17 数据提取过程中的类型化方法

更新时间：2019-06-17 10:56:38

> 智慧，不是死的默念，而是生的沉思。
>
> ——斯宾诺莎

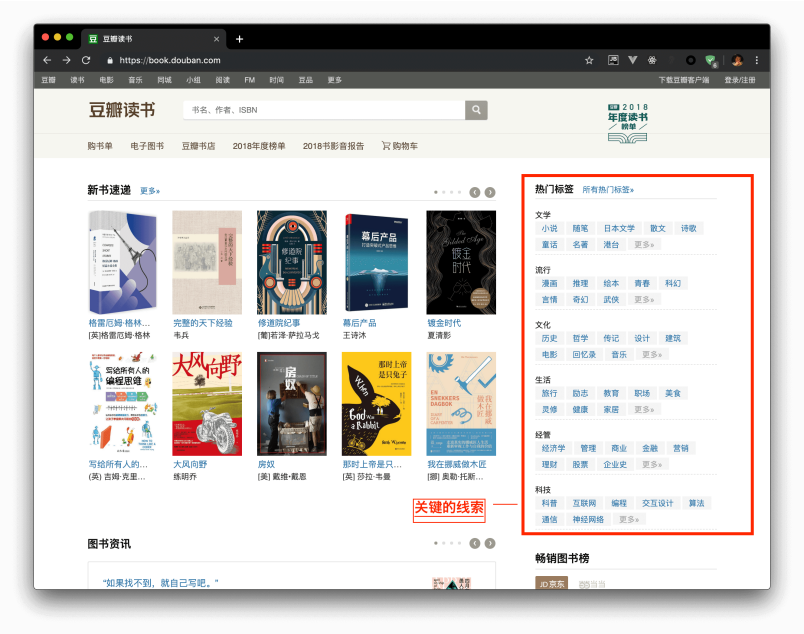
以爬取豆瓣读书中的藏书数据为例，页面图书数据中包含了多种的数据类型，我们往往得需要编写大量的代码将从页面提取的字符串数据转化为它们真正的数据类型，本节将介绍怎么简化提取页面时的这些类型转换代码，并使其可以被重用起来。

前面的章节中的爬虫示例则重于爬虫相关技术的讲解，旨在让各类的读者都能按照我的讲述节奏来动手实践。所选的示例可能会让你稍觉得有点缺乏实用性。故此，我选了一个我认为很具有实用价值的示例来作为本章的实践对象，那就是豆瓣读书爬虫。

我是个很喜欢阅读各类书籍的人，由其希望通过读书系统化地了解更多的新技术、新领域。然，我们这个时代最不缺乏的可能就是内容了，但我们最缺的却是时间，所以我总想在茫茫书海中选出那些最值得一读的书来看。那究竟什么是最“值“得一读呢？应该用什么关键性的数学指标去衡量这个"值"字呢？一次偶然的机会想起了豆瓣上有对书籍的评分机制，就细细地研究起了豆瓣读书。

豆瓣读书的网页分析与设计

豆瓣读书对各类的图书有具体的评分，可以很好地反映图书在读者心目中的受欢迎程度，具有很高的数据价值。



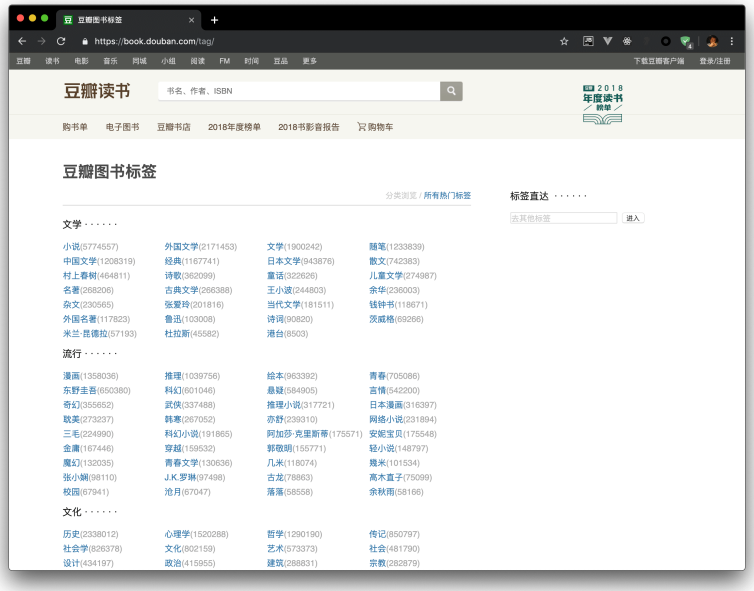
当我看到豆瓣读书主页右侧的这一方挤满标签的区域时，让我灵光一闪：

> 能不能以这个标签作为爬虫的"种子"页开始，将整个豆瓣上的图书都扒下来，然后在数据库里面将评分最高的数据筛选出来那不就找到那些最"值"得看的书了吗？

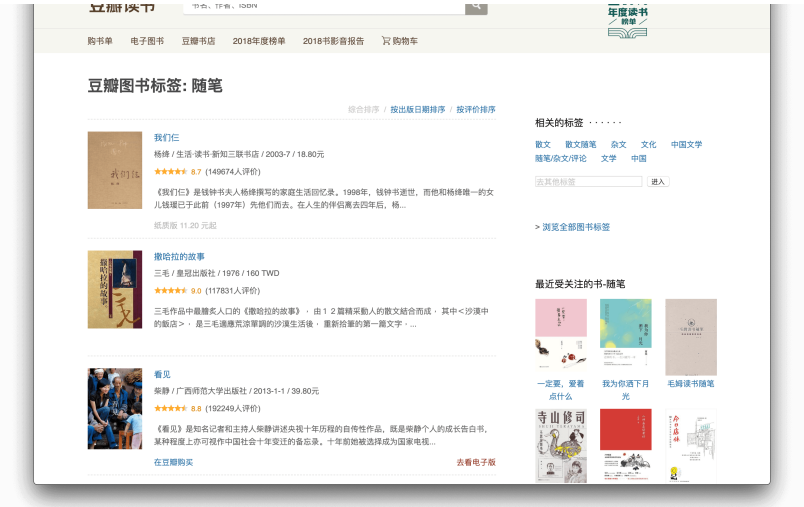
知识就是力量，懂爬虫的就可以数据为王了，说干就干开始由这个思路开始展开分析。

分析思路

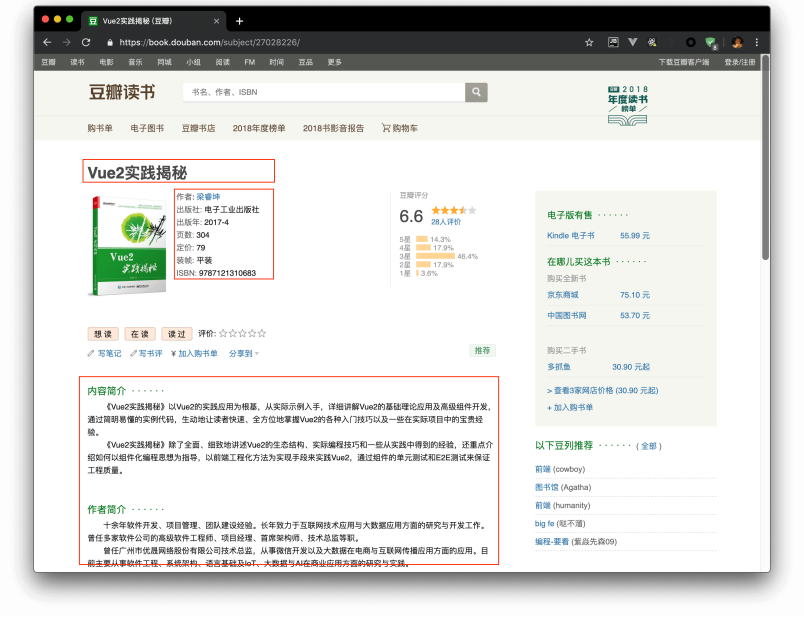
任何的爬虫开始之前一定是先考虑“种子”页在哪里这个首要问题，因为这是爬虫的开端，这在我之前的章节一直都在反复强调的一点。豆瓣读书除了首页上的那个推荐标签区域以外还有一个有一个非常好的种子来源，那就是豆瓣的标签页，如下图所示：



通过这个标签页可以搜索到各种图书的分类列表：



通过分类列表就能直达图书的详情页：



我将需要爬取的数据简单地框了一下。不难分析出上述三个页面的导航关系如下：

```
- 标签页 https://book.douban.com/tag/
    |- 分类列表页 https://book.douban.com/tag/[标签]
        |- 详情页 https://book.douban.com/subject/[图书id]
```

这是一种经典的导航结构，从其中的数据分布情况可以得到进一步的分析思路：

1. 以标签页为"种子"页生成`start_urls`
2. 分类页得到图书的列表，然后可以循环地生成步进详细页的请求
3. 从详情页中提取出目标数据项
4. 将数据保存到后端的数据库中

接下来我们就根据这个分析思路逐一地对各个页面**爬取路径**进行展开深度的学习与分析：

标签页

豆瓣读书将全网的大类标签链接都放于标签页 内，打开开发人员工具窗口仔细探查每个分类链接的HTML标记会发现，分类链接的命名非常规范，都是以 `href="/tag/<分类名>"` 这样的规则，这就很容易编写链接提取器(`LinkExtractor`)中的`allow`表达式了。

那我们可以得到两种设计思路：

1. 一次性提取所有的"标签链接"就生成种子页(`start_urls`)，这种做法可以非常精确地的定位每个爬取入口，坏处是当爬虫进行增量式爬取时有新的标签产生就无法取得了。
2. 以**标签页**为入口遇到"标签链接"规则进行步进，这种做法的好处是作增量式爬取时可以捕获所有的标签入口，坏处就是种子入口单一，要增加链接提取与步进规则。

显而易见，思路2更优于思路1，所以接下来就按思路2作为爬虫出发的起始点。

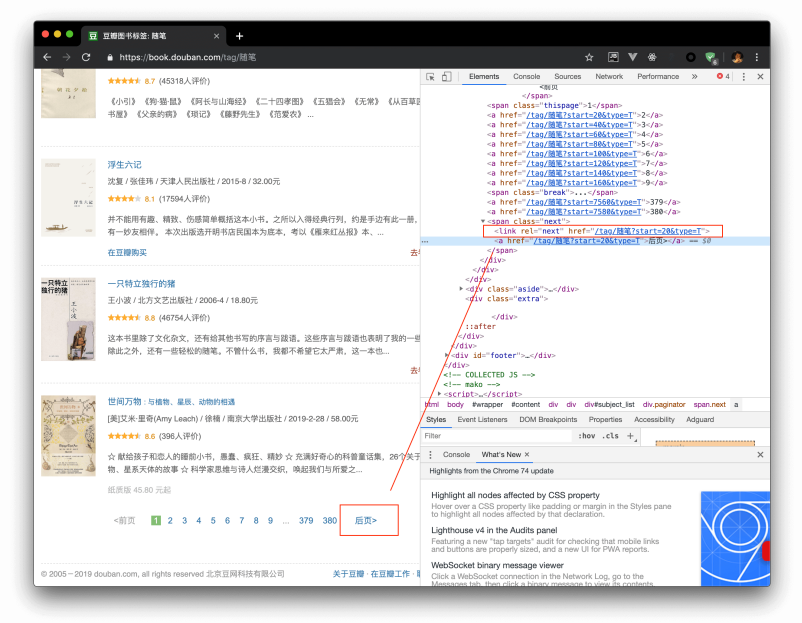
标签分类页

标签分类页(`/tag/<分类>`) 就是一个具体图书的列表，每次显示20本图书，在页面的底部带有刷新性的分页器。这个页面可以获取到部分的图书概要信息，如果你只关心获取图书的评分的话可以将这个页面作为爬虫的终结点直接从该页提取数据就行了，但既然都爬到这里了，为什么不让爬虫多拿点详细的数据呢？因此这里还不是本示例爬虫的终点，而只是一个中转站而已。

这个页面可以直达图书的详情，所以我们的爬虫需要遍历当前标签分类下的所有图书才能得到完整的图书详情的地址集合。那分页器就是我们的重点遍历目标了，只要对每个分页标签作为爬虫的步入点(`follow`)就能完成上述的需求。

分页器每次只显示了前10页的链接标签，如果只是简单地将分页器中的链接作为步进点那么做完10左右的爬取爬虫就会自动止终行动了，显然这将不是我们想要的结果。所以分页器中的数字链接并不是我们真正要找的步进点，但是在分页器的最后一个链接"后页"中，这个链接才是可以用来进行逐步分页的点，通过它我们就可以让爬虫一页一页地**翻**直至末页，这样不就可以无一遗漏地将标签内的所有图书都遍历一次了吗？打开源码分析，在"后页"链接的元素代码的相邻节点上竟然发现了另一个更有用的明确的分页位置标记元素：

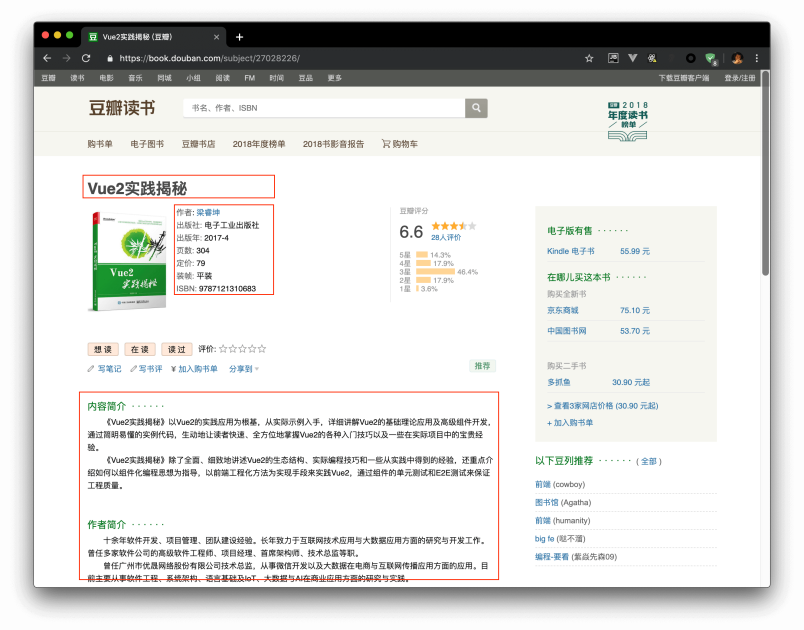
```
<link rel="next" href="/tag/小说?start=60&amp;type=T">
```



直接用一个链接提取器就可以将`<link>`元素中的`href`属性提出来作为步进的入口了！

详情页

打开详情页（`/subject/<图书ID>/`）具体我们需要的数据项位置我已在下图上圈出。



根据框出来的数据区别我们就可以开始设计需要提出来的数据项栏目，具体如下：

栏目	名称	类型
书名	`name`	`String`
作者	`authors`	`String`
出版社	`publishing_house`	`String`
出版时间	`pub_date`	`Date`
原名	`origin_name`	`String`
译者	`translators`	`String`
页数	`pages`	`String`
定价	`price`	`Number`
装订	`pkg_type`	`String`
丛书	`series`	`String`
ISBN	`isbn`	`String`
豆瓣评分	`rates`	`String`
评价数	`rating_count`	`Number`
简介	`summary`	`String`
作者简介	`about_authors`	`String`

数据模型的设计

有了上述的栏目与字段的对照表，我们就能快速地建立一个`BookItem`类来描述图书的数据属性了，先创建`douban`爬虫项目：

```
$ scrapy startproject douban
$ cd douban
$ virtualenv -p python3 venv
$ . venv/bin/activate
$ pip install scrapy
```

直接打开`items.py`文件编写`BookItem`类，代码如下所示：

```python
# coding:utf-8

from scrapy import Item, Field


class BookItem(Item):
    # 书名
    name = Field()
    # 作者
    authors = Field()
    # 出版社
    publishing_house = Field()
    # 原名
    origin_name = Field()
    # 译者
    translators = Field()
    # 出版时间
    pub_date = Field()
    # 页数
    pages = Field()
    # 定价
    price = Field()
    # 装订
    pkg_type = Field()
    # 丛书
    series = Field()
    # ISBN
    isbn = Field()
    # 豆瓣评分
    rates = Field()
    # 评价数
    rating_count = Field()
    # 简介
    summary = Field()
    # 作者简介
    about_authors = Field()
```

BookItem 设计与自定义复杂的 Processor

在栏目与字段对照表中我们发现有几个字段本身是具有数据类型的，虽然我们可以将它们统一地作为字符串类型进行处理，但这样做的最大缺点是使得类型丢失，在以后保存到数据库后就不利于进行统计查询了。

最佳的做法就是在提取数据的时候就将数据值直接进行转换，将其还原为原本类型。我们在第4章第一节"用ItemLoader解决网页数据多样性的问题"中已经学习过使用输入/输出处理器来对数据值进行"加工"的办法。但是Scrapy并没有提供将字符串数据直接转换为数字、日期等常用基本类型的处理器。

那么，借此机会我们就来动手编写几个专门用于做数据类型转换的输入/输出处理器完成这一经常会遇到的类型转换场景，按当前的项目需要我们可以得出需要以下几种输入/输出处理：。

- `Text` - 纯文本(无HTML)类型格式
- `Date` - 日期类型格式
- `Number` - 数字类型格式
- `Price` - 价格类型格式

在项目下创建一个`processors.py`的文件将以下各种格式化输入/输出处理器的代码添加其中。

`Text` 纯文本格式处理器

纯文本类型处理器的设计目标就是去除提取值中HTML标记，这是一种我们经常处理的过程，为了让我们的代码更**DRY** (**D**on't **R**epeat **Y**ourself)是很有必要将这种经常的重复封装成为一个输入/输出处理器类以便于在其它的项目中重用的。

去除HTML标记可以采用w3lib这个包内的`remove_tags`方法进行处理，这是python3的内置标准包，并不需要安装。

因为输入/输出处理器是将类名作为一个函数来调用的:

```
Text(html)
```

所以我们只要直接重写 `__call__`方法，将去除HTML标记的代码写入其中即可，具体代码如下所示：

```
from w3lib.html import remove_tags
class Text():
    def __call__(self, values):
        return [remove_tags(v).strip()
                if v and isinstance(v, six.string_types) else v
                for v in values]
```

`Number` 数字格式处理器

数字类型可以包括整数与小数，但我们并不能确定输入的字符串内的数字究竟有没有小数位，又或者是否带有空格或其它字符的情况。因此，最佳的处理手段就是采用正则表达式来完成这个数字的提取工作。

```
_NUMERIC_ENTITIES = re.compile(r'&#([0-9]+)(?:;|\s)', re.U)
_NUMBER_RE = re.compile(r'(-?\d+(?:\.\d+)?)')

class Number():
    def __call__(self, values):
        numbers = []
        for value in values:
            if isinstance(value, (dict, list)):
                numbers.append(value)
            txt = _NUMERIC_ENTITIES.sub(lambda m: unichr(int(m.groups()[0])),value)
            numbers.append(_NUMBER_RE.findall(txt))
        return list(chain(*numbers))
```

`Price` 价格格式处理

由于价格的格式应该是一个可以支持`,`千分位符的两位数字型的字符串，所以我们先用正则表达式对其进行数字型提取，然后去除千分位符和保留两位小数的操作：

```
_DECIMAL_RE = re.compile(r'(\d[\d\,]*(?:(?:\.\d+)|(?:)))', re.U | re.M)
_VALPARTS_RE = re.compile(r'([\.,]?\d+)')

class Price():
    def __call__(self, values):
        prices = []
        for value in values:
            if isinstance(value, (dict, list)):
                prices.append(value)
            txt = _NUMERIC_ENTITIES.sub(lambda m: unichr(int(m.groups()[0])),value)
            m = _DECIMAL_RE.search(txt)
            if m:
                value = m.group(1)
                parts = _VALPARTS_RE.findall(value)
                decimalpart = parts.pop(-1)
                if decimalpart[0] == "," and len(decimalpart) <= 3:
                    decimalpart = decimalpart.replace(",", ".")
                value = "".join(parts + [decimalpart]).replace(",", "")
                prices.append(value)
        return prices
```

`Date` 日期格式处理器

对于日期类型的数据格式则容易得多可以使用`dateparser`这个工具包来对日期进行格式化操作即可。

安装dataparser:

```
(venv) $ pip install dataparser
```

`Date`输入处理器的代码如下所示：

```
from dateparser.date import DateDataParser

class Date(Text):
    def __init__(self, format='%Y-%m-%dT%H:%M:%S'):
        self.format = format

    def __call__(self, values):
        values = super(Date, self).__call__(values)
        dates = []
        for text in values:
            if isinstance(text, (dict, list)):
                dates.append(text)
            try:
                date = DateDataParser().get_date_data(text)['date_obj']
                dates.append(date.strftime(self.format))
            except ValueError:
                pass
        return dates
```

现在我们可以稍微对`BookItem`作出一些小改动来规范化提取出的数据：

```
class BookItem(Item):
    # 书名
    name = Field()
    # 作者
    authors = Field()
    # 出版社
    publishing_house = Field()
    # 出品方
    publisher = Field()
    # 原名
    origin_name = Field()
    # 译者
    translators = Field()
    # 出版时间
    pub_date =  Field(input_processor=Date(),
                output_processor=TakeFirst())
    # 页数
    pages = Field(input_processor=Number(),
                output_processor=TakeFirst())
    # 定价
    price =  Field(input_processor=Price(),
                output_processor=TakeFirst())
    # 装订
    pkg_type = Field()
    # 丛书
    series = Field()
    # ISBN
    isbn = Field()
    # 豆瓣评分
    rates = Field()
    # 评价数
    rating_count =  Field(input_processor=Number(),
                output_processor=TakeFirst())
    # 简介
    summary = Field(input_processor=Text(),
                output_processor=TakeFirst())
    # 作者简介
    about_authors = Field()
```

`BookSpider`的设计与开发

接下来就是对`BookSpider`的开发了，在命令行键入以下的指命创建蜘蛛类:

```
(venv) $ scrapy genspider -t crawl book book.douban.com
```

按照上文中分析的URL逻辑可以分别得到以下三个正则表达式：

- 标签页：`'\/tag\/(.*?)'`
- 标签分类页：`\/tag\/(.*?)\?start\=`
- 详情页：`'\/subject\/.*'`

对于标签页我们还需要增加对`<link>`元素的限定，则否Scrapy会默认提取`<a>`元素的`href`:

```
LinkExtractor(allow=('\/tag\/(.*?)\?start\='),
              tags=('link'), attrs=('href'))
```

将三套规则写入到`BookSpider`，具体代码如下：

```
# -*- coding: utf-8 -*-
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor
from scrapy.loader import ItemLoader

class BookSpider(CrawlSpider):
    name = "doubanbook"
    start_urls = ['https://book.douban.com/tag/']
    rules = (Rule(LinkExtractor(allow=('\/tag\/(.*?)')), follow=True),
             Rule(LinkExtractor(allow=('\/tag\/(.*?)\?start\='),
                                tags=('link'), attrs=('href')), follow=True),
             Rule(LinkExtractor(allow=('\/subject\/.*'),), follow=False, callback='parse_item'))

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

最后就是用`ItemLoader`定义数据项的提取逻辑了。然而，在豆瓣读书上遇到了一种非常棘手的数据提取场景，先来看以下图：

如果你仔细看以上的图，你会发现的需要提取的内容是没有标签的，也就是说是属于一种不规范化的数据内容，如果使用非pyQuery还可以这样来获取到这个些值，以"出版社"栏目为例如：

```
doc=pq(response.body)
item['publishing_house']= doc('.info .pl:contains("出版社")')[0].tail
```

这是先选出合符件的`<sapn>`元素，然后通过pyQuery的对象的元素对象中的`tail`获取元素尾后的字符，但在`ItemLoader`则只能用纯CSS选择器或者XPath的选择器来提取，这又该如何呢？

CSS选择器我并没有找到什么好的方法，但在XPath中却可以使用`following`这种轴选择的关键词来获取：

```
loader.add_xpath('publishing_house', u'//span[.//text()[normalize-space(.)="出版社:"]]/following::text()[1]')
```

这是一种比较特别的情况一但遇上了就非常棘手了。

按照以述的方法我就可以一个一个地将将目标栏目的值填充到`ItemLoader`中，具体代码如下：

```
    def parse_item(self, response):
        loader = ItemLoader(item=BookItem(), response=response)
        loader.add_css('name',"h1 span[property='v:itemreviewed']") # 标题
        loader.add_css('summary','.related_info #link-report') # 简介
        loader.add_xpath('authors', u'//span[.//text()[normalize-space(.)="作者:"]]/following::text()[1]')
       # loader.add_xpath('authors', u'//span[.//text()[normalize-space(.)="作者:"]]/following::a::text()')
        loader.add_xpath('publishing_house', u'//span[.//text()[normalize-space(.)="出版社:"]]/following::text()[1]')
        #loader.add_xpath('publishing_house', u'//span[.//text()[normalize-space(.)="出版社:"]]/following::text()[1]')
        loader.add_xpath('publisher', u'//span[.//text()[normalize-space(.)="出品方:"]]/following::text()[1]')
        loader.add_xpath('origin_name', u'//span[.//text()[normalize-space(.)="原作名:"]]/following::text()[1]')
        loader.add_xpath('translators', u'//span[.//text()[normalize-space(.)="译者:"]]/following::text()[1]')
        #loader.add_xpath('translators', u'//span[.//text()[normalize-space(.)="译者:"]]/following::text()[1]')
        loader.add_xpath('pub_date', u'//span[.//text()[normalize-space(.)="出版年:"]]/following::text()[1]')
        loader.add_xpath('pages', u'//span[.//text()[normalize-space(.)="页数:"]]/following::text()[1]')
        loader.add_xpath('price', u'//span[.//text()[normalize-space(.)="定价:"]]/following::text()[1]')
        loader.add_xpath('isbn', u'//span[.//text()[normalize-space(.)="ISBN:"]]/following::text()[1]')
        loader.add_css('rates',".rating_num") # 得分
        loader.add_css('rating_count', ".rating_people>span") #投票
        return loader.load_item()
```

至此 `BookSpider` 就编写完成了。

小结

虽然，蜘蛛已经写好了我们还需要将这些数据保存到数据库中，而且我们并不确定按照我们所构想的代码就不存在运行期的缺陷，所以并不急于马上去运行它，在下一节中我会介绍如何来让我们确保代码完全是合符我们设计要求的时间再动手。

本节的重点是学习如何去编写与自定义能格式化固定类型的输入处理器，这样可以在不段的开发出抽象出经常需要重复边写的代码，避免不必要的重复。同时也是对我们之前的课程进行深化，尽量将每项学到的识知更深度地强化形成一种深入脑海的技能。

16 增量式爬虫与增量爬网策略设计

18 测试驱动开发（TDD）网络爬虫项目

精选留言 1

欢迎在这里发表留言，作者筛选后可公开显示

weixin_风一般的梦幻_0

代码很多地方兼容了python2，建议增加一些注释。

0 回复 2019-06-18

DongHj 回复 **weixin_风一般的梦幻_0**

好的同学，你的建议我们已经收到了，正在进行修改

回复 2019-06-25 10:50:55