

# 网络数据采集

## 第七章 构建健壮的爬虫系统

- 讲师姓名：
- 授课时间：
- 共32课时，第23-28课时

### 1 上节回顾

上节课介绍了Selenium爬取网页的基本方法，同时介绍了几个案例。

### 2 本节课程主要内容

内容列表：

- 本节目标
- Scrapy爬虫框架基础
- scrapy具体操作与应用
- 本节总结
- 课后练习

#### 2.1 本节目标

- 掌握Scrapy框架的构成要素
- 掌握Scrapy框架的工作原理
- 能够应用Scrapy设计网络数据采集系统

#### 2.2 Scrapy爬虫框架基础

这一讲，我们介绍网络爬虫框架Scrapy的基本组成、工作原理，并介绍使用Scrapy构建较为健壮的网络爬虫应用程序的方法。



An open source and collaborative framework  
for extracting the data you need from websites.  
In a fast, simple, yet extensible way.

pypi v1.8.0 wheel yes coverage 84%

## 2.2.1 主要功能

Scrapy是一个爬虫框架，主要用于爬取网页、提取数据。可用于：

- 数据挖掘
- 网络监控
- 自动测试等



### Fast and powerful

write the rules to extract the data and let  
Scrapy do the rest



### Easily extensible

extensible by design, plug new functionality  
easily without having to touch the core



### Portable, Python

written in Python and runs on Linux,  
Windows, Mac and BSD

## 2.2.2 Scrapy at a glance

为了从专业角度让大家了解Scrapy的基本应用方式，我们先看一个Scrapy的简单例子。

下面的例子以爬取示例网站 <http://quotes.toscrape.com> 中信息为目标，展现了使用Scrapy构建爬虫的基本方法。

爬虫设计

```

import scrapy

class QuotesSpider(scrapy.Spider):
    name = 'quotes'
    start_urls = [
        'http://quotes.toscrape.com/tag/humor/',
    ]

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').get(),
                'author': quote.xpath('span/small/text()').get(),
            }

        next_page = response.css('li.next a::attr("href")').get()
        if next_page is not None:
            yield response.follow(next_page, self.parse)

```

## 运行爬虫

```
scrapy runspider quotes_spider.py -o quotes.json
```

## 得到结果

```

[{"author": "Jane Austen",
  "text": "\u201cThe person, be it gentleman or lady, who has not pleasure in a good novel, must be intolerably"},
 {"author": "Groucho Marx",
  "text": "\u201cOutside of a dog, a book is man's best friend. Inside of a dog it's too dark to read.\u201d"},
 {"author": "Steve Martin",
  "text": "\u201cA day without sunshine is like, you know, night.\u201d"},
 ...]

```

## 2.2.3 Scrapy特性

从上面的例子中，我们可以看到使用Scrapy的优势：

- 网页请求经过调度和异步处理
  - 意味着scrapy不需要等待某个请求结束或处理完成，就同时可以发送另一个请求或做其他的工作。
- 可以构建多快速的多并发爬虫系统，并且有效控制爬虫的礼貌性。

- 可以在每个请求中，设置下载延迟，限制对每个IP的并发请求数量。
- 可以使用正则表达式、CSS、XPATH从htmlz中获取数据。
- 可以使用交互式shell控制台实现交互式开发过程。
- 支持多种输出格式（JSON、CSV、XML）
- 支持将输出结果导出到多个后端系统（文件、数据库）
- 强大的编码能力和自动检测能力，可以用于检查外来、非标的编码。
- 强大的扩展性支持，可以使用信号或良好的API，实现自定义功能。
- 丰富的内置中间件：
  - cookie
  - 会话处理
  - HTTP压缩、身份验证、缓存
  - Proxy
  - robots.txt
  - 爬行深度限制
- 支持远程控制台
- 大量可重用代码。

## 2.2.4 Scrapy基本组成和工作原理。

Scrapy的主要模块有：

### Scrapy Engine(引擎)

负责Spider、ItemPipeline、Downloader、Scheduler中间的通讯，信号、数据传递等。

### Scheduler(调度器)

它负责接受引擎发送过来的Request请求，并按照一定的方式进行整理排列，入队，当引擎需要时，交还给引擎。

### Downloader（下载器）

负责下载Scrapy Engine(引擎)发送的所有Requests请求，并将其获取到的Responses交还给Scrapy Engine(引擎)，由引擎交给Spider来处理。

### Spider（爬虫）

它负责处理所有Responses,从中分析提取数据，获取Item字段需要的数据，并将需要跟进的URL提交给引擎，再次进入Scheduler(调度器)。

### Item Pipeline(管道)

它负责处理Spider中获取到的Item，并进行进行后期处理（详细分析、过滤、存储等）的地方。以流水线方式对Spider解析后得到的结果（Item），用户可以定义一组操作顺序，包括：清理、检验、查重、存储到数据库等。

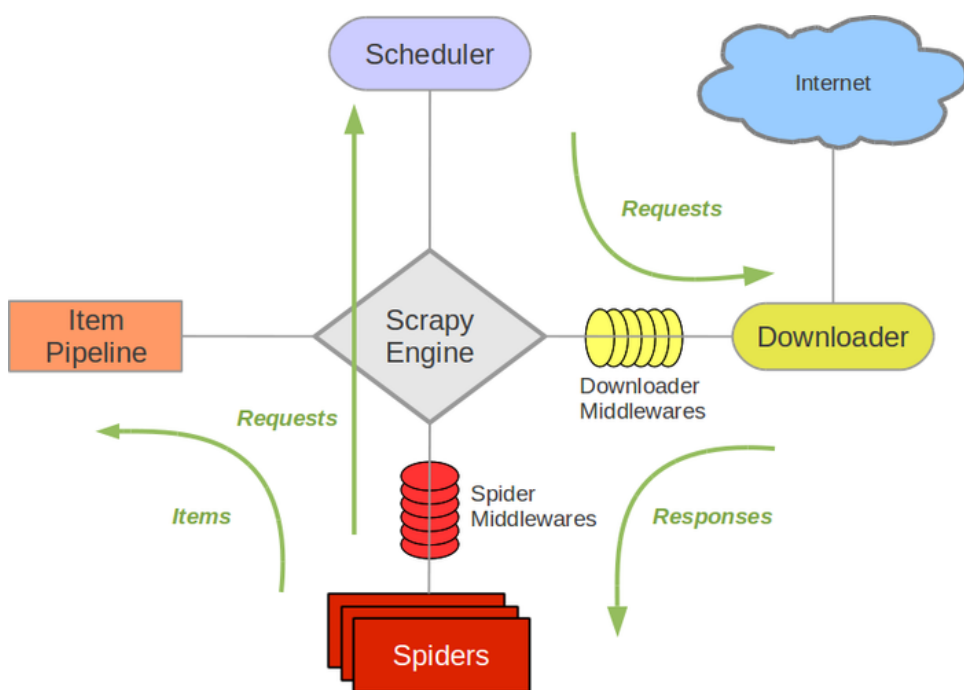
## Downloader Middlewares（下载中间件）

你可以当作是一个可以自定义扩展下载功能的组件。

## Spider Middlewares（Spider中间件）

你可以理解为是一个可以自定扩展和操作引擎和Spider中间通信的功能组件（比如进入Spider的Responses;和从Spider出去的Requests），

下图是Scrapy的基本结构框图：



借助这一幅图，我们再介绍一下scrapy的工作原理：

Scrapy有3条数据流路径：

### 请求发起与调度

- Spiders经Spider中间件向Engine发送Requests;
- Engine将爬取请求交给Scheduler，完成调度。

### 请求执行与响应

- Scheduler将调度后Requests交给Engine;
- Engine将请求转发给Downloader;
- Downloader执行下载并将响应Response对象提交给Engine;
- Engine将Response对象送给Spiders，用于提取Item和跟踪Links。

## 结果输出与新请求的产生

- Spiders处理Response对象，解析出结构化的爬得数据Item和新Links，并送给Engine；
- Engine将Items送给Item pipelines完成输出处理；
- Engine将新Requests送给Scheduler，产生新请求。

通过上面流程的分析，可知入口是Spiders、出口是Item pipelines、核心是Engine、策略由Scheduler制定。

在Scrapy中需要用户编写的模块类及方法有：

- Spiders
- Item pipelines

而Scrapy已经实现的部分（不需用户编写）是：

- Engine
- Scheduler
- Downloader

为了增加用户的灵活性，Scrapy在Engine与Spider之间、Engine与Downloader之间增加了中间件：Downloader Middleware，用户可以定义这些中间件来修改、丢弃和新增请求或响应。Spider Middleware，对请求和爬取项的再处理，可以修改、丢弃、增加请求和爬取项。

## 2.2.5 模块练习与答案

见习题集

## 2.2.6 内容小结

以上就是对scrapy基本组成和工作原理的介绍，希望大家能够细心体会，掌握这些内容将有利于我们后续基于scrapy设计网络爬虫程序。

## 2.3 scrapy具体操作与应用

下面通过具体操作介绍Scrapy基础应用方法。主要内容有：

- 准备工作
- 新建scrapy 项目
- 建立scrapy 爬虫
- 运行scrapy 爬虫
- 使用scrapy shell进行分析

下面我们将使用Scrapy爬取[quotes.toscrape.com](http://quotes.toscrape.com)，中的这是一个收集了一些名人名言的网站。

### 2.3.1 准备工作

为了在Anaconda3中借助scrapy框架编写爬虫程序，我们需要建立一个conda虚拟环境并安装scrapy框架：

1. 打开Anaconda prompt 命令行
2. 进入自己安装anaconda3的目录下的envs目录，查看当前已有的虚拟环境目录名
3. 运行下列命令，建立名为scrapyws的conda虚拟环境,并在其中安装scrapy

```
conda create -n scrapyws scrapy
```

4. 运行下列命令，激活该虚拟环境：

```
(base) D:\pythonspace\anaconda3\envs>conda activate scrapyws
```

## 运行后，会发现下面的路径前括号内的内容变化为scrapyws，正是我们建立的。

<p class="mume-header " id="运行后会发现下面的路径前括号内的内容变化为scrapyws正是我们建立的"></p>

```
(scrapyws) D:\pythonspace\anaconda3\envs>
```

5. 运行下列命令进入scrapyws目录

```
(scrapyws) D:\pythonspace\anaconda3\envs>cd scrapyws
```

```
(scrapyws) D:\pythonspace\anaconda3\envs\scrapyws>dir`
```

## 你会发现这个目录下含有很多已安装的文件

<p class="mume-header " id="你会发现这个目录下含有很多已安装的文件"></p>

注：如果不想使用虚拟环境，可以在Anaconda3命令行中直接运行 `conda install -c conda-forge scrapy` 或 `pip install scrapy`。

Scrapy是一个纯python项目，它依赖的其他python packages:

- lxml, an efficient XML and HTML parser
- parsel, an HTML/XML data extraction library written on top of lxml,
- w3lib, a multi-purpose helper for dealing with URLs and web page encodings
- twisted, an asynchronous networking framework
- cryptography and pyOpenSSL, to deal with various network-level security needs

### 2.3.2 新建 Scrapy 项目

在命令行执行下面语句就可以新建scrapy项目，在这里我们假定项目名为tutorial

```
scrapy startproject tutorial
```

如果你使用了上面推荐的conda虚拟环境安装scrapy，那么上面语句执行成功后，会在Anaconda的envs/myscrapy/中出现一个tutorial目录。

```
tutorial/
  scrapy.cfg          # 部署配置文件
  tutorial/           # 项目的python模块
    __init__.py
    items.py          # 项目输出结果的定义文件
    middlewares.py     # 项目中间件文件
    pipelines.py       # 项目管道文件
    settings.py        # 项目配置文件
    spiders/           # 爬虫文件目录
      __init__.py
```

各文件的基本功能是：

- scrapy.cfg：项目的配置文件
- mySpider/：项目的Python模块，将会从这里引用代码
- mySpider/items.py：项目的目标文件
- mySpider/pipelines.py：项目的管道文件
- mySpider/settings.py：项目的设置文件
- mySpider/spiders/：存储爬虫代码目录

我们将爬取quotes.toscrape.com中的名人名言，希望得到以下信息（JSON格式）：

```
{
  'author': 'Douglas Adams',
  'text': '"I may not have gone where I intended to go, but I think I ..."',
  'tags': ['life', 'navigation']
}
```

### 2.3.3 编写爬虫文件

生成具体爬虫程序

首先，在当前目录下输入命令，将tutorial/spider目录下创建一个名为quotesspider的爬虫，并制定爬取域的范围：

```
tutorial/spider/> scrapy genspider quotesspider "quotes.toscrape.com"
```



自定义的Spider类用于从网站上爬取信息，它需要继承scrapy.Spider，并生成初始请求。类的定义中还可以选择如何跟踪页内的链接，如何解析下载页面的内容。

## 编写爬虫程序

编写下面的爬虫类:

```
# -*- coding: utf-8 -*-
<p class="mume-header " id="-coding-utf-8-"></p>

import scrapy

class QuotesspiderSpider(scrapy.Spider):
    name = "quotesspider"
    allowed_domains = ["quotes.toscrape.com"]
    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()').extract_first(),
                'author': quote.xpath('./small[@class="author"]/text()').extract_first(),
                'tags': quote.xpath('./div[@class="tags"]/a[@class="tag"]/text()').extract()
            }

        next_page_url = response.xpath('//li[@class="next"]/a/@href').extract_first()
        if next_page_url is not None:
            yield scrapy.Request(response.urljoin(next_page_url))
```

## Python 中的 yield

上面代码中使用了yield函数，它使它所在的函数成为了一个python generator。

所谓生成器也是一种产生可迭代对象的函数，它每次产生一个值后，函数就被冻结不再执行，当被唤醒时（需要获取下一个值时）才再次产生一个值。

例如:

```
def gen(n):
    for i in range(n):
        yield i**2

for i in gen(100000000):
    print(i)
```

生成器的优势:

- 内存更加优化
- 速度更快
- 应用更灵活

## 2.3.4 运行Scrapy爬虫

为了使上面编写的爬虫程序工作，需要在命令行中进入项目顶级目录(否则会报错)并运行下面指令：

```
scrapy crawl quotesspider
```

这个命令将启动我们上面编写的名为 quotesspider 的爬虫，它将向quotes.toscrape.com网站发送请求，之后会得到类似下面的输出：

```
(scrapyws) D:\python\space\anaconda3\envs\scrapyws\tutorial\tutorial>scrapy crawl quotesspider
2019-07-30 10:36:58 [scrapy.utils.log] INFO: Scrapy 1.3.3 started (bot: tutorial)
2019-07-30 10:36:58 [scrapy.utils.log] INFO: Overridden settings: {'BOT_NAME': 'tutorial', 'NEWSPIDER_MODULE': 't
2019-07-30 10:36:58 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.corestats.CoreStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.logstats.LogStats']
2019-07-30 10:36:58 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2019-07-30 10:36:58 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']

.....

2019-07-30 10:37:00 [scrapy.core.scrapers] DEBUG: Scraped from <200 http://quotes.toscrape.com/>
{'text': '"The world as we have created it is a process of our thinking. It cannot be changed without changing ou
2019-07-30 10:37:00 [scrapy.core.scrapers] DEBUG: Scraped from <200 http://quotes.toscrape.com/>
{'text': '"It is our choices, Harry, that show what we truly are, far more than our abilities."', 'author': 'J.K.
.....
```

## 2.3.5 存储爬得数据

若要存储爬得数据，简单的方法是通过下面的命令将内容保存为json文件：

```
scrapy crawl quotesspider -o quotes.json
```

上面的命令会生成一个包含所有爬得数据条目（以JSON形式序列化）的quotes.json文件。也可以使用别的存储格式，例如：

```
scrapy crawl quotes -o quotes.jl
```

由于JSON Lines格式是流状文件，所以你可以对其追加新的记录。JSON格式文件是不能追加内容的，如果不小心执行两次之前的命令，会得到一个破损的JSON文件。JSON Lines文件中的每条记录被分割为单独的一行，你可以使用它来在内存中处理大文件，而不需要应对大文件加载的一些麻烦。

如果要对爬得的数据条目执行更为复杂的操作，可以写一个Item Pipeline类。一个服务于Item Pipelines的占位符文件在新建项目时就被生成了，就存放在tutorial/pipelines.py文件中。这一点以后介绍。

## 2.3.6 探查 Scrapy 的命令

可以在能够运行Scrapy命令的目录下，运行 `scrapy -h`，能够发下如下的帮助信息：

```
Scrapy 1.6.0 - project: tutorial
```

```
Usage:
```

```
scrapy <command> [options] [args]
```

```
Available commands:
```

|              |  |
|--------------|--|
| bench        | Run quick benchmark test                                 |
| check        | Check spider contracts                                   |
| crawl        | Run a spider   |
| edit         | Edit spider  |
| fetch        | Fetch a URL using the Scrapy downloader                  |
| genspider    | Generate new spider using pre-defined templates          |
| list         | List available spiders                                   |
| parse        | Parse URL (using its spider) and print the results       |
| runspider    | Run a self-contained spider (without creating a project) |
| settings     | Get settings values                                      |
| shell        | Interactive scraping console                             |
| startproject | Create new project                                       |
| version      | Print Scrapy version                                     |
| view         | Open URL in browser, as seen by Scrapy                   |

```
Use "scrapy <command> -h" to see more info about a command
```

## 2.3.7 使用 scrapy shell 实现交互式爬取

上面的例子对**scrapy**的运作过程进行了简单介绍。可能不少同学会觉得这部分内容过于理想化。在没有对网站的分析和交互式反馈的情况下，我们不可能对**scrapy**有更好的应用。

下面我们尝试使用 **scrapy shell** 对<http://quotes.toscrape.com> 进行交互式的分析，然后根据分析过程和结果验证刚才的爬虫程序。

在使用**scrapy shell**之前，可以对这个**shell**进行设置，选择**bpython**或是**IPython**。在**scrapy.cfg**文件中键入如下内容：

```
[settings]
shell=ipython
```

在**scrapy**工作环境（虚拟环境、**scrapyws**目录都要准备好）下运行**Scrapy shell**的命令是：

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

注意：上面命令中要在**URL**外加引号，否则**scrapy**命令会认为是某个命令行参数。

运行命令后得到的命令行输出大致如下：

```

(scrapyws) D:\pythonspace\anaconda3\envs\scrappyws\tutorial>scrappy shell "http://quotes.toscrape.com"
2019-07-30 11:05:28 [scrappy.utils.log] INFO: Scrappy 1.3.3 started (bot: tutorial)
2019-07-30 11:05:28 [scrappy.utils.log] INFO: Overridden settings: {'BOT_NAME': 'tutorial', 'DUPEFILTER_CLASS': 's
2019-07-30 11:05:28 [scrappy.middleware] INFO: Enabled extensions:
['scrappy.extensions.corestats.CoreStats',
 'scrappy.extensions.telnet.TelnetConsole']
2019-07-30 11:05:28 [scrappy.middleware] INFO: Enabled downloader middlewares:
['scrappy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrappy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrappy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrappy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrappy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrappy.downloadermiddlewares.retry.RetryMiddleware',
 'scrappy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrappy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrappy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrappy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrappy.downloadermiddlewares.stats.DownloaderStats']
2019-07-30 11:05:28 [scrappy.middleware] INFO: Enabled spider middlewares:
['scrappy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrappy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrappy.spidermiddlewares.referer.RefererMiddleware',
 'scrappy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrappy.spidermiddlewares.depth.DepthMiddleware']
2019-07-30 11:05:28 [scrappy.middleware] INFO: Enabled item pipelines:
[]
2019-07-30 11:05:28 [scrappy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023
2019-07-30 11:05:28 [scrappy.core.engine] INFO: Spider opened
2019-07-30 11:05:30 [scrappy.core.engine] DEBUG: Crawled (404) <GET http://quotes.toscrape.com/robots.txt> (referer: None)
2019-07-30 11:05:31 [scrappy.core.engine] DEBUG: Crawled (200) <GET http://quotes.toscrape.com> (referer: None)
[s] Available Scrappy objects:
[s]   scrapy      scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler     <scrappy.crawler.Crawler object at 0x0000029AB98B2E80>
[s]   item        {}
[s]   request     <GET http://quotes.toscrape.com>
[s]   response    <200 http://quotes.toscrape.com>
[s]   settings    <scrappy.settings.Settings object at 0x0000029ABA9B0B38>
[s]   spider      <QuotesspiderSpider 'quotesspider' at 0x29abac35f28>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects (by default, redirects are followed)
[s]   fetch(req)                  Fetch a scrapy.Request and update local objects
[s]   shelp()                     Shell help (print this help)
[s]   view(response)             View response in a browser
>>>

```

可以看到，上面的响应输出中含有：

```
> 2019-07-30 11:05:31 [scrappy.core.engine] DEBUG: Crawled (200) <GET http://quotes.toscrape.com> (referer: None)
```

其中的200表示响应成功，我们可以使用下列命令快速查看scrapy下载的临时文件：

```
>>>view(response)
```

上述命令会打开系统默认浏览器，并加载scrapy下载到本地的临时页面。

之后，尝试运行：

```
>>> response.xpath('//title')
```

结果如下：

```
[<Selector xpath='//title' data='<title>Quotes to Scrape</title>'>]  
>>>
```

上面的命令会得到一个名为SelectorList的列表型对象。这个列表中的Selector对象会包装XML/HTML元素，以便于进行更多的查询来精确选择和提取数据。例如要提取上面例子中的title标签的文本，可以如下操作：

```
>>> response.xpath('//title').extract_first()
```

结果如下：

```
'<title>Quotes to Scrape</title>'  
>>>
```

如果上面不写“::text”或“text()”，会得到整个title标签内容。上例中调用.extract()的结果是一个列表，因为我们处理的是一个SelectorList实例。如果你只想获得列表中的第一个内容，可以使用下列方法：

```
>>> response.xpath('//title/text()').extract_first()
```

结果如下：

```
'Quotes to Scrape'  
>>>
```

另一种获得类似输出的方法是：

```
response.css('title::text')[0].extract()
```

或

```
response.xpath('//title/text()')[0].extract()
```

但是，使用`extract_first()`方法可以在`SelectorList`为空时返回`None`，避免`IndexError`错误。

除了`extract`方法和`extract_first()`方法，还有`re()`方法支持使用正则表达式来选择文本并提取。

例如：

```
response.xpath('//title/text()').re(r'Quotes')
```

结果：['Quotes']

```
response.xpath('//title/text()').re(r'Q\w+')
```

结果：['Quotes']

```
response.xpath('//title/text()').re(r'(\w+) to (\w+)')
```

结果：['Quotes', 'Scrape']

为了能够找到正确的CSS选择器，你可以使用`view(response)`命令查看响应内容；或者使用Chrome或Firefox浏览器的开发者工具分析原页面的CSS样式或XPath表达式。XPath表达式非常强大，而且是Scrapy选择器`Selectors`的基础。事实上，CSS selectors会被转换为XPath。

## 提取名人名言和作者

现在，我们了解了一些选择器`selector`和提取方法，下面尝试着完成名人名言爬虫的完整实现。

<http://quotes.toscrape.com> 中的每个网页均有HTML元素组成，例如：

```
<div class="quote">
  <span class="text">“The world as we have created it is a process of our
  thinking. It cannot be changed without changing our thinking.”</span>
  <span>
    by <small class="author">Albert Einstein</small>
    <a href="/author/Albert-Einstein">(about)</a>
  </span>
  <div class="tags">
    Tags:
    <a class="tag" href="/tag/change/page/1/">change</a>
    <a class="tag" href="/tag/deep-thoughts/page/1/">deep-thoughts</a>
    <a class="tag" href="/tag/thinking/page/1/">thinking</a>
    <a class="tag" href="/tag/world/page/1/">world</a>
  </div>
</div>
```

在设计爬虫解析函数前，我们首先要打开`scrapy shell`来找出如何提取我们想要的数据。可以运行如下命令：

```
scrapy shell "http://quotes.toscrape.org"
```

由于我们希望获取的信息包含在上面的

块中，所以我们尝试下列操作。运行下面命令将会得到一个名言HTML元素的selectors列表：

```
qs = response.css("div.quote")
或
qs = response.xpath('//div[@class="quote"]')
len(qs)
```

### 2.4 输出为10，表示爬取到了10条信息（div）

现在我们可以利用qs列表中第一个对象提取出子标签title、author、tags等内容：

```
>>> qs[0].xpath("./span/text()")
[<Selector xpath='./span/text()' data='“The world as we have created it is a pr">
>>>

>>> qs[0].xpath('./small[@class="author"]/text()').extract_first()
'Albert Einstein'
>>>

>>> qs[0].xpath('./div[@class="tags"]/a[@class="tag"]/text()').extract()
['change', 'deep-thoughts', 'thinking', 'world']
>>>
```

说明：

- 上面的xpath路径前有“.”表示相对当前selector内容进行查找；
- qs[0]表示qs列表中的第一个元素
- extract\_first()方法用于获取第一个selector对象的可打印内容；
- extract()方法获得的是所有可打印内容列表。

## 跟踪链接

如果需要爬取整个网站的内容，就需要对页面中的链接进行获取和跟踪。如何使用Scrapy提取网页上的链接呢？

检查爬得的页面内容，我们可以看到有下一页的链接，例如：

```
<ul class="pager">
  <li class="next">
    <a href="/page/2/">Next <span aria-hidden="true">&rarr;</span></a>
  </li>
</ul>
```



为了保证程序的正确性，首先要在scrapy shell下进行测试：

```
>>> link = response.xpath('//li[@class="next"]/a/@href').extract_first()
>>> link
'/page/2/'
>>>
```

之后，就可以像上面的quotesspider中一样，写出递归爬取新URLs的方法。

## Scrapy示例：进一步地爬取作者信息

下面的例子演示了多个回调函数和跟踪链接的情况：

```
import scrapy

class AuthorSpider(scrapy.Spider):
    name = 'author'

    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        # follow links to author pages
        for href in response.css('.author + a::attr(href)'):
            yield response.follow(href, self.parse_author)

        # follow pagination links
        for href in response.css('li.next a::attr(href)'):
            yield response.follow(href, self.parse)

    def parse_author(self, response):
        def extract_with_css(query):
            return response.css(query).extract_first().strip()

        yield {
            'name': extract_with_css('h3.author-title::text'),
            'birthdate': extract_with_css('.author-born-date::text'),
            'bio': extract_with_css('.author-description::text'),
        }
```

上面例子中的spider将调用parse\_author()方法，从网站的主page开始爬取所有的指向作者页面的链接。在调用回调函数时，我们使用了：

```
response.follow
```

这个方法可以使我们的代码更为精炼。

```

import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"

    def start_requests(self):
        url = 'http://quotes.toscrape.com/'
        tag = getattr(self, 'tag', None)
        if tag is not None:
            url = url + 'tag/' + tag
        yield scrapy.Request(url, self.parse)

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').extract_first(),
                'author': quote.css('small.author::text').extract_first(),
            }

        next_page = response.css('li.next a::attr(href)').extract_first()
        if next_page is not None:
            yield response.follow(next_page, self.parse)

```

## 2.3.8 scrapy spider

Spider类是一组定义了如何获取某个网站的数据的类。包括：

- 执行爬取过程
- 跟踪链接
- 提取数据

Spiders的爬取过程如下：

- 使用第一个URL生成初始请求（默认为start\_urls里的URL，调用start\_requests()方法）；
- 指定要调用的回调函数（默认为parse）；
- 在回调函数中，定义处理数据的过程（例如使用Selectors）；
- 返回items到一个数据库（在item pipeline中定义）或一个文件（使用Feed exports）。

下面是使用一个回调函数返回多个请求和items的示例：

```

import scrapy

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = [
        'http://www.example.com/1.html',
        'http://www.example.com/2.html',
        'http://www.example.com/3.html',
    ]

    def parse(self, response):
        for h3 in response.xpath('//h3').getall():
            yield {"title": h3}

        for href in response.xpath('//a/@href').getall():
            yield scrapy.Request(response.urljoin(href), self.parse)

```

如果不使用`start_urls`，可以使用`start_requests()`，例如：

```

import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']

    def start_requests(self):
        yield scrapy.Request('http://www.example.com/1.html', self.parse)
        yield scrapy.Request('http://www.example.com/2.html', self.parse)
        yield scrapy.Request('http://www.example.com/3.html', self.parse)

    def parse(self, response):
        for h3 in response.xpath('//h3').getall():
            yield MyItem(title=h3)

        for href in response.xpath('//a/@href').getall():
            yield scrapy.Request(response.urljoin(href), self.parse)

```

## 通用爬虫模板

Scrapy提供了几个爬虫模板：

- basic
- crawl
- xml
- sitemap

## basic spider

示例：

```
# -*- coding: utf-8 -*-
import scrapy

class Quotes1Spider(scrapy.Spider):
    name = 'quotes1'
    allowed_domains = ['quotes.toscrape.com']
    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()').extract_first(),
                'author': quote.xpath('./span/small/text()').extract_first(),
                'tags': quote.xpath('./div[@class="tags"]/a[@class="tag"]/text()').extract()
            }

        nextpage = response.xpath('//li[@class="next"]/a/@href').extract_first()
        if nextpage:
            yield scrapy.Request(response.urljoin(nextpage))
```

## crawl spider

这个模板是爬取常规网站的最常用的，因为它通过定义一组规则为跟踪链接提供了便捷机制。

除了从Spider继承的属性之外，此类还支持一个新属性：**rules**

**scrapy.spiders.Rule** (link\_extractor = None, callback = None, cb\_kwargs = None, follow = None, process\_links = None, process\_request = None )

说明：

- **rules**是一个（或多个）**Rule**对象的列表。
- 每个都**Rule** 定义了用于爬网的特定行为。
- 如果多个规则匹配同一链接，则将根据在此属性中定义的顺序使用第一个规则。
- **link\_extractor**是一个链接提取器对象，它定义如何从每个已爬网页面提取链接。
- 每个产生的链接将用于生成一个**Request**对象，该对象将在其**meta**字典中包含链接的文本。
- 如果省略，将使用不带参数创建的默认链接提取器，从而提取所有链接。
- **callback**是为使用指定链接提取器提取的每个链接调用的可调用或字符串

这个蜘蛛还公开了一个可重写的方法：**parse\_start\_url**(回调函数)

- 该方法用于start\_urls响应；
- 它允许解析初始响应，并且必须返回一个Item对象；
- 一个Request对象或包含其中任何一个的可迭代对象。

现在，让我们看一个带有规则的示例CrawlSpider:

```
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor

class MySpider(CrawlSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com']

    rules = (
        # Extract links matching 'category.php' (but not matching 'subsection.php')
        # and follow links from them (since no callback means follow=True by default).
        Rule(LinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

        # Extract links matching 'item.php' and parse them with the spider's method parse_item
        Rule(LinkExtractor(allow=('item\.php', )), callback='parse_item'),
    )

    def parse_item(self, response):
        self.logger.info('Hi, this is an item page! %s', response.url)
        item = scrapy.Item()
        item['id'] = response.xpath('//td[@id="item_id"]/text()').re(r'ID: (\d+)')
        item['name'] = response.xpath('//td[@id="item_name"]/text()').get()
        item['description'] = response.xpath('//td[@id="item_description"]/text()').get()
        item['link_text'] = response.meta['link_text']
        return item
```

## XMLFeedSpider

XMLFeedSpider用于通过迭代一定的节点名称来解析XML源。

迭代器可以选择:

- iternodes (推荐, 性能较优)
- xml
- html

示例:

```

from scrapy.spiders import XMLFeedSpider
from myproject.items import TestItem

class MySpider(XMLFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.xml']
    iterator = 'iternodes' # This is actually unnecessary, since it's the default value
    itertag = 'item' # 用于迭代时使用的节点名称

    def parse_node(self, response, node):
        self.logger.info('Hi, this is a <{s}> node!: {s}', self.itertag, ''.join(node.getall()))

        item = TestItem()
        item['id'] = node.xpath('@id').get()
        item['name'] = node.xpath('name').get()
        item['description'] = node.xpath('description').get()
        return item

```

## CSVFeedSpider

这个spider模板通XMLFeedSpider类似，但它的迭代器是按行取数据的，而不是按节点。而且回调函数是parse\_row()。

示例：

```

from scrapy.spiders import CSVFeedSpider
from myproject.items import TestItem

class MySpider(CSVFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.csv']
    delimiter = ';'
    quotechar = '"'
    headers = ['id', 'name', 'description']

    def parse_row(self, response, row):
        self.logger.info('Hi, this is a row!: {r}', row)

        item = TestItem()
        item['id'] = row['id']
        item['name'] = row['name']
        item['description'] = row['description']
        return item

```

## SitemapSpider

这个模板可以使用Sitemaps爬取某个网站上的所有URLs。Sitemaps (站点地图)是网站站长通知搜索引擎有关其站点上可进行爬网的页面的一种简便方法。Sitemaps通常是一个XML文件，其中列出了站点的URL以及有关每个URL的其他元数据，以便搜索引擎可以更智能地抓取该网站。

它支持嵌套的sitemaps，以及从robots.txt中发现sitemap。

sitemap示例：

```
<?xml version="1.0" encoding="UTF-8"?>

<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">

  <url>

    <loc>http://www.example.com/</loc>

    <lastmod>2005-01-01</lastmod>

    <changefreq>monthly</changefreq>

    <priority>0.8</priority>

  </url>

</urlset>
```

SitemapSpider示例：

```
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]

    other_urls = ['http://www.example.com/about']

    def start_requests(self):
        requests = list(super(MySpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests

    def parse_shop(self, response):
        pass # ... scrape shop here ...

    def parse_other(self, response):
        pass # ... scrape other here ...
```

## 2.3.9 scrapy pipeline

在Scrapy 中运行爬虫，获取到一系列的item后，通常会将这些结果交给 pipeline中的方法去做后续处理。

每个 item pipeline 都是实现简单方法的Python类，它们接收item，并对其进行操作，还决定该item 是否应该继续通过管道或被删除。

Pipeline的常见用途有：

- 清洗数据
- 验证数据
- 去除重复
- 存储数据

Scrapy中的pipeline类可以定义多个，主要目的是：

- 不同的spider，会有不同的结果处理方式；
- 一个spider内部需要不同的操作时，可以对结果做不同处理，例如：存入不同的系统（文件、数据库）
- 在settings.py文件中，权重越小，优先级越高。

### 编写自己的pipeline

每个pipeline组件都是一个Python类，例如：

```
class TutorialPipeline(object):
    def process_item(self, item, spider):
        return item
```

Scrapy 要求pipeline类中必须实现一个process\_item(self,item,spider)方法。而且这个方法必须执行下列操作之一：

- 返回带有数据的字典；
- 返回一个Item对象；
- 返回Twisted Deferred或引发DropItem的异常

除了process\_item() 方法外， pipeline类还可以定义下列函数：

- open\_spider(self,spider)
  - 用于打开某个spider后调用该方法；
- close\_spider(self,spider)
  - 用于关闭某个spider后调用该方法；
- from\_spider(self,spider)



- 作为类方法，用于由某个crawler生成一个pipeline实例，返回一个新的pipeline实例。
- Crawler对象提供了访问所有Scrapy核心部件的能力，例如设置、信号等。
- 这是一种将pipeline访问scrapy核心，并将自己功能挂载到scrapy内核的一个方法。

## 示例1 价格验证

下面的例子展示了验证有价格信息的item，并调整其值，删除不包含价格信息的item。

```
class PricePipeline(object):

    vat_factor = 1.15

    def process_item(self, item, spider):
        if item.get('price'):
            if item.get('price_excludes_vat'):
                item['price'] = item['price'] * self.vat_factor
            return item
        else:
            raise DropItem("Missing price in %s" % item)
```

## 示例2 输出JSON格式数据

下面的例子展示了如果输出items到一个JSON文件。

```
import json

class JsonWriterPipeline(object):

    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
```

## 示例3 将结果写入MongoDB

下面的例子，展示了如何使用pipeline将结果写入mongodb。

这需要先要在settings.py中设置mongodb的地址和数据库名。

请注意下面的from\_crawler()，以及如何清理数据。

```

import pymongo

class MongoPipeline(object):

    collection_name = 'scrapy_items'

    def __init__(self, mongo_uri, mongo_db):
        self.mongo_uri = mongo_uri
        self.mongo_db = mongo_db

    @classmethod
    def from_crawler(cls, crawler):
        return cls(
            mongo_uri=crawler.settings.get('MONGO_URI'),
            mongo_db=crawler.settings.get('MONGO_DATABASE', 'items')
        )

    def open_spider(self, spider):
        self.client = pymongo.MongoClient(self.mongo_uri)
        self.db = self.client[self.mongo_db]

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

    def process_item(self, item, spider):
        self.db[self.collection_name].insert_one(dict(item))
        return item

```

#### 示例4 获取item截图

下面的例子中介绍了使用scrapy-splash爬取页面截图。

```

import scrapy
import hashlib
from urllib.parse import quote

class ScreenshotPipeline(object):
    """Pipeline that uses Splash to render screenshot of
    every Scrapy item."""

    SPLASH_URL = "http://localhost:8050/render.png?url={}"

    def process_item(self, item, spider):
        encoded_item_url = quote(item["url"])
        screenshot_url = self.SPLASH_URL.format(encoded_item_url)
        request = scrapy.Request(screenshot_url)
        dfd = spider.crawler.engine.download(request, spider)
        dfd.addBoth(self.return_item, item)
        return dfd

    def return_item(self, response, item):
        if response.status != 200:
            # Error happened, return item.
            return item

        # Save screenshot to file, filename will be hash of url.
        url = item["url"]
        url_hash = hashlib.md5(url.encode("utf8")).hexdigest()
        filename = "{}.png".format(url_hash)
        with open(filename, "wb") as f:
            f.write(response.body)

        # Store filename in item.
        item["screenshot_filename"] = filename
        return item

```

## 重复过滤器

下面的例子展示了一个用于查找重复项，并删除那些已经处理过的item的过滤器。

假设我们的商品具有唯一的ID，但我们的Spider会返回具有相同ID的多个商品：

```

from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

    def __init__(self):
        self.ids_seen = set()

    def process_item(self, item, spider):
        if item['id'] in self.ids_seen:
            raise DropItem("Duplicate item found: %s" % item)
        else:
            self.ids_seen.add(item['id'])
            return item

```

## 激活项目管道组件

要激活Item Pipeline组件，必须将其类添加到 ITEM\_PIPELINES设置中，如以下示例所示：

```

ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 300,
    'myproject.pipelines.JsonWriterPipeline': 800,
}

```

您在此设置中分配给类的整数值确定它们运行的顺序：项目从值较低的类转到值较高的类。通常将这些数字定义在0-1000范围内。

## 2.3.10 Scrapy项目中的设置

Scrapy中settings参数有四个级别（方面）：

- 命令行选项(Command line Options)(最高优先级)
- 自定义设置模块(Project settings module)
- 命令默认设定模块(Default settings per-command)
- 全局默认设定(Default global settings) (最低优先级)

一般情况下，配置主要在三个地方进行：命令行选项、项目、自定义。

### 命令行选项

所谓的命令行选项，就是我们平时使用命令行 `scrapy crawl somespider` 时，可在命令行后面添加的自定义参数。

例如： `scrapy crawl somespider -s LOG_FILE=test.log -s MACHINE_NAME=123 -s ITEM_PIPELINES={}`

说明：

- 启动spider，设置了LOG\_FILE为test.log
- 自定义了一个变量MACHINE\_NAME为123
- 将ITEM\_PIPELINES设置为空。

命令行选项是所有settings中优先级最高的。

## 自定义设置模块

custom\_settings可以理解为spider的个性设置。通常我们在一个项目目录下会有很多个spider，但是只有一个settings.py全局配置文件，为了让不同的spider应用不同的设置，我们可以在spider代码中加入custom\_settings设置。

例如：

spiders/somespider.py:

```
from ..custom_settings import *

class Spider1(CrawlSpider):
    name = "spider1"
    custom_settings = custom_settings_for_spider1
    pass

class Spider2(CrawlSpider):
    name = "spider2"
    custom_settings = custom_settings_for_spider2
```

此外，有一个 custom\_settings.py:

```

custom_settings_for_spider1 = {
    'LOG_FILE': 'spider1.log',
    'CONCURRENT_REQUESTS': 100,
    'DOWNLOADER_MIDDLEWARES': {
        'spider.middleware_for_spider1.Middleware': 667,
    },
    'ITEM_PIPELINES': {
        'spider.mysql_pipeline_for_spider1.Pipeline': 400,
    },
}

custom_settings_for_spider2 = {
    'LOG_FILE': 'spider2.log',
    'CONCURRENT_REQUESTS': 40,
    'DOWNLOADER_MIDDLEWARES': {
        'spider.middleware_for_spider2.Middleware': 667,
    },
    'ITEM_PIPELINES': {
        'spider.mysql_pipeline_for_spider2.Pipeline': 400,
    },
}

```

在spider里有spider1、spider2，引入了来自custom\_settings的配置变量custom\_settings\_for\_spider1、custom\_settings\_for\_spider2，通过这些变量，我们分别对两个爬虫的log文件、并发数、应用的中间件和管道文件进行了设置。

custom\_settings的优先级在命令行以下，比settings.py要高。

## settings.py

settings.py是爬虫项目的配置文件，通常我们在一个项目目录下会有很多个spider，但是只有一个settings.py全局配置文件，在这个settings.py我们可以设置项目下所有爬虫的一些公共变量。

例如：

- 用什么请求头
- 用哪种schedule
- 怎样去重
- 公共的数据库连接

settings.py的优先级是比较低的，次于命令行变量和custom\_settings变量。

下面给出例子：

```

# Crawl responsibly by identifying yourself (and your website) on the user-agent
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) ' \
              'AppleWebKit/537.36 (KHTML, like Gecko) ' \
              'Chrome/49.0.2623.87 Safari/537.36'

# Disable cookies (enabled by default)
COOKIES_ENABLED = True

# Override the default request headers:
DEFAULT_REQUEST_HEADERS = {
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'zh-CN,zh;q=0.8,en;q=0.6,zh-TW;q=0.4',
    'Accept-Encoding': 'gzip, deflate, sdch',
    'Connection': 'keep-alive'
}

# 广度优先
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeues.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeues.FifoMemoryQueue'

# 项目路径
PROJECT_DIR = os.path.dirname(os.path.abspath(os.path.curdir))

# mongodb配置
MONGO_URI = 'mongodb://localhost:27017'

#Mysql数据库的配置信息
MYSQL_HOST = '127.0.0.1'
MYSQL_DBNAME = 'testdb'           #数据库名字, 请修改
MYSQL_USER = 'root'               #数据库账号, 请修改
MYSQL_PASSWD = '123456'           #数据库密码, 请修改
MYSQL_PORT = 3306                 #数据库端口, 在dbhelper中使用

# 异步任务队列
BROKER_URL = 'amqp://guest:guest@localhost:5672//'

```

通过settings的设置，scrapy在crawler运行spider的时候，就会按照优先级把spider对应的设置用最高优先级的进行填充，这时候你可以在代码中从crawler.settings中获取到相应的设置。

## settings的使用技巧

对于不同类型settings的使用，有一些经验：

- 1.首先是settings.py文件，在一个scrapy项目中，一些通用的设置，比如请求头、代理入口、数据库连接等等，都可以统一写在settings.py中。

- 2.其次是custom\_settings，我一般会在scrapy项目中单独建立一个custom\_settings.py文件，依据不同spider所需要的设置，比如某站点A可能反爬严，我并发设置短点，站点B没反爬，我并发设置高点；又或者A用了中间件MA，B用了中间件MB等，我在custom\_settings.py文件中分别给予设置。
- 3.最后是命令行，都知道scrapy是协程，其不能完全利用好多核的计算机资源，所以我经常在一台机器上开多个scrapy进程，每个进程又对应相同的spider，对于这些进程如何进行管理，例如我要每个进程的日志分别查看，或者追踪每个进程的爬取速度等，这用命令行参数就比较合适。

### 2.3.11 Feed 输出

在编写爬虫程序时，有一个频繁出现的需求，那就是恰当地存储爬得数据。我们通常会将输出数据（称其为 export feed）存放到一个输出文件中。而这些数据将作为其他系统的输入，被后续步骤使用。

Scrapy提供了Feed Exports功能，使用多种序列化格式和存储后台来生成一个爬得数据的feed。

对于爬得数据，支持的输出格式，即序列化格式（Serialization formats）有：

- JSON
- JSON lines
- CSV
- XML
- Pickle
- Marshal
- 在FEED\_EXPORTERS设置项中设置所需的其他格式。

#### JSON格式

- FEED\_FORMAT: json
- Exporter: JsonItemExporter

#### JSON lines

- FEED\_FORMAT: jsonlines
- Exporter: JsonLinesItemExporter

#### CSV

- FEED\_FORMAT: csv
- Exporter: CsvItemExporter

指定要导出的列及其顺序使用 FEED\_EXPORT\_FIELDS。其他Feed导出程序也可以使用此选项，但它对CSV很重要，因为与许多其他导出格式不同，CSV使用固定标头。

#### XML



- FEED\_FORMAT: xml
- Exporter: XmlItemExporter

## Pickle

- FEED\_FORMAT: pickle
- Exporter: PickleItemExporter

## Marshal

- FEED\_FORMAT: marshal
- Exporter: MarshallItemExporter

## 存储

使用Feed导出时，可以使用URI（通过FEED\_URI设置）定义在哪里存储Feed。Feed导出支持由URI方案定义的多个存储后端类型。

支持开箱即用的存储后端包括：

- 本地文件系统
- FTP
- S3（需要 botocore或 boto）
- 标准输出

例如：

# 存储在FTP中使用每个蜘蛛一个目录：

```
ftp://user:password@ftp.example.com/scraping/feeds/%(name)s/%(time)s.json
```

# 存储在S3使用每个蜘蛛一个目录：

```
s3://mybucket/scraping/feeds/%(name)s/%(time)s.json
```

## Storage backends

### Local filesystem

即将数据源存储在本地文件系统中。

URI scheme: file

例如： file:///tmp/export.csv

使用这种方式不需要额外的库支持。

## FTP

即将数据源存放到FTP服务器。

URI scheme: ftp

例如: `ftp://user:pass@ftp.example.com/path/to/export.csv`

使用这种方式不需要额外的库支持。

## S3

即将数据源存放到Amazon S3服务器。

URI scheme: s3

例如:

```
s3://mybucket/path/to/export.csv
s3://aws_key:aws_secret@mybucket/path/to/export.csv
```

要求安装botocore (Python 2 and Python 3) or boto (Python 2 only)库。

此外还需要 AWS credentials :

```
AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
## You can also define a custom ACL for exported feeds using this setting:

FEED_STORAGE_S3_ACL
```

## Standard output

即将数据源存放到标准输出。

URI scheme: stdout

例如: `stdout:`

FEED\_STORAGES\_BASE 默认值如下:

```
{
    '': 'scrapy.extensions.feedexport.FileFeedStorage',
    'file': 'scrapy.extensions.feedexport.FileFeedStorage',
    'stdout': 'scrapy.extensions.feedexport.StdoutFeedStorage',
    's3': 'scrapy.extensions.feedexport.S3FeedStorage',
    'ftp': 'scrapy.extensions.feedexport.FTPFeedStorage',
}
```

FEED\_EXPORTERS\_BASE 默认值如下:

```
{
    'json': 'scrapy.exporters.JsonItemExporter',
    'jsonlines': 'scrapy.exporters.JsonLinesItemExporter',
    'jl': 'scrapy.exporters.JsonLinesItemExporter',
    'csv': 'scrapy.exporters.CsvItemExporter',
    'xml': 'scrapy.exporters.XmlItemExporter',
    'marshal': 'scrapy.exporters.MarshalItemExporter',
    'pickle': 'scrapy.exporters.PickleItemExporter',
}
# 在settings.py中设置下列内容, 可以去除默认FEED_EXPORTERS。
FEED_EXPORTERS = {
    'csv': None,
}
```

## 2.3.12 Scrapy-Splash

Scrapy没有JS engine, 因此它无法爬取JavaScript生成的动态网页, 只能爬取静态网页。

scrapy-splash模块主要使用了Splash. 所谓的Splash, 就是一个Javascript渲染服务。它是一个实现了HTTP API的轻量级浏览器, Splash是用Python实现的, 同时使用Twisted和QT。Twisted (QT) 用来让服务具有异步处理能力, 以发挥webkit的并发能力。

Splash的特点如下:

- 并行处理多个网页
- 得到HTML结果以及 (或者) 渲染成图片
- 关掉加载图片或使用 Adblock Plus规则使得渲染速度更快
- 使用JavaScript处理网页内容
- 使用Lua脚本
- 能在Splash-Jupyter Notebooks中开发Splash Lua scripts
- 能够获得具体的HAR格式的渲染信息

## 2.3.13 scrapy-splash的安装

```
pip3 install scrapy-splash
```

scrapy-splash使用的是Splash HTTP API， 所以需要有一个splash instance， 一般采用docker运行splash， 所以需要安装docker。

```
sudo yum install docker
```

开启docker服务，拉取splash镜像（pull the image）：

```
sudo service docker start
```

```
sudo dock pull scrapinghub/splash
```

开启容器

```
sudo docker run -p 8050:8050 scrapinghub/splash
```

之后就可以使用了。

## 使用实例

在此介绍一个简单的实例，那就是利用百度查询手机号码信息。比如，我们在百度输入框中输入手机号码‘159’，然后查询，得到如下信息：



我们将利用scrapy-splash模拟以上操作并获取手机号码信息。

- 1. 创建scrapy项目phone
- 2. 配置settings.py文件，配置的内容如下：

```
ROBOTSTXT_OBEY = False
```

```
SPIDER_MIDDLEWARES = {  
    'scrapy_splash.SplashDeduplicateArgsMiddleware': 100,  
}
```

```
DOWNLOADER_MIDDLEWARES = {  
    'scrapy_splash.SplashCookiesMiddleware': 723,  
    'scrapy_splash.SplashMiddleware': 725,  
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 810  
}
```

```
SPLASH_URL = 'http://localhost:8050'
```

```
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'  
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

创建爬虫文件`phoneSpider.py`, 代码如下:

```
# -*- coding: utf-8 -*-
<p class="mume-header " id="-coding-utf-8--1"></p>

from scrapy import Spider, Request
from scrapy_splash import SplashRequest

# splash lua script
<p class="mume-header " id="splash-lua-script"></p>

script = """
    function main(splash, args)
        assert(splash:go(args.url))
        assert(splash:wait(args.wait))
        js = string.format("document.querySelector('#kw').value=%s;document.querySelector('#su').click()", a
        splash:evaljs(js)
        assert(splash:wait(args.wait))
        return splash:html()
    end
    """

class phoneSpider(Spider):
    name = 'phone'
    allowed_domains = ['www.baidu.com']
    url = 'https://www.baidu.com'

    # start request
    def start_requests(self):
        yield SplashRequest(self.url, callback=self.parse, endpoint='execute', args={'lua_source': script, 'phone

    # parse the html content
    def parse(self, response):
        info = response.css('div.op_mobilephone_r.c-gap-bottom-small').xpath('span/text()').extract()
        print('='*40)
        print(''.join(info))
        print('='*40)
```

运行爬虫， scrapy crawl phone , 结果如下：

```
2018-03-17 16:11:20 [scrapy.core.engine] DEBUG: Crawled (200) <GET https:
=====
手机号码"159:      " 上海  中国移动
=====
2018-03-17 16:11:20 [scrapy.core.engine] INFO: Closing spider (finished)
```

## 2.3.14 内容小结

本节主要介绍了以下内容：

- conda虚拟环境建立
- scrapy安装

- scrapy 项目建立
- scrapy 爬虫生成
- scrapy 爬虫运行
- scrapy shell的使用

### 2.3.15 模块练习与答案

见习题集

## 3 本节总结

本讲主要介绍了爬虫框架scrapy的基本原理和使用方法。

重点是：

- 深刻理解scrapy的组件结构，能够画出架构示意图
- 能够应用scrapy进行爬虫开发。

## 4 课后练习

见习题集。