

Ajax 数据爬取

有时候我们在浏览器中可以看到的内容，使用 requests 请求后找不到这些数据

Ajax 简介

Asynchronous JavaScript and XML，异步的 JavaScript 和 XML，Ajax 不是一门编程语言，而是利用 JavaScript 在保证页面不被刷新、页面链接不改变的情况下与服务器交换数据并更新部分网页的技术

在浏览网页的时候，我们会发现很多网页都有下滑功能，比如微博的主页，切换到微博页面，一直下滑，可以发现下滑几个微博之后，再向下就没有了，转而会出现一个加载的动画，不一会儿下方就继续出现了新的内容，这个过程就是 Ajax 加载的过程

```
# https://m.weibo.cn/u/2830678474
```



崔庆才 | 静觅



2016-11-27 来自 微博 weibo.com



@一起神回复: 论如何拍出好看的滑雪照片 😊



转发

评论

1

加载中...

下面介绍一下 Ajax 的基本原理，发送 Ajax 请求到网页更新的这个过程可以简单分为以下三步：发送请求、解析内容和渲染网页

发送请求

Ajax 由 JavaScript 实现，实际上就是先新建一个 XMLHttpRequest 对象 xmlhttp，然后调用 onreadystatechange 属性设置监听，最后调用 open 和 send 方法向服务器发送请求

```
var xmlhttp;
if (window.XMLHttpRequest) {
    // code for IE7+, Firefox, Chrome, Opera, Safari
    // 新建 XMLHttpRequest 对象
```

```
xmlhttp=new XMLHttpRequest();
} else {// code for IE6, IE5
    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
// 调用 onreadystatechange 属性设置监听
xmlhttp.onreadystatechange=function() {
    if (xmlhttp.readyState==4 && xmlhttp.status==200) {
        document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
}
// 调用 open() 和 send() 方法向某个链接（也就是服务器）发送了请求
xmlhttp.open("POST","/ajax/",true);
xmlhttp.send();
```

解析内容

得到响应之后，onreadystatechange 属性对应的方法便会被触发，此时利用 xmlhttp 的 responseText 属性便可取到响应内容，这类似于 Python 中利用 requests 向服务器发起请求，然后得到响应的过程

返回的内容可能是 HTML，也可能是 JSON，接下来只需要在方法中用 JavaScript 进一步处理即可

渲染网页

JavaScript 有改变网页内容的能力，解析完响应内容之后，就可以调用 JavaScript 来针对解析完的内容对网页进行下一步处理，比如通过 document.getElementById().innerHTML 可以对某个元素内的源代码进行更改，这样网页显示的内容就改变了，这样的操作也被称作 DOM 操作，即对 Document 网页文档进行操作

发送请求、解析内容和渲染网页这三个步骤都是由 JavaScript 完成的，因此真实的数据都是 Ajax 请求得到的，如果想要抓取这些数据，就需要知道这些请求到底是怎么发送的、发送到哪里、发送了哪些参数，如果知道了这些，就可以使用 Python 模拟这个发送操作，获取其中的数据

Ajax 请求分析

下面我们使用浏览器的开发者工具查看 Ajax 请求，<https://m.weibo.cn/u/2830678474>

微博

m.weibo.cn/u/2830678474

< 返回

崔庆才 | 静觅 🌐

关注 1202 | 粉丝 1.2万

聊天 + 关注

Network

Name	Status	Type	Initiator
chunk-4d847fb4.css	200	fetch	workbox-core.prod.js:1
collect-main-page-profile-statusLite.3f0735fa.css	200	fetch	workbox-core.prod.js:1
collect-main-page-profile-statusLite.b1abaf37.js	200	fetch	workbox-core.prod.js:1
page.a7370fc9.css	200	fetch	workbox-core.prod.js:1
page.d88780c4.js	200	fetch	workbox-core.prod.js:1

查看请求

Ajax 有其特殊的请求类型，叫作 xhr，我们可以发现一个名称以 getIndex 开头的请求，其 Type 为 xhr，这就是一个 Ajax 请求

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights

Preserve log Disable cache No throttling

Filter Invert More filters All Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other

Name	Headers	Payload	Preview	Response	Initiator	Timing
collect-main-page-profile-statusLite.b...						
page.a7370fc9.css						
page.d88780c4.js						
config						
config						
getIndex?type=uid&value=2830678474...						
getIndex?type=uid&value=2830678474...						
getIndex?type=uid&value=2830678474...						
getIndex?type=uid&value=2830678474...						
getIndex?type=uid&value=2830678474...						
getIndex?type=uid&value=2830678474...						
20 / 59 requests 12.2 kB / 62.3 kB transferred	X-Requested-With: XMLHttpRequest					
	X-Xsrf-Token: 0124f7					

点击 Preview 可以看到响应的内容，这里 Chrome 自动解析了这个 JSON，返回结果是作者的个人信息，如昵称、简介、头像等，这也是用来渲染个人主页所使用的数据，JavaScript 接收到这些数据之后，再执行相应的渲染方法，整个页面就渲染出来了

Screenshot of the Network tab in the Chrome DevTools Network panel. The 'Preview' tab is selected, showing a JSON response from a request to 'page.d88780c4.js'. The response contains an 'ok' key with value 1, and a 'data' object with keys: 'isVideoCoverStyle' (1), 'isStarStyle' (0), 'userInfo' (id: 2830678474, screen_name: "崔庆才 | 静觅", ...), 'fans_scheme' (fanscheme URL), 'follow_scheme' (followscheme URL), 'isStarStyle' (0), 'isVideoCoverStyle' (1), 'profile_ext' (touid: 2830678474), 'scheme' (sinaweibo://userinfo?uid=2830678474&type=uid&value=2830678474&luid=10000011&lfid=10760328306784), 'showAppTips' (0), 'tabsInfo' (selectedTab: 1), and 'userAvatar_hd' (avatar URL).

切换到 Response 可以观察到真实的返回数据，和刚才基本一样

Screenshot of the Network tab in the Chrome DevTools Network panel. The 'Response' tab is selected, showing the raw JSON response from the previous screenshot. The response is identical to the one shown in the 'Preview' tab.

如果选择红框这个“ALL”可以看到原始的页面，可以看到原始页面只有 50 行左右的代码，所以我们看到的微博页面的真实数据并不是最原始的页面返回的，而是后来执行 JavaScript 后再次向后台发送了 Ajax 请求，浏览器拿到数据后渲染出来的

Screenshot of the Network tab in the Chrome DevTools Network panel. The 'All' tab is selected, showing the raw HTML response from the previous screenshot. The response is a single line of code containing the entire HTML document, including the head and body sections.

过滤请求

接下来使用 Chrome 的筛选功能筛选出所有的 Ajax 请求，随着不断滑动页面，就得到了作者完整的某一页微博信息，接下来只需要用程序模拟这些 Ajax 请求，就可以提取我们所需要的信息了

The screenshot shows the Chrome DevTools Network tab with the 'Fetch/XHR' filter selected. A single XHR request is highlighted, representing a POST request to 'https://sp1.scrape.center/'. The 'Preview' tab is open, showing the JSON response body. The response contains fields like 'apply_scenario_flag', 'display_text', 'display_text_min_number', 'pending_approval_count', 'pic_ids', 'pic_num', 'raw_text', 'region_name', and 'region_opt'. The 'raw_text' field is explicitly highlighted with a red box, containing the Chinese text '还有这种人' (There are still such people).

Name	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
config							
config							
getIndex?type=uid&value=2830678474&c...							
getIndex?type=uid&value=2830678474...							
getIndex?type=uid&value=2830678474&c...							
getIndex?type=uid&value=2830678474...							
getIndex?type=uid&value=2830678474...							
getIndex?type=uid&value=2830678474...							
getIndex?type=uid&value=2830678474...							
getIndex?type=uid&value=2830678474...							

案例：Ajax 爬取电影信息

这里使用 <https://spa1.scrape.center/> 作为 Ajax 实战的爬取网站

我们查阅网页源码可以发现并没有电影信息

The screenshot shows the Google Chrome Developer Tools Network tab. A red arrow points to the status bar message "DevTools is now available in Chinese! Always match". The Network tab displays two requests:

Name	Status	Protocol	Type	Initiator	Size	Time
movie/?limit=10&offset=0	200	http/1.1	xhr	movie	3.1 kB	284 ms
movie?limit=10&offset=0	200	http/1.1	xhr / Redirect	chunk-vendors.683ca77c.js:235	(disk cache)	Pending

使用开发者工具观察 Ajax 接口，需要 limit 和 offset 两个参数

The screenshot shows a movie search result page. A specific movie entry for "完美的世界 - A Perfect World" is highlighted. The page includes a poster, basic details (Genre: 剧情, 犯罪; Country: 美国 / 138分钟; Release Date: 1993-11-24), and a rating of 8.8.

The screenshot shows the Google Chrome Developer Tools Network tab with a detailed view of a movie API request. The request is for "movie/?limit=10&offset=10". The Headers tab is selected, showing:

- Request URL: https://spa1.scrape.center/api/movie/?limit=10&offset=10
- Request Method: GET
- Status Code: 200 OK
- Remote Address: 111.6.217.225:443
- Referrer Policy: strict-origin-when-cross-origin

经过调试进行爬取

```
import requests
import logging
import time
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# https://spa1.scrape.center/api/movie/?limit=10&offset=10
# index_url = 'https://spa1.scrape.center/api/movie/?limit={limit}&offset={offset}'
limit = 10
total_page = 10

# 爬取页面
def scrape_api(url):
    logging.info(f'Scraping {url}')
    try:
        response = requests.get(url)
        if response.status_code == 200:
            # 解析响应内容并转换成 JSON
            return response.json()
        logging.error(f'get invalid status code {response.status_code} while scraping {url}')
    except requests.exceptions.RequestException as e:
        # 设置 exc_info=True 可以打印 Traceback 错误堆栈信息
        logging.error(f'error occurred while scraping {url}', exc_info=True)

# 列表页的爬取
def scrape_index(page):
    url = f'https://spa1.scrape.center/api/movie/?limit={limit}&offset={limit * (page - 1)}'
    return scrape_api(url)

# 详情页的爬取
def scrape_detail(id):
    url = f'https://spa1.scrape.center/api/movie/{id}'
    return scrape_api(url)

# 保存数据
from pymongo import MongoClient

# 连接 MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['crawler']
collection = db['movies_ajax']

def save_data(data):
    # 存在即更新，不存在就插入
    collection.update_one({'name': data.get('name')}, {'$set': data}, upsert=True)

def main():
    for page in range(1, total_page + 1):
        index_data = scrape_index(page)
        for item in index_data.get('results'):
            id = item.get('id')
            detail_data = scrape_detail(id)
            logging.info(f'detail_data: {detail_data}')
            save_data(detail_data)
            logging.info('data saved successfully')
            time.sleep(1)
```

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

存储结果如下

The screenshot shows the MongoDB Compass interface with the following details:

- Path:** localhost:27017 > crawler > movies_ajax
- Documents:** 100 (highlighted in green)
- Aggregations, Schema, Indexes, Validation:** Other tabs in the top navigation.
- Query Bar:** Type a query: { field: 'value' } or [Generate query](#).
- Buttons:** Explain, Reset, Find, Options.
- Action Buttons:** ADD DATA, EXPORT DATA, UPDATE, DELETE.
- Result View:** Displays two movie documents. The first document is for "霸王别姬" (The Last Emperor) and the second for "这个杀手不太冷" (Léon).

```
_id: ObjectId('6787b335b8eedf565221913f')
name : "霸王别姬"
actors : Array (30)
alias : "Farewell My Concubine"
categories : Array (2)
cover : "https://p0.meituan.net/movie/ce4da3e03e655b5b88ed31b5cd7896cf62472.jpg..."
directors : Array (1)
drama : "影片借一出《霸王别姬》的京戏，牵扯出三个人之间一段随时代风云变幻的爱恨情仇。段小楼（张丰毅 饰）与程蝶衣（张国荣 饰）是一对打小一起长大的师..."
id : 1
minute : 171
photos : Array (616)
published_at : "1993-07-26"
rank : 1
regions : Array (2)
score : 9.5
updated_at : "2020-03-07T16:31:36.967843Z"

_id: ObjectId('6787b336b8eedf5652219143')
name : "这个杀手不太冷"
actors : Array (80)
alias : "Léon"
categories : Array (3)
cover : "https://p1.meituan.net/movie/6bea9af4524dfbd0b668eaa7e187c3df767253.jpg..."
directors : Array (1)
```

案例：存储作者微博信息

首先分析一下页面 <https://m.weibo.cn/u/2830678474>

打开浏览器工具，选择 Fetch/XHR 过滤器，然后一直滑动页面加载新的微博内容，选定其中一个请求，分析它的参数信息，根据请求 URL 可以发现参数有 type、value、containerid

DevTools is now available in Chinese! [Always match Chrome's language](#) [Switch DevTools to Chinese](#) [Don't show again](#)

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights ▲ 1

Preserve log Disable cache No throttling Filter Invert More filters All Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other

Name Headers Payload Response Initiator Timing

Request URL: https://m.weibo.cn/api/container/getIndex?type=uid&value=2830678474&containerid=1076032830678474
Request Method: GET
Status Code: 200 OK (from service worker)
Referrer Policy: strict-origin-when-cross-origin

Response Headers

Content-Encoding:	gzip
Content-Security-Policy:	upgrade-insecure-requests
Content-Type:	application/json; charset=utf-8
Date:	Fri, 03 Jan 2025 06:57:57 GMT
Lb:	111.13.134.130
Proc_node:	mweibo-h5-v8-web-5c9c9b56-g9mn2
Server:	SHANHAI-SERVER
Ssl_node:	mapi-10-54-7-26.hj4.intra.weibo.cn
Vary:	Accept-Encoding

21 / 102 requests 59.2 kB / 85.3 kB transferred 360 X-Log-Uid:

观察这个请求的响应内容，这里最关键的两部分信息是 cardlistInfo 和 cards，前者包含一个比较重要的信息 total，其实是微博的总数量；后者是一个列表，它包含 10 个元素

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights

Preserve log Disable cache No throttling Filter Invert More filters All Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other

Name Headers Payload Preview Response Initiator Timing

page.92500443.js

```

{
  "ok": 1,
  "data": {
    "cardlistInfo": {
      "containerid": "1076032830678474",
      "v_p": 42,
      "show_style": 1,
      "total": 2489,
      "autoLoadMoreIndex": 10,
      "autoLoadMoreIndex": 10
    },
    "data": [
      {
        "cardlistInfo": {
          "containerid": "1076032830678474",
          "v_p": 42,
          "show_style": 1,
          "since_id": 5070486381790320,
          "total": 2489
        },
        "cards": [
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5117859504063656"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5117859504063656"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5117429564312301"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5117422770585679"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5117421919405054"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5108511876844136"
          },
          {
            "card_type": 9,
            "profile_type_id": "proweibo_5100869923637349"
          }
        ]
      }
    ]
  }
}

```

9 / 88 requests 53.7 kB / 79.8 kB transferred

展开其中一个元素，发现一个比较重要的字段 mblog，包含的正是微博的一些信息，比如 attitudes_count（点赞数）、comments_count（评论数）、reposts_count（转发数）、created_at（发布时间）、text（微博正文）等

DevTools is now available in Chinese! [Always match Chrome's language](#) [Switch DevTools to Chinese](#) [Don't show again](#)

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights

Filter Invert More filters All Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other

20000 ms	40000 ms	60000 ms	80000 ms	100000 ms	120000 ms	140000 ms	160000 ms	180000 ms	200000 ms	220000 ms	240000 ms	260000 ms	280000 ms	300000 ms	320000 ms

Name

- page.92500443.js
- config
- config
- getIndex?type=uid&value=2830678474&...
- getIndex?type=uid&value=2830678474...
- getIndex?type=uid&value=2830678474&...
- getIndex?type=uid&value=2830678474...
- getIndex?type=uid&value=2830678474...
- config
- config

11 / 92 requests | 54.6 kB / 80.7 kB transferred

我们使用程序模拟 Ajax 请求，将作者的微博爬取下来

```

import time
from urllib.parse import urlencode
import requests
from pyquery import PyQuery as pq

base_url = 'https://m.weibo.cn/api/container/getIndex?'

headers = {
    'Host': 'm.weibo.cn',
    'Referer': 'https://m.weibo.cn/u/2830678474',
    'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36',
    'X-Requested-With': 'XMLHttpRequest',
}

# 构造参数字典，其中 type、value、containerid 是固定参数，page 是可变参数
def get_page(page):
    params = {
        'type': 'uid',
        'value': '2830678474',
        'containerid': '1076032830678474',
        'page': page
    }
    # 将参数转化成 类似于 type=uid&value=2145291155&containerid=1076032145291155&page=2 的形式
    url = base_url + urlencode(params)
    try:
        # 发送请求
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            return response.json()
    except Exception as e:
        print(e)
        return None

```

```

print('请求成功 --> ', url)
# 如果请求成功，直接调用 json 方法将内容解析为 JSON
return response.json()

except requests.ConnectionError as e:
    print('Error', e.args)

# 定义一个解析方法，从结果中提取想要的信息，比如微博的 id、正文、点赞数、评论数和转发数
def parse_page(json):
    if json:
        items = json.get('data').get('cards')
        for item in items:
            item = item.get('mblog')
            # 创建字典
            weibo = {}
            weibo['id'] = item.get('id')
            weibo['text'] = pq(item.get('text')).text()
            weibo['attitudes'] = item.get('attitudes_count')
            weibo['comments'] = item.get('comments_count')
            weibo['reposts'] = item.get('reposts_count')
            yield weibo

# 将结果保存到 MongoDB 数据库
def page_to_mongodb(results):
    import pymongo
    from pymongo import MongoClient
    # 连接 MongoDB
    client = MongoClient('mongodb://localhost:27017/')
    # 指定数据库
    db = client['crawler']
    # 指定集合（表）
    collection = db['weibo_info']
    insert_info = collection.insert_many(results)
    print(insert_info)

if __name__ == '__main__':
    for page in range(1, 11):
        json = get_page(page)
        results = parse_page(json)
        print('存储中... ')
        page_to_mongodb(results)
        time.sleep(13)

```

最终数据落库如下

localhost:27017 > crawler > weibo_info

Documents 9 Aggregations Schema Indexes 1 Validation

Type a query: { field: 'value' } or [Generate query](#)

Explain Reset Find Options ▾

[ADD DATA](#) [EXPORT DATA](#) [UPDATE](#) [DELETE](#)

50 51 - 100 of 106

```

{
  "_id": {},
  "id": "4921038087651774",
  "text": "转发微博",
  "attitudes": 2,
  "comments": 0,
  "reposts": 0
}

{
  "_id": {},
  "id": "4918761163326875",
  "text": "//@Easy:转需//@蚁工厂:去这里: 智能法律咨询, 一步一步的输入对方欠多少钱约定的还款期和你有的证据啥的, 最后生成一个法律咨询书, 包括真要诉讼的具体",
  "attitudes": 1,
  "comments": 0,
  "reposts": 10
}

{
  "_id": {},
  "id": "4916622017956803",
  "text": "//@互联网的那点事:【美团: 王慧文因个人健康原因辞任美团董事】美团在港交所公布, 王慧文先生因个人健康原因, 已提出辞去本公司非执行董事、本公司董事
  "attitudes": 0,
  "comments": 0,
  "reposts": 0
}

```

模拟登录

在很多情况下，网站的信息需要登录后才可以查看，模拟登录主要分为两种模式，一种是基于 session 和 cookie 的模拟登录，另一种是基于 JWT (JSON Web Token) 的模拟登录

网站登录验证

登录一般都需要用户名和密码，也有的网站是填写手机号获取验证码，或者微信扫码，或者 OAuth 验证等，从根本上看，这些方式都是把一些可供认证的信息提交给服务器

就拿用户名和密码来说，用户在一个网页表单里面输入这两个内容，然后在点击登录按钮的一瞬间，浏览器客户端会向服务器发送一个登录请求，这个请求里肯定包含刚输入的用户名和密码

这时服务器需要处理这些内容，然后返回给客户端一个类似凭证的东西，有了这个凭证，客户端再去访问某些需要登录才能查看的页面时，服务器自然就会“放行”，并返回对应的内容或执行对应的操作

这就像在坐火车前，乘客要先用钱买票，有了票之后，让进站口查验一下，没问题就可以去候车了，这个票就是坐火车时的凭证

基于 Session 和 Cookie

不同网站对于用户登录状态的实现可能是不同的，下面两种情况几乎能涵盖大部分网站，具体的实现逻辑因服务器而异，但 Session 和 Cookie 一定是要相互配合的

Cookie 里可能只保存了 Session ID 相关的信息，服务器能根据这个信息找到对应的 Session，当用户登录后，服务器会在对应的 Session 里标记一个字段，代表用户已处于登录状态或者其他信息，用户每次访问网站的时候都带着 Cookie，服务器每次都找到对应的 Session，然后看一下用户的状态是否为登录状态，再决定返回什么结果或执行什么操作

Cookie 里也可能直接保存了某些凭证信息，例如用户发起登录请求，服务器校验通过后，返回给客户端的响应头里面可能带有 Set-Cookie 字段，里面就包含着类似凭证的信息，客户端会执行设置 Cookie 的操作，将那些类似凭证的信息保存到 Cookie 里，以后再访问网站时都携带着 Cookie，服务器拿着其中的信息进行校验，自然也能检测登录状态

基于 JWT

Web 开发技术一直在发展，前些年（现在 2025 年）前后端分离的开发模式越来越火，传统的基于 Session 和 Cookie 的校验又存在一定问题，例如服务器需要维护登录用户的 Session 信息，而且分布式部署也不方便，不太适合前后端分离的项目，所以 JWT 技术应运而生

JWT (JSON Web Token) 是为了在网络应用环境中传递声明而执行的一种基于 JSON 的开放标准，实际上就是在每次登录时都通过一个 Token 字段校验登录状态

JWT 的声明一般用来在身份提供者和服务提供者之间传递要认证的用户身份信息，以便从资源服务器获取资源，此外可以增加一些业务逻辑必需的声明信息，总之 Token 可以直接用于认证，也可以传递一些额外信息

有了 JWT，一些认证就不需要借助于 Session 和 Cookie 了，服务器也无须维护 Session 信息，从而减少了开销，只需要有一个校验 JWT 的功能就够了，同时还支持分布式部署和跨语言开发

JWT 一般是一个经过 Base64 编码技术加密的字符串，有自己的标准，格式类似下面这样

```
eyJ0eXAi  
joMTIzNCIsImFsZzIiOiJhZG1pbjIISInRSCCI6IkpxVCISImFsZyI6IkhtMiU2Ino.ey]Vc2VyswQi0jEyMywivXNlc  
kshbwui0iJhZG1pbjIISImv4CCI6MTU1MI4NCONi44NZCOMDE4fQ.pEgdmFAy73wa1FonEm2zbxg460th3d1T02HR9ivz  
xa8
```

其中有两个起分隔作用的“.”，因此可以把 JWT 看成一个三段式的加密字符串，这三部分分别是 Header、Payload 和 Signature

Header 声明了 JWT 的签名算法（如 RSA、SHA256），还可能包含 JWT 编号或类型等数据

Payload 通常是一些业务需要但不敏感的信息（如 User ID），另外还有很多默认字段，如 JWT 签发者、JWT 接受者、JWT 过期时间等

Signature 就是一个签名，是利用密钥 secret 对 Header、Payload 的信息进行加密后形成的，这个密钥保存在服务端，不会轻易泄露，如果 Payload 的信息被篡改，服务器就能通过 Signature 判断出这是非法请求，拒绝提供服务

用户通过用户名和密码登录，然后服务器生成 JWT 字符串返回给客户端，之后客户端每次请求都带着这个 JWT，服务器会自动判断其有效情况，如果有效就返回对应的数据

JWT 的传递方式多种多样，可以放在请求头中，也可以放在 URL 里，甚至把它放在 Cookie 里

账号池

如果爬取的数据量比较大或爬取速度比较快，网站又有单账号并发限制或者访问状态检测等反爬虫手段，我们的账号可能就无法访问网站或者面临封号的风险

这时可以建立一个账号池进行分流，用多个账号随机访问网站或爬取数据，这样能大幅提高爬虫的并发量，降低被封号的风险

例如准备 100 个账号，然后这 100 个账号都模拟登录，并保存对应的 Cookie 或 JWT，每次都随机从中选取一个来访，账号越多，每个账号被选取的概率就越小，也就避免了单账号并发量过大的问题，从而降低封号风险

案例：基于 Session 和 Cookie 的模拟登录

首先使用浏览器工具查看一下登录 <https://login2.scrape.center/> 的过程

The screenshot shows the Network tab in Chrome DevTools. A red arrow points from the 'Headers' tab to the 'Payload' tab, which is highlighted. Below the tabs, there are two expanded entries. The top entry, under 'Res', shows a POST request to <https://login2.scrape.center/login?next=/> with a status of 302 Found. The bottom entry, under 'Req', shows a GET request to <https://login2.scrape.center/> with a status of 200 OK.

发现浏览器进行了两次请求，第一个是带有用户名和密码的 form 表单形式，第二个是 GET 请求

The screenshot shows the Network tab in Chrome DevTools. A red arrow points from the 'Headers' tab to the 'Payload' tab, which is highlighted. Below the tabs, there are two expanded entries. The top entry, under 'Query String Parameters', shows a 'next:' parameter with a value of '/'. The bottom entry, under 'Form Data', shows 'username: admin' and 'password: admin'.

我们使用代码模拟这两个请求

```
import requests
from urllib.parse import urljoin

def login2_pre():
    base_url = 'https://login2.scrape.center/'
    login_url = urljoin(base_url, '/login')
    index_url = urljoin(base_url, '/page/1')

    username = 'admin'
    password = 'admin'

    response_login = requests.post(login_url, data={'username': username, 'password': password})
```

```
response_index = requests.get(index_url)

print('状态码: ', response_index.status_code)
print('URL: ', response_index.url)

# 状态码: 200
# URL: https://login2.scrape.center/login?next=/page/1
```

可是运行结果打印的不是电影列表页，依然是登录页，明明我们第一个 POST 请求已经登录了，为什么之后的 GET 请求依然需要登录？

这是因为模拟发出的两个请求，两次对应的 Session 不是同一个，模拟登录的关键在于两次发出请求的 Cookie 相同，因此这里可以把第一次模拟登录后的 Cookie 保存下来，在第二次请求时加上这个 Cookie

```
import requests
from urllib.parse import urljoin


def login2_simple():
    base_url = 'https://login2.scrape.center/'
    login_url = urljoin(base_url, '/login')
    index_url = urljoin(base_url, '/page/1')

    username = 'admin'
    password = 'admin'
    # allow_redirects=False 设置 requests 不自动处理重定向
    response_login = requests.post(login_url, data={'username': username, 'password': password},
                                    allow_redirects=False)
    cookies = response_login.cookies
    print('Cookies: ', cookies)
    response_index = requests.get(index_url, cookies=cookies)

    print('状态码: ', response_index.status_code)
    print('URL: ', response_index.url)


def login2_session():
    base_url = 'https://login2.scrape.center/'
    login_url = urljoin(base_url, '/login')
    index_url = urljoin(base_url, '/page/1')
    username = 'admin'
    password = 'admin'
    session = requests.session()
    #
    response_login = requests.post(login_url, data={'username': username, 'password': password})
    cookies = session.cookies
```

```

print('Cookies: ', cookies)
response_index = session.get(index_url)
print('状态码: ', response_index.status_code)
print('URL: ', response_index.url)

# login2_pre()
# login2_simple()
# login2_session()

```

不过这个例子有些简单，对于稍微复杂些的网站就无法使用了，下面使用 selenium 进行模拟登录

```

import time
import requests

from urllib.parse import urljoin
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.by import By

def login2_selenium():
    base_url = 'https://login2.scrape.center/'
    login_url = urljoin(base_url, '/login')
    index_url = urljoin(base_url, '/page/1')
    username = 'admin'
    password = 'admin'
    chrome_driver_path = './chromedriver.exe'
    chrome_options = Options()
    # 忽略浏览器证书验证
    chrome_options.add_argument('--ignore-certificate-errors')
    service = Service(executable_path=chrome_driver_path)
    browser = webdriver.Chrome(service=service, options=chrome_options)
    browser.get(base_url)
    time.sleep(3)
    browser.find_element(by=By.CSS_SELECTOR,
value='input[name="username"]').send_keys(username)
    browser.find_element(by=By.CSS_SELECTOR,
value='input[name="password"]').send_keys(password)
    browser.find_element(by=By.CSS_SELECTOR, value='input[type="submit"]').click()
    time.sleep(9)
    # 获取 Cookie
    cookies = browser.get_cookies()
    print('Cookies: ', cookies)
    browser.close()
    # 使用 Cookie
    session = requests.Session()
    for cookie in cookies:
        session.cookies.set(cookie['name'], cookie['value'])
    response_index = session.get(index_url)

```

```
print('状态码: ', response_index.status_code)
print('URL: ', response_index.url)

login2_selenium()
```

运行效果如下

The screenshot shows a web browser window with the title 'Scrape | Movie'. The URL in the address bar is 'login2.scrape.center'. A message at the top of the page reads 'Chrome 正受到自动测试软件的控制。' (Chrome is being controlled by an automatic testing software). The main content displays two movie reviews:

霸王别姬 - Farewell My Concubine (1993)

主演: 张国荣, 梁家辉, 张丰毅
导演: 钟孟宏
类型: 剧情, 爱情
地区: 中国内地, 中国香港 / 171 分钟
上映日期: 1993-07-26

这个杀手不太冷 - Léon (1994)

主演: 让·雷诺, 莫妮卡·贝鲁奇, 伊莲·朋佩奥
导演: 鲁本·奥斯特伦德
类型: 剧情, 动作, 犯罪
地区: 法国 / 110 分钟
上映日期: 1994-09-14

Both movies have a rating of **9.5** and five yellow star icons below the score.

案例：基于 JWT 的模拟登录

首先使用浏览器登录 <https://login3.scrape.center/>，基于 JWT 的网站通常采用前后端分离，前端的数据传输依赖于 Ajax，登录验证依赖于 JWT 这个本身就是 token 的值，如果 JWT 经验证是有效的，服务器就会返回相应的数据。

Name

- login
- element-icons.535877f5.woff
- chunk-f52d396c.4f574d24.css
- chunk-f52d396c.b7026cff.js
- book?limit=18&offset=0
- book/?limit=18&offset=0
- s33523792.jpg
- s1120387.jpg
- s27264181.jpg
- s24514468.jpg
- s29053580.jpg

Request URL: https://login3.scrape.center/api/login

Request Method: POST

Status Code: 200 OK

Remote Address: 183.192.184.218:443

Referrer Policy: no-referrer

Response Headers (11)

Request Headers	Raw
Accept:	application/json, text/plain, */*
Accept-Encoding:	gzip, deflate, br, zstd
Accept-Language:	zh-CN,zh;q=0.9

我们进行翻页发现请求头里多了一个 Authorization 字段

Name

- s2120993.jpg
- s33463759.jpg
- s29652928.jpg
- s26739368.jpg
- s1144911.jpg
- book?limit=18&offset=18
- book/?limit=18&offset=18
- s1078958.jpg
- s4371408.jpg
- s1595557.jpg
- s2988680.ipq

Request Headers

Response	Initiator	Timing
application/json, text/plain, */*		
gzip, deflate, br, zstd		
zh-CN,zh;q=0.9		
jwt		
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkjoxLClc2VybmtzSl6ImFkbWluliwzXhwlijoxNzQwMTczNTQ0LCJlbWFpbCI6ImFkbWluQGFkbWluLnNvbSlsm9yaWdfaWF0ljoxNzQwMTMwMzQ0fQ.hZL_TjVzYwZwgu6m9UXNayGRwRkcBggIPo3UxjrzgY.		
keep-alive		
Connection:		
Host:		
Sec-Ch-Ua:		
Sec-Ch-Ua-Mobile:		
Sec-Ch-Ua-Platform:		

经过观察我们知道需要分成两个步骤：首先模拟登录请求，带上必要的登录信息，获取返回的 JWT；之后发送请求时，在请求头里面加上 Authorization 字段，值就是 JWT 对应的内容

```
import requests
from urllib.parse import urljoin

base_url = 'https://login3.scrape.center/'
login_url = urljoin(base_url, '/api/login')
index_url = urljoin(base_url, '/api/book')
username = 'admin'
password = 'admin'

response_login = requests.post(login_url, json={
    'username': username,
    'password': password
})
data = response_login.json()
```

```

print('Response JSON: ', data)

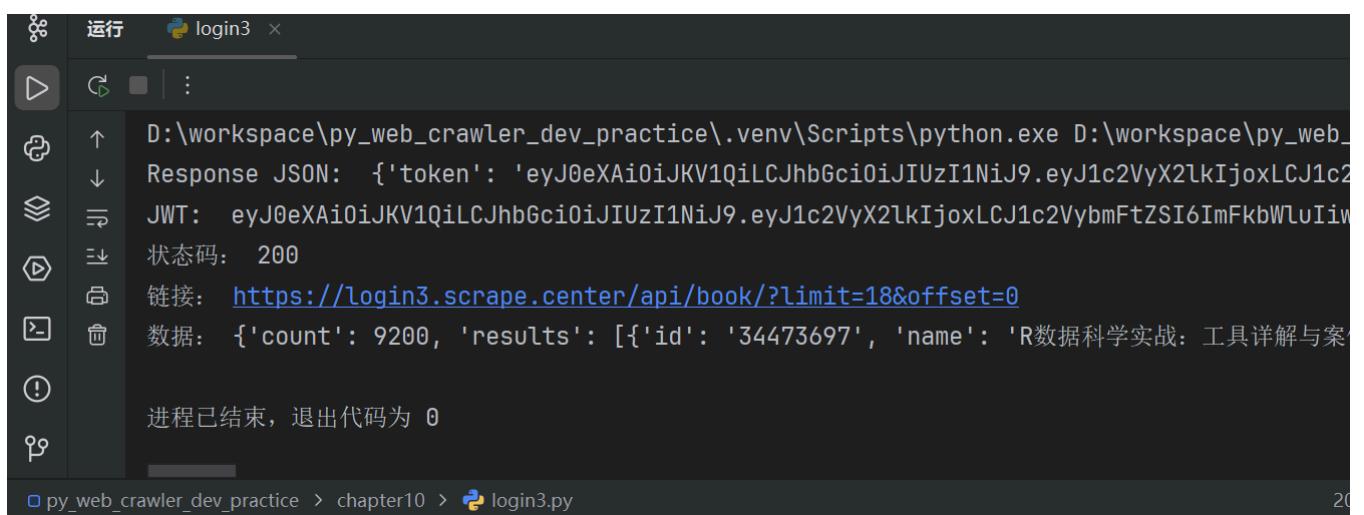
jwt = data.get('token')
print('JWT: ', jwt)

header = {
    'Authorization': f'jwt {jwt}'
}
response_index = requests.get(index_url, params={
    'limit': 18,
    'offset': 0
}, headers=header)

print('状态码: ', response_index.status_code)
print('链接: ', response_index.url)
print('数据: ', response_index.json())

```

运行效果如下



案例：大规模账号池搭建

本次案例使用网站 <https://antispider6.scrape.center/>，当我们使用 admin 账号登录后多次刷新后会被禁止访问



所以需要搭建一个账号池，维护 100 个账号信息以及对应的 Cookie，并存放到数据库中，每次爬取时随机取用其中一个账号的 Cookie。

账号池的核心模块有存储模块、获取模块、检测模块和接口模块，本次需要的环境有：pip install redis selenium flask loguru environs

<https://github.com/Python3WebSpider/AccountPool/tree/antispider6>

页面智能解析

异步爬虫

打开浏览器访问一下 <https://www.httpbin.org/delay/5> 这个链接，发现等待了 5 秒后才出现页面

下面使用了循环的方法请求 100 次该网址，发现最终耗时超过 10 分钟了，这显然是有些慢的，我们可以使用协程提高爬取的速度

```
import requests
import logging
import time

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

url = 'https://www.httpbin.org/delay/5'
total_number = 100

start_time = time.time()
for i in range(1, total_number + 1):
    logging.info(f'scraping {url}')
    response = requests.get(url)
end_time = time.time()
# 2025-01-16 09:23:43,910 - INFO - Total time: 622.6026341915131 seconds
logging.info(f'Total time: {end_time - start_time} seconds')
```

协程

在介绍协程之前需要了解阻塞和非阻塞、同步和异步、多进程和协程的概念

协程是一种运行在用户态的轻量级线程，本质上是一个单进程

协程用法

使用协程最常用的库是 asyncio

```
import asyncio # 引入 asyncio 才能使用关键字 async、await

def async_intro01():
    # 定义一个 execute() 方法
    async def execute(x):
        print('执行: ', x)

    # 直接调用后并不会执行，而是返回一个 coroutine 对象
    coroutine = execute(1)
    print('coroutine 对象:', coroutine)

    # 创建一个事件循环 loop
    loop = asyncio.get_event_loop()
    # 将协程对象注册到事件循环中
    loop.run_until_complete(coroutine)
    print('事件循环 loop')

# coroutine 对象: <coroutine object async_intro01.<locals>.execute at
0x000001693DE33510>
# 执行: 1
# 事件循环 loop

....
```

coroutine: 在 Python 中常指代协程对象类型，我们可以将协程对象注册到事件循环中，它会被事件循环调用。可以使用 `async` 关键字来定义一个方法，这个方法在调用时不会立即被执行，而是会返回一个协程对象

event_loop: 事件循环，相当于一个无限循环，我们可以把一些函数注册到这个事件循环上，当满足发生条件的时候，就调用对应的处理方法

task: 任务，这是对协程对象的进一步封装，包含协程对象的各个状态

future: 代表将来执行或者没有执行的任务的结果，实际上和 `task` 没有本质区别

随着 Python 的更新，上面的写法已经不再适用

```
def async_intro02():
    # 定义一个 execute() 方法
    async def execute(x):
        print('运行 execute 函数:', x)

    # 使用 asyncio.run() 来运行协程
    asyncio.run(execute(1))

    # 这行代码将在协程执行完毕后执行
    print('loop')

    # 运行 execute 函数: 1
    # loop
```

```
def async_intro03():
    # 定义一个 execute() 方法
    async def execute(x):
        print('运行 execute 函数:', x)
        return x

    # 直接调用后并不会执行，而是返回一个 coroutine 对象
    coroutine = execute(1)
    print('coroutine 对象:', coroutine)

    # 创建一个事件循环 loop
    loop = asyncio.get_event_loop()
    # 将协程对象转化为 task 对象
    task = loop.create_task(coroutine)
    print('转化 task:', task)
    # 将task对象注册到事件循环中
    loop.run_until_complete(task)
    print('注册 task:', task)
    print('loop')

# coroutine 对象: <coroutine object async_intro03.<locals>.execute at
0x0000023685123510>
# 转化 task: <Task pending name='Task-1' coro=<async_intro03.<locals>.execute() running
at D:\workspace\crawler_async\async_base.py:52>>
# 运行 execute 函数: 1
# 注册 task: <Task finished name='Task-1' coro=<async_intro03.<locals>.execute() done,
defined at D:\workspace\crawler_async\async_base.py:52> result=1>
# loop

def async_intro04():
    # 定义一个 execute() 方法
    async def execute(x):
        print('运行 execute 函数:', x)
        return x

    # 直接调用后并不会执行，而是返回一个 coroutine 对象
    coroutine = execute(1)
    print('coroutine 对象:', coroutine)

    # 定义 task 对象
    task = asyncio.ensure_future(coroutine)
    # 创建一个事件循环 loop
    loop = asyncio.get_event_loop()
    # 将 task 对象注册到事件循环中
    loop.run_until_complete(task)
    print('注册 task:', task)
    print('loop')

# coroutine 对象: <coroutine object async_intro04.<locals>.execute at
0x0000013DAF7D3510>
# 运行 execute 函数: 1
# 注册 task: <Task finished name='Task-1' coro=<async_intro04.<locals>.execute() done,
defined at D:\workspace\crawler_async\async_base.py:79> result=1>
```

```
# loop

# async_intro01()
# async_intro02()
# async_intro03()
# async_intro04()
```

绑定回调

下面我们给 task 对象绑定回调方法

```
import asyncio
import requests

# 定义 request 方法，注意没有打印语句
async def request():
    url = 'https://www.baidu.com'
    status = requests.get(url)
    return status

# 定义 callback 方法，打印 task 对象
def callback(task):
    print('task 对象的状态: ', task.result())

# 当协程对象执行完成后，再执行回调方法

# 定义协程对象
coroutine = request()
# 定义 task 对象
task = asyncio.ensure_future(coroutine)
# 将回调函数传递给 task 对象
task.add_done_callback(callback)
print('task 对象: ', task)

# 定义事件循环
loop = asyncio.get_event_loop()
# 将 task 对象注册到事件循环中
loop.run_until_complete(task)
print('task 注册: ', task)
print('loop')

# task 对象: <Task pending name='Task-1' coro=<request() running at
D:\workspace\crawler_async\async_callback.py:5> cb=[callback() at
D:\workspace\crawler_async\async_callback.py:11]>
# task 对象的状态: <Response [200]>
# task 注册: <Task finished name='Task-1' coro=<request() done, defined at
D:\workspace\crawler_async\async_callback.py:5> result=<Response [200]>>
# loop
```

多任务协程

执行多次请求时可以定义一个 task 列表

```
import asyncio
import requests

async def request():
    url = 'https://www.baidu.com'
    status = requests.get(url)
    return status

# 创建 5 个 task
tasks = [asyncio.ensure_future(request()) for _ in range(5)]
print('task:', tasks)

loop = asyncio.get_event_loop()
# 先传递给 wait() 方法，再注册到事件循环中
loop.run_until_complete(asyncio.wait(tasks))

for task in tasks:
    print(task.result())
```

协程的实现

这里先演示几个错误用法

```
import asyncio
import requests
import time

start = time.time()

async def request():
    url = 'https://www.httpbin.org/delay/5'
    response = requests.get(url)
    # print(response.text)

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('用时: ', end - start)
```

实现异步处理，需要有挂起操作，await 关键字可以将耗时等待的操作挂起，直到其他协程挂起或者执行完毕

```
import asyncio
import requests
import time

start = time.time()

async def request():
    url = 'https://www.httpbin.org/delay/5'
    response = await requests.get(url)

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('用时: ', end - start)
# TypeError: object Response can't be used in 'await' expression
```

但 await 不能和 requests 的 response 对象共同使用，我们把 await 后面的对象改为协程对象

```
import asyncio
import requests
import time

start = time.time()

async def get(url):
    return requests.get(url)

async def request():
    url = 'https://www.httpbin.org/delay/5'
    print('waiting for', url)
    response = await get(url)
    print('Response:', response)

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('用时: ', end - start)
# 用时: 61.90195631980896
```

但协程还不是异步执行的，只有使用支持异步操作的请求方式才可以实现真正的异步，这需要使用 aiohttp

```
import asyncio
import aiohttp
import time

start = time.time()

async def get(url):
    session = aiohttp.ClientSession()
    # 这里我们使用了 await，后面跟着 get 方法
    response = await session.get(url)
    await response.text()
    await session.close()
    return response

async def request():
    url = 'https://www.httpbin.org/delay/5'
    print('Waiting for:', url)
    response = await get(url)
    print('Response:', response)

tasks = [asyncio.ensure_future(request()) for _ in range(10)]
loop = asyncio.get_event_loop()
# 如果遇到 await，就会将当前协程挂起，转而执行其他协程，直到其他协程也挂起或执行完毕，再执行下一个协程
loop.run_until_complete(asyncio.wait(tasks))

end = time.time()
print('用时：', end - start)
# 用时： 6.60272216796875
```

下面以网站 <https://ssr1.scrape.center/> 为例，测试一下多个并发量耗时

```
import asyncio
import aiohttp
import time

def test(number):
    start = time.time()

    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://ssr1.scrape.center/'
        await get(url)

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

    end = time.time()
    print('用时：', end - start)
    # 用时： 0.000272216796875
```

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

end = time.time()
print('并发量: ', number, ' 用时: ', end - start)

for number in [1, 3, 5, 10, 15, 30, 50, 75, 100, 200]:
    test(number)
    time.sleep(1)

"""
并发量: 1 用时: 0.24360132217407227
并发量: 3 用时: 0.2914865016937256
并发量: 5 用时: 0.23633575439453125
并发量: 10 用时: 0.48740434646606445
并发量: 15 用时: 0.41086578369140625
并发量: 30 用时: 0.7979156970977783
并发量: 50 用时: 1.4050536155700684
并发量: 75 用时: 2.0033068656921387
并发量: 100 用时: 2.7045936584472656
并发量: 200 用时: 5.42927098274231
"""

```

aiohttp

前面介绍的 `asyncio` 模块，其内部实现了对 TCP、UDP、SSL 协议的异步操作，但是对于 HTTP 请求需要使用 `aiohttp` 实现

`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

使用服务端可以搭建一个支持异步处理的服务器，这个服务器就是用来处理请求并返回响应的，类似于 Django、Flask、Tornado 等一些 Web 服务器，客户端可以用来发起请求，类似于使用 `requests` 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，`aiohttp` 则是异步的

基本用法

下面介绍一下 `aiohttp` 的基本用法，爬取作者的个人博客

```
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__':
    asyncio.run(main())
```

传递参数

对于 URL 中需要加参数的请求可以使用 params 传参

```
import aiohttp
import asyncio

async def main():
    url = 'https://www.httpbin.org/get'
    params = {'name': 'Tom', 'age': 24}
    async with aiohttp.ClientSession() as session:
        async with session.get(url, params=params) as response:
            print(await response.text())

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

POST 请求

aiohttp 也支持其他类型的请求，如 POST、PUT、DELETE、HEAD、OPTIONS、PATCH

```
import aiohttp
import asyncio

async def main():
    url = 'https://www.httpbin.org/post'
    data = {'name': 'Tom', 'age': 24}
    async with aiohttp.ClientSession() as session:
        # POST 表单提交
        async with session.post(url, data=data) as response:
```

```
# 如果是 POST JSON 数据提交, 需要使用 json 参数
# async with session.post(url, json=data) as response:
#     print(await response.text())

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

响应信息

获取响应中的状态码、响应头、响应体等内容

```
import aiohttp
import asyncio

async def main():
    url = 'https://www.httpbin.org/post'
    data = {'name': 'Tom', 'age': 24}
    async with aiohttp.ClientSession() as session:
        async with session.post(url, data=data) as response:
            print('状态码: ', response.status)
            print('响应头: ', response.headers)
            # 返回的是协程对象就需要加 await
            print('响应体: ', await response.text())
            print('响应体二进制内容: ', await response.read())
            print('响应体 JSON 内容: ', await response.json())

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

超时设置

使用 ClientTimeout 对象设置超时

```
import aiohttp
import asyncio

async def main():
    url = 'https://www.httpbin.org/get'
    timeout = aiohttp.ClientTimeout(total=3)

    async with aiohttp.ClientSession(timeout=timeout) as session:
        async with session.get(url) as response:
            print('Status: ', response.status)
```

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

并发限制

超高并发对于目标网站来说是很危险的，我们可以使用 Semaphore 控制并发量

```
import aiohttp
import asyncio

# 最大并发量
concurrency = 3
url = 'https://www.baidu.com/'

# 创建一个信号量对象
semaphore = asyncio.Semaphore(concurrency)
session = None

async def scrape_api():
    async with semaphore:
        print(f'Scraping {url}')
        async with session.get(url) as response:
            await asyncio.sleep(1)
            return await response.text()

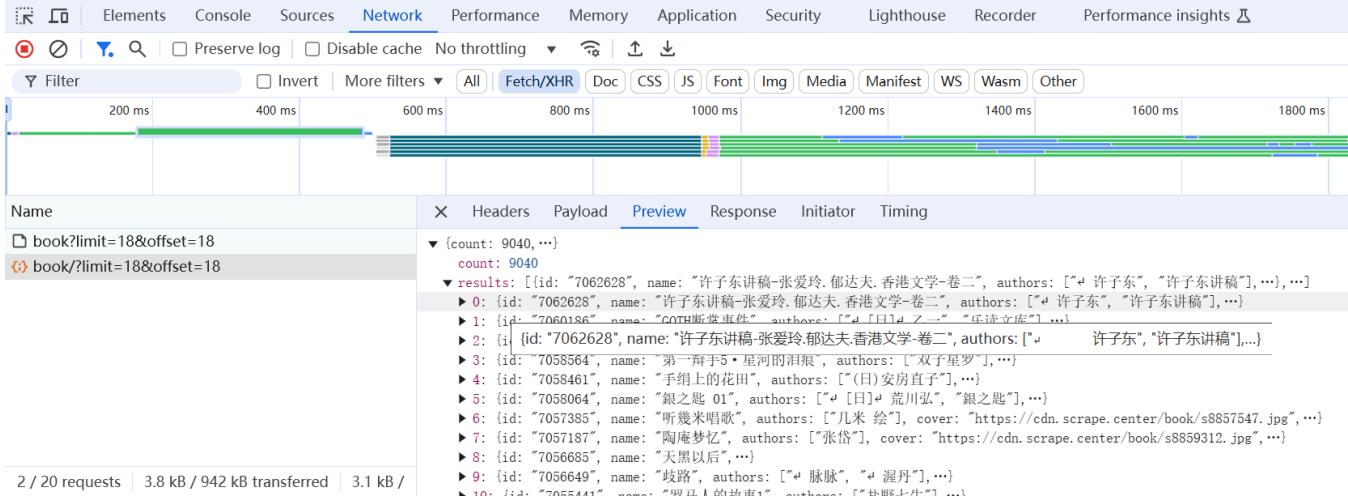
async def main():
    global session
    session = aiohttp.ClientSession()
    scrape_index_tasks = [asyncio.ensure_future(scrape_api()) for _ in range(10)]
    await asyncio.gather(*scrape_index_tasks)

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

案例：异步爬取图书信息

使用 aiohttp 爬取图书数据，并异步保存到 MongoDB 中

首先分析页面，这个网页 <https://spa5.scrape.center> 和之前 Ajax 的类似



经过编写和调试代码如下

```

import json
import aiohttp
import asyncio
import logging

from motor.motor_asyncio import AsyncIOMotorClient

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# https://spa5.scrape.center/api/book/?limit=18&offset=18
index_url = 'https://spa5.scrape.center/api/book/?limit=18&offset={offset}'
detail_url = 'https://spa5.scrape.center/api/book/{id}'

# 每页 18 本图书
page_book_number = 18
# 共 100 页
page_number = 100
# 并发量
concurrency = 3

# 数据库相关配置
from pymongo import MongoClient

client_string = 'mongodb://localhost:27017/'
client = AsyncIOMotorClient(client_string)
db = client['crawler']
collection = db['books_async']

semaphore = asyncio.Semaphore(concurrency)
session = None

# 列表页、详情页 通用爬取
async def scrape_api(url):
    pass

```

```
# 引入信号量作为上下文
async with semaphore:
    try:
        logging.info(f'Scraping {url}')
        # 请求网页, 返回 JSON 数据
        async with session.get(url) as response:
            return await response.json()
    except aiohttp.ClientError:
        logging.error(f'Failed to scrape {url}', exc_info=True)

# 保存数据
async def save_data(data):
    logging.info(f'Saving data {data} to database')
    if data:
        return await collection.update_one({'id': data['id']}, {'$set': data}, upsert=True)

# 爬取列表页
async def scrape_index(page):
    url = index_url.format(offset=page_book_number * (page - 1))
    return await scrape_api(url)

# 爬取详情页
async def scrape_detail(id):
    url = detail_url.format(id=id)
    data = await scrape_api(url)
    await save_data(data)

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))

    # 增加 results 解析
    id_list = []
    for index_data in results:
        if not index_data:
            continue
        for item in index_data.get('results'):
            id_list.append(item.get('id'))

    # 爬取详情页
    scrape_detail_tasks = [asyncio.ensure_future(scrape_detail(id)) for id in id_list]
    await asyncio.wait(scrape_detail_tasks)
    await session.close()

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

记录一个报错 ModuleNotFoundError: No module named 'backports' , 直接安装这个特定的子模块就行

The screenshot shows a PyCharm interface. In the top navigation bar, it says "PP py_web_crawler_dev_practice master". The left sidebar shows a project structure with chapters 01 through 07. The main code editor window has "netutil.py" selected, displaying Python code. The terminal window at the bottom shows a pip installation command:

```
安装最新的 PowerShell, 了解新功能和改进! https://aka.ms/PSWindows
(.venv) PS D:\workspace\py_web_crawler_dev_practice> pip install backports -i https://pypi.tuna.tsinghua.edu.cn/simple
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
ERROR: Could not find a version that satisfies the requirement backports (from versions: none)
ERROR: No matching distribution found for backports
(.venv) PS D:\workspace\py_web_crawler_dev_practice> pip install backports.ssl_match_hostname
Collecting backports.ssl_match_hostname
  Downloading backports.ssl_match_hostname-3.7.0.1.tar.gz (5.7 kB)
```

The command "pip install backports.ssl_match_hostname" is highlighted with a red box.

爬取结果如下

The screenshot shows the MongoDB Compass interface. On the left, there's a tree view of databases and collections. The "books_async" collection under the "crawler" database is selected. The main pane displays a list of document results. One specific document is expanded to show its fields:

```
_id: ObjectId('678a678487f570389a3fdc72')
id: "7916054"
authors: Array (2)
catalog: "
    小记
    无灯无月何妨(1)
    那些名字那些人(6)
    杨花满路春归了((11))
    ...
comments: Array (9)
cover: "https://cdn.scrape.center/book/s27250764.jpg"
introduction: ""
isbn: "9787511007230"
name: "清白家风"
page_number: 127
price: "15.00"
published_at: "2014-04-20T16:00:00Z"
publisher: "海豚出版社"
score: "7.5"
tags: Array (8)
translators: Array (empty)
updated_at: "2020-03-21T16:50:24.831994Z"
url: "https://book.douban.com/subject/7916054/"
```

代理的使用