8小时学会Scrapy网络爬虫

通过这个为期8小时的学习计划
你将从零开始掌握Scrapy网络爬虫的基本技能，并能独立完成数据抓取任务。

1 - 网络爬虫与Scrapy简介

- 网络爬虫概述
- Scrapy简介
- 练习
  - 安装Scrapy并创建一个简单的Scrapy项目

2 - 理解Scrapy项目结构

- Scrapy项目结构
- 练习
  - 创建并配置一个新的Scrapy项目

3 - 编写第一个爬虫

- 创建爬虫
- 运行爬虫
- 练习
  - 编写一个爬虫来抓取一个简单的网页并提取数据

4 - 数据提取与解析

- 使用Selectors
- 处理多页数据
- 练习
  - 使用CSS和XPath选择器提取网页中的特定数据
  - 添加标题、轴标签和图例

5 - 存储抓取的数据

- 数据存储选项
- 配置Item Pipeline
- 练习
  - 编写Item Pipeline，将抓取的数据保存为JSON文件或数据库

6 - 处理复杂的网站和数据

- 处理动态内容
- 登录和表单提交
- 练习
  - 抓取一个需要登录的网站或一个动态生成内容的网站

7 - 调试与优化

- 调试技巧
- 优化爬虫
- 练习
  - 使用Scrapy shell调试一个爬虫，并优化其抓取速度

8 - 综合项目实践

- 项目选择
- 项目实施
- 练习
  - 完成一个综合项目，将所学的技能全部应用到实际项目中

记得多实践，巩固所学内容。

@lifeboat

第1小时：网络爬虫与Scrapy简介
第2小时：理解Scrapy项目结构
第3小时：编写第一个爬虫
第4小时：数据提取与解析
第5小时：存储抓取的数据
第6小时：处理复杂的网站和数据
第7小时：调试与优化
第8小时：综合项目实践

第1小时：网络爬虫与Scrapy简介

目标： 了解网络爬虫的基本概念以及Scrapy框架。

- **网络爬虫概述：**
  - **定义和应用：** 网络爬虫是通过程序自动化从网页上抓取数据的技术，广泛应用于数据分析、信息收集等领域。
  - **法律和伦理考虑：** 抓取数据时需遵守相关法律法规，避免侵犯版权和隐私权，尊重网站的robots.txt文件。
- **Scrapy简介：**
  - **什么是Scrapy：** Scrapy是一个用于爬取网站并从网页中提取数据的Python框架。它高效、灵活、可扩展，适合处理复杂的爬取任务。
  - **为什么使用Scrapy：** Scrapy具有高效的数据提取、处理和存储能力，支持并发抓取，提供丰富的中间件和扩展功能。
  - **安装和设置：**
    1. 安装Scrapy：

       ```
       pip install scrapy
       ```

    2. 创建一个Scrapy项目：

       ```
       scrapy startproject myproject
       ```

  - **基本概念：**
    - **爬虫（Spiders）：** 定义如何从网站（或一组网站）抓取信息的类。
    - **Items：** 定义抓取的数据结构。
    - **Pipelines：** 处理抓取数据的组件。

练习：

- **安装Scrapy并创建一个简单的Scrapy项目：**
  1. 打开终端，安装Scrapy：

     ```
     pip install scrapy
     ```

  2. 创建一个新的Scrapy项目：

     ```
     scrapy startproject myproject
     ```

  3. 进入项目目录：

     ```
     cd myproject
     ```

  4. 创建一个新的爬虫：

     ```
     scrapy genspider example example.com
     ```

  5. 运行爬虫：

     ```
     scrapy crawl example
     ```

通过以上练习，你将学会安装和设置Scrapy，并创建和运行一个简单的Scrapy项目，为接下来的学习打下基础。

第2小时：理解Scrapy项目结构

目标： 学习Scrapy项目的结构和组成部分。

- **Scrapy项目结构：**
  - **项目目录和文件解释：**
    - **myproject/**: 项目根目录。
    - **myproject/settings.py**: 配置文件，用于设置项目参数。
    - **myproject/items.py**: 定义数据结构。
    - **myproject/middlewares.py**: 定义中间件。
    - **myproject/pipelines.py**: 处理抓取数据的管道。
    - **myproject/spiders/**: 存放爬虫文件的目录。
  - **settings.py的配置：**
    - 设置USER_AGENT：

```
USER_AGENT = 'myproject (+http://www.yourdomain.com)'
```

- 设置下载延迟：

```
DOWNLOAD_DELAY = 2
```

- 启用管道：

```
ITEM_PIPELINES = {
    'myproject.pipelines.MyProjectPipeline': 300,
}
```

- **items.py和pipelines.py的用途：**
  - **items.py**: 定义要抓取的数据字段。

```
import scrapy

class MyProjectItem(scrapy.Item):
    title = scrapy.Field()
    link = scrapy.Field()
    desc = scrapy.Field()
```

  - **pipelines.py**: 处理和存储抓取的数据。

```
class MyProjectPipeline:
    def process_item(self, item, spider):
        # 处理item
        return item
```

练习：

- **创建并配置一个新的Scrapy项目：**
  1. 创建一个新的Scrapy项目：

```
scrapy startproject newproject
```

  2. 进入项目目录：

```
cd newproject
```

  3. 配置settings.py文件：

```
USER_AGENT = 'newproject (+http://www.yourdomain.com)'
DOWNLOAD_DELAY = 2
ITEM_PIPELINES = {
    'newproject.pipelines.NewProjectPipeline': 300,
}
```

4. 在items.py中定义数据结构：

```
import scrapy

class NewProjectItem(scrapy.Item):
    title = scrapy.Field()
    link = scrapy.Field()
    desc = scrapy.Field()
```

5. 在pipelines.py中编写处理数据的代码：

```
class NewProjectPipeline:
    def process_item(self, item, spider):
        # 处理item
        return item
```

通过以上练习，你将理解Scrapy项目的结构，并能够配置settings.py文件和定义items及pipelines，为编写爬虫打下坚实的基础。

第3小时：编写第一个爬虫

目标： 学习如何编写和运行基本的Scrapy爬虫。

- **创建爬虫：**
  - **爬虫类和start_requests方法：**
    - 在Scrapy中，爬虫类继承自`scrapy.Spider`类。
    - `start_requests`方法定义了爬虫的初始请求，可以覆盖该方法自定义起始请求。

```
import scrapy

class ExampleSpider(scrapy.Spider):
    name = "example"
    start_urls = ['http://example.com']

    def start_requests(self):
        for url in self.start_urls:
            yield scrapy.Request(url=url,
callback=self.parse)
```

- **编写parse方法来解析网页内容：**
  - parse方法用于处理响应数据，提取所需信息。

```
def parse(self, response):
    for quote in response.css('div.quote'):
        yield {
            'text': quote.css('span.text::text').get(),
            'author':
quote.css('small.author::text').get(),
            'tags': quote.css('div.tags
a.tag::text').getall(),
        }
```

- **运行爬虫：**
  - 使用Scrapy命令行工具运行爬虫。

```
scrapy crawl example
```

练习：

- **编写一个爬虫来抓取一个简单的网页并提取数据：**
  1. 在spiders目录下创建一个新的爬虫文件example_spider.py：

```
import scrapy

class ExampleSpider(scrapy.Spider):
    name = "example"
    start_urls = ['http://quotes.toscrape.com']

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
```

```
                'text': quote.css('span.text::text').get(),
                'author':
quote.css('small.author::text').get(),
                'tags': quote.css('div.tags
a.tag::text').getall(),
            }

        next_page = response.css('li.next
a::attr(href)').get()
        if next_page is not None:
            next_page = response.urljoin(next_page)
            yield scrapy.Request(next_page,
callback=self.parse)
```

2. 运行爬虫：

```
scrapy crawl example
```

通过以上练习，你将学会如何编写和运行基本的Scrapy爬虫，能够抓取简单网页并提取数据，为进一步学习数据提取和处理打下基础。

第4小时：数据提取与解析

目标： 学习如何从网页中提取有用的数据。

- **使用Selectors：**
  - **CSS选择器和XPath选择器：**
    - **CSS选择器：** 使用CSS选择器提取网页中的元素，类似于jQuery的选择器语法。

      ```
      title = response.css('title::text').get()
      ```

    - **XPath选择器：** 使用XPath语法提取网页中的元素，类似于XML路径语法。

      ```
      title = response.xpath('//title/text()').get()
      ```

  - **使用response对象提取数据：**
    - response对象包含了网页的内容和其他有用的信息，可以使用css和xpath方法进行数据提取。

```
text = response.css('div.quote span.text::text').get()
author = response.css('small.author::text').get()
tags = response.css('div.tags a.tag::text').getall()
```

- **处理多页数据：**
  - **处理分页和链接追踪：**
    - 使用response对象获取下一页的链接，继续抓取数据。

```
next_page = response.css('li.next a::attr(href)').get()
if next_page is not None:
    next_page = response.urljoin(next_page)
    yield scrapy.Request(next_page, callback=self.parse)
```

练习：

- **使用CSS和XPath选择器提取网页中的特定数据：**
  1. 修改example_spider.py，使用CSS选择器提取网页中的标题、作者和标签：

```
import scrapy

class ExampleSpider(scrapy.Spider):
    name = "example"
    start_urls = ['http://quotes.toscrape.com']

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').get(),
                'author':
quote.css('small.author::text').get(),
                'tags': quote.css('div.tags
a.tag::text').getall(),
            }

        next_page = response.css('li.next
a::attr(href)').get()
        if next_page is not None:
            next_page = response.urljoin(next_page)
            yield scrapy.Request(next_page,
callback=self.parse)
```

  2. 使用XPath选择器提取相同的数据：

```
import scrapy

class ExampleSpider(scrapy.Spider):
    name = "example"
    start_urls = ['http://quotes.toscrape.com']

    def parse(self, response):
        for quote in response.xpath('//div[@class="quote"]'):
            yield {
                'text':
quote.xpath('span[@class="text"]/text()').get(),
                'author':
quote.xpath('span/small[@class="author"]/text()').get(),
                'tags':
quote.xpath('div[@class="tags"]/a[@class="tag"]/text()').getal
l(),
            }

        next_page =
response.xpath('//li[@class="next"]/a/@href').get()
        if next_page is not None:
            next_page = response.urljoin(next_page)
            yield scrapy.Request(next_page,
callback=self.parse)
```

通过以上练习，你将掌握使用CSS和XPath选择器从网页中提取数据的技巧，并能够处理多页数据，为后续的数据存储和处理奠定基础。

第5小时：存储抓取的数据

目标： 学习如何将抓取的数据保存到不同的格式和数据库中。

- **数据存储选项：**
  - **JSON文件存储：**
    - Scrapy可以将抓取的数据直接保存为JSON文件。

      ```
      scrapy crawl example -o quotes.json
      ```
  - **CSV文件存储：**
    - Scrapy可以将抓取的数据直接保存为CSV文件。

      ```
      scrapy crawl example -o quotes.csv
      ```

- **数据库存储（如SQLite、MongoDB）：**
  - 使用Scrapy Item Pipeline将数据保存到数据库中。

- **配置Item Pipeline：**
  - **编写并配置Item Pipeline来处理数据存储：**
    - 在pipelines.py文件中编写Pipeline类，将数据保存到JSON文件或数据库中。

```
import json

class JsonWriterPipeline:
    def open_spider(self, spider):
        self.file = open('items.json', 'w')

    def close_spider(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item
```

    - 在settings.py中启用Pipeline。

```
ITEM_PIPELINES = {
    'myproject.pipelines.JsonWriterPipeline': 300,
}
```

练习：

- **编写Item Pipeline，将抓取的数据保存为JSON文件或数据库：**
  1. 在pipelines.py文件中编写将数据保存为JSON文件的Pipeline类。

```
import json

class JsonWriterPipeline:
    def open_spider(self, spider):
        self.file = open('items.json', 'w')

    def close_spider(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item
```

2. 在settings.py中启用Pipeline。

```
ITEM_PIPELINES = {
    'newproject.pipelines.JsonWriterPipeline': 300,
}
```

3. 如果需要将数据保存到SQLite数据库中，可以编写SQLite Pipeline。

```
import sqlite3

class SQLitePipeline:
    def open_spider(self, spider):
        self.connection = sqlite3.connect("quotes.db")
        self.cursor = self.connection.cursor()
        self.cursor.execute('''
            CREATE TABLE quotes(
                text TEXT,
                author TEXT,
                tags TEXT
            )
        ''')
        self.connection.commit()

    def close_spider(self, spider):
        self.connection.close()

    def process_item(self, item, spider):
        self.cursor.execute('''
            INSERT INTO quotes (text, author, tags) VALUES(?,
?, ?)
```

```
            ''', (item.get('text'), item.get('author'),
','.join(item.get('tags'))))
            self.connection.commit()
            return item
```

4. 在settings.py中启用SQLite Pipeline。

```
ITEM_PIPELINES = {
    'newproject.pipelines.SQLitePipeline': 300,
}
```

通过以上练习，你将学会如何将抓取的数据保存为JSON文件、CSV文件或存储到数据库中，并能够配置和使用Scrapy的Item Pipeline来处理数据存储，为数据的进一步分析和使用做好准备。

第6小时：处理复杂的网站和数据

目标： 学习处理复杂的网站结构和数据。

- **处理动态内容：**
  - **使用Scrapy与Selenium结合：**
    - Scrapy本身无法处理JavaScript生成的内容，但可以结合Selenium来解决这个问题。
    - 安装Selenium和浏览器驱动：

      ```
      pip install selenium
      ```

    - 示例代码：

      ```
      from selenium import webdriver
      from scrapy import signals
      from scrapy.http import HtmlResponse

      class SeleniumMiddleware:
          @classmethod
          def from_crawler(cls, crawler):
              s = cls()
              crawler.signals.connect(s.spider_closed,
      signal=signals.spider_closed)
              return s

          def __init__(self):
      ```

```
        self.driver = webdriver.Chrome()

    def process_request(self, request, spider):
        self.driver.get(request.url)
        body = self.driver.page_source
        return HtmlResponse(self.driver.current_url,
body=body, encoding='utf-8', request=request)

    def spider_closed(self, spider):
        self.driver.quit()
```

- **处理JavaScript生成的内容：**
  - 使用Selenium加载页面，并将加载后的内容交给Scrapy处理。
  - 在settings.py中启用Selenium中间件：

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.SeleniumMiddleware': 543,
}
```

- **登录和表单提交：**
  - **模拟登录和表单提交：**
    - 使用Scrapy的FormRequest类来提交表单。
    - 示例代码：

```
import scrapy

class LoginSpider(scrapy.Spider):
    name = 'login'
    start_urls = ['http://example.com/login']

    def parse(self, response):
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'yourusername',
'password': 'yourpassword'},
            callback=self.after_login
        )

    def after_login(self, response):
        # 检查是否成功登录
        if "authentication failed" in response.body:
            self.logger.error("Login failed")
```

```
            return
        # 继续抓取其他页面
        yield
scrapy.Request(url='http://example.com/after_login',
callback=self.parse_page)

    def parse_page(self, response):
        # 解析登录后的页面
        pass
```

练习：

- **抓取一个需要登录的网站或一个动态生成内容的网站：**
  1. **处理动态内容：**
     - 编写一个新的Scrapy项目，配置Selenium中间件，编写爬虫处理JavaScript生成的内容。
     - 示例：

```
import scrapy
from selenium import webdriver
from scrapy.http import HtmlResponse

class DynamicContentSpider(scrapy.Spider):
    name = 'dynamic'
    start_urls = ['http://example.com/dynamic']

    def __init__(self):
        self.driver = webdriver.Chrome()

    def parse(self, response):
        self.driver.get(response.url)
        body = self.driver.page_source
        response = HtmlResponse(self.driver.current_url,
body=body, encoding='utf-8', request=response)
        # 解析动态内容
        for quote in response.css('div.quote'):
            yield {
                'text':
quote.css('span.text::text').get(),
                'author':
quote.css('small.author::text').get(),
                'tags': quote.css('div.tags
a.tag::text').getall(),
```

```
            }
        self.driver.quit()
```

2. **模拟登录和表单提交：**
    - 编写一个新的Scrapy项目，使用FormRequest模拟登录并抓取登录后的页面。
    - 示例：

```
import scrapy

class LoginSpider(scrapy.Spider):
    name = 'login'
    start_urls = ['http://quotes.toscrape.com/login']

    def parse(self, response):
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'yourusername',
'password': 'yourpassword'},
            callback=self.after_login
        )

    def after_login(self, response):
        if "Login failed" in response.body:
            self.logger.error("Login failed")
            return
        yield
scrapy.Request(url='http://quotes.toscrape.com/',
callback=self.parse_page)

    def parse_page(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text':
quote.css('span.text::text').get(),
                'author':
quote.css('small.author::text').get(),
                'tags': quote.css('div.tags
a.tag::text').getall(),
            }
```

通过以上练习，你将学会处理动态内容和模拟登录及表单提交的技巧，能够抓取复杂的网站数据，为实际应用中的复杂爬取任务做好准备。

第7小时：调试与优化

目标： 学习如何调试和优化Scrapy爬虫。

- **调试技巧：**
  - **使用Scrapy shell调试：**
    - Scrapy shell是一个交互式工具，用于测试和调试选择器、XPath等。
    - 启动Scrapy shell：

      ```
      scrapy shell 'http://quotes.toscrape.com'
      ```

    - 测试选择器和提取数据：

      ```
      response.css('title::text').get()
      response.xpath('//title/text()').get()
      ```

  - **常见错误和解决方法：**
    - 404错误：检查URL是否正确。
    - 选择器未匹配到数据：检查选择器语法是否正确。
    - 爬虫卡住：检查是否进入了死循环，或者网页加载时间过长。
- **优化爬虫：**
  - **并发抓取设置：**
    - 增加并发请求数，提高抓取速度：

      ```
      CONCURRENT_REQUESTS = 32
      ```

    - 减少每个域名的并发请求数，避免封禁：

      ```
      CONCURRENT_REQUESTS_PER_DOMAIN = 16
      ```

  - **避免被封禁的技巧（如设置延迟、使用代理）：**
    - 设置下载延迟，避免过于频繁的请求：

      ```
      DOWNLOAD_DELAY = 3
      ```

    - 使用随机User-Agent：

      ```
      USER_AGENT = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
      AppleWebKit/537.36 (KHTML, like Gecko)
      Chrome/58.0.3029.110 Safari/537.3'
      ```

- 使用代理：

```
PROXY = 'http://yourproxy.com:8000'
```

- 使用中间件设置代理：

```
DOWNLOADER_MIDDLEWARES = {

 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddlew
are': 1,
    'myproject.middlewares.ProxyMiddleware': 100,
}

class ProxyMiddleware:
    def process_request(self, request, spider):
        request.meta['proxy'] =
'http://yourproxy.com:8000'
```

练习：

- **使用Scrapy shell调试一个爬虫，并优化其抓取速度：**
  1. 启动Scrapy shell，测试选择器：

```
scrapy shell 'http://quotes.toscrape.com'
```

     测试选择器提取数据：

```
response.css('div.quote span.text::text').getall()
response.xpath('//div[@class="quote"]/span[@class="text"]/text
()').getall()
```

  2. 优化爬虫的抓取速度：
     - 修改settings.py文件，设置并发请求数和下载延迟：

```
CONCURRENT_REQUESTS = 32
DOWNLOAD_DELAY = 2
CONCURRENT_REQUESTS_PER_DOMAIN = 16
```

     - 使用随机User-Agent：

```
USER_AGENT = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/58.0.3029.110 Safari/537.3'
```

- 使用代理：

```
PROXY = 'http://yourproxy.com:8000'
```

在middlewares.py中添加代理中间件：

```
class ProxyMiddleware:
    def process_request(self, request, spider):
        request.meta['proxy'] =
'http://yourproxy.com:8000'
```

第8小时：综合项目实践

目标： 综合运用所学知识，完成一个实际的Scrapy项目。

- **项目选择：**
  - 选择一个包含多页数据、动态内容或需要登录的网站进行数据抓取。例如，可以选择一个新闻网站或电子商务网站。
- **项目实施：**
  - **步骤1：创建Scrapy项目**
    - 创建并配置Scrapy项目，包括settings.py、items.py和pipelines.py。

```
scrapy startproject complexproject
cd complexproject
```

  - **步骤2：编写爬虫**
    - 创建爬虫类，处理网站的抓取和数据提取。

```
import scrapy
from selenium import webdriver
from scrapy.http import HtmlResponse

class ComplexSpider(scrapy.Spider):
    name = 'complex'
    start_urls = ['http://example.com/login']

    def __init__(self):
        self.driver = webdriver.Chrome()

    def parse(self, response):
        # 模拟登录
```

```
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'yourusername', 'password':
'yourpassword'},
            callback=self.after_login
        )

    def after_login(self, response):
        if "Login failed" in response.body:
            self.logger.error("Login failed")
            return
        # 登录成功后开始抓取数据
        for url in self.start_urls:
            yield scrapy.Request(url=url,
callback=self.parse_page)

    def parse_page(self, response):
        self.driver.get(response.url)
        body = self.driver.page_source
        response = HtmlResponse(self.driver.current_url,
body=body, encoding='utf-8', request=response)
        # 解析数据
        for item in response.css('div.item'):
            yield {
                'title': item.css('h2.title::text').get(),
                'description':
item.css('div.description::text').get(),
            }
        # 处理分页
        next_page = response.css('a.next::attr(href)').get()
        if next_page is not None:
            next_page = response.urljoin(next_page)
            yield scrapy.Request(next_page,
callback=self.parse_page)

    def closed(self, reason):
        self.driver.quit()
```

- **步骤3：配置数据存储**
  - 在pipelines.py中配置数据存储，例如保存为JSON文件或存储到数据库。

```
import json
import sqlite3
```

```
class JsonWriterPipeline:
    def open_spider(self, spider):
        self.file = open('items.json', 'w')

    def close_spider(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item

class SQLitePipeline:
    def open_spider(self, spider):
        self.connection = sqlite3.connect("items.db")
        self.cursor = self.connection.cursor()
        self.cursor.execute('''
            CREATE TABLE items(
                title TEXT,
                description TEXT
            )
        ''')
        self.connection.commit()

    def close_spider(self, spider):
        self.connection.close()

    def process_item(self, item, spider):
        self.cursor.execute('''
            INSERT INTO items (title, description) VALUES(?,
?)
        ''', (item.get('title'), item.get('description')))
        self.connection.commit()
        return item
```

- 在settings.py中启用Pipeline。

```
ITEM_PIPELINES = {
    'complexproject.pipelines.JsonWriterPipeline': 300,
    'complexproject.pipelines.SQLitePipeline': 400,
}
```

练习：

- **完成一个综合项目，将所学的技能全部应用到实际项目中：**
    1. **创建并配置项目：**

    ```
    scrapy startproject complexproject
    cd complexproject
    ```

    2. **编写爬虫：**
        - 在spiders目录下创建爬虫文件complex_spider.py，编写爬虫类，处理登录、数据提取和分页。
    3. **配置数据存储：**
        - 在pipelines.py中编写保存数据的Pipeline类。
        - 在settings.py中启用Pipeline。
    4. **运行爬虫：**

    ```
    scrapy crawl complex
    ```

通过以上练习，你将能够综合运用所学的Scrapy爬虫知识，完成一个实际的爬虫项目，抓取复杂网站的数据并进行处理和存储，为实际应用中的数据抓取任务提供支持。

通过以上8小时的学习计划，你将能够从零开始掌握Scrapy网络爬虫的基本技能，并能独立完成数据抓取任务。