11 编写网易爬虫NeteaseSpider让它“爬”起来

更新时间：2019-06-14 14:34:54

> 时间像海绵里的水，只要你愿意挤，总还是有的。
>
> ——鲁迅

将对目标网页的分析融入到爬虫代码中，怎样在蜘蛛内部用最简洁的办法从复杂的网页内容中提取数据。从认识 scrapy 的配置文件`settings.py`理解 scrapy 的组件化思想。

在重新进入我们的网易爬虫示例之前，先来回顾一下前面章节的内容。首先在第一节我介绍了分页网页结构的基本方法与思路，重点是了解`<a>`和`<link>`这两类蜘蛛最关注元素，如果按层层步进的方式爬取网页内容称之为**深度爬网**，与之对应的是在同一链接层次上先爬取所有相邻页面节点的内容，再向下步进称之为**广度爬网**，还有就是将两者相结合的**泛爬网**方式。从这些爬取路径了解到蜘蛛是怎么在网站上**爬行**的。由此引入了`CrawlSpider`来实现**泛爬**。

编写 `NeteaseSpider` 蜘蛛

开始`CrawlSpider`的学习就很自然地接触到**正则表达式**，也从这个点上我用了整整三个小节来系统化地介绍了正则表达式的相关内容，其实当我们完全理解了正则表达式之后，就可以说是大体上掌握了对`CrawlSpider`的基本用法，它的主要使用路径可以总结如下：

1. 找出种子页
2. 分析与定义Item类的结构
3. 根据你的爬网策略编写`rules`属性内的爬网规则和链接提取器
4. 编写`parse_item`方法分析网页的响应内容并从中提取出 Item 所需要的数据内容

我们在第一节已经知道网易的网站结构是由多个不同子域来划分，例如：

- https://news.163.com - 新闻

- https://wars.163.com - 军事
- https://sports.163.com -体育
- https://v.163.com -直播
- 诸如此类

那么既然有这么多的子域，我们是不是都要一一加入到`start_urls`内作为种子页呢？对于这种问题可以分两种情况来分析：

1. 如果你已经很明确要爬取的内容是在哪个子域下，又或者你只想爬取某几个子域下的内容就可以在`start_urls`下进行逐一添加，并通过`allowed_domains`声明只可以爬取的域；
2. 如果我们并不知道网站有多少个子域并且我们希望爬虫可以爬取整个网站内的每个子域，那么我们只需要将`www.163.com`(首页)作为唯一的种子页，由首页进入，对不符合目标网页的URL规则执行步进，当遇到符合规则的URL则进行数据的提取并停止步进。

对于我们的示例将会采用第二种爬取策略，尽可能多地从网站中爬取各个栏目的内容。首先可以打开几个网易上的新闻详情页，不难发现这些页面都与以下的地址形式类似:

```
https://news.163.com/19/0501/08/EE32CO3K000189FH.html
```

由此对URL进行一般化的定义：

```
[协议定义]://[域]/[2位数字]/[4位数字]/[2位数字]/[页面名]
```

那么与之对应的正则表达式则如下所示：

```
(\w+):\/\/([^/:]+)\/(\d{2})+\/(\d{4})+\/(\d{2})+\/([^#]*)
```

对于符合上述URL规则的地址就让蜘蛛执行`parse_item`方法。代码如下所示：

```
# -*- coding: utf-8 -*-
import scrapy
from scrapy import Selector
from scrapy.linkextractors import LinkExtractor
from scrapy.spiders import CrawlSpider, Rule
from ..items import NewsItem

class NeteaseSpider(CrawlSpider):
    name = 'netease'
    allowed_domains = ['163.com']
    start_urls = ['http://163.com/']
    rules = (
    Rule(LinkExtractor(allow=r'(\w+):\/\/([^/:]+)\/(\d{2})+\/(\d{4})+\/(\d{2})+\/([^#]*)'), callback='parse_item', follow=T
rue),
    )
```

编写 `parse_item` 分析代码

根据在第一节中对新闻详情页分析得出的元素与字段关系映射表：

名称	字段	选择器
标题	title	`#epContentLeft>h1`
发表日期	pub_date	`#epContentLeft .post_time_source`
摘要	desc	`#epContentLeft .post_desc`
正文	body	`#endText`
链接	link	当前请求中的URL

很容易得到`parse_item`的代码内容：

```
def parse_item(self,response):
  item = NewsItem() # NewsItem编写详情见本章第一小节
  selector = Selector(response)
  item['title'] = selector.css('#epContentLeft>h1::text').get()
  item['pub_date'] = selector.css('#epContentLeft .post_time_source::text').get()
  if item['pub_date'] is not None:
    item['pub_date'] = item['pub_date'].split()[0]
  item['desc'] = selector.css('#epContentLeft .post_desc::text').get()
  if item['desc'] is not None:
    item['desc'] = item['desc'].strip()
  item['body'] = selector.css('#endText::text').get()
  if item['body'] is not None:
    item['body'] = item['body'].strip()
  item['link'] = response.url
  return item
```

深入选择器 xpath, css

我在"动手开发最简单的单文件爬虫"一节中就使用过`pyQuery`对返回的响应内容进行分析提取，如果由此入手去百度或谷歌一下你会发现不少关于 beautifulsoup4 与 pyQuery 的对比的文章，过去我曾对此产生过选择困难症：到底 beautifulsoup4、pyQuery 和 Scrapy 内自带的选择器到底哪个更好呢 ？

我们就借本节的示例用具体代码来对比一下这三者的优劣然后再作出选择，首先是beautifulsoup4 ，用bs4的`parse_item`代码如下所示：

```
from bs4 import BeautifulSoup

def parse_item(self, response):
  soup = BeautifulSoup(response.body, 'html.parser')
  item = NewsItem()
  item['title'] = soup.select('#epContentLeft>h1').text()
  item['pub_date'] = soup.select('#epContentLeft .post_time_source').text()
  item['desc'] = soup.select('#epContentLeft .post_desc').text()
  item['body'] = soup.select('#endText').text()
  return item
```

如果换成pyQuery则代码如下所示：

```
from pyQuery import PyQuery as pq

def parse_item(self, response):
  doc = pq(response.body)
  item = NewsItem()
  item['title'] = doc('#epContentLeft>h1').text()
  item['pub_date'] = doc('#epContentLeft .post_time_source').text()
  item['desc'] = doc('#epContentLeft .post_desc').text()
  item['body'] = doc('#endText').text()
  return item
```

beautifulsoup4、pyQuery 在具体使用时两者语法大同小异，bs4 更 python 化一点，而 pyQuery 则更像python中的 jQuery。bs4 年代比较久远名声也大一些，不过性能确实很差。对爬取任务量巨大(爬取量在万级以上)时会有很明显的速度滞后的感觉，而且它对内存的消耗也比较大。而 pyQuery 则比 bs4 的性能要好一些，如果你曾对 jQuery 非常熟悉的话也推荐使用 pyQuery。

如果对 bs4 与 pyQuery 本身都没有什么认知的话，我建议"两者皆可抛"，直接使用 Scrapy 自带的选择器就好了，毕竟这是学习成本最低的，而且 Scrapy 自带的选择器速度一定不比前两者慢，甚至更好一些。

Scrapy 提供了 xpath 和 CSS 两种选择器，如果你熟悉XPath 那当然使用 XPath 的选择语法一定是速度最高的。悻然，即使你不懂XPath语法只学会 CSS 选择器的语法也能有 XPath 的分析速度，那是因为 Scrapy 会将 CSS 选择器直接转换为 XPath！这也是 Scrapy 在不断迭代更新后的一个非常友好的功能，如果去查看`scrapy.Selector.css`方法的源代码，你会发现这个方法是调用了`Selector`内部的一个`_css2xpath`方法实现的，`_css2xpath`正是一个自动将css选择器语法转成xpath语法的私有方法。我是个崇尚简洁实用主义的开发者，所以在 bs4 ， pyQuery 和 Selector 这三者之间，我会推荐你使用学习成本最低，速度最高的`scrapy.Selector`。

编写完`parse_item`方法，这个网易爬虫也可以宣告完成，在终端执行以下指令来执行：

```
$ scrapy crawl netease -o netease.json -s FEED_EXPORT_ENCODING='utf-8'
```

认识基本配置 —— scrapy 的组件化设计思想

Scrapy是由核心（core）模块、扩展插件（extension）、蜘蛛(spider)、管道(pipeline)、下载器中间件(Download middleware)和存储后端(Storage Backend)等多个部分的组成，这些模块各自负责爬取过程中对应环节中的某项细化的处理。默认情况下我们只需要编写蜘蛛就能完成基本的爬网任务，对于不同的爬取场景我们能通过修改模块的配置甚至增加某些定义的模块就能适应具体的爬取场景。

Scrapy这种强大的可自由组合与自由**装配**的能力得益于它高度组件化的设计理念。**每个模块只做一件事**，这是一种设计原则称为“单一职责原则“，模块之间相对独立，它们能被组合在一起协同工作依赖于Scrapy引擎在执行爬取时根据`settings.py`配置的内容对各个模块的的运行参数进行微调、将自定义的模块装配到Scrapy的执行环境中。

Scrapy配置文件`settings.py`有很多的配置项，如果一股脑地去理解其中的内容并不容易，按照本专栏的思路我们可以从最常用最简单的入手，从本章示例开始我们将会频繁地与这个`settings.py`文件打交道。

当我们使用`scrapy startproject`指令创建爬虫项目后 Scrapy 就会创建一个默认的`settings.py`文件，这个文件内有很多的注释项，我们可以暂时保留并忽略它们。在我们后面的内容中对最要的配置项目都会一一提及（详细的`settings.py`的配置可以参见官方说明）。打开`settings.py`文件，具体内容如下所示:

```
BOT_NAME = 'netease'

SPIDER_MODULES = ['netease_crawler.spiders']
NEWSPIDER_MODULE = 'netease_crawler.spiders'

# Obey robots.txt rules
ROBOTSTXT_OBEY = True
```

上面例出的4个配置是最重要的配置项，`BOT_NAME`是声明当前爬虫项目默认的蜘蛛名称，这个名字必须与蜘蛛的`name`属性相同。

```
class NeteaseSpider(CrawlSpider):
    name = 'netease'
```

`SPIDER_MODULES`是声明所有蜘蛛模块的命名空间，`NEWSPIDER_MODULE`是指向`BOT_NAME`所在的蜘蛛的命名空间，如果在创建项目后没有更改文件名或文件夹的情况下，这两个设置是不需要变动的。

`robot.txt`

`ROBOTSTXT_OBEY` 是声明是否在爬取之前先获取目标网站上的`robot.txt`文件，然后让蜘蛛根据`robot.txt`文件中制定的规则进行爬取。

robot.txt 是一个纯文本文件，是网站用来专门告诉爬虫，网站内哪些内容可以爬哪些内容不希望爬虫进入的一个文件。这个文件是网站对爬虫的一种**礼遇**，如果爬虫按照里面的规定**有礼貌**地爬取内容，一般上是不会触发网站的反爬机制。这只是一份**君子协议**，爬虫是可以不遵守这份协定而突破网站的任意区域获取数据的，但会触发什么后果也只能是"看着办"。

robot.txt 文件的内容非常简单，只有两个配置:

- `User-agent` - 声明只允许哪些浏览器进入，如果被设置为`*`则是说明任意的浏览器都可以访问(在后面的反爬技术章节中会有专门讲述这个User Agent的内容)
- `Disallow` - 声明哪些URL下的内容是不允许爬虫进入的。(可以有多行的声明)

robot.txt 是一个由来已久的协定，最初是为了给搜索引擎的蜘蛛准备的，用于屏蔽掉一些对于搜索引擎蜘蛛无用的文件，避免索引质量下降而采用的协定。我们之所以要知道它，是因为我们的爬虫也可以**装**作是一种搜索引擎发出的蜘蛛，从最小的程度上**麻痹**对方的反爬机制。

以上是一种背景知识，因为Scrapy会默认让蜘蛛遵守`robot.txt`的协议内容的，当`ROBOTSTXT_OBEY`被设置为`True`时，如果对方网站的根目录下根本没有放置 robot.txt文件，会导致爬虫执行失败的，所以推荐这个选项设置为`False`：

```
ROBOTSTXT_OBEY=False
```

为了让爬虫的执行速度能更快，建议将以下的选项打开（将配置项的注释去掉就好）：

```
HTTPCACHE_ENABLED = True
TELNETCONSOLE_ENABLED = False
```

`HTTPCACHE_ENABLED`会将爬取过的内容自动存到本地，下次重复爬取相同URL时就直接从本地的文件中读取而不再向网站发出请求，这个功能在开发期需要反复调试爬虫的情况是非常有用的。

`TELNETCONSOLE_ENABLED` 这个选项是被默认打开的，它的作用就是打开一个 Telnet 服务让我们可以通过终端用 Telnet 来与 Scrapy 的爬虫实例进行观察，这是一种非常古老的调试手段了，而且它的启动很影响性能，所以建议将这个选项直接关闭。如果对Telnet调试有兴趣的读者可以参考官方关于Telnet的控制台的内容

保存 Item 到 JSON 的方法

如果执行爬虫的命令行参数过多，实在是一种非常不友好的使用方式：

```
$ scrapy crawl netease -o netease.json -s FEED_EXPORT_ENCODING='utf-8'
```

我们同样可以通过配置文件`settings.py`来简化执行指令，具体配置方法如下所示：

```
FEED_FORMAT = 'json'
FEED_URI = 'result.json'
FEED_EXPORT_ENCODING='utf-8'
```

用`FEED_FORMAT`与`FEED_URI`取代输出参数`-o`，直接指定输出格式与输出文件位置。(Windows环境下需指定`FEED_URI`的绝对路径，如`file:///D:/Spider/netease_crawler/result.json`)。`FEED_EXPORT_ENCODING`指定输出文件的编码方式以解决中文乱码的问题。

至此，我们可以得到一份完整实用的`settings.py`配置文件：

```
BOT_NAME = 'netease'

SPIDER_MODULES = ['netease_crawler.spiders']
NEWSPIDER_MODULE = 'netease_crawler.spiders'

ROBOTSTXT_OBEY=False
HTTPCACHE_ENABLED = True
TELNETCONSOLE_ENABLED = False

FEED_FORMAT = 'json'
FEED_URI = 'result.json'
FEED_EXPORT_ENCODING='utf-8'
```

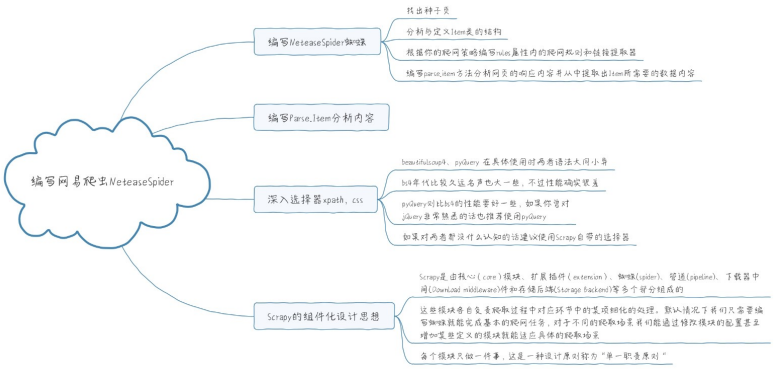
完成配置工作后只要按如下方式启动执行爬网就可以了

```
$ scrapy crawl netease
```

小结

虽然本章用了五个小节来讲述一个"网易泛爬蜘蛛"开发过程，实际上代码量只是非常少量。只要你运行起这个爬虫就会发现它的性能惊人，在一个小时内`result.json`的大小已经超过几百兆，请注意这可是纯文本的内容！

一但我们站在蜘蛛的视觉角度建立起一张可爬行的蜘网的概念，用正则表达式来提取出这个网的规则，完全掌握页面分析与数据提取的方法，无论面对规模多么庞大的网站，蜘蛛都可以畅通无阻地在其中穿行。



> 注：你可以去GitHub获取本章源代码

10 Python中的正则表达式用法

12 用 ItemLoader 解决网页数据多样性的问题