请求处理函数，该函数是一个异步函数，接收的一个传入参数是请求对象(aiohttp. web. Request)，返回一个响应对象(aiohttp. web. Respone)，并将该函数映射到服务器指定的路径上，代码如下：

```
"""
第 3 章/aiohttp_simple_server/server.py
```

```
"""
from aiohttp import web


# 请求处理函数,接收一个参数为请求对象,返回信息将被传到浏览器端
async def hello(request: web.Request):
    return web.Response(text = "Hello, world")


# 创建一个服务器应用
app = web.Application()
# 将 hello 处理函数映射到网站根路径 /
app.router.add_get("/", hello)
# 启动服务器应用
web.run_app(app)
```

在该示例中,将 hello 函数映射到网站根路径上,这时访问网站主页,将看到页面中显示信息 Hello, world,结果如图 3-2 所示。

图 3-2 AIOHTTP Hello,world 示例

接下来为该服务器配置容器化支持,在项目目录下创建 Dockerfile 文件,在其中输入如下代码:

```
# 第 3 章/aiohttp_simple_server/Dockerfile
FROM python:3.8 - slim
RUN pip install  - i https://pypi.tuna.tsinghua.edu.cn/simple aiohttp
```

脚本指定该容器基于 python:3.8-slim,这是一个功能完整但轻量级的 Linux 环境,在第 2 行使用清华镜像安装项目依赖项 aiohttp。

在项目目录下创建一个 docker-compose.yml 文件,在其中输入如下代码:

```
# 第 3 章/aiohttp_simple_server/docker - compose.yml
version: "3"

services:
  web:
    build: .
    ports:
      - "8080:8080"
    # 如果指定该容器,则会在意外停止后自动重启
    restart: always
    # 配置路径映射,将当前目录映射到容器中的 /opt 目录
    volumes:
      - ".:/opt/"
    # 指定容器中应用程序的工作目录为 /opt
    working_dir: "/opt/"
    # 配置容器启动时执行的命令,这里我们启动服务器
    command: python3 ./server.py
```

所有工作已经准备就绪，下面在终端里键入 docker-compose up-d 以自动构建镜像并启动，如图 3-3 所示。

图 3-3 启动 Docker 服务

3.2 路由



4min

路由(route)是一种将路径映射到处理函数的机制，可以一个一个地单独配置，示例代码如下：

```
"""
第 3 章/aiohttp_route/config_route_one_by_one.py
"""
from aiohttp import web


async def hello(request: web.Request):
    return web.Response(text = "Hello, world")


app = web.Application()
app.router.add_get("/", hello)
web.run_app(app)
```

可以批量配置，代码如下：

```
"""
第 3 章/aiohttp_route/config_routes.py
"""
from aiohttp import web


async def hello(request: web.Request):
    return web.Response(text = "Hello, world")


async def users(req):
    return web.Response(text = "All users are here")


app = web.Application()
app.add_routes([
```

```
    web.get("/", hello),
    web.get("/users", users)
])
web.run_app(app)
```

也可以通过装饰器进行配置，代码如下：

```
"""
第 3 章/aiohttp_route/config_route_with_decorator.py
"""
from aiohttp import web

routes = web.RouteTableDef()


@routes.get('/')
async def hello(request):
    return web.Response(text = "Hello, world")


app = web.Application()
app.add_routes(routes)
web.run_app(app)
```

还可以通过路由配置接收其他的 HTTP 方法，代码如下：

```
@routes.post('/save_user_info')
async def save_user_info(req):
    return web.Response(text = "Saved")
```

如果仅仅支持配置路径映射，那么适用的场景就太少了，AIOHTTP 的路由还支持路径参数。例如，在实现一个博客类的网站时，通常会根据文章的 id 来定位该篇文章，为了对搜索引擎更友好，我们还会将文章的地址静态化，将类似 http://xxx.com? p=12 这样的路径改为 http://xxx.com/p/12，这就是服务器端开发常用的伪静态技术。代码如下：

```
"""
第 3 章/aiohttp_route/pages.py
"""
from aiohttp import web

routes = web.RouteTableDef()


@routes.get('/p/{page_id}')
async def page(req: web.Request):
    return web.Response(text = f"Page id is {req.match_info['page_id']}")


@routes.get('/')
```

```
async def hello(request):
    return web.Response(text = "Hello, world")


app = web.Application()
app.add_routes(routes)
web.run_app(app)
```

访问页面结果如图 3-4 所示。

图 3-4 解析路径参数

此外还可以使用正则表达式对路径中的参数进行约束，如果我们想让 page_id 只接收数字类型的参数，则可对代码进行修改，代码如下：

```
@routes.get(r'/p/{page_id:\d + }')
async def page(req: web.Request):
    return web.Response(text = f"Page id is {req.match_info['page_id']}")
```

本节中我们一共写了 4 个单独的示例文件，如图 3-5 所示。

接下来我们打算用一个 docker-compose 配置文件一键启动所有示例，并运行在不同的端口上，此时需要在项目目录中创建文件 Dockerfile 和 docker-compose.yml，文件结构如图 3-6 所示。

在 Dockerfile 中输入代码如下：

```
# 第 3 章/aiohttp_route/Dockerfile
FROM python:3.8 - slim
RUN pip install  - i https://pypi.tuna.tsinghua.edu.cn/simple aiohttp
```

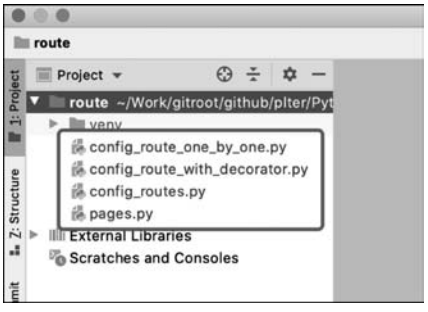

图 3-5 所有示例文件

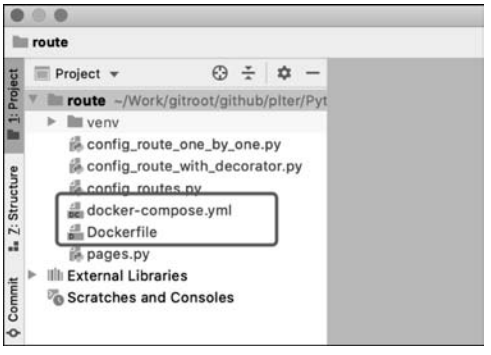

图 3-6 Docker 配置文件所在位置

在 docker-compose.yml 中输入代码如下：

```
# 第 3 章/aiohttp_route/docker - compose.yml
version: "3"

services:
```

```
config_route_one_by_one:
  build: .
  ports:
    - "8080:8080"
  restart: always
  volumes:
    - ".:/opt/"
  working_dir: "/opt/"
  command: python3 ./config_route_one_by_one.py

config_route_with_decorator:
  build: .
  ports:
    - "8081:8080"
  restart: always
  volumes:
    - ".:/opt/"
  working_dir: "/opt/"
  command: python3 ./config_route_with_decorator.py

config_routes:
  build: .
  ports:
    - "8082:8080"
  restart: always
  volumes:
    - ".:/opt/"
  working_dir: "/opt/"
  command: python3 ./config_routes.py

pages:
  build: .
  ports:
    - "8083:8080"
  restart: always
  volumes:
    - ".:/opt/"
  working_dir: "/opt/"
  command: python3 ./pages.py
```

在该配置文件中创建了 4 个服务，将端口分别映射在本机的 8080、8081、8082、8083 端口，启动后可以分别通过这 4 个端口访问 4 个不同的示例。

3min

3.3 静态文件处理

一个完整的网站除了服务器应用之外，一定会存在很多静态文件，例如图片、js 脚本等。

在大型网站的部署上，静态资源会单独使用 CDN 进行管理，所以应用层可以不用考虑这个问题，直接通过 CDN 去引用静态资源文件即可。

在中小型网站的部署上，传统阻塞型 IO 编程模式下，由于应用服务器并不擅长处理高并发请求，一般来说会把应用服务器与基础 HTTP 服务器（如：Apache 或者 Nginx）集成并把静态文件的处理工作交给基础 HTTP 服务器。但在异步 IO 编程模型下，完全可以直接使用 Python 来处理静态资源请求。

AIOHTTP 已经内置了静态资源文件处理的功能，因此可以直接使用，非常方便，代码如下：

```
"""
第 3 章/aiohttp_static_resource/server.py
"""

from aiohttp import web
import os

routes = web.RouteTableDef()


@routes.get('/')
async def hello(request):
    return web.Response(text = "Hello, world")


app = web.Application()
app.add_routes(routes)
#配置一个静态文件目录
app.router.add_static(
    "/static",
    os.path.join(os.path.dirname(__file__), 'static')
)
web.run_app(app)
```

配置一个静态文件目录，通过网站的 /static 路径进行访问，对应的文件都配置在项目目录下的 static 目录中，项目结构如图 3-7 所示。

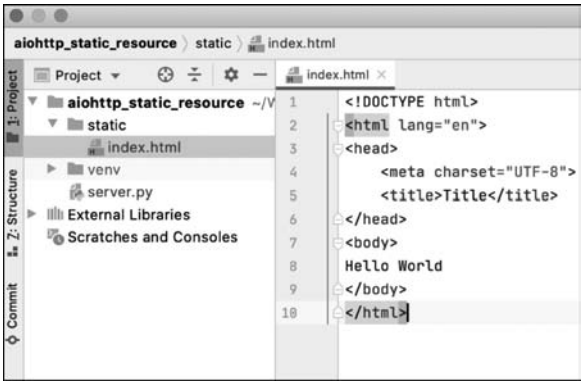

图 3-7 项目文件结构

启动服务器后可通过浏览器访问地址http://127.0.0.1:8080/static/index.html，结果如图3-8所示。

图3-8 静态文件访问结果

与该项目相关的Dockerfile文件内容如下：

```
#第3章/aiohttp_static_resource/Dockerfile
FROM python:3.8-slim
RUN pip install -i https://pypi.tuna.tsinghua.edu.cn/simple aiohttp
```

与该项目相关的docker-compose.yml文件内容如下：

```
#第3章/aiohttp_static_resource/docker-compose.yml
version: "3"

services:
  web:
    build: .
    ports:
      - "8080:8080"
    restart: always
    volumes:
      - ".:/opt/"
    working_dir: "/opt/"
    command: python3 ./server.py
```

9min

3.4 模板渲染

如果每个页面都只是直接使用字符串拼接而成，那么实际的工作量也是很大的，所以人们发明了模板技术，用现成的HTML文件经过处理(渲染)，并将处理之后的结果返回给浏览器端就可以大大节省开发成本。

在AIOHTTP服务端应用开发时，可配套使用aiohttp_jinja2库实现模板渲染，通过命令pip install aiohttp_jinja2进行安装。首先需要指定模板文件的根目录，并将模板系统与服务端应用集成，代码如下：

```
aiohttp_jinja2.setup(
    app,
    loader=jinja2.FileSystemLoader(
        os.path.join(os.path.dirname(__file__), "tpls")
    )
)
```

然后在需要渲染的请求处理函数前添加装饰器以指定要渲染的模板的路径，代码如下：

```
@routes.get('/')
@aiohttp_jinja2.template("index.html")
```

```
async def hello(request):
    return dict()
```

完整的代码如下：

```
"""
第 3 章/aiohttp_template/server.py
"""

from aiohttp import web
import jinja2
import aiohttp_jinja2
import os

app = web.Application()
aiohttp_jinja2.setup(
    app,
    loader = jinja2.FileSystemLoader(
        os.path.join(os.path.dirname(__file__), "tpls")
    )
)
routes = web.RouteTableDef()


@routes.get('/')
@aiohttp_jinja2.template("index.html")
async def hello(request):
    return dict()


app.add_routes(routes)
web.run_app(app)
```

项目文件结构如图 3-9 所示。

其中模板文件 index.html 的内容如下：

```
<!-- 第 3 章/aiohttp_template/tpls/index.html -->
<!DOCTYPE html>
<html lang = "en">
<head>
<meta charset = "UTF - 8">
<title> Title </title>
</head>
<body>
Hello World
</body>
</html>
```

启动服务器后，最终浏览器运行结果如图 3-10 所示。

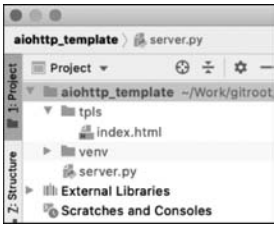

图 3-9　项目文件结构

图 3-10　模板渲染结果

如果仅仅读取静态文件并且呈现出来，那么模板技术显得没什么用处。而实际上，模板技术能够允许我们从后端传参数给模板并最终由模板引擎渲染成页面，例如从数据库中读取用户的信息并传给模板。

若要从 Python 代码中向模板文件传参数，只需请求处理函数返回一个字典，代码如下：

```
@routes.get('/')
@aiohttp_jinja2.template("index.html")
async def hello(request):
    return dict(name = "小云", age = 20)
```

模板文件中使用参数的代码如下：

```
<!-- 第 3 章/aiohttp_template/tpls/index.html -->
<!DOCTYPE html>
<html lang = "en">
<head>
<meta charset = "UTF - 8">
<title> Title </title>
</head>
<body>
<ul>
<li>名字:{{ name }}</li>
<li>年龄:{{ age }}</li>
</ul>
</body>
</html>
```

最终页面渲染的结果如图 3-11 所示。

此外，为了减少不必要的重复性工作，模板还支持继承、循环、条件渲染等。下面我们以常规网站页面为例，深入了解更多与模板引擎使用相关的知识。

实现效果如图 3-12 所示。

图 3-11　向模板传参数

图 3-12　通用网站框架结构

主页按钮背景和页面标题会随着页面的变化而变化，在博客页面还渲染了一个列表，如图 3-13 所示。

图 3-13 博客页面示例

在该项目的实现过程中，还使用了前端界面框架 Bootstrap，接下来一步一步实现这个常规网站的基本框架。

使用 PyCharm 创建一个新的项目，并命名为 aiohttp_template_pages，在终端中使用 npm init 命令初始化 npm 环境（使用该命令之前需要先安装 nodejs，到 https://nodejs.org 下载并安装）。所有选项均选择默认即可（在所有遇到需要输入信息时单击回车键直到完成），如图 3-14 所示。

```
Terminal: Local +
(venv) yunp@yunps-MBP aiohttp_template_pages % npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (aiohttp_template_pages)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/yunp/Work/gitroot/github/plter/PythonAIOProgramming/

{
  "name": "aiohttp_template_pages",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this OK? (yes)
(venv) yunp@yunps-MBP aiohttp_template_pages %
```

图 3-14 初始化 npm 项目

命令执行结束后将在项目目录下创建一个 package.json 文件，如图 3-15 所示。

通过 npm 命令 npm install--save jquery bootstrap--registry＝https://registry.npm.taobao.org 安装前端依赖项 jQuery 和 Bootstrap，在执行该命令时可指定使用淘宝的仓库，这样速度更快，执行完毕后，将在 node_modules 目录下新增 bootstrap 和 jquery 两个目录，如图 3-16 所示。

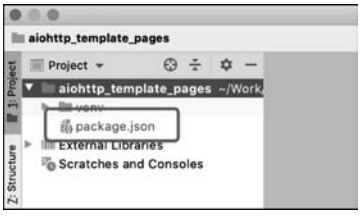

图 3-15　package.json 文件

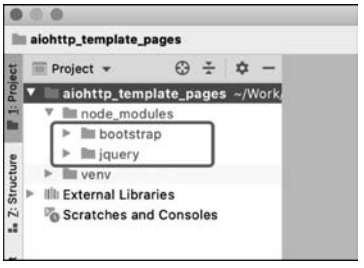

图 3-16　安装的前端依赖项

建立 tpls 目录，用于存放模板文件，在其中创建 layout.html、index.html、news.html、blog.html 4 个文件，结构如图 3-17 所示。

其中 blog.html、index.html、news.html 3 个模板分别对应 3 个页面，这 3 个页面的框架布局是一样的，所以我们将框架抽象出来并创建一个父级模板 layout.html。

其中模板 layout.html 文件的代码如下：

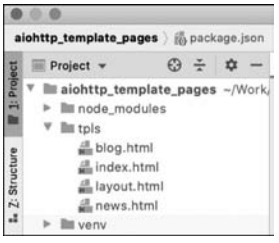

图 3-17　模板文件

```
<!-- 第 3 章/aiohttp_template_pages/tpls/layout.html -->
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<!--
要求传一个名为 title 的参数，这样不同的页面传来不同
的值以实现不同页面拥有不同标题的效果
    -->
<title>
        {{ title }}
</title>

<!--
引用前端页面的依赖项，使用的前端界面框架是 Bootstrap，
因为 Bootstrap 依赖 jQuery，所以我们也引用了 jQuery，
并且 jQuery 的引用语句必须写在 Bootstrap 引用语句的前面
     -->
<link rel="stylesheet" href="/node_modules/bootstrap/dist/css/bootstrap.min.css">
<script src="/node_modules/jquery/dist/jquery.min.js"></script>
<script src="/node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
```

```
</head>
<body>
<nav class="navbar navbar-light bg-light" style="justify-content:start">
<!-- 配置顶部菜单 -->
<a class="navbar-brand" href="/">小云站</a>

<ul class="nav nav-pills">
<li class="nav-item">
<!--
使用了模板引擎支持的判断语句,根据当前页面路径选择性
地给菜单项添加 active 类,以使它处于激活状态,这里
需要注意的是我们使用了 request 这个参数,意味着在渲
染时需要传入 request 对象
              -->
<a class="nav-link {% if request.path == '/' %}active{% endif %}"
                href="/">
首页
</a>
</li>
<li class="nav-item">
<a class="nav-link {% if request.path == '/news' %}active{% endif %}"
                href="/news">
动态
</a>
</li>
<li class="nav-item">
<a class="nav-link {% if request.path == '/blog' %}active{% endif %}"
                href="/blog">
博客
</a>
</li>
</ul>
</nav>
<div class="container-fluid">
<!-- 声明一个名为 body 的代码块,子模板可以重写该
代码块以自定义页面的内容 -->
    {% block body %}
    {% endblock %}
</div>
</body>
</html>
```

模板 index.html 的代码如下：

```
<!-- 第 3 章/aiohttp_template_pages/tpls/index.html -->
<!-- 指定继承模板 layout.html -->
{% extends "layout.html" %}
{% block body %}
```

```
<!-- 重写代码块 body 的内容 -->
这是首页
{% endblock %}
```

接下来创建 server.py 文件,用于配置服务器,其代码如下:

```
"""
第 3 章/aiohttp_template_pages/server.py
"""
from aiohttp import web
import jinja2
import aiohttp_jinja2
import os

app = web.Application()
aiohttp_jinja2.setup(
    app,
    loader = jinja2.FileSystemLoader(
        os.path.join(os.path.dirname(__file__), "tpls")
    )
)
routes = web.RouteTableDef()


@routes.get('/')
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request):
    return dict(title = "首页", request = request)


@routes.get('/news')
@aiohttp_jinja2.template("news.html")
async def news(request: web.Request):
    return dict(title = "动态", request = request)


@routes.get('/blog')
@aiohttp_jinja2.template("blog.html")
async def blog(request: web.Request):
    return dict(
        title = "博客", request = request,
        # 向模板传递数组
        posts = [
            {"title": "文章 1", "content": "内容 1"},
            {"title": "文章 2", "content": "内容 2"}
        ]
    )
```

```
app.add_routes(routes)
app.router.add_static(
    "/node_modules",
    os.path.join(os.path.dirname(__file__), "node_modules")
)
web.run_app(app)
```

其中/blog 页面向模板传递了数组，则对应的模板代码如下：

```
<!-- 第 3 章/aiohttp_template_pages/tpls/blog.html -->
{ % extends "layout.html" % }
{ % block body % }
这是博客页面
<div>
    { % for p in posts % }
<!-- 使用模板引擎支持的循环语句生成 html 代码 -->
<div>
<div class = "font - weight - bold">{{ p.title }}</div>
<div>{{ p.content }}</div>
</div>
    { % endfor % }
</div>
{ % endblock % }
```

与该项目对应的 Dockerfile 文件代码如下：

```
# 第 3 章/aiohttp_template_pages/Dockerfile
FROM python:3.8 - slim
RUN pip install - i https://pypi.tuna.tsinghua.edu.cn/simple aiohttp
RUN pip install - i https://pypi.tuna.tsinghua.edu.cn/simple aiohttp_jinja2
```

与该项目对应的 docker-compose.yml 代码如下：

```
# 第 3 章/aiohttp_template_pages/docker - compose.yml
version: "3"

services:
  web:
    build: .
    ports:
      - "8080:8080"
    restart: always
    volumes:
      - ".:/opt/"
    working_dir: "/opt/"
    command: python3 ./server.py
```

3.5 处理表单提交

处理表单提交是一个 Web 服务器最基本的功能，AIOHTTP 也提供了强大的表单处理功能，首先看一下效果，如图 3-18 所示。

图 3-18 表单页面渲染效果

在表单中输入信息后提交则可呈现结果页面，如图 3-19 所示。

图 3-19 表单提交后呈现结果

这个项目使用了 Bootstrap 来搭建前端页面，同样的使用流程不再赘述，这里只讲解最核心的处理逻辑。

在首页中编写一个表单，代码如下：

```
<!-- 第 3 章/aiohttp_submitform/tpls/index.html -->
{ % extends "layout.html" % }
{ % block body % }
<!-- 配置该表单提交的方式为 post 方式，提交的目标页面路径
是/login，并且以 URL 参数对的形式提交数据 -->
< form method = "post" action = "/login" enctype = "application/x - www - form - URLencoded">
< div class = "card" style = "margin: 2rem auto 0 auto;width:300px;">
< div class = "card - header">登录</div >
```

```
<div class="card-body">
<div class="form-group row">
<label for="inputName" class="col-sm-3 col-form-label">姓名</label>
<div class="col-sm-9">
<!-- 指定姓名输入框的字段名为 name -->
<input type="text" name="name" class="form-control" id="inputName">
</div>
</div>
<div class="form-group row">
<label for="inputAge" class="col-sm-3 col-form-label">年龄</label>
<div class="col-sm-9">
<!-- 指定年龄输入框的字段名为 age -->
<input type="number" name="age" class="form-control" id="inputAge">
</div>
</div>
<div class="form-group row">
<div class="col-sm">
<button type="submit" class="btn btn-primary">登录</button>
</div>
</div>
</div>
</div>
</form>
{% endblock %}
```

该模板所继承的父级模板源码如下：

```
<!-- 第 3 章/aiohttp_submitform/tpls/layout.html -->
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>{{ title }}</title>
<link rel="stylesheet" href="/node_modules/bootstrap/dist/css/bootstrap.min.css">
<script src="/node_modules/jquery/dist/jquery.min.js"></script>
<script src="/node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
</head>
<body>
{% block body %}
{% endblock %}
</body>
</html>
```

与/login 页面对应的处理函数源码如下：

```
@routes.post("/login")
@aiohttp_jinja2.template("login.html")
async def login(request: web.Request):
    #读取表单数据
    user = await request.post()
    return dict(title="登录结果", user=user)
```

通过 request.post 函数解析前端以 post 方式传来的数据，并将数据传给模板，由模板进行渲染，与 /login 对应的模板文件源码如下：

```
<!-- 第 3 章/aiohttp_submitform/tpls/login.html -->
{ % extends "layout.html" % }
{ % block body % }
<div class = "card" style = "margin: 2rem auto 0 auto;width:300px;">
<div class = "card - header">接收的数据</div>
<div>
<table class = "table table - bordered">
<tbody>
<tr>
<td>姓名</td>
<td style = "width:200px;">{{ user.name }}</td>
</tr>
<tr>
<td>年龄</td>
<td>{{ user.age }}</td>
</tr>
</tbody>
</table>
</div>
</div>
{ % endblock % }
```

6min

3.6 文件上传

本节计划实现一个上传图片后将此图片呈现在页面中的过程。

上传表单如图 3-20 所示。

图 3-20 文件上传表单

选择一张图片，上传后效果如图 3-21 所示。

图 3-21 文件上传后的效果

上传文件表单的源码如下：

```
<!-- 第 3 章/aiohttp_upload_file/tpls/index.html -->
{% extends "layout.html" %}
{% block body %}
<!-- 指定提交表单的方式为 post,提交的目标页面是 /upload,
并且编码方式为 multipart/form-data,上传文件时,这几项
必须这样配置,否则后端将无法正确接收数据 -->
<form method="post" action="/upload" enctype="multipart/form-data">
<div class="card" style="margin: 2rem auto 0 auto;width:400px;">
<div class="card-header">上传图片</div>
<div class="card-body">
<div class="form-group row">
<label for="inputName" class="col-sm-2 col-form-label">图片</label>
<div class="col-sm-10">
<!-- 指定输入框的类型为 file,这会在
页面中呈现一个浏览本地文件的按钮 -->
<input type="file" name="file" id="inputName">
</div>
</div>
<div class="form-group row">
<div class="col-sm">
<button type="submit" class="btn btn-primary">上传</button>
</div>
</div>
</div>
</div>
</form>
{% endblock %}
```

则对应的服务端处理逻辑源码如下：

```
"""
第 3 章/aiohttp_upload_file/server.py
"""
from aiohttp import web
import os, datetime, aiofile, aiohttp_jinja2, jinja2

APP_ROOT = os.path.dirname(__file__)
app = web.Application()
aiohttp_jinja2.setup(
    app,
    loader = jinja2.FileSystemLoader(
        os.path.join(os.path.dirname(__file__), "tpls")
    )
)
routes = web.RouteTableDef()


@routes.get('/')
@aiohttp_jinja2.template("index.html")
async def index(request: web.Request):
    return dict(title = "首页")


@routes.post("/upload")
@aiohttp_jinja2.template("upload.html")
async def login(request: web.Request):
    #解析前端上传的数据
    data = await request.post()
    #根据表单字段名从上传数据中取得文件对象,与< input name = "file">对应
    file_object = data['file']
    #用当前时间当作要保存的文件名
    file_name = f"{datetime.datetime.now().timestamp()}"
    #以写入二进制数据(wb)的方式打开文件
    file = await aiofile.open_async(
        os.path.join(APP_ROOT, "uploads", file_name), "wb"
    )
    #将数据读取出来并保存到目标文件中
    await file.write(file_object.file.read())
    #关闭文件 IO
    await file.close()
    #将该文件在网站中的路径传给模板
    return dict(title = "上传结果", file_path = f"/uploads/{file_name}")


app.add_routes(routes)
app.router.add_static("/node_modules", os.path.join(APP_ROOT, "node_modules"))
app.router.add_static("/uploads", os.path.join(APP_ROOT, "uploads"))
web.run_app(app)
```

实现的基本逻辑是先解析上传的数据，从数据库获得文件对象，并将文件对象的内容读取出来，再以写入二进制数据的方式保存到指定的文件中，最后将保存后的文件在网站中的路径传给模板，由模板渲染出来。这里我们使用当前时间对文件进行重命名，目的是保证上传的文件名不重复，在实际使用中，这种方式并不是最可靠的，在高并发的情况下，有可能存在同一时间上传多个文件，所以在实际开发工作中，为了保证文件名的唯一性，我们可以把时间与用户 id 关联起来形成真正唯一的文件名。保证文件名唯一的方式还有很多种，例如使用 uuid，不管使用哪种方式，我们的目标是保证文件名唯一。

在这段代码里用到了 aiofile，这是在 1.5 节文件异步 IO 中所实现的库，如果还有印象可重复利用该代码，如果没有印象需要回去再看看那一节。

3.7 Session

会话(Session)是一种在服务器端缓存用户数据的机制，主要用于保存用户的登录信息。如果要在 AIOHTTP 中支持 Session，需要第三方库 aiohttp_session，用命令 pip install aiohttp_session 进行安装。

在使用 Session 功能之前需要先为应用配置 Session，代码如下：

```
aiohttp_session.setup(app, aiohttp_session.SimpleCookieStorage())
```

然后在请求处理函数中使用 Session 的代码如下：

```
session = await aiohttp_session.get_session(request)
```

可运行的完整示例代码如下：

```
"""
第 3 章/aiohttp_session_counter/server.py
"""
from aiohttp import web
import aiohttp_session

routes = web.RouteTableDef()


@routes.get('/')
async def index(request: web.Request):
    session = await aiohttp_session.get_session(request)
    session['count'] = (session['count'] if 'count' in session else 0) + 1
    return web.Response(text = f"Count is {session['count']}")


app = web.Application()
app.add_routes(routes)
aiohttp_session.setup(app, aiohttp_session.SimpleCookieStorage())
web.run_app(app)
```

这段程序实现了当前用户的访问计数，用户每刷新一次页面，则计数加 1，如图 3-22 所示。

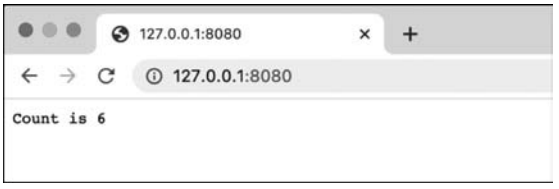

图 3-22　Session 计数器

如果希望以加密的方式在浏览器与服务器之间传输 Session ID，则可使用 aiohttp_session.cookie_storage.EncryptedCookieStorage 进行实现，代码如下：

```
"""
第 3 章/aiohttp_ssession_counter/server.py
"""
import base64

from aiohttp import web
from aiohttp_session import setup, get_session
from aiohttp_session.cookie_storage import EncryptedCookieStorage
from cryptography import fernet

routes = web.RouteTableDef()


@routes.get('/')
async def index(request: web.Request):
    session = await get_session(request)
    session['count'] = \
        (session['count'] if 'count' in session else 0) + 1
    return web.Response(text = f"Count is {session['count']}")


app = web.Application()
app.add_routes(routes)
fernet_key = fernet.Fernet.generate_key()
# secret_key 必须是 URL 安全的、base64 编码的 32 字节数据
# URL 安全是指不能包括 '+' 与 '/',在实际使用中会使用 '-' 代替 '+',使用'_' 代替 '/'
secret_key = base64.URLsafe_b64decode(fernet_key)
setup(app, EncryptedCookieStorage(secret_key))
web.run_app(app)
```

与该示例配套的 Dockerfile 内容如下：

```
# 第 3 章/aiohttp_ssession_counter/Dockerfile
FROM python:3.8 - slim
RUN pip install - i https://pypi.tuna.tsinghua.edu.cn/simple \
    aiohttp cryptography aiohttp_session
```

配套的 docker-compose.yml 文件内容如下：

```
#第 3 章/aiohttp_ssession_counter/docker - compose.yml
version: "3"

services:
  web:
    build: .
    ports:
      - "8080:8080"
    restart: always
    volumes:
      - ".:/opt/"
    working_dir: "/opt/"
    command: python3 ./server.py
```

3.8 HTTP 客户端

AIOHTTP 库实现了完整而健壮的 HTTP 客户端。在 1.8 节异步 HTTP 客户端中，我们已经介绍过如何实现 HTTP 客户端，但是 HTTP 的标准非常多，如果按 HTTP 协议去逐个实现，工作量也是非常庞大的。但如果使用 AIOHTTP 来作为开发框架，则可以直接使用已经实现的、功能完整的 HTTP 客户端。

在不使用 AIOHTTP 作为 Web 开发框架的情况下，是否也可以使用 AIOHTTP 的客户端库呢？当然可以，AIOHTTP 的客户端库是一套功能完整的、独立实现的库，可以应用在任何 Python 异步编程中。

接下来使用 AIOHTTP 的客户端去连接第 3 章/aiohttp_session_counter 项目，客户端代码如下：

```
session = aiohttp.ClientSession() #创建客户端会话上下文
response = await session.get('http://127.0.0.1:8080') #连接服务器
print(f"[{time.strftime('%X')}] {await response.text()}") #读取返回数据
await session.close() #关闭会话
```

AIOHTTP 的客户端还可以模拟 Cookie 机制，这意味着 AIOHTTP 客户端可以很方便地用于模拟登录等其他复杂的应用场景，下面代码演示了 AIOHTTP 客户端对服务器端会话(Session)机制的支持：

```
"""
第 3 章/aiohttp_client/app.py
"""
import asyncio, time

import aiohttp
```

```
async def main():
    session = aiohttp.ClientSession(
        #使用最简单的 Cookie 机制
        cookie_jar=aiohttp.CookieJar(unsafe=True)
    )
    for i in range(10):
        response = await session.get('http://127.0.0.1:8080')
        print(f"[{time.strftime('%X')}] {await response.text()}")
        #每次请求后休眠 1s
        await asyncio.sleep(1)
    await session.close()


if __name__ == '__main__':
    asyncio.run(main())
```

代码运行的结果如图 3-23 所示。

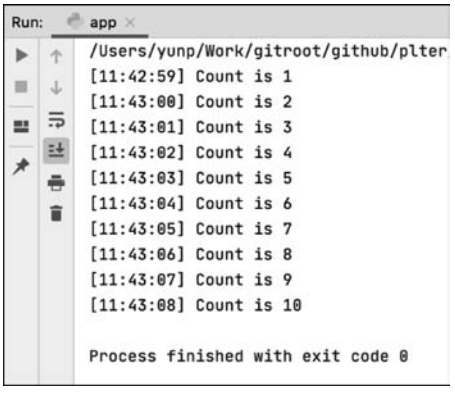

图 3-23　AIOHTTP 客户端对服务器端 Session 的支持

在使用 AIOHTTP 客户端时，只需创建一个 ClientSession 实例，之后可以使用该实例去执行所有的请求任务，强烈推荐这种方法，如果是同时请求多个网站，则需要分别为每个网站创建单独的 ClientSession 实例。

3.9　HTTPS 支持

互联网技术已经发展了很多年，至今在世界范围内有无数的网站，今天人类的生活已经很难完全脱离互联网。在我们享受科技的同时，也担心隐私泄露等问题，作为一家网站，有义务保护用户隐私，所以今天的世界要求每一家网站都通过 HTTPS 来服务用户。

每一家云主机服务商都提供免费的 HTTPS 证书，通常有效期为一年，到期后网站维护人员应重新申请证书并进行部署。所以将网站 HTTPS 化并不会增加成本，作为网站的运营者和开发者，更加有义务为用户提供安全的服务。

如果网站以 HTTP 直接部署，则用户在访问时浏览器会提示用户这是不安全的链接，如图 3-24 所示。

用户在访问这类网站时就会明白，隐私信息可能会泄露给第三方（例如木马、监控软件等），遇到这种情况用户一般会直接离开该网站，对于网站来说，也就会遇到极大的推广障碍。如果采用的是 HTTPS 的安全链接，则浏览器会提示安全或者不提示，如图 3-25 所示。

图 3-24 不安全的链接

图 3-25 安全链接

在 AIOHTTP 框架中，配置支持 HTTPS 非常方便，示例代码如下：

```
"""
第 3 章/aiohttps/server.py
"""
from aiohttp import web
import ssl, os

routes = web.RouteTableDef()


@routes.get('/')
async def index(request: web.Request):
    return web.Response(text = "Hello World")


app = web.Application()
app.add_routes(routes)
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
# 加载网站证书文件
ssl_context.load_cert_chain(
    os.path.join(os.path.dirname(__file__), "ssl", "cert.pem"),
    os.path.join(os.path.dirname(__file__), "ssl", "cert.key"),
)
web.run_app(app, port = 443, ssl_context = ssl_context)
```

在该示例中，使用两个文件，一个是证书文件，另一个是密钥文件，在你申请 HTTPS 证书完成后可以获得这两个文件。

AIOHTTP 创建的 HTTPS 服务器默认使用端口 8443，而 HTTPS 的默认端口号是 443，所以如果想使用 HTTPS 默认端口（访问时域名后可省略端口号），须手动指定使用 443 端口。