

在上一课时我们介绍了异步爬虫的基本原理和 `asyncio` 的基本用法，另外在最后简单提及了 `aiohttp` 实现网页爬取的过程，这一可是我们来介绍一下 `aiohttp` 的常见用法，以及通过一个实战案例来介绍下使用 `aiohttp` 完成网页异步爬取的过程。

## aiohttp

前面介绍的 `asyncio` 模块内部实现了对 TCP、UDP、SSL 协议的异步操作，但是对于 HTTP 请求的异步操作来说，我们就需要用到 `aiohttp` 来实现了。

`aiohttp` 是一个基于 `asyncio` 的异步 HTTP 网络模块，它既提供了服务端，又提供了客户端。其中我们用服务端可以搭建一个支持异步处理的服务器，用于处理请求并返回响应，类似于 Django、Flask、Tornado 等一些 Web 服务器。而客户端我们就可以用来发起请求，就类似于 `requests` 来发起一个 HTTP 请求然后获得响应，但 `requests` 发起的是同步的网络请求，而 `aiohttp` 则发起的是异步的。

本课时我们就主要来了解一下 `aiohttp` 客户端部分的使用。

## 基本使用

### 基本实例

首先我们来看一个基本的 `aiohttp` 请求案例，代码如下：

```
import aiohttp
import asyncio
async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text(), response.status
async def main():
    async with aiohttp.ClientSession() as session:
        html, status = await fetch(session, 'https://cuiqingcai.com')
        print(f'html: {html[:100]}...')
        print(f'status: {status}')
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

在这里我们使用 `aiohttp` 来爬取了我的个人博客，获得了源码和响应状态码并输出，运行结果如下：

```
html: <!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<meta name="baidu-tc-verification" content=...
status: 200
```

这里网页源码过长，只截取输出了一部分，可以看到我们成功获取了网页的源代码及响应状态码 200，也就完成了一次基本的 HTTP 请求，即我们成功使用 `aiohttp` 通过异步的方式进行了网页的爬取，当然这个操作之前我们所讲的 `requests` 同样也可以做到。

我们可以看到其请求方法的定义和之前有了明显的区别，主要有如下几点：

- 首先在导入库的时候，我们除了必须要引入 `aiohttp` 这个库之外，还必须要引入 `asyncio` 这个库，因为要实现异步爬取需要启动协程，而协程则需要借助于 `asyncio` 里面的事件循环来执行。除了事件循环，`asyncio` 里面也提供了很多基础的异步操作。
- 异步爬取的方法的定义和之前有所不同，在每个异步方法前面统一要加 `async` 来修饰。
- `with as` 语句前面同样需要加 `async` 来修饰，在 Python 中，`with as` 语句用于声明一个上下文管理器，能够帮我们自动分配和释放资源，而在异步方法中，`with as` 前面加上 `async` 代表声明一个支持异步的上下文管理器。
- 对于一些返回 `coroutine` 的操作，前面需要加 `await` 来修饰，如 `response` 调用 `text` 方法，查询 API 可以发现其返回的是 `coroutine` 对象，那么前面就要加 `await`；而对于状态码来说，其返回值就是一个数值类型，那么前面就不需要加 `await`。所以，这里可以按照实际情况处理，参考官方文档说明，看看其对应的返回值是怎样的类型，然后决定加不加 `await` 就可以了。
- 最后，定义完爬取方法之后，实际上是 `main` 方法调用了 `fetch` 方法。要运行的话，必须要启用事件循环，事件循环就需要使用 `asyncio` 库，然后使用 `run_until_complete` 方法来运行。

注意在 Python 3.7 及以后的版本中，我们可以使用 `asyncio.run(main())` 来代替最后的启动操作，不需要显式声明事件循环，`run` 方法内部会自动启动一个事件循环。但这里为了兼容更多的 Python 版本，依然是显式声明了事件循环。

### URL 参数设置

对于 URL 参数的设置，我们可以借助于 `params` 参数，传入一个字典即可，示例如下：

```
import aiohttp
import asyncio
async def main():
    params = {'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())
if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())
```

运行结果如下：

```
{
  "args": {
    "age": "25",
    "name": "germey"
  },
  "headers": {
    "Accept": "**/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "Python/3.7 aiohttp/3.6.2",
    "X-Amzn-Trace-Id": "Root=1-5e85eed2-d240ac90f4ddd40b4723ef0"
  },
  "origin": "17.20.255.122",
  "url": "https://httpbin.org/get?name=germey&age=25"
}
```

这里可以看到，其实际请求的 URL 为 <https://httpbin.org/get?name=germey&age=25>，其 URL 请求参数就对应了 `params` 的内容。

### 其他请求类型

另外 `aiohttp` 还支持其他的请求类型，如 POST、PUT、DELETE 等等，这个和 `requests` 的使用方式有点类似，示例如下：

```
session.post('http://httpbin.org/post', data=b'data')
session.put('http://httpbin.org/put', data=b'data')
session.delete('http://httpbin.org/delete')
session.head('http://httpbin.org/get')
session.options('http://httpbin.org/get')
session.patch('http://httpbin.org/patch', data=b'data')
```

### POST 数据

对于 POST 表单提交，其对应的请求头的 `Content-type` 为 `application/x-www-form-urlencoded`，我们可以用如下方式来实现，代码示例如下：

```
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__':
    asyncio.get_event_loop().run_until_complete(main())

```

运行结果如下:

```

{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "age": "25",
    "name": "germey"
  },
  "headers": {
    "Accept": "**/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "18",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Python/3.7 aiohttp/3.6.2",
    "X-Amzn-Trace-Id": "Root=1-5e85f0b2-9017ea603a68dc285e0552d0"
  },
  "json": null,
  "origin": "17.20.255.58",
  "url": "https://httpbin.org/post"
}

```

对于 POST JSON 数据提交, 其对应的请求头的 Content-type 为 application/json, 我们只需要将 post 方法的数据参数改成 json 即可, 代码示例如下:

```

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

```

运行结果如下:

```

{
  "args": {},
  "data": "{\"name\": \"germey\", \"age\": 25}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "**/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "29",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "Python/3.7 aiohttp/3.6.2",
    "X-Amzn-Trace-Id": "Root=1-5e85f03e-c91c9a20c79b9780d7540"
  },
  "json": {
    "age": 25,
    "name": "germey"
  },
  "origin": "17.20.255.58",
  "url": "https://httpbin.org/post"
}

```

## 响应字段

对于响应来说, 我们可以用如下的方法分别获取响应的状态码、响应头、响应体、响应体二进制内容、响应体 JSON 结果, 代码示例如下:

```

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)
            print('headers:', response.headers)
            print('body:', await response.text())
            print('bytes:', await response.read())
            print('json:', await response.json())
if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())

```

运行结果如下:

```

status: 200
headers: <CIMultiDictProxy('Date': 'Thu, 02 Apr 2020 14:13:05 GMT', 'Content-Type': 'application/json', 'Content-Length': '503', 'Connection': 'keep-alive', 'Server': 'gunicorn/19.9.0')>
body: {
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "age": "25",
    "name": "germey"
  },
  "headers": {
    "Accept": "**/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "18",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "Python/3.7 aiohttp/3.6.2",
    "X-Amzn-Trace-Id": "Root=1-5e85f2f1-f55326ff5800b15886c8e029"
  },
  "json": null,
  "origin": "17.20.255.58",
  "url": "https://httpbin.org/post"
}
bytes: b'{"args": {}, "data": "", "files": {}, "form": {"age": "25", "name": "germey"}, "headers": {"Accept": "**/*", "Accept-Encoding": "gzip, deflate", "Content-Length": "18", "Content-T

```

这里我们可以看到有些字段前面需要加 await, 有的则不需要。其原则是, 如果其返回的是一个 coroutine 对象 (如 async 修饰的方法), 那么前面就要加 await, 具体可以看 aiohttp 的 API, 其链接为: [https://docs.aiohttp.org/en/stable/client\\_reference.html](https://docs.aiohttp.org/en/stable/client_reference.html).

## 超时设置

对于超时的设置, 我们可以借助于 ClientTimeout 对象, 比如这里我要设置 1 秒的超时, 可以这么来实现:

```

import aiohttp
import asyncio
async def main():

```

```

timeout = aiohttp.ClientTimeout(total=1)
async with aiohttp.ClientSession(timeout=timeout) as session:
    async with session.get('https://httpbin.org/get') as response:
        print('status:', response.status)
if __name__ == '__main__':
    asyncio.get_event_loop().run_until_complete(main())

```

如果在 1 秒之内成功获取响应的的话，运行结果如下：

```
200
```

如果超时的话，会抛出 `TimeoutError` 异常，其类型为 `asyncio.TimeoutError`，我们再进行异常捕获即可。

另外 `ClientTimeout` 对象声明时还有其他参数，如 `connect`、`socket_connect` 等，详细说明可以参考官方文档：[https://docs.aiohttp.org/en/stable/client\\_quickstart.html#timeouts](https://docs.aiohttp.org/en/stable/client_quickstart.html#timeouts)。

### 并发限制

由于 `aiohttp` 可以支持非常大的并发，比如上万、十万、百万都是能做到的，但这么大的并发量，目标网站是很可能在短时间内无法响应的，而且很可能瞬间将目标网站爬挂掉。所以我们需要控制一下爬取的并发量。

在一般情况下，我们可以借助于 `asyncio` 的 `Semaphore` 来控制并发量，代码示例如下：

```

import asyncio
import aiohttp
CONCURRENCY = 5
URL = 'https://www.baidu.com'
semaphore = asyncio.Semaphore(CONCURRENCY)
session = None
async def scrape_api():
    async with semaphore:
        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__':
    asyncio.get_event_loop().run_until_complete(main())

```

在这里我们声明了 `CONCURRENCY` 代表爬取的最大并发量为 5，同时声明爬取的目标 URL 为百度。接着我们借助于 `Semaphore` 创建了一个信号量对象，赋值为 `semaphore`，这样我们就可以用它来控制最大并发量了。怎么使用呢？我们这里把它直接放置在对应的爬取方法里面，使用 `async with` 语句将 `semaphore` 作为上下文对象即可。这样的话，信号量可以控制进入爬取的最大协程数量，最大数量就是我们声明的 `CONCURRENCY` 的值。

在 `main` 方法里面，我们声明了 10000 个 `task`，传递给 `gather` 方法运行。倘若不加以限制，这 10000 个 `task` 会被同时执行，并发数量太大。但有了信号量的控制之后，同时运行的 `task` 的数量最大会被控制在 5 个，这样就能给 `aiohttp` 限制速度了。

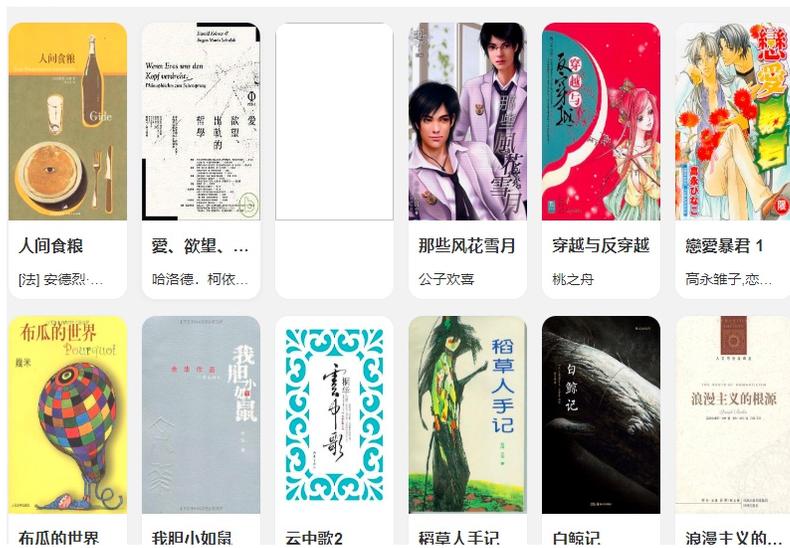
在这里，`aiohttp` 的基本使用就介绍这么多，更详细的内容还是推荐你到官方文档查阅，链接：<https://docs.aiohttp.org/>。

### 爬取实战

上面我们介绍了 `aiohttp` 的基本用法之后，下面我们来根据一个实例实现异步爬虫的实战演练吧。

本次我们要爬取的网站是：<https://dynamic5.scrape.cuiqingcai.com/>，页面如图所示。

#### Scrape



这是一个书籍网站，整个网站包含了数千本书籍信息，网站是 JavaScript 渲染的，数据可以通过 Ajax 接口获取到，并且接口没有设置任何反爬措施和加密参数，另外由于这个网站比之前的电影案例网站数据量大一些，所以更加适合做异步爬取。

本课时我们要完成的目标有：

- 使用 `aiohttp` 完成全站的书籍数据爬取。
- 将数据通过异步的方式保存到 `MongoDB` 中。

在本课时开始之前，请确保你已经做好了如下准备工作：

- 安装好了 Python（最低为 Python 3.6 版本，最好为 3.7 版本或以上），并能成功运行 Python 程序。
- 了解了 Ajax 爬取的一些基本原理和模拟方法。
- 了解了异步爬虫的基本原理和 `asyncio` 库的基本用法。
- 了解了 `aiohttp` 库的基本用法。
- 安装并成功运行了 `MongoDB` 数据库，并安装了异步存储库 `motor`。

注：这里要实现 `MongoDB` 异步存储，需要异步 `MongoDB` 存储库，叫作 `motor`，安装命令为：`pip3 install motor`

### 页面分析

在之前我们讲解了 Ajax 的基本分析方法，本课时的站点结构和之前 Ajax 分析的站点结构类似，都是列表页加详情页的结构，加载方式都是 Ajax，所以我们能轻松分析到如下信息：

- 列表页的 Ajax 请求接口格式为：<https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset={offset}>，limit 的值即为每一页的书的个数，offset 的值为每一页的偏移量，其计算公式为  $offset = limit * (page - 1)$ ，如第 1 页 offset 的值为 0，第 2 页 offset 的值为 18，以此类推。
- 列表页 Ajax 接口返回的数据里 results 字段包含当前页 18 本书的信息，其中每本书的数据里面包含一个字段 id，这个 id 就是书本身的 ID，可以用来进一步请求详情页。
- 详情页的 Ajax 请求接口格式为：<https://dynamic5.scrape.cuiqingcai.com/api/book/{id}>，id 即为书的 ID，可以从列表页的返回结果中获取。

如果你掌握了 Ajax 爬取实战一课时的内容话，上面的内容应该很容易分析出来。如有难度，可以复习下之前的知识。

## 实现思路

其实一个完善的异步爬虫应该能够充分利用资源进行全速爬取，其思路是维护一个动态变化的爬取队列，每产生一个新的 task 就会将其放入队列中，有专门的爬虫消费者从队列中获取 task 并执行，能做到在最大并发量的前提下充分利用等待时间进行额外的爬取处理。

但上面的实现思路整体较为烦琐，需要设计爬取队列、回调函数、消费者等机制，需要实现的功能较多。由于我们刚刚接触 aiohttp 的基本用法，本课时也主要是了解 aiohttp 的实战应用，所以这里我们将爬取案例的实现稍微简化一下。

在这里我们将爬取的逻辑拆分成两部分，第一部分为爬取列表页，第二部分为爬取详情页。由于异步爬虫的关键点在于并发执行，所以我们可以将爬取拆分为两个阶段：

- 第一阶段为所有列表页的异步爬取，我们可以将所有的列表页的爬取任务集合起来，声明为 task 组成的列表，进行异步爬取。
- 第二阶段则是拿到上一列表页的所有内容并解析，拿到所有书的 id 信息，组合为所有详情页的爬取任务集合，声明为 task 组成的列表，进行异步爬取，同时爬取的结果也以异步的方式存储到 MongoDB 里面。

因为两个阶段的拆分之后需要串行执行，所以可能达不到协程的最佳调度方式和资源利用情况，但也差不了很多。但这个实现思路比较简单清晰，代码实现也比较简单，能够帮我们快速了解 aiohttp 的基本使用。

## 基本配置

首先我们先配置一些基本的变量并引入一些必需的库，代码如下：

```
import asyncio
import aiohttp
import logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s: %(message)s')
INDEX_URL = 'https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset={offset}'
DETAIL_URL = 'https://dynamic5.scrape.cuiqingcai.com/api/book/{id}'
PAGE_SIZE = 18
PAGE_NUMBER = 100
CONCURRENCY = 5
```

在这里我们导入了 asyncio、aiohttp、logging 这三个库，然后定义了 logging 的基本配置。接着定义了 URL、爬取页码数量 PAGE\_NUMBER、并发量 CONCURRENCY 等信息。

## 爬取列表页

首先，第一阶段我们就来爬取列表页，还是和之前一样，我们先定义一个通用的爬取方法，代码如下：

```
semaphore = asyncio.Semaphore(CONCURRENCY)
session = None
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)
```

在这里我们声明了一个信号量，用来控制最大并发数量。

接着我们定义了 scrape\_api 方法，该方法接收一个参数 url。首先使用 async with 引入信号量作为上下文，接着调用了 session 的 get 方法请求这个 url，然后返回响应的 JSON 格式的结果。另外这里还进行了异常处理，捕获了 ClientError，如果出现错误，会输出异常信息。

接着，对于列表页的爬取，实现如下：

```
async def scrape_index(page):
    url = INDEX_URL.format(offset=PAGE_SIZE * (page - 1))
    return await scrape_api(url)
```

这里定义了一个 scrape\_index 方法用于爬取列表页，它接收一个参数为 page，然后构造了列表页的 URL，将其传给 scrape\_api 方法即可。这里注意方法同样需要用 async 修饰，调用的 scrape\_api 方法前面需要加 await，因为 scrape\_api 调用之后本身会返回一个 coroutine。另外由于 scrape\_api 返回结果就是 JSON 格式，因此 scrape\_index 的返回结果就是我们想要爬取的信息，不需要再额外解析了。

好，接着我们定义一个 main 方法，将上面的方法串联起来调用一下，实现如下：

```
import json
async def main():
    global session
    session = aiohttp.ClientSession()
    scrape_index_tasks = [asyncio.ensure_future(scrape_index(page)) for page in range(1, PAGE_NUMBER + 1)]
    results = await asyncio.gather(*scrape_index_tasks)
    logging.info('results %s', json.dumps(results, ensure_ascii=False, indent=2))

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

这里我们首先声明了 session 对象，即最初声明的全局变量，将 session 作为全局变量的话我们就不要每次在各个方法里面传递了，实现比较简单。

接着我们定义了 scrape\_index\_tasks，它就是爬取列表页的所有 task，接着我们调用 asyncio 的 gather 方法并传入 task 列表，将结果赋值为 results，它是所有 task 返回结果组成的列表。

最后我们调用 main 方法，使用事件循环启动该 main 方法对应的协程即可。

运行结果如下：

```
2020-04-03 03:45:54,692 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=0
2020-04-03 03:45:54,707 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=18
2020-04-03 03:45:54,707 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=36
2020-04-03 03:45:54,708 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=54
2020-04-03 03:45:54,708 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=72
2020-04-03 03:45:56,431 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=90
2020-04-03 03:45:56,435 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/?limit=18&offset=108
```

可以看到这里就开始异步爬取了，并发量是由我们控制的，目前为 5，当然也可以进一步调高并发量，在网站能承受的情况下，爬取速度会进一步加快。

最后 results 就是所有列表页得到的结果，我们将其赋值为 results 对象，接着我们就可以用它来进行第二阶段的爬取了。

## 爬取详情页

第二阶段就是爬取详情页并保存数据了，由于每个详情页对应一本书，每本书需要一个 ID，而这个 ID 又正好存在 results 里面，所以下面我们就需要将所有详情页的 ID 获取出来。

在 main 方法里增加 results 的解析代码，实现如下：

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

这样 `ids` 就是所有书的 `id` 了，然后我们用所有的 `id` 来构造所有详情页对应的 `task`，来进行异步爬取即可。

那么这里再定义一个爬取详情页和保存数据的方法，实现如下：

```
from motor.motor_asyncio import AsyncIOMotorClient
MONGO_CONNECTION_STRING = 'mongodb://localhost:27017'
MONGO_DB_NAME = 'books'
MONGO_COLLECTION_NAME = 'books'
client = AsyncIOMotorClient(MONGO_CONNECTION_STRING)
db = client[MONGO_DB_NAME]
collection = db[MONGO_COLLECTION_NAME]
async def save_data(data):
    logging.info('saving data %s', data)
    if data:
        return await collection.update_one({
            'id': data.get('id')
        }, {
            '$set': data
        }, upsert=True)
async def scrape_detail(id):
    url = DETAIL_URL.format(id=id)
    data = await scrape_api(url)
    await save_data(data)
```

这里我们定义了 `scrape_detail` 方法用于爬取详情页数据并调用 `save_data` 方法保存数据，`save_data` 方法用于将数据库保存到 MongoDB 里面。

在这里我们用到了支持异步的 MongoDB 存储库 `motor`，MongoDB 的连接声明和 `pymongo` 是类似的，保存数据的调用方法也是基本一致，不过整个都换成了异步方法。

好，接着我们就在 `main` 方法里面增加 `scrape_detail` 方法的调用即可，实现如下：

```
scrape_detail_tasks = [asyncio.ensure_future(scrape_detail(id)) for id in ids]
await asyncio.wait(scrape_detail_tasks)
await session.close()
```

在这里我们先声明了 `scrape_detail_tasks`，即所有详情页的爬取 `task` 组成的列表，接着调用了 `asyncio` 的 `wait` 方法调用执行即可，当然这里也可以用 `gather` 方法，效果是一样的，只不过返回结果略有差异。最后全部执行完毕关闭 `session` 即可。

一些详情页的爬取过程运行如下：

```
2020-04-03 04:00:32,576 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/2301475
2020-04-03 04:00:32,576 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/2351866
2020-04-03 04:00:32,577 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/2828384
2020-04-03 04:00:32,577 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/3040352
2020-04-03 04:00:32,578 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/3074810
2020-04-03 04:00:44,858 - INFO: saving data {'id': '3040352', 'comments': [{'id': '387952888', 'content': '温馨文，青梅竹马神马的很有爱~}， ..., {'id': '2005314253', 'content': '沈晋&秦央，文}
...
2020-04-03 04:00:44,859 - INFO: scraping https://dynamic5.scrape.cuiqingcai.com/api/book/2994915
...
```

最后我们观察下，爬取到的数据也都保存到 MongoDB 数据库里面了，如图所示：

db.getCollection('books').find({})

localhost localhost:27017 books

db.getCollection('books').find({})

books 0.001 sec.

Key	Value	Type
▶ (1) ObjectId("5e863034afa2e239ec1699c9")	{ 18 fields }	Object
▶ (2) ObjectId("5e863036afa2e239ec1699cc")	{ 18 fields }	Object
▶ (3) ObjectId("5e86303aafa2e239ec1699ce")	{ 18 fields }	Object
▶ (4) ObjectId("5e86303aafa2e239ec1699d0")	{ 18 fields }	Object
▶ (5) ObjectId("5e86303aafa2e239ec1699d2")	{ 18 fields }	Object
▶ (6) ObjectId("5e86446cafa2e239ec169cd8")	{ 18 fields }	Object
▶ (7) ObjectId("5e86446eafa2e239ec169cdb")	{ 18 fields }	Object
▼ (8) ObjectId("5e86446fafa2e239ec169cdd")	{ 18 fields }	Object
_id	ObjectId("5e86446fafa2e239ec169cdd")	Object
id	2301475	String
▶ authors	[ 1 element ]	Array
catalog	纪德的写作状态人间食粮1927年版序言引言第一篇第二...	String
▶ comments	[ 10 elements ]	Array
cover	https://img1.doubanio.com/view/subject/l/public/s276...	String
introduction		String
isbn	9787208074149	String
name	人间食粮	String
page_number	339	Int32
price	28.00	String
published_at	2007-11-20T16:00:00Z	String
publisher	上海人民出版社	String
score	8.3	String
▶ tags	[ 8 elements ]	Array
▶ translators	[ 1 element ]	Array
updated_at	2020-03-21T16:59:37.786465Z	String
url	https://book.douban.com/subject/2301475/	String
▶ (9) ObjectId("5e864479afa2e239ec169ce0")	{ 18 fields }	Object
▶ (10) ObjectId("5e864486afa2e239ec169ce3")	{ 18 fields }	Object
▶ (11) ObjectId("5e864487afa2e239ec169ce5")	{ 18 fields }	Object
▶ (12) ObjectId("5e864487afa2e239ec169ce7")	{ 18 fields }	Object

至此，我们就使用 aiohttp 完成了书籍网站的异步爬取。

## 总结

本课时内容较多，我们了解了 aiohttp 的基本用法，然后通过一个实例讲解了 aiohttp 异步爬虫的具体实现。学习过程我们可以发现，相比普通的单线程爬虫来说，使用异步可以大大提高爬取效率，后面我们也可以多多使用。

本课时代码：<https://github.com/Gemey/ScrapeDynamic5>。