Scrapy爬虫框架

Scrapy是一个用于爬取网页、提取结构化数据的应用框架，可用于数据挖掘、信息处理或历史归档等多种应用系统。

Scrapy Installation

Scrapy可运行于Python 2.7和Python 3.4或以上版本之上。如果你是用了Anaconda或Miniconda，你可以通过conda-forge渠道安装，目前提供了支持Linux、Windows和OS X等版本的Scrapy。

- 使用conda安装时：

> conda install -c conda-forge scrapy 或者 pip install Scrapy

我们推荐在某种python虚拟环境中安装和使用scrapy，以避免scrapy与其他python程序因版本问题而发生纠缠和使用故障。

在windows环境下，可在命令行窗口中键入如下命令：

> pip install virtualenv

如果对上述命令有疑问，可以参考https://virtualenv.pypa.io/en/stable/userguide/ 获得更多的指导。

如果使用Anaconda，还可以使用已安装的conda虚拟环境，步骤如下：

1. 进入Anaconda prompt命令行窗口，键入如下命令：

> conda create -n myscrapy scrapy

注意：myscrapy是自定义的虚拟环境目录；scrapy是需要在虚拟环境中新安装的python库。

如果安装过程非常慢，或者失败，可以将conda源改为国内大学的镜像站点。设置命令如下：

> conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/ conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/msys2/
>
> ### 设置搜索时显示通道地址
>
> conda config --set show_channel_urls yes
>
> conda config --add channels https://mirrors.ustc.edu.cn/anaconda/pkgs/main/ conda config --add channels https://mirrors.ustc.edu.cn/anaconda/pkgs/free/ conda config --add channels https://mirrors.ustc.edu.cn/anaconda/cloud/conda-forge/ conda config --add channels

> https://mirrors.ustc.edu.cn/anaconda/cloud/msys2/ conda config --add channels
> https://mirrors.ustc.edu.cn/anaconda/cloud/bioconda/ conda config --add channels
> https://mirrors.ustc.edu.cn/anaconda/cloud/menpo/ conda config --set show_channel_urls yes

有时在windows中安装成功后，生成新的scrapy项目仍会报错：

> from cryptography.hazmat.bindings._openssl import ffi, lib ImportError: DLL load failed: 操作系统无法运行 %1。

有两种修复方法：

- 可以尝试删除windows/system32/目录下的libeay32.dll和ssleay32.dll，这两个文件引起了冲突。
- 或者，在conda虚拟环境myscrapy下运行下列安装命令：

> pip install -I cryptography

注：pip install -I 表示忽略已安装情况，重新安装cryptography。

安装完成后，你可以在%Anaconda Home%/envs/中找到myscrapy目录，而目录内有一系列新安装的scrapy库支持文件。

可以运行下列命令测试：

> scrapy startproject tutorial 结果如下： New Scrapy project 'tutorial', using template directory 'D:\pythonspace\Anaconda3\envs\myscrapy\lib\site-packages\scrapy\templates\project', created in: D:\pythonspace\Anaconda3\envs\myscrapy\tutorial You can start your first spider with: cd tutorial scrapy genspider example example.com (myscrapy) D:\pythonspace\Anaconda3\envs\myscrapy>

1. 启动虚拟环境

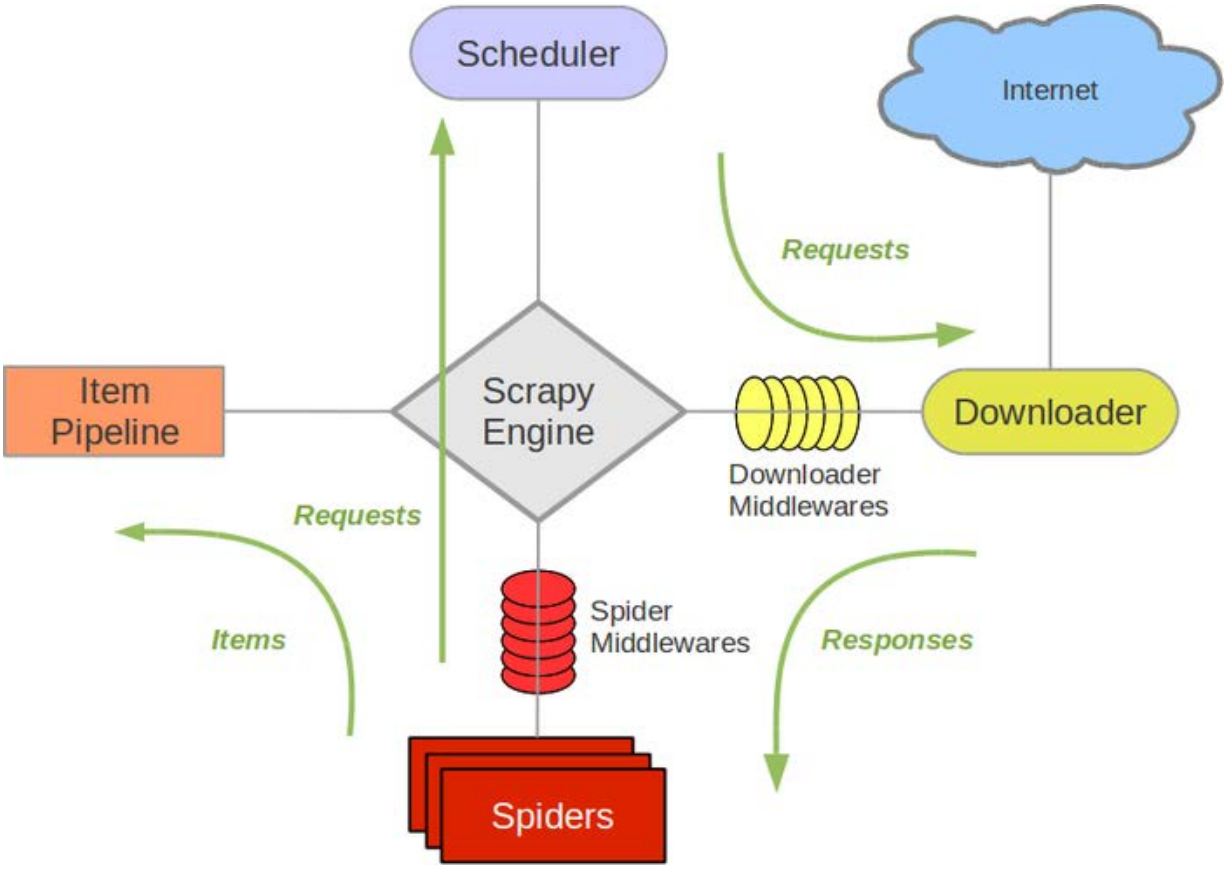
> cd \Anaconda3\envs\myscrapy conda activate myscrapy (myscrapy)\Anaconda3\envs\myscrapy>

预备知识

在学习或使用scrapy之前，最好掌握以下概念和一定的使用技巧：

- lxml: 一种有效的XML和HTML解析器；
- parsel: 一种基于lxml的HTML/XML数据提取库；
- w3lib：一种用于处理URLs和网页编码的多用途服务库；
- twisted：一种异步网络框架（异步I/O)；
- crptography 和 pyOpenSSL：用于处理各种网络层安全需求。

Scrapy结构概览



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）

Scrapy运行流程

Scrapy有3条数据流路径:

1. 请求发起与调度
   A. Spiders经Spider中间件向Engine发送Requests；
   B. Engine将爬取请求交给Scheduler，完成调度。
2. 请求执行与响应
   A. Scheduler将调度后Requests交给Engine；
   B. Engine将请求转发给Downloader；
   C. Downloader执行下载并将响应Response对象提交给Engine；
   D. Engine将Response对象送给Spiders，用于提取Item和跟踪Links。
3. 结果输出与新请求的产生
   A. 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；
   B. Engine将Items送给Item pipelines完成输出处理；
   C. 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

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。

Scrapy初试

下面我们将使用Scrapy爬取quotes.toscrape.com，这是一个收集了众多名人名言的网站。

下面的例子将带你完成如下任务：

1. 新建Scrapy项目；
2. 编写爬取网站和提取数据的爬虫；
3. 使用命令行输出已爬取数据；
4. 递归使用爬虫遍历所有链接；
5. 使用爬虫参数，升级程序功能。

新建项目

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

定制结果（明确目标）

为了爬取quotes.toscrape.com中的名人名言，一个典型的Scrapy爬虫需要完成如下步骤:

1. 打开tutorail目录下的items.py;
2. 在items.py中定义结构化数据字段，用于保存将来爬得的数据。这一结构类似python字典，但提供了一些减少错误的保护检查;
3. 可以考虑创建一个scrapy.item类，并且定义类型为scrapy.Field的类属性来定义一个item（类似于ORM映射关系）;

```
import scrapy
class QuotesItem(scrapy.Item):
    name = scrapy.Field()
    level = scrapy.Field()
    info = scrapy.Field()
```

编写爬虫文件

Scrapy中爬虫分为2步:

1. 爬取数据 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。

例如，我们可以编写下面的爬虫类:

In [ ]:
```
import scrapy

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

    def start_requests(self):
        urls = [
            'http://quotes.toscrape.com/page/1/',
            'http://quotes.toscrape.com/page/2/',
        ]
        for url in urls:
            yield scrapy.Request(url=url, callback=self.parse)

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = 'quotes-%s.html' % page
        with open(filename, 'wb') as f:
            f.write(response.body)
        self.log('Saved file %s' % filename)
```

下面对上面的代码中的关键部分进行说明:

- name
  - name是爬虫的标识符，在本项目的所有爬虫中，每个爬虫的name是唯一的。
- start_requests()
  - 必须返回一个可迭代的Requests（例如返回一个requests列表，或者写一个生成器函数），这个Spider将从这些请求开始爬取网站。
- parse()
  - 解析内容的函数，这个方法将会在每次获得Response之后调用，响应参数是TextResponse类的对象，里面存有页面内容和可用于处理页面的有用方法。解析函数通常需要抽取出所需的内容和新的URLs，新的URLs将用于生成新的请求。

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 quotes

这个命令将启动我们上面编写的名为quotes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:

```
(omitted for brevity)
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Spider opened
2016-12-16 21:24:05 [scrapy.extensions.logstats] INFO: Crawled 0 pages
(at 0 pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:24:05 [scrapy.extensions.telnet] DEBUG: Telnet console
listening on 127.0.0.1:6023
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (404) <GET
http://quotes.toscrape.com/robots.txt> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET
```

```
http://quotes.toscrape.com/page/1/> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET
http://quotes.toscrape.com/page/2/> (referer: None)
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-1.html
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-2.html
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Closing spider (finished)
```

此时，检查一下当前目录，你会注意到有两个新的文件被生成了：

> quotes-1.html quotes-2.html

这就是在上面程序中parse方法中处理的两个URLs。

如果不想实现start_requests()方法来根据URLs生成scrapy.Request对象，也可以只定以一个start_urls列表，使其作为类属性。这个列表将用作默认的start_requests()来生成初始化的requests。例如：

In [ ]:
```
import scrapy


class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        'http://quotes.toscrape.com/page/1/',
        'http://quotes.toscrape.com/page/2/',
    ]

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = 'quotes-%s.html' % page
        with open(filename, 'wb') as f:
            f.write(response.body)
```

上面的代码中，我们没有指定解析结果的回调函数，但是scrapy仍然会调用parse，因为scrapy默认的处理response对象的回调函数就是parse()。

提取数据

学习如何使用Scrapy提取数据的方法是尝试使用Scrapy shell时使用选择器。运行Scrapy shell的命令是：

> scrapy shell "http://quotes.toscrape.com/page/1"

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：

```
[ ... Scrapy log here ... ]
2016-09-19 12:09:27 [scrapy.core.engine] DEBUG: Crawled (200) <GE
T http://quotes.toscrape.com/page/1/> (referer: None)
[s] Available Scrapy objects:
[s]   scrapy     scrapy module (contains scrapy.Request, scrapy.Se
lector, etc)
[s]   crawler    <scrapy.crawler.Crawler object at 0x7fa91d888c90>
[s]   item       {}
```

```
[s]   request    <GET http://quotes.toscrape.com/page/1/>
[s]   response   <200 http://quotes.toscrape.com/page/1/>
[s]   settings   <scrapy.settings.Settings object at 0x7fa91d888c
10>
[s]   spider     <DefaultSpider 'default' at 0x7fa91c8af990>
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local o
bjects
[s]   view(response)    View response in a browser
>>>
```

之后，尝试运行:

> response.css('title') 或 response.xpath('//title')

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.css('title::text').extract() 或 response.xpath('//title/text()').extract() 结果为: ['Quotes to Scrape']

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.css['title::text'].extract_first() 或
> response.xpath('//title/text()').extract_first() 结果是: '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') ['Quotes']
> response.xpath('//title/text()').re(r'Q\w+') ['Quotes']
> response.xpath('//title/text()').re(r'(\w+) to (\w+)') ['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 proce
ss 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-thou
ghts</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.com"

执行成功后，运行下面命令将会得到一个名言HTML元素的selectors列表：

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

每次查询返回的selectors，允许我们运行更多的子元素查询。例如：

> quote = response.css("div.quote")[0] 或 response.xpath('//div[@class="quote"]')[0]

现在我们可以利用quote对象提取出子标签title、author、tags等内容：

> title = quote.xpath("//span/text()").extract_first() 或 title = quote.xpath('./span[@itemprop="text"]/text()').extract_first() title "The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."
>
> author = quote.xpath('//small[@class="author"]/text()').extract_first() author "Albert Einstein" tags = quote[0].xpath('.//div[@class="tags"]/a[@class="tag"]/text()').extract() tags ['change', 'deep-thoughts', 'thinking', 'world']

在自定义爬虫中提取数据

通过scrapy shell的手动分析，可以明确提取数据的基本方法。现在回到我们自定义的spider类，完成parse方法。

一个Scrapy爬虫通常会生成许多包含从网页爬得数据的字典。为了令parse方法生成可迭代的字典，我们通常在callback回调处，使用yield关键字。下面的代码给出了示例：

In [ ]:
```
# -*- coding: utf-8 -*-
import scrapy


class QuotespiderSpider(scrapy.Spider):
    name = 'quotespider'
    allowed_domains = ["quotes.toscrape.com"]
    start_urls = ["http://quotes.toscrape.com/page/1/"]

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

注意

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

存储爬得数据

若要存储爬得数据，最简单的方法是通过下面的命令使用Feed exports：

> scrapy crawl quotes -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qutoes.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 Pipline类。一个服务于Item Pipelines的占位符文件在新建项目时就已被生成了，就存放在tutorial/pipelines.py文件中。

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">→</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：

> links = response.xpath('//li[@class="next"]/a').re(r'\/page\/\d+').extract_first() links
> '/page/2'

之后，在自定义爬虫类中，我们设计一个递归爬取新URLs的方法：

In [ ]:
```
# -*- coding: utf-8 -*-
import scrapy


class QuotespiderSpider(scrapy.Spider):
    name = 'quotespider'
    allowed_domains = ["quotes.toscrape.com"]
    start_urls = ["http://quotes.toscrape.com/page/1/"]

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.xpath(".//span/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 = response.xpath('//li[@class="next"]/a').re(r'\/page\
/\d+')
        if next_page is not None:
            next_page = response.urljoin(next_page[0])
            yield scrapy.Request(next_page, callback=self.parse)
```

更多实例

下面的例子演示了多个回调函数和跟踪链接的情况：

In [ ]:
```
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main 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。

The parse_author callback defines a helper function to extract and cleanup the data from a CSS query and yields the Python dict with the author data.

Another interesting thing this spider demonstrates is that, even if there are many quotes from the same author, we don't need to worry about visiting the same author page multiple times. By default, Scrapy filters out duplicated requests to URLs already visited, avoiding the problem of hitting servers too much because of a programming mistake. This can be configured by the setting DUPEFILTER_CLASS.

Hopefully by now you have a good understanding of how to use the mechanism of following links and callbacks with Scrapy.

As yet another example spider that leverages the mechanism of following links, check out the CrawlSpider class for a generic spider that implements a small rules engine that you can use to write your crawlers on top of it.

Also, a common pattern is to build an item with data from more than one page, using a trick to pass additional data to the callbacks.

Using spider arguments You can provide command line arguments to your spiders by using the -a option when running them:

scrapy crawl quotes -o quotes-humor.json -a tag=humor These arguments are passed to the Spider's **init** method and become spider attributes by default.

In this example, the value provided for the tag argument will be available via self.tag. You can use this to make your spider fetch only quotes with a specific tag, building the URL based on the argument:

In [ ]:
```
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)
```

If you pass the tag=humor argument to this spider, you'll notice that it will only visit URLs from the humor tag, such as http://quotes.toscrape.com/tag/humor.

You can learn more about handling spider arguments here.