19 测试BookSpider

更新时间：2019-06-17 16:11:54

> 世上无难事,只要肯登攀。
>
> ——毛泽东

在上一节我们完成了`Processor`的测试编写，`Processor`比较简单只是测试一个输入输出，那如何来测试`BookSpider`呢？首先，Scrapy是一个高度模块化的框架，蜘蛛只是其中提供爬网路径与内容分析的其中的个模块。

另外，更重要的一点还是回归到"我们想测试什么"，测试的期望值是什么。我们可以回顾一下上一节的内容，这个`BookSpider`蜘蛛是怎么设计出来的，我们最终的思路是什么，那么这个就是预期：

1. 蜘蛛是否能按照我们的期望执行从 标签页->标签分类页->图书详情页这样的路径前行。
2. `ItemLoader`中的CSS选择器和XPath选择器的写法是否正确，提取出来的数据项(`Item`)中的内容是否正确。

在`tests`目录下创建一个`test_spiders.py`的文件并添加以下的代码:

```
# coding:utf-8
import unittest
import os
from douban.spiders.book import BookSpider
from scrapy.http import HtmlResponse, Request

class SpiderTest(unittest.TestCase):

  def test_bookspider(self):
      pass
```

测试有时可以很直接，既然我们的测试目标就是`BookSpider`那么就直接实例化它，然后调用里面的方法看最终输出结果不就行了？与实际运行时不同的只不过是引导程序不一样而已，实际运行是依赖于scrapy cli来引导执行，而测试不过是通过unittest引导。

直接实例化`BookSpider`：

```
spider = BookSpider()
```

我们要测试蜘蛛就得深入了解蜘蛛的动作机理，由于`BookSpider`是从`CrawlSpider`继承而来，我们就可以好好地读一读`CrawlSpider`是怎么实现的，那个才是`CrawlSpider`的入口方法。

蜘蛛在Scrapy是负责处理返回请求的，至于`start_urls`是告诉Scrapy从哪里开始发请请求，`Rules`则是告诉Scrapy在处理完当前请求后如何生成一下步的请求，真正发出请求的是由下载器中间件`DownloaderMiddleware`完成处理的。

查看`CrawlSpider`你会找到一个`parse`函数，是提供给Scrapy当生成了请求的响应结果后调用的。它就是我们要找的函数！

```
def parse(self, response):
 return self._parse_response(response, self.parse_start_url, cb_kwargs={}, follow=True)

def _parse_response(self, response, callback, cb_kwargs, follow=True):
  if callback:
    cb_res = callback(response, **cb_kwargs) or ()
    cb_res = self.process_results(response, cb_res)
    for requests_or_item in iterate_spider_output(cb_res):
      yield requests_or_item
```

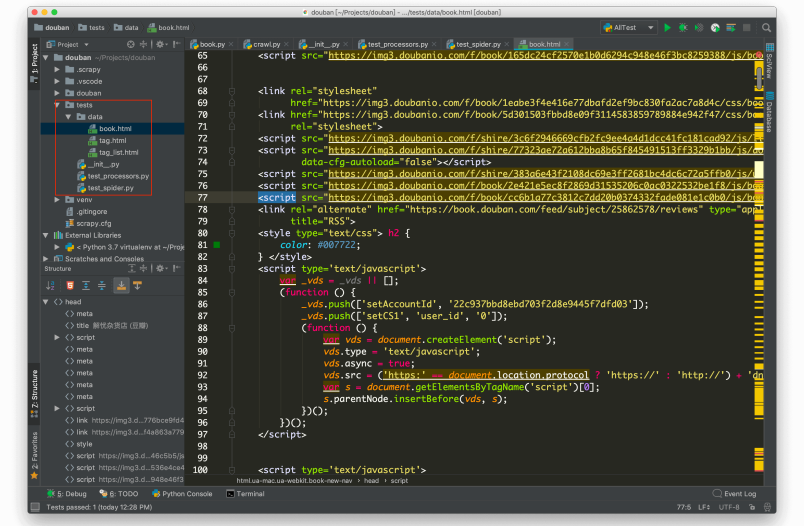
`parse`函数调用了`_parse_reponse`这个私有函数，是分析相应结果中是否有符合`Rule`中定义的提取结果，如果符合则会判断是继续生成新请求还是执行蜘蛛内的响应分析方法返回数据项(`Item`)

那么，我们只要模拟Scrapy分别向标签页、标签分类页与详情页发出请求，然后生成对应的响应对象调用这个`parse`函数并判断这个函数中返回的请求对象或数据项是否合符我们的预期就表示测试成功了。

测试数据的准备

要模拟发出的这个请求并不是真的将请求发送到的豆瓣上去，测试一定做到无依赖可独立执行才是一个真正合格的测试。因此，我们可以直接访问那三个待页面并且直接在浏览器中将访问的页面的源代码保存下来作为**仿真数据**样本。

建立一个`data`目录，将这三个html页面保存其中。如下图所示：



数据仿真 Mock

接下来我们就需要编写一个函数来产生仿真的请求与响应，因为我们已经有响应结果了所以并不需要真正地发出请求，而只是直接返回想要的请求结果对象就行了。代码如下所示：

```
def mock_response(self, target_url, mock_file):
  with open(mock_file, 'r') as f:
    body = f.read()

    return HtmlResponse(url=target_url,
                        request = Request(target_url),
                        body=body,
                        encoding="utf-8")
```

这个`mock_response`函数只要输入请求地址与该请求对应的网页就可以直接生成一个`HtmlResponse`对象实例了，这一过程就相当于跳过了Scrapy在真实运行的时候执行的请求过程，Scrapy的处理过程我们是不用理会的，因为那个根本不会出错所以就没有测试的必要。

接下来就是真正单元测试函数的内容了：

```
def test_crawl_rules(self):
  spider = BookSpider()

  url_and_files = [
    ("https://book.douban.com/tag", './data/tag.html'),
    ("https://book.douban.com/tag", './data/tag_list.html'),
    ("https://book.douban.com/tag", './data/book.html')
  ]

  item_or_request = spider.parse(self.build_response(*url_and_files[0]))
  ##TODO:对标签页处理结果进行判断

  item_or_request = spider.parse(self.build_response(*url_and_files[1]))
  ##TODO:对标签页列表处理结果进行判断

  item_or_request = spider.parse(self.build_response(*url_and_files[2]))
  ##TODO:对ItemLoader的提取内容进行测试判断
```

我将对应的请求地址与结果文件放在一个由元组(Tuple)数组中，然后分别调用三次`spider.parse`函数(实际上Scrapy也是这样运行的)。先把这个爬行路径的执行逻辑编写好。暂时还不清楚要怎么写断言则先用`TODO`加上标记，下一步来做。

编写断言

标签页的断言

标签页是依赖于`CrawlSpider`按照`Rule`中的`LinkExtracotr`定义的提取规则从响应页面读取标签链接然后进一步地生成新的`Request`对象来完成页面的跟进的(follow)。所以第一次的调用`parse`返回的应该是一个`Request`的对象列表，而且它们的URL都是与`\/tag\/(.*?)`匹配的。那么这个断言的写法就应该如下：

```
tag_pattern = u'\/tag\/(.*?)'
for request in item_or_request:
 self.assertIsNotNone(re.search(tag_pattern, request.url))
```

> 如果是自己动手的同学要注意，在实例化`BookSpider`后要手动设置`self._follow=True`否则会报错，因为这个值本来是由Scrapy内部设置的，我们在模拟时就需要改为手动设置了。

标签分类页的断言

对于标签分类页也是相同的逻辑，只是正则表达式有所不同罢了：

```
list_pattern = u'\/tag\/(.*?)\?start\='
for request in item_or_request:
result = re.search(list_pattern, request.url) is not None or \
re.search(tag_pattern, request.url) is not None
self.assertIsNotNone(result)
```

对于以上的代码你一定会产生疑惑：

```
result = re.search(list_pattern, request.url) is not None or \
re.search(tag_pattern, request.url) is not None
```

为什么要对两个规则同时判断？这是因为`CrawlSpider`在每次返回的响应对象是对所有的`Rules`中的`LinkExtractor`都执行一次的，而不是只执行其中符合条件的一个。这是你不做测试是难以观察到的干货。

详细页的断言

同样地，将最后规则加到其中:

```
book_pattern = u'\/subject\/.*'
for request in item_or_request:
result = re.search(list_pattern, request.url) is not None or \
re.search(tag_pattern, request.url) is not None or \
re.search(book_pattern, request.url) is not None
self.assertIsNotNone(result)
```

以上的代码其实逻辑上都是重复的，我们可以精简一下代码把重用的部分都给抽出来：

```
def test_crawl_rules(self):
  spider = BookSpider()
  spider._follow_links = True
  url_and_files = [
    ("https://book.douban.com/tag", 'tests/data/tag.html'),
    ("https://book.douban.com/tag", 'tests/data/tag_list.html'),
    ("https://book.douban.com/tag", 'tests/data/book.html')
  ]

  def check_result(request):
    return re.search(u'\/tag\/(.*?)\?start\=', request.url) is not None or \
  re.search(u'\/tag\/(.*?)', request.url) is not None or \
  re.search(u'\/subject\/.*', request.url) is not None

  for kv in url_and_files:
    [self.assertIsNotNone(check_result(r)) for r in spider.parse(self.build_response(*kv))]
```

这样我们就完成对的三个不同页的爬取路径的测试了，最后就是对`parse_item`函数的单元测试了。因为我们跳过了一些处理机制，直接调用`parse`并没有办法执行`Rule`中的回调函数，所以我们就得手工测试`parse_item`了。

测试`ItemLoader`

`ItemLoader`的测试就更简单了，直接调用`parse_item`然后将采集的仿真数据中值与分析得到的值直接对比就可以知道那个提取逻辑有问题了：

```
def test_parse_item(self):
  spider = BookSpider()
  item = spider.parse_item(self.mock_response("https://book.douban.com/tag", 'tests/data/book.html'))
  self.assertEqual(item['name'], u'解忧杂货店')
  self.assertEqual(item['authors'], u'[日]东野圭吾')
  self.assertEqual(item['publishing_house'], u'南海出版公司')
  self.assertEqual(item['publisher'], u'新经典文化')
  self.assertEqual(item['origin_name'], u'ナミヤ雑貨店の奇蹟')
  self.assertEqual(item['translators'], u'李盈春')
  self.assertEqual(item['pub_date'], u'2014-05-02T00:00:00')
  self.assertEqual(item['pages'], u'291')
  self.assertEqual(item['price'], u'39.50')
  self.assertEqual(item['isbn'], u'9787544270878')
  self.assertEqual(item['rates'], u'8.5')
  self.assertEqual(item['rating_count'], u'438151')
  self.assertIsNotNone(item['summary'])
```

上述的这个测试看似无聊，但却让我改了10个以上的错误！包括`BookItem`：

```python
# coding:utf-8

from scrapy import Item, Field
from .processors import Number, Date, Price, Text,CleanText
from scrapy.loader.processors import TakeFirst, Join


class BookItem(Item):
    # 书名
    name = Field(input_processor=CleanText(),
                 output_processor=TakeFirst())
    # 作者
    authors = Field(input_processor=CleanText(),
                    output_processor=TakeFirst())
    # 出版社
    publishing_house = Field(input_processor=CleanText(),
                             output_processor=TakeFirst())
    # 出品方
    publisher = Field(input_processor=CleanText(),
                      output_processor=TakeFirst())
    # 原名
    origin_name = Field(input_processor=CleanText(),
                        output_processor=TakeFirst())
    # 译者
    translators = Field(input_processor=CleanText(),
                        output_processor=TakeFirst())
    # 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())
    # ISBN
    isbn = Field(input_processor=CleanText(),
                 output_processor=TakeFirst())
    # 豆瓣评分
    rates = Field(input_processor=Number(),
                  output_processor=TakeFirst())
    # 评价数
    rating_count = Field(input_processor=Number(),
                         output_processor=TakeFirst())
    # 简介
    summary = Field(input_processor=Text(),
                    output_processor=Join())
    # 作者简介
    about_authors = Field(input_processor=CleanText(),
                          output_processor=TakeFirst())
```

还有`ItemLoader`的提取逻辑:

```
def parse_item(self, response):
  loader = ItemLoader(item=BookItem(), response=response)
  loader.add_css('name',"h1 span::text") # 标题
  loader.add_css('summary','.related_info #link-report .intro p::text') # 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::text()[2]')
  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('publisher', u'//span[.//text()[normalize-space(.)="出品方:"]]/following::text()[2]')
  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()[2]')
  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::text") # 得分
  loader.add_css('rating_count', ".rating_people>span::text") #投票
  return loader.load_item()
```

个中的不同你可以与上一节的内容进行对照。

小结

我发现源代码中的错误需要运行超过20次以上，到底是在命令行运行`scrapy crawl`等待它运行到需要调试的位置快呢还是我直接在IDE上按F9运行调试直接观察错误引发的位置更快呢？你可以实际动手尝试一下体验两者的对比。只有真的尝试过测试所带来的好处你才会真正认识到测试的重要性。测试带给我们的开发过程是：编码->测试->重构然后往复循环，在重构中改善代码。当测试已经成为你开发的一种习惯以后，这个过程会演变为：测试->编码->重构。

> 注：你可以去 Github 获取本节课的源代码

18 测试驱动开发（TDD）网络爬虫项目

20 基于SQL的数据导出机制

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示

目前暂无任何讨论