异步爬虫

```
# -*- coding: utf-8 -*-
# @Author  : quanchenliu
# @Time    : 2024/1/28
# @Function:
```

一、异步爬虫概述

爬虫是IO密集型任务，当我们发出一个请求之后，程序必须等待网站返回响应，才能接着运行，而在等待响应的过程中，整个爬虫程序是一直在等待的，实际上没有做任何事情。对此情况，提出优化方案**异步爬虫**。

要实现异步爬虫，那就与**协程**脱不了关系。注意区别：单线程、多线程、协程

二、协程的基本原理

1、基础知识：

- 阻塞：程序未得到所需的计算机资源而被挂起的状态。
- 非阻塞：程序在等待某操作的过程中，自身不被阻塞，可以继续执行其他任务，则称该程序在操作上是非阻塞的。
    - 非阻塞不总是存在，只有当程序封装的级别可以囊括独立的子程序时，程序才可能出现非阻塞状态；
    - 非阻塞因阻塞的存在而存在，因为阻塞的存在，所以我们需要非阻塞。
- 同步：同步意味着有序。
- 异步：异步意味着无序。
- 多进程：同一时间并行执行多个任务。
- 协程：运行在用户态的轻量级线程。
    - 协程有自己的寄存器上下文、栈帧，本质上是一个单线程；
    - 相对于多线程而言，没有线程上下文切换的开销，也没有原子操作锁定及同步的开销。

2、协程的用法：

(1) `asyncio`库：

Python中使用协程最常用的库就是`asyncio`。我们首先需要了解几个相关的概念：

- `event_loop`：事件循环，相对于无限循环——当满足发生条件时，就调用对应的处理方法；
- `coroutine`：协程对象类型；

- `task`：对协程对象的进一步封装；比协程对象多了运行状态
- `async`：用于定义一个方法，这个方法在被调用时不会执行，而是会返回一个协程对象；

（2）定义协程：

我们使用 `async` 关键字定义一个方法，调用该方法会返回一个协程对象。然后，**调用 `get_event_loop` 方法创建一个事件循环 `loop`，并调用 `loop` 对象 `run_until_complete` 方法将协程对象注册到事件循环中**。此后，才可以执行使用 `async` 关键字定义的方法。

事实上，**在将协程对象注册的过程中，执行了这样一个操作：将协程对象 `coroutine` 封装成 `task` 对象**。对此，我们可以显式的声明。**调用 `create_task` 方法**将协程对象封装成 `task` 对象。然后将 `task` 对象添加进事件循环中执行。

还有另外一种定义 `task` 对象的方法：**直接调用 `asyncio` 库中的 `ensure_future` 方法**。这样做的好处是，在声明事件循环 `loop` 之前，就提前定义好 `task` 对象。

```
"""
 -*- coding : utf-8 -*-
 @Author    : quanchenliu
 @Time      : 2024/1/30
 @Function  : 协程的用法（event_loop、coroutine、task（create_task）、task
（ensure_future））
"""
import asyncio

async def execute(x):
    print('Number:', x)

def main():
    # 有三种创建 task 对象的方法：create_task(显式) 和 ensure_future(显式)、直
接调用run_until_complete方法(隐式)
    method = {'MODE': 'ensure_future'}
    coroutine = execute(1)                          # 调用 execute 方法，但
并不执行，而是返回一个 coroutine 协程对象
    print('Coroutine:', coroutine)                  # Coroutine: <coroutine
object execute at 0x000001E2BD4C26C0>

    if method['MODE'] == 'Notask':
        # 创建一个事件循环 loop，调用 loop 对象的 run_until_complete 方法，将
协程对象注册到了事件循环中，并执行 execute 方法
        loop = asyncio.get_event_loop()
        loop.run_until_complete(coroutine)          # Number: 1

    if method['MODE'] == 'create_task':
        loop = asyncio.get_event_loop()
        task = loop.create_task(coroutine)
        # print('Task:', task)
        loop.run_until_complete(task)               # 等价于：
loop.run_until_complete(coroutine)
        # print('Task:', task)
```

```
    if method['MODE'] == 'ensure_future':
        # 使用 ensure_future 方法创建 task 对象----即使还没有声明 loop 也可以提前定义好 task 对象
        task = asyncio.ensure_future(coroutine)
        # print('Task:', task)
        loop = asyncio.get_event_loop()
        loop.run_until_complete(task)
        # print('Task:', task)


if __name__ == "__main__":
    main()
```

(3) 多任务协程:

在上面的例子中，只执行了一次请求，如果想执行多次请求，应该怎么做呢？我们首先思考我们的需求：

- [ ] 首先，我们需要**同时执行多个请求**——需要定义一个 `task` 列表，并在事件循环中使用 `wait` 方法执行；
- [ ] 其次，要实现异步处理，得先有**挂起**操作——由此引入 `await` 方法；
- [ ] 最后，仅仅将涉及 `IO` 的操作封装进 `async` 定义方法里是不够的，只有使用支持异步操作的请求方式才能实现真正的异步——由此引入 `aiohttp` 库；

```
"""
 -*- coding : utf-8 -*-
 @Author    : quanchenliu
 @Time      : 2024/1/30
 @Function  : 协程实现
"""
import asyncio
import time
import aiohttp

async def get(url):
    session = aiohttp.ClientSession()
    response = await session.get(url)
    await response.text()
    await session.close()
    return response

async def request():
    url = 'https://httpbin.org/delay/5'
    print('Waiting for ', url)
    response = await get(url)           # await 后面可以跟一个协程对象，而不能跟 requests 返回的 Response 对象
    print('Get response from ', url, 'response', response)

def main():
    start = time.time()
```

```
    tasks = [asyncio.ensure_future(request()) for _ in range(10)]
    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(tasks))

    end = time.time()
    print('Cost time:', end - start)

if __name__ == "__main__":
    main()
```

（4）await 的使用：

`requests`返回的`Response`对象不能和`await`一起使用，通过**查阅官方文档**可知，`await`后面的对象必须是如下格式之一：

- 一个原生协程对象（如：`coroutine`、`task`）；
- 一个由`type.coroutine`修饰的生成器（这个生成器可以返回协程对象）；
- 由一个包含`__await__`方法对象返回的一个迭代器。

三、`aiohttp`的使用

`aiohttp`是一个支持异步请求的库，它和`asyncio`配合使用，可以使我们非常方便地实现异步请求。

`asyncio`实现了对TCP、UDP、SSL协议的异步操作，但是对于HTTP请求来说，就需要使用`aiohttp`请求实现了。`aiohttp`是一个基于`asyncio`的异步HTTP 网络模块，既提供了服务端，提供了客户端。我们可以利用服务端搭建一个支持异步请求的服务器，可以利用客户端发起异步请求。

1、基本实例：

```
import aiohttp
import asyncio

async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text(), response.status

async def main():
    url = 'https://cuiqingcai.com'
    html, status = await get(url)
    print(f'html: {html[:100]}...')
    print(f'status: {status}')

if __name__ == '__main__':
    asyncio.run(main())                    # 总会出现 RuntimeError: Event loop
is closed 的错误
    ''' 一种可行的解决方案是将 asyncio.run(协程主函数名())修改为:
```

```
        loop = asyncio.get_event_loop()
        loop.run_until_complete(协程主函数名())'''
```

通过上述的基本实例可以发现，`aiohttp` 请求和之前的请求方法有明显的区别：

- 除了必须引入`aiohttp` 这个库之外，还**必须引入** `asyncio` **库。**
- 在定义每个异步方法的时候，都必须添加 `async` 关键字进行修饰。
- with as 语句同样需要 `async` 关键字进行修饰。
- 对于一些返回协程对象的操作，需要添加 `await` 进行修饰。
- 要运行上述代码，必须启用事件循环，而事件循环需要使用 `asyncio` 库，然后调用 `run_until_complete` 方法来执行。

2、URL参数设置：

对 URL 参数的设置，我们可以借助 params 参数，传入一个字典即可

```
params = {                      # 对 URL 参数的设置，我们可以借助 params 参数，传入一个字典即可
        'name': 'germey',
        'age': 25
}
async with aiohttp.ClientSession() as session:
    async with session.get('https://httpbin.org/get', params=params) as response:
        print(await response.text())
```

3、其他请求类型：

`aiohttp` 还支持其他请求类型————— POST、PUT、DELETE，下面以 POST 为例，其余请求类型发使用与之相同。

对于 POST表 单提交，其对于请求头中的 Content-Type 为 application/x-www-form-urlencoded，我们可以使用 data 参数实现。

对于 POST JSON 数据提交，其对应请求头中的 Content-Type 为 application/json，我们要将 post 方法里的 data 参数修改为 json 参数。

```
import aiohttp
import asyncio

async def main():
    data = {'name': 'germey', 'age': 25}
    async with aiohttp.ClientSession() as session:
        async with session.post('https://httpbin.org/post', data=data) as response:
            print(await response.text())

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

4、响应结果获取：

有些响应体的字段前面需要加 await ，有些则不需要。其原则是，如果返回的是一个协程对象（如 `async` 修饰的方法），那么前面就需要加await方法。具体可以参数 `aiohttp` 的API：https://docs.aiohttp.org/en/stable/client_reference.html

```
import aiohttp
import asyncio

async def main():
    data = {'name': 'germey', 'age': 25}
    async with aiohttp.ClientSession() as session:
        async with session.post('https://httpbin.org/post', data=data) as
response:
            print('status:', response.status)                 # 状态码
            print('headers:', response.headers)               # 响应头
            print('body:', await response.text())             # 响应体
            print('bytes:', await response.read())            # 响应体二
进制内容
            print('json:', await response.json())             # 响应体
JSON 结果

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

5、超时设置：

我们可以借助 `ClientTimeout` 对象设置超时。

```
async def main():
    timeout = aiohttp.ClientTimeout(total=0.1)                  # 设置
超时时间为 1s
    try:
        async with aiohttp.ClientSession(timeout=timeout) as session:
            async with session.get('https://httpbin.org/get') as response:
                print('status:', response.status)
    except Exception as e:
        print('Error: ', repr(e))
```

6、并发限制：

借助 Semaphore 创建一个信号量对象 semaphore，用于**控制最大并发量**。将 semaphore 放入爬取方法中，并使用 `async with` 语句将 semaphore 作为上下文对象即可。

在 main() 方法中，我们声明了10000个 task，并将其**传递给 gather 方法执行**。

在Python中，`*` **是解包操作符**。在 `asyncio.gather` 中，它接受多个可迭代对象作为参数。

通过使用 `*scrape_index_tasks`，实际上是将列表 `scrape_index_tasks` 解包为单独的参数，传递给 `asyncio.gather`。

如果不使用 `*`，将传递整个列表作为单个参数，而不是将列表中的元素作为单独的参数。

在这个特定的情境中，我们希望 `asyncio.gather` 接受多个任务作为参数，而不是一个包含任务的列表。

所以，`await asyncio.gather(*scrape_index_tasks)` 的写法**确保将列表中的每个任务都作为独立的参数传递给 `asyncio.gather`**

```
import asyncio
import aiohttp

CONCURRENCY = 5                                 # 声明爬虫最大并发数: 5
URL = 'https://www.baidu.com'
semaphore = asyncio.Semaphore(CONCURRENCY)  # 创建一个信号量对象 semaphore，用于控制最大并发量
session = None

async def Scrape_api():
    async with semaphore:                       # 将 semaphore 放入爬取方法中，并使用 async with 语句将 semaphore 作为上下文对象
        print('Scraping: ', URL)
        async with session.get(URL) as response:
            await asyncio.sleep(1)
            return await response.text()

async def main():
    global session
    session = aiohttp.ClientSession()
    scrape_index_tasks = [asyncio.ensure_future(Scrape_api()) for _ in range(10000)]
    await asyncio.gather(*scrape_index_tasks)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

四、总结

如何定义并使用异步爬虫：

1. 定义**协程或任务列表**：使用 `async` 关键字定义一个方法，调用该方法会返回一个协程对象

```
# 定义 task 列表，其中 request() 是使用 async 关键字定义的方法
tasks = [asyncio.ensure_future(request()) for _ in range(10)]
```

2. 定义**事件循环**：调用 `get_event_loop` 方法**创建一个事件循环** `loop`，并调用 `loop` 对象 `run_until_complete` 方法**将协程对象或任务列表注册到事件循环中**

```
loop = asyncio.get_event_loop()
loop.run_until_complete(tasks)
```

3. 使用**异步请求**：通过定义协程或任务列表、定义事件循环等操作仍然不能实现异步操作。实际上，**只有使用异步操作的请求才能实现真正的异步**，一个请求示例如下：

```
async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text(), response.status,
response.json()
```

4. 在实现异步的基础上，我们还需要对异步操作进行**并发限制**：借助 Semaphore 创建一个**信号量对象 semaphore**，用于**控制最大并发量**。将 semaphore 放入爬取方法中，并使用 `async with` 语句将 semaphore 作为上下文对象即可。

```
CONCURRENCY = 5                                 # 声明爬虫最大并发数：5
semaphore = asyncio.Semaphore(CONCURRENCY)  # 创建一个信号量对象
semaphore，用于控制最大并发量

async def Scrape_api():
    async with semaphore:                       # 将 semaphore 放入爬取方法
中，并使用 async with 语句将 semaphore 作为上下文对象
        print('Scraping: ', URL)
        async with aiohttp.ClientSession() as session:
            async with session.get(URL) as response:
                await asyncio.sleep(1)
                return await response.text()

async def main():
    scrape_index_tasks = [asyncio.ensure_future(Scrape_api()) for _ in
range(10000)]
    await asyncio.gather(*scrape_index_tasks)
```

5. **await 与 async 关键字**的使用：
    - `async` **关键字**：`async` 用于定义异步函数，它可以包含 `await` 表达式，使得函数的执行可以在 `await` 处暂停，并在异步操作完成后继续执行
    - `await` **关键字**：`await` 用于在异步函数内部等待异步操作的完成。当遇到 `await` 表达式时，异步函数会暂停执行，让出事件循环的控制权，直到 `await` 表达式的异步操作完成为止。

五、异步爬虫实战

```
import asyncio
import json
import time
import aiohttp
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %
(levelname)s: %(message)s')
INDEX_URL = 'https://spa1.scrape.center/api/movie/?limit={limit}&offset=
{offset}'
DETAIL_URL = 'https://spa1.scrape.center/api/movie/{id}'
LIMIT = 18
TOTAL_PAGE = 11
COUNTER = 5
semaphore = asyncio.Semaphore(COUNTER)
session = None
RESULTS_DIR = 'C:/Users/DELL/Desktop/python爬虫基础/3.AjaxTest/data'

async def save_data(data):
    name = data.get('name')
    data_path = f'{RESULTS_DIR}/{name}.json'
    json_data = {
        "电影名称": name,
        "电影种类": data.get('categories'),
        "制作国家": data.get('regions'),
        "电影评分": data.get('score'),
        "电影时长": data.get('minute'),
        "电影简介": data.get('drama'),
        "上映时间": data.get('published_at')
    }
    json.dump(json_data, open(data_path, 'w', encoding='utf-8'),
              ensure_ascii=False, indent=2)

async def Scrape_api(url):
    async with semaphore:
        try:
            logging.info('Scraping %s', url)
            async with session.get(url) as response:
                return await response.json()
        except aiohttp.ClientError:
            logging.error('Error occurred while scraping %s', url,
exc_info=True)

async def Scrape_index(page):
    url = INDEX_URL.format(limit=LIMIT, offset=LIMIT * (page - 1))
    return await Scrape_api(url)

async def Scrape_detail(id):
    url = DETAIL_URL.format(id=id)
```

```
    data = await scrape_api(url)
    await save_data(data)

async def main():
    start = time.time()

    global session
    session = aiohttp.ClientSession()
    scrape_index_tasks = [asyncio.ensure_future(scrape_index(page)) for
page in range(1, TOTAL_PAGE + 1)]
    results = await asyncio.gather(*scrape_index_tasks)

    ids = []
    for index_data in results:
        if not index_data: continue
        for item in index_data.get('results'):
            ids.append(item.get('id'))

    scrape_detial_tasks = [asyncio.ensure_future(scrape_detail(id)) for id
in ids]
    await asyncio.gather(*scrape_detial_tasks)
    await session.close()

    end = time.time()
    print('Using time:', end - start)


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```