Приглашаем преподавателей поучаствовать в интервью с нашей командой исследований. Если вы готовы участвовать, пожалуйста, заполните форму. Приглашение на интервью.

Урок aiohttp

Использование aiohttp для создания веб-сервисов и запросов

1. Введение

Как мы уже выяснили на прошлых уроках, запросы в интернете, это то место, где выигрыш от использования асинхронных технологий будет максимальный. Поэтому большую часть оставшегося времени мы посветим тому, что будем знакомиться с библиотеками, которые направлены именно на эту предметную область. И начнем мы с библиотеки `aiohttp`.

`aiohttp` — это одна из самых популярных асинхронных библиотек для работы с HTTP в Python, которая основана на `asyncio`. В отличие от библиотеки `requests`, она предоставляет возможности для создания не только http-клиентов, но и серверов, а также позволяет работать с web-сокетами, что делает её универсальным и довольно ультимативным инструментом для решения широкого набора задач.

Для начала установим библиотеку `aiohttp` с помощью pip:

```
pip install aiohttp
```

2. Создание HTTP-клиентов

Начнем с того, чтобы писать наши первые http-запросы. В общем случае, код для выполнения запросов, чуть сложнее, чем тот, который вы писали с использованием библиотеки `requests`, потому что до отправки запросов нам надо будет создать сессию, которая будет обрабатывать все наши запросы. Но не переживайте, это не так сложно, как кажется. Давайте посмотрим на пример:

```
import aiohttp
import asyncio

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            print(f"Статус: {response.status}")
            print(f"Тело ответа: {await response.text()}")

asyncio.run(fetch('http://example.com'))
```

Что мы тут делаем:

1. Создаём сессию, которая будет обрабатывать наши запросы.
2. Выполняем запрос.
3. Выводим статус и тело ответа.

Из нового — использование асинхонного менеджера контекста `async with`, который работает точно также как и синхронный, но для асинхронных объектов. Если мы хотим получить такое поведение для своих объектов нужно определить магические методы `__aenter__` и `__aexit__`.

Из важного, стоит обратить внимание, что `response.text()` и `response.json()` тоже асинхронные методы, как и `response.status`. Поэтому нам нужно использовать `await` перед этими методами, а также вызывать их внутри `async with`, потому что за пределами работы контекстного менеджера запроса мы не можем быть уверены, что объект ответа еще существует.

Работа с get и post параметрами устроена в `aiohttp` точно также, как и в библиотеке `requests`: мы делаем словарь с параметрами и передаем его в нужный параметр при создании запроса.

Давайте посмотрим на пример работы с POST-запросом:

```
import aiohttp
import asyncio

async def post_data(url, data):
    async with aiohttp.ClientSession() as session:
        async with session.post(url, json=data) as response:
```

```
            print(f"Статус: {response.status}")
            print(f"Тело ответа: {await response.text()}")

data = {"ключ": "значение"}
asyncio.run(post_data('http://example.com/api', data))
```

И еще на пример работы с заголовками:

```
import aiohttp
import asyncio

async def fetch_with_headers(url):
    headers = {'User-Agent': 'Mozilla/5.0'}
    async with aiohttp.ClientSession() as session:
        async with session.get(url, headers=headers) as response
            print(f"Статус: {response.status}")
            print(f"Заголовки: {response.headers}")
            print(f"Тело ответа: {await response.text()}")

asyncio.run(fetch_with_headers('http://example.com'))
```

В качестве последнего примера на http-клиент давайте рассмотрим обработку таймаута запроса. В `aiohttp` мы можем установить таймаут на весь запрос или на отдельные его части, например на подключение к серверу или чтение ответа.

Для этого мы используем класс `aiohttp.ClientTimeout`, который принимает следующие параметры:

- connect — таймаут на подключение к серверу
- read — таймаут на чтение ответа
- total — общий таймаут на весь запрос

```
import aiohttp
import asyncio

async def fetch_with_timeout(url):
    timeout = aiohttp.ClientTimeout(total=2)  # Таймаут на весь
    async with aiohttp.ClientSession(timeout=timeout) as sessior
        try:
            async with session.get(url) as response:
                print(f"Статус: {response.status}")
                print(f"Тело ответа: {await response.text()}")
        except asyncio.TimeoutError:
            print("Запрос превысил время ожидания")
```

```
asyncio.run(fetch_with_timeout('http://example.com'))
```

3. Асинхронный HTTP-сервер

Асинхронный HTTP-сервер `aiohttp` позволяет создавать HTTP-сервера, которые могут обрабатывать запросы асинхронно. Создание таких приложений в общем случает достаточно сильно напоминает работу с таким синхронным сервером, как `flask`. Хотя `aiohttp` не предоставляет никаких дополнительных функций, таких как, например, шаблонизатор.

Чтобы создать наш первый сервер на `aiohttp`, нам нужно сделать три простых шага:

- Создать экземпляр класса `web.Application` — само приложение
- Определить маршруты и соответствующие им обработчики
- Запустить приложение на указанном хосте и порте

```
from aiohttp import web

async def handle(request):
    return web.Response(text="Hello, World!")

app = web.Application()
app.add_routes([web.get('/', handle)])

if __name__ == '__main__':
    web.run_app(app)
```

Приложение по умолчанию будет запущено на порту 8080 на локальном ip-адресе, но можно указать и конкретные зеначения порта и хоста через необязательные параметры метода `web.run_app` — `host` и `port`.

Разумеется, наше приложение может обрабатывать не только get-запросы, но и любые другие.

```
from aiohttp import web

async def handle_post(request):
    data = await request.json()
```

```
    return web.json_response({"message": "Data received", "data'

app = web.Application()
app.add_routes([web.post('/submit', handle_post)])

if __name__ == '__main__':
    web.run_app(app)
```

Работать с get-параметрами:

```
from aiohttp import web

async def handle(request):
    name = request.rel_url.query.get('name', 'World')
    return web.Response(text=f"Hello, {name}!")

app = web.Application()
app.add_routes([web.get('/', handle)])

if __name__ == '__main__':
    web.run_app(app)
```

И еще многое другое. Давайте посмотрим чуть более комплексный пример, который покажет, как работать с post-параметрами, заголовками, а также скачивать и отправлять файлы. Для последнего нам понадобится еще библиотека `aiofiles` (которую тоже необходимо установить из Pypi).

```
from aiohttp import web
import aiofiles
import os

# Обработчик для работы с POST параметрами
async def handle_post_params(request):
    data = await request.post()
    name = data.get('name', 'Anonymous')
    age = data.get('age', 'Unknown')
    return web.json_response({'message': f'Hello, {name}! Your a

# Обработчик для работы с заголовками
async def handle_headers(request):
    user_agent = request.headers.get('User-Agent', 'Unknown')
    return web.json_response({'User-Agent': user_agent})

# Обработчик для загрузки файлов
```

```python
async def handle_upload_file(request):
    reader = await request.multipart()
    field = await reader.next()
    assert field.name == 'file'
    filename = field.filename

    # Сохранение файла на диск
    size = 0
    async with aiofiles.open(os.path.join('uploads', filename),
        while True:
            chunk = await field.read_chunk()
            if not chunk:
                break
            size += len(chunk)
            await f.write(chunk)
    return web.json_response({'filename': filename, 'size': size

# Обработчик для отправки файлов
async def handle_send_file(request):
    filename = request.rel_url.query.get('filename')
    if not filename:
        return web.Response(text='Filename not provided', status

    filepath = os.path.join('uploads', filename)
    if not os.path.exists(filepath):
        return web.Response(text='File not found', status=404)

    return web.FileResponse(filepath)

app = web.Application()
app.add_routes([
    web.post('/post-params', handle_post_params),
    web.get('/headers', handle_headers),
    web.post('/upload', handle_upload_file),
    web.get('/download', handle_send_file),
])

if __name__ == '__main__':
    os.makedirs('uploads', exist_ok=True)
    web.run_app(app, host='127.0.0.1', port=8080)
```

4. Работа с WebSocket

Последней сильной стороной `aiohttp` является поддержка технологии `WebSocket`, которая позволяет реализовать двустороннюю связь между клиентом и сервером. Тут мы не будем очень подробно останавливаться, просто рассмотрим базовый пример.

WebSocket сервер

```
from aiohttp import web

async def websocket_handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)

    async for msg in ws:
        if msg.type == web.WSMsgType.TEXT:
            await ws.send_str(f"Message received: {msg.data}")

    return ws

app = web.Application()
app.add_routes([web.get('/ws', websocket_handler)])

if __name__ == '__main__':
    web.run_app(app)
```

Тут мы также используем создание веб-приложения, добавляем маршрут для обработки веб-сокетов и создаем обработчик, который будет отправлять обратно клиенту сообщение, которое он отправил.

WebSocket клиент

```
import aiohttp
import asyncio

async def websocket_client():
    session = aiohttp.ClientSession()
    async with session.ws_connect('http://localhost:8080/ws') as
        await ws.send_str("Hello, server")
        async for msg in ws:
            if msg.type == aiohttp.WSMsgType.TEXT:
```

```
                print(f"Message from server: {msg.data}")
                break

    await session.close()

asyncio.run(websocket_client())
```

Тут мы создаём сессию для подключения к серверу, а затем отправляем сообщение и обрабатываем ответ.

5. Заключение

На этом уроке мы познакомились с библиотекой `aiohttp`, изучили основные концепции и научились использовать её для создания асинхронных HTTP-клиентов и серверов. Мы также прикоснулись к тому, как работать с WebSocket (но эта тема сама по себе очень большая и интересная).

Информационные материалы представлены Яндексом и АНО ДПО «Образовательные технологии Яндекса» в соответствии с Условиями использования. Не является образовательной услугой.

© 2018 – 2025 Яндекс

Справка