scrapy-cookbook Documentation

发布 *0.2.2*

Xiong Neng

3 月 07, 2017

Contents

Contents:

1 Scrapy 教程 01- 入门篇

Scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架。可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。其最初是为了页面抓取 (更确切来说, 网络抓取) 所设计的，也可以应用在获取 API 所返回的数据 (比如 Web Services) 或者通用的网络爬虫。

Scrapy 也能帮你实现高阶的爬虫框架，比如爬取时的网站认证、内容的分析处理、重复抓取、分布式爬取等等很复杂的事。

1.1 安装 scrapy

我的测试环境是 centos6.5

升级 python 到最新版的 2.7，下面的所有步骤都切换到 root 用户

由于 scrapy 目前只能运行在 python2 上，所以先更新 centos 上面的 python 到最新的 Python 2.7.11，具体方法请 google 下很多这样的教程。

先安装一些依赖软件

```
yum install python-devel
yum install libffi-devel
yum install openssl-devel
```

然后安装 pyopenssl 库

```
pip install pyopenssl
```

安装 xlml

```
yum install python-lxml
yum install libxml2-devel
yum install libxslt-devel
```

安装 service-identity

```
pip install service-identity
```

安装 twisted

```
pip install scrapy
```

安装 scrapy

```
pip install scrapy -U
```

测试 scrapy

```
scrapy bench
```

最终成功，太不容易了！

1.2 简单示例

创建一个 python 源文件，名为 stackoverflow.py，内容如下：

```
import scrapy


class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow'
    start_urls = ['http://stackoverflow.com/questions?sort=votes']

    def parse(self, response):
        for href in response.css('.question-summary h3 a::attr(href)'):
            full_url = response.urljoin(href.extract())
            yield scrapy.Request(full_url, callback=self.parse_question)

    def parse_question(self, response):
        yield {
            'title': response.css('h1 a::text').extract()[0],
            'votes': response.css('.question .vote-count-post::text').
↪extract()[0],
            'body': response.css('.question .post-text').extract()[0],
            'tags': response.css('.question .post-tag::text').extract(),
            'link': response.url,
        }
```

运行：

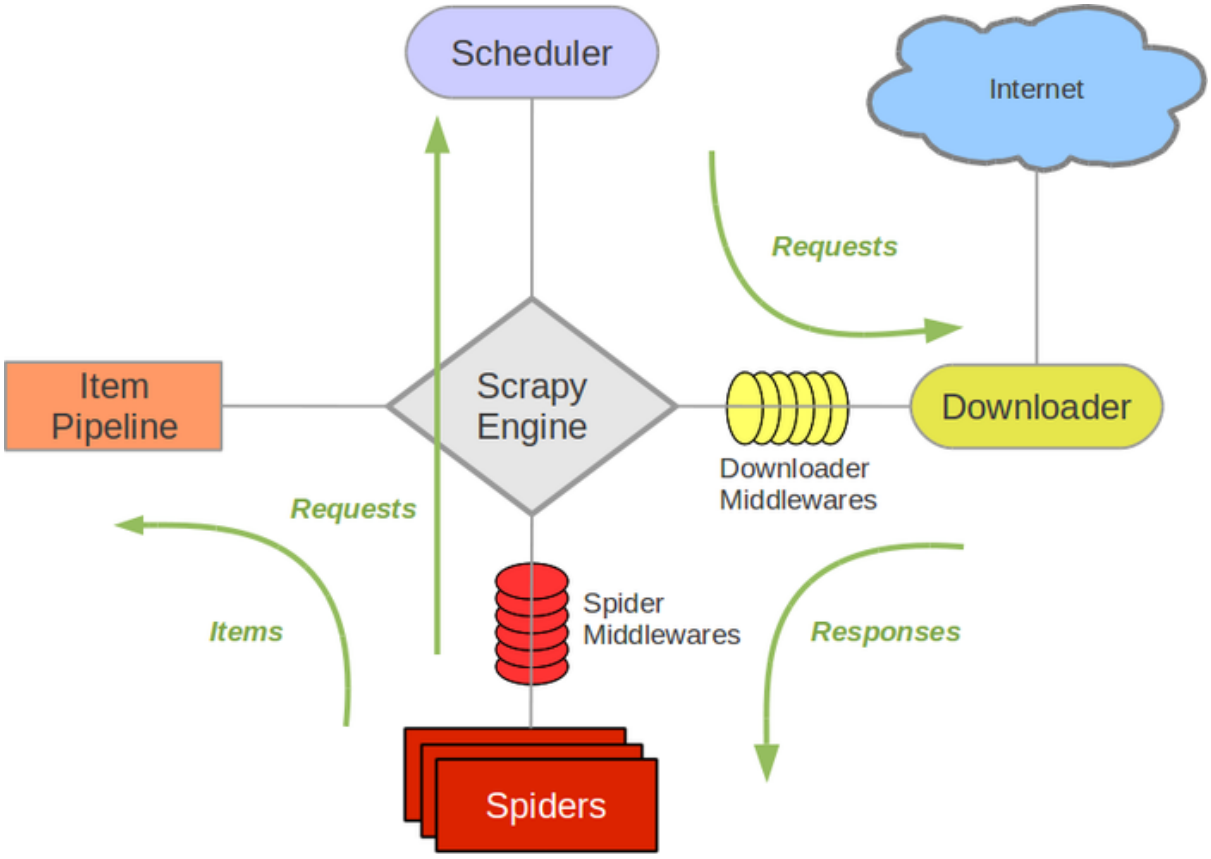
```
scrapy runspider stackoverflow_spider.py -o top-stackoverflow-questions.json
```

结果类似下面：

```
[{
    "body": "... LONG HTML HERE ...",
    "link": "http://stackoverflow.com/questions/11227809/why-is-processing-a-
↪sorted-array-faster-than-an-unsorted-array",
    "tags": ["java", "c++", "performance", "optimization"],
    "title": "Why is processing a sorted array faster than an unsorted array?
↪",
    "votes": "9924"
},
{
    "body": "... LONG HTML HERE ...",
    "link": "http://stackoverflow.com/questions/1260748/how-do-i-remove-a-git-
↪submodule",
    "tags": ["git", "git-submodules"],
    "title": "How do I remove a Git submodule?",
    "votes": "1764"
},
...]
```

当你运行 `scrapy runspider somefile.py` 这条语句的时候，Scrapy 会去寻找源文件中定义的一个 spider 并且交给爬虫引擎来执行它。`start_urls` 属性定义了开始的 URL，爬虫会通过它来构建初始的请求，返回 response 后再调用默认的回调方法 `parse` 并传入这个 response。我们在 `parse` 回调方法中通过使用 css 选择器提取每个提问页面链接的 href 属性值，然后 `yield` 另外一个请求，并注册 `parse_question` 回调方法，在这个请求完成后被执行。

处理流程图：



Scrapy 的一个好处是所有请求都是被调度并异步处理，就算某个请求出错也不影响其他请求继续被处理。

我们的示例中将解析结果生成 json 格式，你还可以导出为其他格式（比如 XML、CSV），或者是将其存储到 FTP、Amazon S3 上。你还可以通过pipeline 将它们存储到数据库中去，这些数据保存的方式各种各样。

1.3 Scrapy 特性一览

你已经可以通过 Scrapy 从一个网站上面爬取数据并将其解析保存下来了，但是这只是 Scrapy 的皮毛。Scrapy 提供了更多的特性来让你爬取更加容易和高效。比如：

1. 内置支持扩展的 CSS 选择器和 XPath 表达式来从 HTML/XML 源码中选择并提取数据，还能使用正则表达式
2. 提供交互式 shell 控制台试验 CSS 和 XPath 表达式，这个在调试你的蜘蛛程序时很有用
3. 内置支持生成多种格式的订阅导出（JSON、CSV、XML）并将它们存储在多个位置（FTP、S3、本地文件系统）
4. 健壮的编码支持和自动识别，用于处理外文、非标准和错误编码问题
5. 可扩展，允许你使用signals 和友好的 API(middlewares, extensions, 和 pipelines) 来编写自定义插件功能。
6. 大量的内置扩展和中间件供使用：

- cookies and session handling
- HTTP features like compression, authentication, caching
- user-agent spoofing
- robots.txt
- crawl depth restriction
- and more

7. 还有其他好多好东东，比如可重复利用蜘蛛来爬取Sitemaps和 XML/CSV 订阅，一个跟爬取元素关联的媒体管道来 自动下载图片，一个缓存 DNS 解析器等等。

2 Scrapy 教程 02- 完整示例

这篇文章我们通过一个比较完整的例子来教你使用 Scrapy，我选择爬取虎嗅网首页的新闻列表。

这里我们将完成如下几个步骤：

- 创建一个新的 Scrapy 工程
- 定义你所需要要抽取的 Item 对象
- 编写一个 spider 来爬取某个网站并提取出所有的 Item 对象
- 编写一个 Item Pipline 来存储提取出来的 Item 对象

Scrapy 使用 Python 语言编写，如果你对这门语言还不熟，请先去学习下基本知识。

2.1 创建 Scrapy 工程

在任何你喜欢的目录执行如下命令

```
scrapy startproject coolscrapy
```

将会创建 coolscrapy 文件夹，其目录结构如下：

```
coolscrapy/
    scrapy.cfg            # 部署配置文件

    coolscrapy/           # Python 模块，你所有的代码都放这里面
        __init__.py

        items.py          # Item 定义文件

        pipelines.py      # pipelines 定义文件

        settings.py       # 配置文件
```

```
        spiders/           # 所有爬虫 spider 都放这个文件夹下面
            __init__.py
            ...
```

2.2 定义我们的 Item

我们通过创建一个 scrapy.Item 类，并定义它的类型为 scrapy.Field 的属性，我们准备将虎嗅网新闻列表的名称、链接地址和摘要爬取下来。

```
import scrapy

class HuxiuItem(scrapy.Item):
    title = scrapy.Field()    # 标题
    link = scrapy.Field()     # 链接
    desc = scrapy.Field()     # 简述
    posttime = scrapy.Field() # 发布时间
```

也许你觉得定义这个 Item 有点麻烦，但是定义完之后你可以得到许多好处，这样你就可以使用 Scrapy 中其他有用的组件和帮助类。

2.3 第一个 Spider

蜘蛛就是你定义的一些类，Scrapy 使用它们来从一个 domain（或 domain 组）爬取信息。在蜘蛛类中定义了一个初始化的 URL 下载列表，以及怎样跟踪链接，如何解析页面内容来提取 Item。

定义一个 Spider，只需继承 `scrapy.Spider` 类并定于一些属性：

- name: Spider 名称，必须是唯一的
- start_urls: 初始化下载链接 URL
- parse(): 用来解析下载后的 Response 对象，该对象也是这个方法的唯一参数。它负责解析返回页面数据并提取出相应的 Item（返回 Item 对象），还有其他合法的链接 URL（返回 Request 对象）。

我们在 coolscrapy/spiders 文件夹下面新建 `huxiu_spider.py`，内容如下：

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: sample
Desc :
"""
from coolscrapy.items import HuxiuItem
import scrapy

class HuxiuSpider(scrapy.Spider):
    name = "huxiu"
```

```
    allowed_domains = ["huxiu.com"]
    start_urls = [
        "http://www.huxiu.com/index.php"
    ]

    def parse(self, response):
        for sel in response.xpath('//div[@class="mod-info-flow"]/div/
↪div[@class="mob-ctt"]'):
            item = HuxiuItem()
            item['title'] = sel.xpath('h3/a/text()')[0].extract()
            item['link'] = sel.xpath('h3/a/@href')[0].extract()
            url = response.urljoin(item['link'])
            item['desc'] = sel.xpath('div[@class="mob-sub"]/text()')[0].
↪extract()
            print(item['title'],item['link'],item['desc'])
```

2.4 运行爬虫

在根目录执行下面的命令，其中 huxiu 是你定义的 spider 名字：

```
scrapy crawl huxiu
```

如果一切正常，应该可以打印出每一个新闻

2.5 处理链接

如果想继续跟踪每个新闻链接进去，看看它的详细内容的话，那么可以在 parse() 方法中返回一个 Request 对象，然后注册一个回调函数来解析新闻详情。

```
from coolscrapy.items import HuxiuItem
import scrapy

class HuxiuSpider(scrapy.Spider):
    name = "huxiu"
    allowed_domains = ["huxiu.com"]
    start_urls = [
        "http://www.huxiu.com/index.php"
    ]

    def parse(self, response):
        for sel in response.xpath('//div[@class="mod-info-flow"]/div/
↪div[@class="mob-ctt"]'):
            item = HuxiuItem()
            item['title'] = sel.xpath('h3/a/text()')[0].extract()
            item['link'] = sel.xpath('h3/a/@href')[0].extract()
            url = response.urljoin(item['link'])
            item['desc'] = sel.xpath('div[@class="mob-sub"]/text()')[0].
↪extract()
```

```
            # print(item['title'],item['link'],item['desc'])
            yield scrapy.Request(url, callback=self.parse_article)

    def parse_article(self, response):
        detail = response.xpath('//div[@class="article-wrap"]')
        item = HuxiuItem()
        item['title'] = detail.xpath('h1/text()')[0].extract()
        item['link'] = response.url
        item['posttime'] = detail.xpath(
            'div[@class="article-author"]/span[@class="article-time"]/text()
↪')[0].extract()
        print(item['title'],item['link'],item['posttime'])
        yield item
```

现在 parse 只提取感兴趣的链接，然后将链接内容解析交给另外的方法去处理了。你可以基于这个构建更加复杂的爬虫程序了。

2.6 导出抓取数据

最简单的保存抓取数据的方式是使用 json 格式的文件保存在本地，像下面这样运行：

```
scrapy crawl huxiu -o items.json
```

在演示的小系统里面这种方式足够了。不过如果你要构建复杂的爬虫系统，最好自己编写Item Pipeline。

2.7 保存数据到数据库

上面我们介绍了可以将抓取的 Item 导出为 json 格式的文件，不过最常见的做法还是编写 Pipeline 将其存储到数据库中。我们在 coolscrapy/pipelines.py 定义

```
# -*- coding: utf-8 -*-
import datetime
import redis
import json
import logging
from contextlib import contextmanager

from scrapy import signals
from scrapy.exporters import JsonItemExporter
from scrapy.pipelines.images import ImagesPipeline
from scrapy.exceptions import DropItem
from sqlalchemy.orm import sessionmaker
from coolscrapy.models import News, db_connect, create_news_table, Article


class ArticleDataBasePipeline(object):
    """ 保存文章到数据库 """
```

```
    def __init__(self):
        engine = db_connect()
        create_news_table(engine)
        self.Session = sessionmaker(bind=engine)

    def open_spider(self, spider):
        """This method is called when the spider is opened."""
        pass

    def process_item(self, item, spider):
        a = Article(url=item["url"],
                    title=item["title"].encode("utf-8"),
                    publish_time=item["publish_time"].encode("utf-8"),
                    body=item["body"].encode("utf-8"),
                    source_site=item["source_site"].encode("utf-8"))
        with session_scope(self.Session) as session:
            session.add(a)

    def close_spider(self, spider):
        pass
```

上面我使用了 python 中的 SQLAlchemy 来保存数据库，这个是一个非常优秀的 ORM 库，我写了篇关于它的入门教程，可以参考下。

然后在 setting.py 中配置这个 Pipeline，还有数据库链接等信息：

```
ITEM_PIPELINES = {
    'coolscrapy.pipelines.ArticleDataBasePipeline': 5,
}

# linux pip install MySQL-python
DATABASE = {'drivername': 'mysql',
            'host': '192.168.203.95',
            'port': '3306',
            'username': 'root',
            'password': 'mysql',
            'database': 'spider',
            'query': {'charset': 'utf8'}}
```

再次运行爬虫

```
scrapy crawl huxiu
```

那么所有新闻的文章都存储到数据库中去了。

2.8 下一步

本章只是带你领略了 scrapy 最基本的功能，还有很多高级特性没有讲到。接下来会通过多个例子向你展示 scrapy 的其他特性，然后再深入讲述每个特性。

3 Scrapy 教程 03- Spider 详解

Spider 是爬虫框架的核心，爬取流程如下：

1. 先初始化请求 URL 列表，并指定下载后处理 response 的回调函数。初次请求 URL 通过 `start_urls` 指定，调用 `start_requests()` 产生 `Request` 对象，然后注册 `parse` 方法作为回调
2. 在 parse 回调中解析 response 并返回字典,`Item` 对象,`Request` 对象或它们的迭代对象。`Request` 对象还会包含回调函数，之后 Scrapy 下载完后会被这里注册的回调函数处理。
3. 在回调函数里面，你通过使用选择器（同样可以使用 BeautifulSoup,lxml 或其他工具）解析页面内容，并生成解析后的结果 Item。
4. 最后返回的这些 Item 通常会被持久化到数据库中 (使用Item Pipeline) 或者使用Feed exports将其保存到文件中。

尽管这个流程适合于所有的蜘蛛，但是 Scrapy 里面为不同的使用目的实现了一些常见的 Spider。下面我们把它们列出来。

3.1 CrawlSpider

链接爬取蜘蛛，专门为那些爬取有特定规律的链接内容而准备的。如果你觉得它还不足以适合你的需求，可以先继承它然后覆盖相应的方法，或者自定义 Spider 也行。

它除了从 `scrapy.Spider` 类继承的属性外，还有一个新的属性 `rules`, 它是一个 `Rule` 对象列表，每个 `Rule` 对象定义了某个规则，如果多个 `Rule` 匹配一个连接，那么使用第一个，根据定义的顺序。

一个详细的例子：

```
from coolscrapy.items import HuxiuItem
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor


class LinkSpider(CrawlSpider):
    name = "link"
    allowed_domains = ["huxiu.com"]
    start_urls = [
        "http://www.huxiu.com/index.php"
    ]

    rules = (
        # 提取匹配正则式 '/group?f=index_group' 链接 (但是不能匹配 'deny.php')
        # 并且会递归爬取 (如果没有定义 callback, 默认 follow=True).
        Rule(LinkExtractor(allow=('/group?f=index_group', ), deny=('deny\.php
↪', ))),
        # 提取匹配 '/article/\d+/\d+.html' 的链接，并使用 parse_item 来解析它们下载后
的内容，不递归
```

```
        Rule(LinkExtractor(allow=('/article/\d+/\d+\.html', )), callback=
↪'parse_item'),
    )

    def parse_item(self, response):
        self.logger.info('Hi, this is an item page! %s', response.url)
        detail = response.xpath('//div[@class="article-wrap"]')
        item = HuxiuItem()
        item['title'] = detail.xpath('h1/text()')[0].extract()
        item['link'] = response.url
        item['posttime'] = detail.xpath(
            'div[@class="article-author"]/span[@class="article-time"]/text()
↪')[0].extract()
        print(item['title'],item['link'],item['posttime'])
        yield item
```

3.2 XMLFeedSpider

XML 订阅蜘蛛，用来爬取 XML 形式的订阅内容，通过某个指定的节点来遍历。可使用 iternodes, xml, 和 html 三种形式的迭代器，不过当内容比较多的时候推荐使用 iternodes，默认也是它，可以节省内存提升性能，不需要将整个 DOM 加载到内存中再解析。而使用 html 可以处理 XML 有格式错误的内容。处理 XML 的时候最好先Removing namespaces

接下来我通过爬取我的博客订阅 XML 来展示它的使用方法。

```
from coolscrapy.items import BlogItem
import scrapy
from scrapy.spiders import XMLFeedSpider


class XMLSpider(XMLFeedSpider):
    name = "xml"
    namespaces = [('atom', 'http://www.w3.org/2005/Atom')]
    allowed_domains = ["github.io"]
    start_urls = [
        "http://www.pycoding.com/atom.xml"
    ]
    iterator = 'xml'  # 缺省的 iternodes，貌似对于有 namespace 的 xml 不行
    itertag = 'atom:entry'

    def parse_node(self, response, node):
        # self.logger.info('Hi, this is a <%s> node!', self.itertag)
        item = BlogItem()
        item['title'] = node.xpath('atom:title/text()')[0].extract()
        item['link'] = node.xpath('atom:link/@href')[0].extract()
        item['id'] = node.xpath('atom:id/text()')[0].extract()
        item['published'] = node.xpath('atom:published/text()')[0].extract()
```

```
        item['updated'] = node.xpath('atom:updated/text()')[0].extract()
        self.logger.info('|'.join([item['title'],item['link'],item['id'],item[
↪'published']]))
        return item
```

3.3 CSVFeedSpider

这个跟上面的 XMLFeedSpider 很类似，区别在于它会一行一行的迭代，而不是一个节点一个节点的迭代。每次迭代行的时候会调用 `parse_row()` 方法。

```
from coolscrapy.items import BlogItem
from scrapy.spiders import CSVFeedSpider


class CSVSpider(CSVFeedSpider):
    name = "csv"
    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 = BlogItem()
        item['id'] = row['id']
        item['name'] = row['name']
        return item
```

3.4 SitemapSpider

站点地图蜘蛛，允许你使用Sitemaps发现 URL 后爬取整个站点。还支持嵌套的站点地图以及从 `robots.txt` 中发现站点 URL

4 Scrapy 教程 04- Selector 详解

在你爬取网页的时候，最普遍的事情就是在页面源码中提取需要的数据，我们有几个库可以帮你完成这个任务：

1. BeautifulSoup是 python 中一个非常流行的抓取库, 它还能合理的处理错误格式的标签，但是有一个唯一缺点就是：它运行很慢。
2. lxml是一个基于ElementTree的 XML 解析库 (同时还能解析 HTML), 不过 lxml 并不是 Python 标准库

而 Scrapy 实现了自己的数据提取机制，它们被称为选择器，通过XPath或CSS表达式在 HTML 文档中来选择特定的部分

XPath是一用来在 XML 中选择节点的语言，同时可以用在 HTML 上面。CSS是一种 HTML 文档上面的样式语言。

Scrapy 选择器构建在 lxml 基础之上，所以可以保证速度和准确性。

本章我们来详细讲解下选择器的工作原理，还有它们极其简单和相似的 API，比 lxml 的 API 少多了，因为 lxml 可以用于很多其他领域。

完整的 API 请查看Selector 参考

4.1 关于选择器

Scrapy 帮我们下载完页面后，我们怎样在满是 html 标签的内容中找到我们所需要的元素呢，这里就需要使用到选择器了，它们是用来定位元素并且提取元素的值。先来举几个例子看看：

- /html/head/title: 选择 `<title>` 节点, 它位于 html 文档的 `<head>` 节点内
- /html/head/title/text(): 选择上面的 `<title>` 节点的内容.
- //td: 选择页面中所有的元素
- //div[@class=”mine”]: 选择所有拥有属性 `class="mine"` 的 div 元素

Scrapy 使用 css 和 xpath 选择器来定位元素，它有四个基本方法：

- xpath(): 返回选择器列表，每个选择器代表使用 xpath 语法选择的节点
- css(): 返回选择器列表，每个选择器代表使用 css 语法选择的节点
- extract(): 返回被选择元素的 unicode 字符串
- re(): 返回通过正则表达式提取的 unicode 字符串列表

4.2 使用选择器

下面我们通过 Scrapy shell 演示下选择器的使用，假设我们有如下的一个网页http://doc.scrapy.org/en/latest/_static/selectors-sample1.html，内容如下：

```
<html>
 <head>
  <base href='http://example.com/' />
  <title>Example website</title>
 </head>
 <body>
  <div id='images'>
   <a href='image1.html'>Name: My image 1 <br /><img src='image1_thumb.jpg' />
↪</a>
   <a href='image2.html'>Name: My image 2 <br /><img src='image2_thumb.jpg' />
↪</a>
```

```
   <a href='image3.html'>Name: My image 3 <br /><img src='image3_thumb.jpg' />
↪</a>
   <a href='image4.html'>Name: My image 4 <br /><img src='image4_thumb.jpg' />
↪</a>
   <a href='image5.html'>Name: My image 5 <br /><img src='image5_thumb.jpg' />
↪</a>
  </div>
 </body>
</html>
```

首先我们打开 shell

```
scrapy shell http://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

运行

```
>>> response.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
>>> response.css('title::text')
[<Selector (text) xpath=//title/text()>]
```

结果可以看出,xpath() 和 css() 方法返回的是 SelectorList 实例，是一个选择器列表，你可以选择嵌套的数据：

```
>>> response.css('img').xpath('@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

必须使用.extract() 才能提取最终的数据，如果你只想获得第一个匹配的，可以使用.extract_first()

```
>>> response.xpath('//div[@id="images"]/a/text()').extract_first()
u'Name: My image 1 '
```

如果没有找到，会返回 None，还可选择默认值

```
>>> response.xpath('//div[@id="not-exists"]/text()').extract_first(default=
↪'not-found')
'not-found'
```

而 CSS 选择器还可以使用 CSS3 标准：

```
>>> response.css('title::text').extract()
[u'Example website']
```

下面是几个比较全面的示例：

```
>>> response.xpath('//base/@href').extract()
[u'http://example.com/']

>>> response.css('base::attr(href)').extract()
[u'http://example.com/']

>>> response.xpath('//a[contains(@href, "image")]/@href').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.css('a[href*=image]::attr(href)').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.xpath('//a[contains(@href, "image")]/img/@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']

>>> response.css('a[href*=image] img::attr(src)').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

4.3 嵌套选择器

xpath() 和 css() 返回的是选择器列表，所以你可以继续使用它们的方法。举例来讲：

```
>>> links = response.xpath('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>
↪',
 u'<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>
↪',
 u'<a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>
↪',
 u'<a href="image4.html">Name: My image 4 <br><img src="image4_thumb.jpg"></a>
↪',
 u'<a href="image5.html">Name: My image 5 <br><img src="image5_thumb.jpg"></a>
↪']
```

```
>>> for index, link in enumerate(links):
...     args = (index, link.xpath('@href').extract(), link.xpath('img/@src').
↪extract())
...     print 'Link number %d points to url %s and image %s' % args

Link number 0 points to url [u'image1.html'] and image [u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and image [u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and image [u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and image [u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and image [u'image5_thumb.jpg']
```

4.4 使用正则表达式

Selector 有一个 re() 方法通过正则表达式提取数据，它返回的是 unicode 字符串列表，你不能再去嵌套使用

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']

>>> response.xpath('//a[contains(@href, "image")]/text()').re_first(r
↪'Name:\s*(.*)')
u'My image 1'
```

4.5 XPath 相对路径

当你嵌套使用 XPath 时候，不要使用/开头的，因为这个会相对文档根节点开始算起，需要使用相对路径

```
>>> divs = response.xpath('//div')
>>> for p in divs.xpath('.//p'):  # extracts all <p> inside
...     print p.extract()

# 或者下面这个直接使用 p 也可以
>>> for p in divs.xpath('p'):
...     print p.extract()
```

4.6 XPath 建议

使用 text 作为条件时

避免使用.//text(), 直接使用.

```
>>> sel.xpath("//a[contains(., 'Next Page')]").extract()
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
```

//node[1] 和 (//node)[1] 区别

- //node[1]: 选择所有位于第一个子节点位置的 node 节点
- (//node)[1]: 选择所有的 node 节点，然后返回结果中的第一个 node 节点

通过 class 查找时优先考虑 CSS

```
>> from scrapy import Selector
>>> sel = Selector(text='<div class="hero shout"><time datetime="2014-07-23␣
↪19:00">Special date</time></div>')
>>> sel.css('.shout').xpath('./time/@datetime').extract()
[u'2014-07-23 19:00']
```

5 Scrapy 教程 05- Item 详解

Item 是保存结构数据的地方，Scrapy 可以将解析结果以字典形式返回，但是 Python 中字典缺少结构，在大型爬虫系统中很不方便。

Item 提供了类字典的 API，并且可以很方便的声明字段，很多 Scrapy 组件可以利用 Item 的其他信息。

5.1 定义 Item

定义 Item 非常简单，只需要继承 scrapy.Item 类，并将所有字段都定义为 scrapy.Field 类型即可

```
import scrapy

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    last_updated = scrapy.Field(serializer=str)
```

5.2 Item Fields

`Field` 对象可用来对每个字段指定元数据。例如上面 `last_updated` 的序列化函数指定为 `str`，可任意指定元数据，不过每种元数据对于不同的组件意义不一样。

5.3 Item 使用示例

你会看到 Item 的使用跟 Python 中的字典 API 非常类似

创建 Item

```
>>> product = Product(name='Desktop PC', price=1000)
>>> print product
Product(name='Desktop PC', price=1000)
```

获取值

```
>>> product['name']
Desktop PC
>>> product.get('name')
Desktop PC

>>> product['price']
1000

>>> product['last_updated']
Traceback (most recent call last):
    ...
KeyError: 'last_updated'

>>> product.get('last_updated', 'not set')
not set

>>> product['lala'] # getting unknown field
Traceback (most recent call last):
    ...
KeyError: 'lala'

>>> product.get('lala', 'unknown field')
'unknown field'

>>> 'name' in product  # is name field populated?
True

>>> 'last_updated' in product  # is last_updated populated?
False
```

```
>>> 'last_updated' in product.fields  # is last_updated a declared field?
True

>>> 'lala' in product.fields  # is lala a declared field?
False
```

设置值

```
>>> product['last_updated'] = 'today'
>>> product['last_updated']
today

>>> product['lala'] = 'test' # setting unknown field
Traceback (most recent call last):
    ...
KeyError: 'Product does not support field: lala'
```

访问所有的值

```
>>> product.keys()
['price', 'name']

>>> product.items()
[('price', 1000), ('name', 'Desktop PC')]
```

5.4 Item Loader

Item Loader 为我们提供了生成 Item 的相当便利的方法。Item 为抓取的数据提供了容器，而 Item Loader 可以让我们非常方便的将输入填充到容器中。

下面我们通过一个例子来展示一般使用方法：

```
from scrapy.loader import ItemLoader
from myproject.items import Product

def parse(self, response):
    l = ItemLoader(item=Product(), response=response)
    l.add_xpath('name', '//div[@class="product_name"]')
    l.add_xpath('name', '//div[@class="product_title"]')
    l.add_xpath('price', '//p[@id="price"]')
    l.add_css('stock', 'p#stock]')
    l.add_value('last_updated', 'today') # you can also use literal values
    return l.load_item()
```

注意上面的 name 字段是从两个 xpath 路径添累加后得到。

5.5 输入/输出处理器

每个 Item Loader 对每个 Field 都有一个输入处理器和一个输出处理器。输入处理器在数据被接受到时执行，当数据收集完后调用 ItemLoader.load_item() 时再执行输出处理器，返回最终结果。

```
l = ItemLoader(Product(), some_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_css('name', css) # (3)
l.add_value('name', 'test') # (4)
return l.load_item() # (5)
```

执行流程是这样的：

1. xpath1 中的数据被提取出来，然后传输到 name 字段的输入处理器中，在输入处理器处理完后生成结果放在 Item Loader 里面 (这时候没有赋值给 item)
2. xpath2 数据被提取出来，然后传输给 (1) 中同样的输入处理器，因为它们都是 name 字段的处理器，然后处理结果被附加到 (1) 的结果后面
3. 跟 2 一样
4. 跟 3 一样，不过这次是直接的字面字符串值，先转换成一个单元素的可迭代对象再传给输入处理器
5. 上面 4 步的数据被传输给 name 的输出处理器，将最终的结果赋值给 name 字段

5.6 自定义 Item Loader

使用类定义语法，下面是一个例子

```
from scrapy.loader import ItemLoader
from scrapy.loader.processors import TakeFirst, MapCompose, Join

class ProductLoader(ItemLoader):

    default_output_processor = TakeFirst()

    name_in = MapCompose(unicode.title)
    name_out = Join()

    price_in = MapCompose(unicode.strip)

    # ...
```

通过 _in 和 _out 后缀来定义输入和输出处理器，并且还可以定义默认的 ItemLoader.default_input_processor 和 ItemLoader.default_input_processor.

5.7 在 Field 定义中声明输入/输出处理器

还有个地方可以非常方便的添加输入/输出处理器，那就是直接在 Field 定义中

```
import scrapy
from scrapy.loader.processors import Join, MapCompose, TakeFirst
from w3lib.html import remove_tags

def filter_price(value):
    if value.isdigit():
        return value

class Product(scrapy.Item):
    name = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=Join(),
    )
    price = scrapy.Field(
        input_processor=MapCompose(remove_tags, filter_price),
        output_processor=TakeFirst(),
    )
```

优先级：

1. 在 Item Loader 中定义的 `field_in` 和 `field_out`
2. Filed 元数据 (`input_processor` 和 `output_processor` 关键字)
3. Item Loader 中的默认的

Tips：一般来讲，将输入处理器定义在 Item Loader 的定义中 `field_in`，然后将输出处理器定义在 Field 元数据中

5.8 Item Loader 上下文

Item Loader 上下文被所有输入/输出处理器共享，比如你有一个解析长度的函数

```
def parse_length(text, loader_context):
    unit = loader_context.get('unit', 'm')
    # ... length parsing code goes here ...
    return parsed_length
```

初始化和修改上下文的值

```
loader = ItemLoader(product)
loader.context['unit'] = 'cm'

loader = ItemLoader(product, unit='cm')

class ProductLoader(ItemLoader):
    length_out = MapCompose(parse_length, unit='cm')
```

5.9 内置的处理器

1. `Identity` 啥也不做
2. `TakeFirst` 返回第一个非空值，通常用作输出处理器
3. `Join` 将结果连起来，默认使用空格' '
4. `Compose` 将函数链接起来形成管道流，产生最后的输出
5. `MapCompose` 跟上面的 `Compose` 类似，区别在于内部结果在函数中的传递方式. 它的输入值是可迭代的，首先将第一个函数依次作用于所有值，产生新的可迭代输入，作为第二个函数的输入，最后生成的结果连起来返回最终值，一般用在输入处理器中。
6. `SelectJmes` 使用 json 路径来查询值并返回结果

6 Scrapy 教程 06- Item Pipeline

当一个 item 被蜘蛛爬取到之后会被发送给 Item Pipeline，然后多个组件按照顺序处理这个 item。每个 Item Pipeline 组件其实就是一个实现了一个简单方法的 Python 类。他们接受一个 item 并在上面执行逻辑，还能决定这个 item 到底是否还要继续往下传输，如果不要了就直接丢弃。

使用 Item Pipeline 的常用场景：

- 清理 HTML 数据
- 验证被抓取的数据 (检查 item 是否包含某些字段)
- 重复性检查 (然后丢弃)
- 将抓取的数据存储到数据库中

6.1 编写自己的 Pipeline

定义一个 Python 类，然后实现方法 `process_item(self, item, spider)` 即可，返回一个字典或 Item，或者抛出 `DropItem` 异常丢弃这个 Item。

或者还可以实现下面几个方法：

- `open_spider(self, spider)` 蜘蛛打开的时执行
- `close_spider(self, spider)` 蜘蛛关闭时执行
- `from_crawler(cls, crawler)` 可访问核心组件比如配置和信号，并注册钩子函数到 Scrapy 中

6.2 Item Pipeline 示例

价格验证

我们通过一个价格验证例子来看看怎样使用

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

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

将 item 写入 json 文件

下面的这个 Pipeline 将所有的 item 写入到一个单独的 json 文件，一行一个 item

```
import json

class JsonWriterPipeline(object):

    def __init__(self):
        self.file = open('items.jl', 'wb')

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

将 item 存储到 MongoDB 中

这个例子使用pymongo来演示怎样讲 item 保存到 MongoDB 中。MongoDB 的地址和数据库名在配置中指定，这个例子主要是向你展示怎样使用 `from_crawler()` 方法，以及如何清理资源。

```
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(dict(item))
        return item
```

重复过滤器

假设我们的 item 里面的 id 字典是唯一的，但是我们的蜘蛛返回了多个相同 id 的 item

```
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
```

6.3 激活一个 Item Pipeline 组件

你必须在配置文件中将你需要激活的 Pipline 组件添加到 ITEM_PIPELINES 中

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

后面的数字表示它的执行顺序，从低到高执行，范围 0-1000

6.4 Feed exports

这里顺便提下 Feed exports，一般有的爬虫直接将爬取结果序列化到文件中，并保存到某个存储介质中。只需要在 settings 里面设置几个即可：

```
* FEED_FORMAT= json # json|jsonlines|csv|xml|pickle|marshal
* FEED_URI= file:///tmp/export.csv|ftp://user:pass@ftp.example.com/path/to/
↪export.csv|s3://aws_key:aws_secret@mybucket/path/to/export.csv|stdout:
* FEED_EXPORT_FIELDS = ["foo", "bar", "baz"] # 这个在导出 csv 的时候有用
```

6.5 请求和响应

Scrapy 使用 Request 和 Response 对象来爬取网站。Request 对象被蜘蛛生成，然后被传递给下载器，之后下载器处理这个 Request 后返回 Response 对象，然后返回给生成 Request 的这个蜘蛛。

给回调函数传递额外的参数

Request 对象生成的时候会通过关键字参数 callback 指定回调函数，Response 对象被当做第一个参数传入，有时候我们想传递额外的参数，比如我们构建某个 Item 的时候，需要两步，第一步是链接属性，第二步是详情属性，可以指定 Request.meta

```
def parse_page1(self, response):
    item = MyItem()
    item['main_url'] = response.url
    request = scrapy.Request("http://www.example.com/some_page.html",
                             callback=self.parse_page2)
    request.meta['item'] = item
    return request

def parse_page2(self, response):
    item = response.meta['item']
    item['other_url'] = response.url
    return item
```

Request 子类

Scrapy 为各种不同的场景内置了很多 Request 子类，你还可以继承它自定义自己的请求类。

FormRequest 这个专门为 form 表单设计，模拟表单提交的示例

```
return [FormRequest(url="http://www.example.com/post/action",
                    formdata={'name': 'John Doe', 'age': '27'},
                    callback=self.after_post)]
```

我们再来一个例子模拟用户登录，使用了 FormRequest.from_response()

```
import scrapy

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

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

    def after_login(self, response):
        # check login succeed before going on
        if "authentication failed" in response.body:
            self.logger.error("Login failed")
            return

        # continue scraping with authenticated session...
```

Response 子类

一个 scrapy.http.Response 对象代表了一个 HTTP 相应，通常是被下载器下载后得到，并交给 Spider 做进一步的处理。Response 也有很多默认的子类，用于表示各种不同的响应类型。

- TextResponse 在基本 Response 类基础之上增加了编码功能，专门用于二进制数据比如图片、声音或其他媒体文件
- HtmlResponse 此类是 TextResponse 的子类，通过查询 HTML 的 meta http-equiv 属性实现了编码自动发现
- XmlResponse 此类是 TextResponse 的子类，通过查询 XML 声明实现编码自动发现

7 Scrapy 教程 07- 内置服务

Scrapy 使用 Python 内置的的日志系统来记录事件日志。日志配置

```
LOG_ENABLED = true
LOG_ENCODING = "utf-8"
LOG_LEVEL = logging.INFO
LOG_FILE = "log/spider.log"
LOG_STDOUT = True
LOG_FORMAT = "%(asctime)s [%(name)s] %(levelname)s: %(message)s"
LOG_DATEFORMAT = "%Y-%m-%d %H:%M:%S"
```

使用也很简单

```
import logging
logger = logging.getLogger(__name__)
logger.warning("This is a warning")
```

如果在 Spider 里面使用，那就更简单了，因为 logger 就是它的一个实例变量

```
import scrapy

class MySpider(scrapy.Spider):

    name = 'myspider'
    start_urls = ['http://scrapinghub.com']

    def parse(self, response):
        self.logger.info('Parse function called on %s', response.url)
```

7.1 发送 email

Scrapy 发送 email 基于Twisted non-blocking IO实现，只需几个简单配置即可。

初始化

```
mailer = MailSender.from_settings(settings)
```

发送不包含附件

```
mailer.send(to=["someone@example.com"], subject="Some subject", body="Some␣
↪body", cc=["another@example.com"])
```

配置

```
MAIL_FROM = 'scrapy@localhost'
MAIL_HOST = 'localhost'
MAIL_PORT = 25
MAIL_USER = ""
MAIL_PASS = ""
MAIL_TLS = False
MAIL_SSL = False
```

7.2 同一个进程运行多个 Spider

```
import scrapy
from scrapy.crawler import CrawlerProcess
```

```
class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):
    # Your second spider definition
    ...

process = CrawlerProcess()
process.crawl(MySpider1)
process.crawl(MySpider2)
process.start() # the script will block here until all crawling jobs are␣
↪finished
```

7.3 分布式爬虫

Scrapy 并没有提供内置的分布式抓取功能，不过有很多方法可以帮你实现。

如果你有很多个 spider，最简单的方式就是启动多个 Scrapyd 实例，然后将 spider 分布到各个机器上面。

如果你想多个机器运行同一个 spider，可以将 url 分片后交给每个机器上面的 spider。比如你把 URL 分成 3 份

```
http://somedomain.com/urls-to-crawl/spider1/part1.list
http://somedomain.com/urls-to-crawl/spider1/part2.list
http://somedomain.com/urls-to-crawl/spider1/part3.list
```

然后运行 3 个 Scrapyd 实例，分别启动它们，并传递 part 参数

```
curl http://scrapy1.mycompany.com:6800/schedule.json -d project=myproject -d␣
↪spider=spider1 -d part=1
curl http://scrapy2.mycompany.com:6800/schedule.json -d project=myproject -d␣
↪spider=spider1 -d part=2
curl http://scrapy3.mycompany.com:6800/schedule.json -d project=myproject -d␣
↪spider=spider1 -d part=3
```

7.4 防止被封的策略

一些网站实现了一些策略来禁止爬虫来爬取它们的网页。有的比较简单，有的相当复杂，如果你需要详细了解可以咨询商业支持

下面是对于这些网站的一些有用的建议：

- 使用 user agent 池。也就是每次发送的时候随机从池中选择不一样的浏览器头信息，防止暴露爬虫身份

- 禁止 Cookie，某些网站会通过 Cookie 识别用户身份，禁用后使得服务器无法识别爬虫轨迹
- 设置 download_delay 下载延迟，数字设置为 5 秒，越大越安全
- 如果有可能的话尽量使用Google cache获取网页，而不是直接访问
- 使用一个轮转 IP 池，例如免费的Tor project或者是付费的ProxyMesh
- 使用大型分布式下载器，这样就能完全避免被封了，只需要关注怎样解析页面就行。一个例子就是Crawlera

如果这些还是无法避免被禁，可以考虑商业支持

8 Scrapy 教程 08- 文件与图片

Scrapy 为我们提供了可重用的item pipelines为某个特定的 Item 去下载文件。通常来说你会选择使用 Files Pipeline 或 Images Pipeline。

这两个管道都实现了：

- 避免重复下载
- 可以指定下载后保存的地方 (文件系统目录中,Amazon S3 中)

Images Pipeline 为处理图片提供了额外的功能：

- 将所有下载的图片格式转换成普通的 JPG 并使用 RGB 颜色模式
- 生成缩略图
- 检查图片的宽度和高度确保它们满足最小的尺寸限制

管道同时会在内部保存一个被调度下载的 URL 列表，然后将包含相同媒体的相应关联到这个队列上来，从而防止了多个 item 共享这个媒体时重复下载。

8.1 使用 Files Pipeline

一般我们会按照下面的步骤来使用文件管道：

1. 在某个 Spider 中，你爬取一个 item 后，将相应的文件 URL 放入 `file_urls` 字段中
2. item 被返回之后就会转交给 item pipeline
3. 当这个 item 到达 `FilesPipeline` 时，在 `file_urls` 字段中的 URL 列表会通过标准的 Scrapy 调度器和下载器来调度下载，并且优先级很高，在抓取其他页面前就被处理。而这个 `item` 会一直在这个 pipeline 中被锁定，直到所有的文件下载完成。
4. 当文件被下载完之后，结果会被赋值给另一个 `files` 字段。这个字段包含一个关于下载文件新的字典列表，比如下载路径，源地址，文件校验码。`files` 里面的顺序和 `file_url` 顺序是一致的。要是某个写文件下载出错就不会出现在这个 `files` 中了。

8.2 使用 Images Pipeline

ImagesPipeline 跟 FilesPipeline 的使用差不多，不过使用的字段名不一样，image_urls 保存图片 URL 地址，images 保存下载后的图片信息。

使用 ImagesPipeline 的好处是你可以通过配置来提供额外的功能，比如生成文件缩略图，通过图片大小过滤需要下载的图片等。

ImagesPipeline 使用Pillow来生成缩略图以及转换成标准的 JPEG/RGB 格式。因此你需要安装这个包，我们建议你使用 Pillow 而不是 PIL。

8.3 使用例子

要使用媒体管道，请先在配置文件中打开它

```
# 同时使用图片和文件管道
ITEM_PIPELINES = {
                  'scrapy.pipelines.images.ImagesPipeline': 1,
                  'scrapy.pipelines.files.FilesPipeline': 2,
                 }
FILES_STORE = '/path/to/valid/dir'  # 文件存储路径
IMAGES_STORE = '/path/to/valid/dir' # 图片存储路径
# 90 days of delay for files expiration
FILES_EXPIRES = 90
# 30 days of delay for images expiration
IMAGES_EXPIRES = 30
# 图片缩略图
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (270, 270),
}
# 图片过滤器，最小高度和宽度
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

一个使用了缩略图的下载例子会生成如下图片：

```
<IMAGES_STORE>/full/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/small/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/big/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
```

然后，某个 Item 返回时，有 file_urls 或 image_urls，并且存在相应的 files 或 images 字段

```
import scrapy

class MyItem(scrapy.Item):

    # ... other item fields ...
```

```
    image_urls = scrapy.Field()
    images = scrapy.Field()
```

8.4 自定义媒体管道

如果你还需要更加复杂的功能，想自定义下载媒体逻辑，请参考扩展媒体管道

不管是扩展 FilesPipeline 还是 ImagesPipeline, 都只需重写下面两个方法

- get_media_requests(self, item, info), 返回一个 Request 对象
- item_completed(self, results, item, info), 当上门的 Request 下载完成后回调这个方法，然后填充 files 或 images 字段

下面是一个扩展 ImagesPipeline 的例子，我只取 path 信息，并将它赋给 image_paths 字段，而不是默认的 images

```python
import scrapy
from scrapy.pipelines.images import ImagesPipeline
from scrapy.exceptions import DropItem

class MyImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield scrapy.Request(image_url)

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
            raise DropItem("Item contains no images")
        item['image_paths'] = image_paths
        return item
```

9 Scrapy 教程 09- 部署

本篇主要介绍两种部署爬虫的方案。如果仅仅在开发调试的时候在本地部署跑起来是很容易的，不过要是生产环境，爬虫任务量大，并且持续时间长，那么还是建议使用专业的部署方法。主要是两种方案：

- Scrapyd 开源方案
- Scrapy Cloud 云方案

9.1 部署到 Scrapyd

Scrapyd是一个开源软件，用来运行蜘蛛爬虫。它提供了 HTTP API 的服务器，还能运

行和监控 Scrapy 的蜘蛛

要部署爬虫到 Scrapyd，需要使用到scrapyd-client部署工具集，下面我演示下部署的步骤

Scrapyd 通常以守护进程 daemon 形式运行，监听 spider 的请求，然后为每个 spider 创建一个进程执行 `scrapy crawl myspider`, 同时 Scrapyd 还能以多进程方式启动，通过配置 `max_proc` 和 `max_proc_per_cpu` 选项

安装

使用 pip 安装

```
pip install scrapyd
```

在 ubuntu 系统上面

```
apt-get install scrapyd
```

配置

配置文件地址，优先级从低到高

- /etc/scrapyd/scrapyd.conf (Unix)
- /etc/scrapyd/conf.d/* (in alphabetical order, Unix)
- scrapyd.conf
- ~/.scrapyd.conf (users home directory)

具体参数参考scrapyd 配置

简单的例子

```
[scrapyd]
eggs_dir    = eggs
logs_dir    = logs
items_dir   =
jobs_to_keep = 5
dbs_dir     = dbs
max_proc    = 0
max_proc_per_cpu = 4
finished_to_keep = 100
poll_interval = 5
bind_address = 0.0.0.0
http_port   = 6800
debug       = off
runner      = scrapyd.runner
application = scrapyd.app.application
launcher    = scrapyd.launcher.Launcher
webroot     = scrapyd.website.Root
```

```
[services]
schedule.json     = scrapyd.webservice.Schedule
cancel.json       = scrapyd.webservice.Cancel
addversion.json   = scrapyd.webservice.AddVersion
listprojects.json = scrapyd.webservice.ListProjects
listversions.json = scrapyd.webservice.ListVersions
listspiders.json  = scrapyd.webservice.ListSpiders
delproject.json   = scrapyd.webservice.DeleteProject
delversion.json   = scrapyd.webservice.DeleteVersion
listjobs.json     = scrapyd.webservice.ListJobs
daemonstatus.json = scrapyd.webservice.DaemonStatus
```

部署

使用scrapyd-client最方便，Scrapyd-client 是scrapyd的一个客户端，它提供了 scrapyd-deploy 工具将工程部署到 Scrapyd 服务器上面

通常将你的工程部署到 Scrapyd 需要两个步骤：

1. 将工程打包成 python 蛋，你需要安装setuptools
2. 通过addversion.json终端将蟒蛇蛋上传至 Scrapd 服务器

你可以在你的工程配置文件 scrapy.cfg 定义 Scrapyd 目标

```
[deploy:example]
url = http://scrapyd.example.com/api/scrapyd
username = scrapy
password = secret
```

列出所有可用目标使用命令

```
scrapyd-deploy -l
```

列出某个目标上面所有可运行的工程，执行命令

```
scrapyd-deploy -L example
```

先 cd 到工程根目录，然后使用如下命令来部署：

```
scrapyd-deploy <target> -p <project>
```

你还可以定义默认的 target 和 project，省的你每次都去敲代码

```
[deploy]
url = http://scrapyd.example.com/api/scrapyd
username = scrapy
password = secret
project = yourproject
```

这样你就直接取执行

```
scrapyd-deploy
```

如果你有多个 target，那么可以使用下面命令将 project 部署到多个 target 服务器上面

```
scrapyd-deploy -a -p <project>
```

9.2 部署到 Scrapy Cloud

Scrapy Cloud是一个托管的云服务器，由 Scrapy 背后的公司Scrapinghub维护

它免除了安装和监控服务器的需要，并提供了非常美观的 UI 来管理各个 Spider，还能查看被抓取的 Item，日志和状态等。

你可以使用shub命令行工具来讲 spider 部署到 Scrapy Cloud。更多请参考官方文档

Scrapy Cloud 和 Scrapyd 是兼容的，你可以根据需要在两者之前切换，配置文件也是 `scrapy.cfg`，跟 `scrapyd-deploy` 读取的是一样的。

10 Scrapy 教程 10- 动态配置爬虫

有很多时候我们需要从多个网站爬取所需要的数据，比如我们想爬取多个网站的新闻，将其存储到数据库同一个表中。我们是不是要对每个网站都得去定义一个 Spider 类呢？其实不需要，我们可以通过维护一个规则配置表或者一个规则配置文件来动态增加或修改爬取规则，然后程序代码不需要更改就能实现多个网站爬取。

要这样做，我们就不能再使用前面的 `scrapy crawl test` 这种命令了，我们需要使用编程的方式运行 Scrapy spider，参考官方文档

10.1 脚本运行 Scrapy

可以利用 scrapy 提供的核心 API通过编程方式启动 scrapy，代替传统的 `scrapy crawl` 启动方式。

Scrapy 构建于 Twisted 异步网络框架基础之上，因此你需要在 Twisted reactor 里面运行。

首先你可以使用 `scrapy.crawler.CrawlerProcess` 这个类来运行你的 spider，这个类会为你启动一个 Twisted reactor，并能配置你的日志和 shutdown 处理器。所有的 scrapy 命令都使用这个类。

```
import scrapy
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings

process = CrawlerProcess(get_project_settings())
```

```
process.crawl(MySpider)
process.start() # the script will block here until the crawling is finished
```

然后你就可以直接执行这个脚本

```
python run.py
```

另外一个功能更强大的类是 scrapy.crawler.CrawlerRunner，推荐你使用这个

```
from twisted.internet import reactor
import scrapy
from scrapy.crawler import CrawlerRunner
from scrapy.utils.log import configure_logging

class MySpider(scrapy.Spider):
    # Your spider definition
    ...

configure_logging({'LOG_FORMAT': '%(levelname)s: %(message)s'})
runner = CrawlerRunner()

d = runner.crawl(MySpider)
d.addBoth(lambda _: reactor.stop())
reactor.run() # the script will block here until the crawling is finished
```

10.2 同一进程运行多个 spider

默认情况当你每次执行 scrapy crawl 命令时会创建一个新的进程。但我们可以使用核心 API 在同一个进程中同时运行多个 spider

```
import scrapy
from twisted.internet import reactor
from scrapy.crawler import CrawlerRunner
from scrapy.utils.log import configure_logging

class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):
    # Your second spider definition
    ...

configure_logging()
runner = CrawlerRunner()
runner.crawl(MySpider1)
runner.crawl(MySpider2)
d = runner.join()
```

```
d.addBoth(lambda _: reactor.stop())

reactor.run() # the script will block here until all crawling jobs are␣
↪finished
```

10.3 定义规则表

好了言归正传，有了前面的脚本启动基础，就可以开始我们的动态配置爬虫了。我们的需求是这样的，从两个不同的网站爬取我们所需要的新闻文章，然后存储到 article 表中。

首先我们需要定义规则表和文章表，通过动态的创建蜘蛛类，我们以后就只需要维护规则表即可了。这里我使用 SQLAlchemy 框架来映射数据库。

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 定义数据库模型实体
Desc :
"""
import datetime

from sqlalchemy.engine.url import URL
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import create_engine, Column, Integer, String, Text, DateTime
from coolscrapy.settings import DATABASE

Base = declarative_base()

class ArticleRule(Base):
    """ 自定义文章爬取规则"""
    __tablename__ = 'article_rule'

    id = Column(Integer, primary_key=True)
    # 规则名称
    name = Column(String(30))
    # 运行的域名列表，逗号隔开
    allow_domains = Column(String(100))
    # 开始 URL 列表，逗号隔开
    start_urls = Column(String(100))
    # 下一页的 xpath
    next_page = Column(String(100))
    # 文章链接正则表达式（子串）
    allow_url = Column(String(200))
    # 文章链接提取区域 xpath
    extract_from = Column(String(200))
    # 文章标题 xpath
    title_xpath = Column(String(100))
    # 文章内容 xpath
```

```
    body_xpath = Column(Text)
    # 发布时间 xpath
    publish_time_xpath = Column(String(30))
    # 文章来源
    source_site = Column(String(30))
    # 规则是否生效
    enable = Column(Integer)


class Article(Base):
    """ 文章类"""
    __tablename__ = 'articles'

    id = Column(Integer, primary_key=True)
    url = Column(String(100))
    title = Column(String(100))
    body = Column(Text)
    publish_time = Column(String(30))
    source_site = Column(String(30))
```

10.4 定义文章 Item

这个很简单了，没什么需要说明的

```
import scrapy


class Article(scrapy.Item):
    title = scrapy.Field()
    url = scrapy.Field()
    body = scrapy.Field()
    publish_time = scrapy.Field()
    source_site = scrapy.Field()
```

10.5 定义 ArticleSpider

接下来我们将定义爬取文章的蜘蛛，这个 spider 会使用一个 Rule 实例来初始化，然后根据 Rule 实例中的 xpath 规则来获取相应的数据。

```
from coolscrapy.utils import parse_text
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor
from coolscrapy.items import Article


class ArticleSpider(CrawlSpider):
    name = "article"
```

```
    def __init__(self, rule):
        self.rule = rule
        self.name = rule.name
        self.allowed_domains = rule.allow_domains.split(",")
        self.start_urls = rule.start_urls.split(",")
        rule_list = []
        # 添加 `下一页 `的规则
        if rule.next_page:
            rule_list.append(Rule(LinkExtractor(restrict_xpaths=rule.next_
↪page)))
        # 添加抽取文章链接的规则
        rule_list.append(Rule(LinkExtractor(
            allow=[rule.allow_url],
            restrict_xpaths=[rule.extract_from]),
            callback='parse_item'))
        self.rules = tuple(rule_list)
        super(ArticleSpider, self).__init__()

    def parse_item(self, response):
        self.log('Hi, this is an article page! %s' % response.url)

        article = Article()
        article["url"] = response.url

        title = response.xpath(self.rule.title_xpath).extract()
        article["title"] = parse_text(title, self.rule.name, 'title')

        body = response.xpath(self.rule.body_xpath).extract()
        article["body"] = parse_text(body, self.rule.name, 'body')

        publish_time = response.xpath(self.rule.publish_time_xpath).extract()
        article["publish_time"] = parse_text(publish_time, self.rule.name,
↪'publish_time')

        article["source_site"] = self.rule.source_site

        return article
```

要注意的是 start_urls，rules 等都初始化成了对象的属性，都由传入的 rule 对象初始化，parse_item 方法中的抽取规则也都有 rule 对象提供。

10.6 编写 pipeline 存储到数据库中

我们还是使用 SQLAlchemy 来将文章 Item 数据存储到数据库中

```
@contextmanager
def session_scope(Session):
    """Provide a transactional scope around a series of operations."""
```

```
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
        session.close()


class ArticleDataBasePipeline(object):
    """ 保存文章到数据库"""

    def __init__(self):
        engine = db_connect()
        create_news_table(engine)
        self.Session = sessionmaker(bind=engine)

    def open_spider(self, spider):
        """This method is called when the spider is opened."""
        pass

    def process_item(self, item, spider):
        a = Article(url=item["url"],
                    title=item["title"].encode("utf-8"),
                    publish_time=item["publish_time"].encode("utf-8"),
                    body=item["body"].encode("utf-8"),
                    source_site=item["source_site"].encode("utf-8"))
        with session_scope(self.Session) as session:
            session.add(a)

    def close_spider(self, spider):
        pass
```

10.7 修改 run.py 启动脚本

我们将上面的 run.py 稍作修改即可定制我们的文章爬虫启动脚本

```
import logging
from spiders.article_spider import ArticleSpider
from twisted.internet import reactor
from scrapy.crawler import CrawlerRunner
from scrapy.utils.project import get_project_settings
from scrapy.utils.log import configure_logging
from coolscrapy.models import db_connect
from coolscrapy.models import ArticleRule
from sqlalchemy.orm import sessionmaker
```

```
if __name__ == '__main__':
    settings = get_project_settings()
    configure_logging(settings)
    db = db_connect()
    Session = sessionmaker(bind=db)
    session = Session()
    rules = session.query(ArticleRule).filter(ArticleRule.enable == 1).all()
    session.close()
    runner = CrawlerRunner(settings)

    for rule in rules:
        # stop reactor when spider closes
        # runner.signals.connect(spider_closing, signal=signals.spider_closed)
        runner.crawl(ArticleSpider, rule=rule)

    d = runner.join()
    d.addBoth(lambda _: reactor.stop())

    # blocks process so always keep as the last statement
    reactor.run()
    logging.info('all finished.')
```

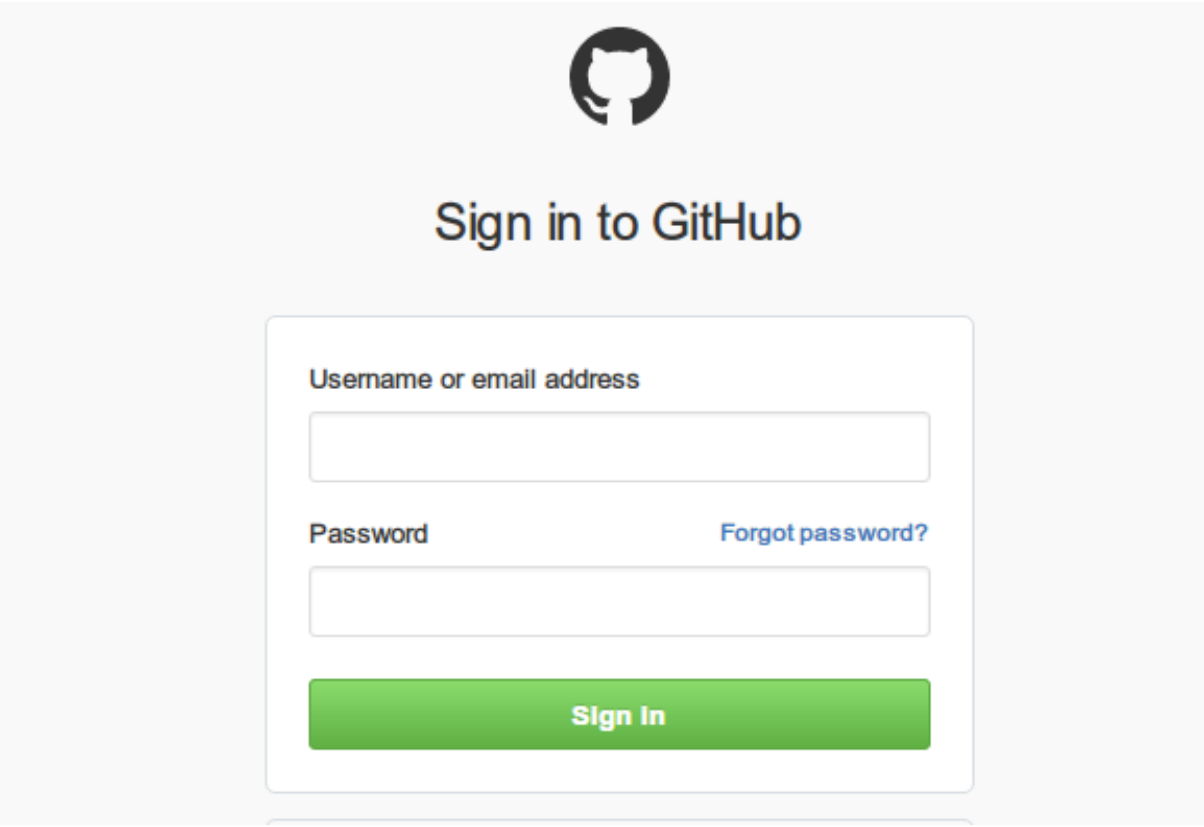
OK，一切搞定。现在我们可以往 ArticleRule 表中加入成百上千个网站的规则，而不用添加一行代码，就可以对这成百上千个网站进行爬取。当然你完全可以做一个 Web 前端来完成维护 ArticleRule 表的任务。当然 ArticleRule 规则也可以放在除了数据库的任何地方，比如配置文件。

你可以在GitHub上看到本文的完整项目源码。

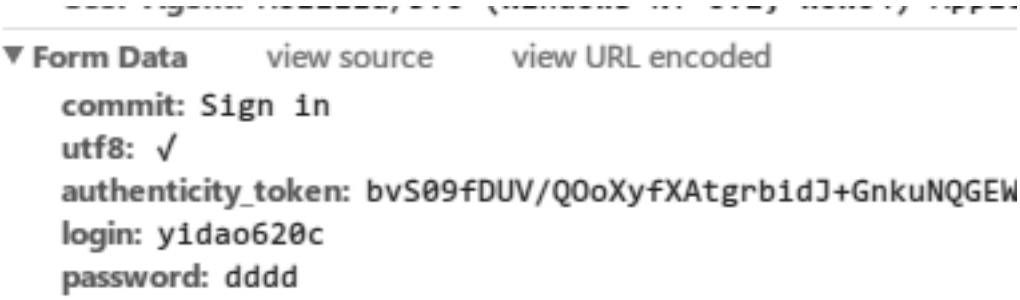
11 Scrapy 教程 11- 模拟登录

有时候爬取网站的时候需要登录，在 Scrapy 中可以通过模拟登录保存 cookie 后再去爬取相应的页面。这里我通过登录 github 然后爬取自己的 issue 列表来演示下整个原理。

要想实现登录就需要表单提交，先通过浏览器访问 github 的登录页面https://github.com/login，然后使用浏览器调试工具来得到登录时需要提交什么东西:



我这里使用 chrome 浏览器的调试工具，F12 打开后选择 Network，并将 Preserve log 勾上。我故意输入错误的用户名和密码，得到它提交的 form 表单参数还有 POST 提交的 URL:



去查看 html 源码会发现表单里面有个隐藏的 `authenticity_token` 值，这个是需要先获取然后跟用户名和密码一起提交的:

11.1 重写 start_requests 方法

要使用 cookie，第一步得打开它呀，默认 scrapy 使用 CookiesMiddleware 中间件，并且打开了。如果你之前禁止过，请设置如下

```
COOKIES_ENABLES = True
```

我们先要打开登录页面，获取 authenticity_token 值，这里我重写了 start_requests 方法

```
# 重写了爬虫类的方法，实现了自定义请求，运行成功后会调用 callback 回调函数
def start_requests(self):
    return [Request("https://github.com/login",
                    meta={'cookiejar': 1}, callback=self.post_login)]

# FormRequeset
def post_login(self, response):
    # 先去拿隐藏的表单参数 authenticity_token
    authenticity_token = response.xpath(
        '//input[@name="authenticity_token"]/@value').extract_first()
    logging.info('authenticity_token=' + authenticity_token)
    pass
```

start_requests 方法指定了回调函数，用来获取隐藏表单值 authenticity_token，同时我们还给 Request 指定了 cookiejar 的元数据，用来往回调函数传递 cookie 标识。

11.2 使用 FormRequest

Scrapy 为我们准备了 FormRequest 类专门用来进行 Form 表单提交的

```
# 为了模拟浏览器，我们定义 httpheader
post_headers = {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
↪webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "zh-CN,zh;q=0.8,en;q=0.6",
    "Cache-Control": "no-cache",
    "Connection": "keep-alive",
    "Content-Type": "application/x-www-form-urlencoded",
    "User-Agent": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/49.0.2623.75 Safari/537.36",
    "Referer": "https://github.com/",
}
# 使用 FormRequeset 模拟表单提交
def post_login(self, response):
    # 先去拿隐藏的表单参数 authenticity_token
    authenticity_token = response.xpath(
        '//input[@name="authenticity_token"]/@value').extract_first()
    logging.info('authenticity_token=' + authenticity_token)
```

```
    # FormRequeset.from_response 是 Scrapy 提供的一个函数，用于 post 表单
    # 登陆成功后，会调用 after_login 回调函数，如果 url 跟 Request 页面的一样就省略掉
    return [FormRequest.from_response(response,
                                      url='https://github.com/session',
                                      meta={'cookiejar': response.meta[
↪'cookiejar']},
                                      headers=self.post_headers,  # 注意此处的
headers
                                      formdata={
                                          'utf8': '✓',
                                          'login': 'yidao620c',
                                          'password': '******',
                                          'authenticity_token': authenticity_
↪token
                                      },
                                      callback=self.after_login,
                                      dont_filter=True
                                      )]

def after_login(self, response):
    pass
```

FormRequest.from_response() 方法让你指定提交的 url，请求头还有 form 表单值，注意我们还通过 meta 传递了 cookie 标识。它同样有个回调函数，登录成功后调用。下面我们来实现它

```
def after_login(self, response):
    # 登录之后，开始进入我要爬取的私信页面
    for url in self.start_urls:
        # 因为我们上面定义了 Rule，所以只需要简单的生成初始爬取 Request 即可
        yield Request(url, meta={'cookiejar': response.meta['cookiejar']})
```

这里我通过 start_urls 定义了开始页面，然后生成 Request，具体爬取的规则和下一页规则在前面的 Rule 里定义了。注意这里我继续传递 cookiejar，访问初始页面时带上 cookie 信息。

11.3 重写 _requests_to_follow

有个问题刚开始困扰我很久就是这里我定义的 spider 继承自 CrawlSpider，它内部自动去下载匹配的链接，而每次去访问链接的时候并没有自动带上 cookie，后来我重写了它的 _requests_to_follow() 方法解决了这个问题

```
def _requests_to_follow(self, response):
    """ 重写加入 cookiejar 的更新"""
    if not isinstance(response, HtmlResponse):
        return
    seen = set()
    for n, rule in enumerate(self._rules):
        links = [l for l in rule.link_extractor.extract_links(response) if l␣
↪not in seen]
```

```
        if links and rule.process_links:
            links = rule.process_links(links)
        for link in links:
            seen.add(link)
            r = Request(url=link.url, callback=self._response_downloaded)
            # 下面这句是我重写的
            r.meta.update(rule=n, link_text=link.text, cookiejar=response.
↪meta['cookiejar'])
            yield rule.process_request(r)
```

11.4 页面处理方法

在规则 Rule 里面我定义了每个链接的回调函数 parse_page，就是最终我们处理每个 issue 页面提取信息的逻辑

```
def parse_page(self, response):
    """ 这个是使用 LinkExtractor 自动处理链接以及 `下一页 `"""
    logging.info(u'---------------消息分割线-----------------')
    logging.info(response.url)
    issue_title = response.xpath(
        '//span[@class="js-issue-title"]/text()').extract_first()
    logging.info(u'issue_title: ' + issue_title.encode('utf-8'))
```

11.5 完整源码

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""
Topic: 登录爬虫
Desc : 模拟登录 https://github.com 后将自己的 issue 全部爬出来
tips: 使用 chrome 调试 post 表单的时候勾选 Preserve log 和 Disable cache
"""
import logging
import re
import sys
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor
from scrapy.http import Request, FormRequest, HtmlResponse

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(filename)s[line:%(lineno)d]
↪%(levelname)s %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S',
                    handlers=[logging.StreamHandler(sys.stdout)])
```

```
class GithubSpider(CrawlSpider):
    name = "github"
    allowed_domains = ["github.com"]
    start_urls = [
        'https://github.com/issues',
    ]
    rules = (
        # 消息列表
        Rule(LinkExtractor(allow=('/issues/\d+',),
                           restrict_xpaths='//ul[starts-with(@class, "table-
↪list")]/li/div[2]/a[2]'),
             callback='parse_page'),
        # 下一页, If callback is None follow defaults to True, otherwise it␣
↪defaults to False
        Rule(LinkExtractor(restrict_xpaths='//a[@class="next_page"]')),
    )
    post_headers = {
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
↪image/webp,*/*;q=0.8",
        "Accept-Encoding": "gzip, deflate",
        "Accept-Language": "zh-CN,zh;q=0.8,en;q=0.6",
        "Cache-Control": "no-cache",
        "Connection": "keep-alive",
        "Content-Type": "application/x-www-form-urlencoded",
        "User-Agent": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/49.0.2623.75 Safari/537.36",
        "Referer": "https://github.com/",
    }

    # 重写了爬虫类的方法, 实现了自定义请求, 运行成功后会调用 callback 回调函数
    def start_requests(self):
        return [Request("https://github.com/login",
                        meta={'cookiejar': 1}, callback=self.post_login)]

    # FormRequeset
    def post_login(self, response):
        # 先去拿隐藏的表单参数 authenticity_token
        authenticity_token = response.xpath(
            '//input[@name="authenticity_token"]/@value').extract_first()
        logging.info('authenticity_token=' + authenticity_token)
        # FormRequeset.from_response 是 Scrapy 提供的一个函数, 用于 post 表单
        # 登陆成功后, 会调用 after_login 回调函数, 如果 url 跟 Request 页面的一样就省
略掉
        return [FormRequest.from_response(response,
                                          url='https://github.com/session',
                                          meta={'cookiejar': response.meta[
↪'cookiejar']},
                                          headers=self.post_headers,  # 注意此
处的 headers
                                          formdata={
```

```
                                            'utf8': '✓',
                                            'login': 'yidao620c',
                                            'password': '******',
                                            'authenticity_token':␣
↪authenticity_token
                                        },
                                        callback=self.after_login,
                                        dont_filter=True
                                        )]

    def after_login(self, response):
        for url in self.start_urls:
            # 因为我们上面定义了 Rule，所以只需要简单的生成初始爬取 Request 即可
            yield Request(url, meta={'cookiejar': response.meta['cookiejar']})

    def parse_page(self, response):
        """ 这个是使用 LinkExtractor 自动处理链接以及 `下一页 `"""
        logging.info(u'--------------消息分割线-----------------')
        logging.info(response.url)
        issue_title = response.xpath(
            '//span[@class="js-issue-title"]/text()').extract_first()
        logging.info(u'issue_title: ' + issue_title.encode('utf-8'))

    def _requests_to_follow(self, response):
        """ 重写加入 cookiejar 的更新"""
        if not isinstance(response, HtmlResponse):
            return
        seen = set()
        for n, rule in enumerate(self._rules):
            links = [l for l in rule.link_extractor.extract_links(response)␣
↪if l not in seen]
            if links and rule.process_links:
                links = rule.process_links(links)
            for link in links:
                seen.add(link)
                r = Request(url=link.url, callback=self._response_downloaded)
                # 下面这句是我重写的
                r.meta.update(rule=n, link_text=link.text, cookiejar=response.
↪meta['cookiejar'])
                yield rule.process_request(r)
```

你可以在GitHub上看到本文的完整项目源码，还有另外一个自动登陆 iteye 网站的例子。

12 Scrapy 教程 12- 抓取动态网站

前面我们介绍的都是去抓取静态的网站页面，也就是说我们打开某个链接，它的内容全部呈现出来。但是如今的互联网大部分的 web 页面都是动态的，经常逛的网站例如京东、淘宝等，商品列表都是 js，并有 Ajax 渲染，下载某个链接得到的页面里面含有异步加载的内容，这样再使用之前的方式我们根本获取不到异步加载的这些网页内容。

使用 Javascript 渲染和处理网页是种非常常见的做法，如何处理一个大量使用 Javascript 的页面是 Scrapy 爬虫开发中一个常见的问题，这篇文章将说明如何在 Scrapy 爬虫中使用scrapy-splash来处理页面中得 Javascript。

12.1 scrapy-splash 简介

scrapy-splash 利用Splash将 javascript 和 Scrapy 集成起来，使得 Scrapy 可以抓取动态网页。

Splash 是一个 javascript 渲染服务，是实现了 HTTP API 的轻量级浏览器，底层基于 Twisted 和 QT 框架，Python 语言编写。所以首先你得安装 Splash 实例

12.2 安装 docker

官网建议使用 docker 容器安装方式 Splash。那么首先你得先安装 docker

参考官方安装文档，这里我选择 Ubuntu 12.04 LTS 版本安装

升级内核版本，docker 需要 3.13 内核

```
$ sudo apt-get update
$ sudo apt-get install linux-image-generic-lts-trusty
$ sudo reboot
```

安装 CA 认证

```
$ sudo apt-get install apt-transport-https ca-certificates
```

增加新的 GPGkey

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-
↪keys 58118E89F3A912897C070ADBF76221572C52609D
```

打开/etc/apt/sources.list.d/docker.list，如果没有就创建一个，然后删除任何已存在的内容，再增加下面一句

```
deb https://apt.dockerproject.org/repo ubuntu-precise main
```

更新 APT

```
$ sudo apt-get update
$ sudo apt-get purge lxc-docker
$ apt-cache policy docker-engine
```

安装

```
$ sudo apt-get install docker-engine
```

启动 docker 服务

```
$ sudo service docker start
```

验证是否启动成功

```
$ sudo docker run hello-world
```

上面这条命令会下载一个测试镜像并在容器中运行它，它会打印一个消息，然后退出。

12.3 安装 Splash

拉取镜像下来

```
$ sudo docker pull scrapinghub/splash
```

启动容器

```
$ sudo docker run -p 5023:5023 -p 8050:8050 -p 8051:8051 scrapinghub/splash
```

现在可以通过 0.0.0.0:8050(http),8051(https),5023 (telnet) 来访问 Splash 了。

12.4 安装 scrapy-splash

使用 pip 安装

```
$ pip install scrapy-splash
```

12.5 配置 scrapy-splash

在你的 scrapy 工程的配置文件 settings.py 中添加

```
SPLASH_URL = 'http://192.168.203.92:8050'
```

添加 Splash 中间件，还是在 settings.py 中通过 DOWNLOADER_MIDDLEWARES 指定，并且修改 HttpCompressionMiddleware 的优先级

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

默认情况下，HttpProxyMiddleware 的优先级是 750，要把它放在 Splash 中间件后面

设置 Splash 自己的去重过滤器

```
DUPEFILTER_CLASS = 'scrapy_splash.SplashAwareDupeFilter'
```

如果你使用 Splash 的 Http 缓存，那么还要指定一个自定义的缓存后台存储介质，scrapy-splash 提供了一个 scrapy.contrib.httpcache.FilesystemCacheStorage 的子类

```
HTTPCACHE_STORAGE = 'scrapy_splash.SplashAwareFSCacheStorage'
```

如果你要使用其他的缓存存储，那么需要继承这个类并且将所有的 scrapy.util.request.request_fingerprint 调用替换成 scrapy_splash.splash_request_fingerprint

12.6 使用 scrapy-splash

SplashRequest

最简单的渲染请求的方式是使用 scrapy_splash.SplashRequest，通常你应该选择使用这个

```
yield SplashRequest(url, self.parse_result,
    args={
        # optional; parameters passed to Splash HTTP API
        'wait': 0.5,

        # 'url' is prefilled from request url
        # 'http_method' is set to 'POST' for POST requests
        # 'body' is set to request body for POST requests
    },
    endpoint='render.json', # optional; default is render.html
    splash_url='<url>',     # optional; overrides SPLASH_URL
    slot_policy=scrapy_splash.SlotPolicy.PER_DOMAIN,  # optional
)
```

另外，你还可以在普通的 scrapy 请求中传递 splash 请求 meta 关键字达到同样的效果

```
yield scrapy.Request(url, self.parse_result, meta={
    'splash': {
        'args': {
```

```
            # set rendering arguments here
            'html': 1,
            'png': 1,

            # 'url' is prefilled from request url
            # 'http_method' is set to 'POST' for POST requests
            # 'body' is set to request body for POST requests
        },

        # optional parameters
        'endpoint': 'render.json',  # optional; default is render.json
        'splash_url': '<url>',      # optional; overrides SPLASH_URL
        'slot_policy': scrapy_splash.SlotPolicy.PER_DOMAIN,
        'splash_headers': {},       # optional; a dict with headers sent to␣
↪Splash
        'dont_process_response': True, # optional, default is False
        'dont_send_headers': True,  # optional, default is False
        'magic_response': False,    # optional, default is True
    }
})
```

Splash API 说明，使用 SplashRequest 是一个非常便利的工具来填充 request.meta['splash'] 里的数据

- meta['splash']['args'] 包含了发往 Splash 的参数。
- meta['splash']['endpoint'] 指定了 Splash 所使用的 endpoint，默认是render.html
- meta['splash']['splash_url'] 覆盖了 settings.py 文件中配置的 Splash URL
- meta['splash']['splash_headers'] 运行你增加或修改发往 Splash 服务器的 HTTP 头部信息，注意这个不是修改发往远程 web 站点的 HTTP 头部
- meta['splash']['dont_send_headers'] 如果你不想传递 headers 给 Splash，将它设置成 True
- meta['splash']['slot_policy'] 让你自定义 Splash 请求的同步设置
- meta['splash']['dont_process_response'] 当你设置成 True 后，SplashMiddleware 不会修改默认的 scrapy.Response 请求。默认是会返回 SplashResponse 子类响应比如 SplashTextResponse
- meta['splash']['magic_response'] 默认为 True，Splash 会自动设置 Response 的一些属性，比如 response.headers,response.body 等

如果你想通过 Splash 来提交 Form 请求，可以使用 scrapy_splash.SplashFormRequest，它跟 SplashRequest 使用是一样的。

Responses

对于不同的 Splash 请求，scrapy-splash 返回不同的 Response 子类

- SplashResponse 二进制响应，比如对/render.png 的响应

- SplashTextResponse 文本响应，比如对/render.html 的响应
- SplashJsonResponse JSON 响应，比如对/render.json 或使用 Lua 脚本的/execute 的响应

如果你只想使用标准的 Response 对象，就设置 `meta['splash']['dont_process_response']=True`

所有这些 Response 会把 `response.url` 设置成原始请求 URL(也就是你要渲染的页面 URL)，而不是 Splash endpoint 的 URL 地址。实际地址通过 `response.real_url` 得到

Session 的处理

Splash 本身是无状态的，那么为了支持 scrapy-splash 的 session 必须编写 Lua 脚本，使用/execute

```
function main(splash)
    splash:init_cookies(splash.args.cookies)

    -- ... your script

    return {
        cookies = splash:get_cookies(),
        -- ... other results, e.g. html
    }
end
```

而标准的 scrapy session 参数可以使用 `SplashRequest` 将 cookie 添加到当前 Splash cookiejar 中

12.7 使用实例

接下来我通过一个实际的例子来演示怎样使用，我选择爬取京东网首页的异步加载内容。

京东网打开首页的时候只会将导航菜单加载出来，其他具体首页内容都是异步加载的，下面有个”猜你喜欢”这个内容也是异步加载的，我现在就通过爬取这个”猜你喜欢”这四个字来说明下普通的 Scrapy 爬取和通过使用了 Splash 加载异步内容的区别。

首先我们写个简单的测试 Spider，不使用 splash：

```
class TestSpider(scrapy.Spider):
    name = "test"
    allowed_domains = ["jd.com"]
    start_urls = [
        "http://www.jd.com/"
    ]

    def parse(self, response):
        logging.info(u'---------我这个是简单的直接获取京东网首页测试---------')
```

```
        guessyou = response.xpath('//div[@id="guessyou"]/div[1]/h2/text()').
↪extract_first()
        logging.info(u"find: %s" % guessyou)
        logging.info(u'---------------success----------------')
```

然后运行结果：

```
2016-04-18 14:42:44 test_spider.py[line:20] INFO ---------我这个是简单的直接获取京
东网首页测试---------
2016-04-18 14:42:44 test_spider.py[line:22] INFO find: None
2016-04-18 14:42:44 test_spider.py[line:23] INFO ---------------success-------
↪---------
```

我找不到那个” 猜你喜欢” 这四个字

接下来我使用 splash 来爬取

```
import scrapy
from scrapy_splash import SplashRequest


class JsSpider(scrapy.Spider):
    name = "jd"
    allowed_domains = ["jd.com"]
    start_urls = [
        "http://www.jd.com/"
    ]

    def start_requests(self):
        splash_args = {
            'wait': 0.5,
        }
        for url in self.start_urls:
            yield SplashRequest(url, self.parse_result, endpoint='render.html
↪',
                                args=splash_args)

    def parse_result(self, response):
        logging.info(u'----------使用 splash 爬取京东网首页异步加载内容----------
↪')
        guessyou = response.xpath('//div[@id="guessyou"]/div[1]/h2/text()').
↪extract_first()
        logging.info(u"find: %s" % guessyou)
        logging.info(u'---------------success----------------')
```

运行结果：

```
2016-04-18 14:42:51 js_spider.py[line:36] INFO ----------使用 splash 爬取京东网首
页异步加载内容----------
2016-04-18 14:42:51 js_spider.py[line:38] INFO find: 猜你喜欢
2016-04-18 14:42:51 js_spider.py[line:39] INFO ---------------success---------
↪-------
```

可以看出结果里面已经找到了这个” 猜你喜欢”，说明异步加载内容爬取成功！

13 联系我

- Email：yidao620@gmail.com
- 博客：https://www.xncoding.com/
- GitHub：https://github.com/yidao620c

微信扫一扫，请博主喝杯咖啡