

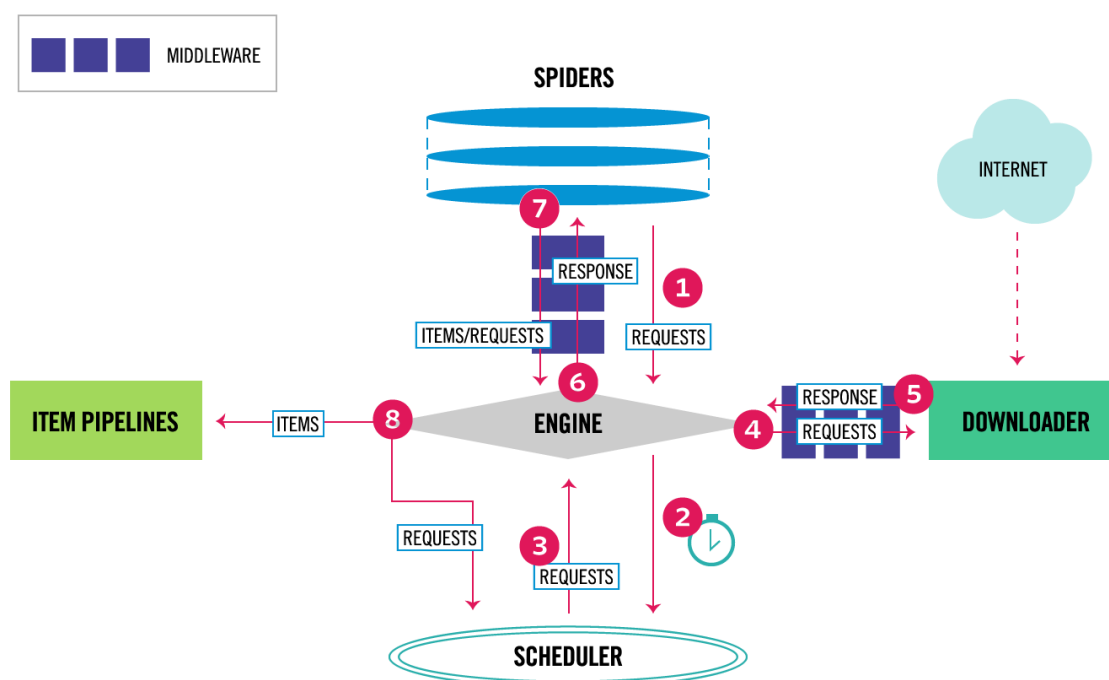
Scrapy爬虫框架高级应用

Spider的用法

在Scrapy框架中，我们自定义的蜘蛛都继承自`scrapy.spiders.Spider`，这个类有一系列的属性和方法，具体如下所示：

1. `name`：爬虫的名字。
2. `allowed_domains`：允许爬取的域名，不在此范围的链接不会被跟进爬取。
3. `start_urls`：起始URL列表，当我们没有重写`start_requests()`方法时，就会从这个列表开始爬取。
4. `custom_settings`：用来存放蜘蛛专属配置的字典，这里的设置会覆盖全局的设置。
5. `crawler`：由`from_crawler()`方法设置的和蜘蛛对应的Crawler对象，Crawler对象包含了很多项目组件，利用它我们可以获取项目的配置信息，如调用`crawler.settings.get()`方法。
6. `settings`：用来获取爬虫全局设置的变量。
7. `start_requests()`：此方法用于生成初始请求，它返回一个可迭代对象。该方法默认是使用GET请求访问起始URL，如果起始URL需要使用POST请求来访问就必须重写这个方法。
8. `parse()`：当Response没有指定回调函数时，该方法就会被调用，它负责处理Response对象并返回结果，从中提取出需要的数据和后续的请求，该方法需要返回类型为Request或Item的可迭代对象（生成器当前也包含在其中，因此根据实际需要可以用`return`或`yield`来产生返回值）。
9. `closed()`：当蜘蛛关闭时，该方法会被调用，通常用来做一些释放资源的善后操作。

中间件的应用



下载中间件

scrapy组件

首先我们看下scrapy官网提供的新结构图，乍一看这画的是啥啊，这需要你慢慢的理解其原理就很容易看懂了，这些都是一个通用爬虫框架该具有的一些基本组件。上一篇博客说了项目管道(也就是图中的ITEM PIPELINES)，可以看到中间的引擎(ENGINE)将item传递给了项目管道，也就是让项目管道来处理

抓取到的内容。另外图中的所谓的组件只是抽象出来的东西比较容易让人理解，其实这些都是python的类实例化的东西。

下载中间件处于引擎和下载器的中间，就像是引擎和下载器有一根管道，管道上面有两个路障就是下载中间件了，而引擎和下载器之间只传递请求(REQUESTS)和响应(RESPONSE)，所以这两个路障很明显的的作用就是拦截请求和响应(至于拦截之后想干嘛就随便了，比如更改请求或响应内容，更改请求头或响应头，或者什么都不干，记录一下就放过去)。

蜘蛛中间件处于蜘蛛和引擎之间，这里说的蜘蛛就是我们在spider目录下编写的蜘蛛类，已经写过蜘蛛的应该知道，在蜘蛛中一般会处理响应(默认解析是parse方法),返回item，或者返回(准确点应该是迭代)请求(scrapy.Request)。而在图中也很明显，蜘蛛和引擎之间会传递请求(REQUESTS)、响应(RESPONSE)和items。响应是由下载器从网站上下载得到的，在传递给引擎后给蜘蛛来处理，而items一般是由蜘蛛从响应内容中解析出来传递给引擎在给用户管道处理，而请求一般是由蜘蛛中的start_urls中定义的URL和蜘蛛解析出的新的URL准备传递给引擎在交由调度器统一管理。

上面的一堆屁话说的：中间件的作用就是拦截两个组件传递的内容。

看了这些组件的功能和使用你会发现为什么要这么麻烦，你看我用requests库只需要请求一下得到响应直接处理就行了，这引擎和调度器感觉没什么存在的意义啊。这就需要慢慢理解了，或者自己写一个简单通用爬虫框架可能就知道了。

中间件的编写

中间件的启用

```
DOWNLOADER_MIDDLEWARES = {
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
    'newspider.middlewares.UAMiddleware': 500 #newspider为scrapy项目名称,也就是
    settings.py中BOT_NAME的值, UAMiddleware为定义的中间件的名字
}
```

字典的值如何选择，500这个值改成其他的行不行呢？先看一下官方是怎么说的：

scrapy会将DOWNLOADER_MIDDLEWARES设置与DOWNLOADER_MIDDLEWARES_BASE设置合并，然后按值的顺序排序以获取启用的中间件的最终排序列表：值最小的中间件离引擎更近，最大的中间件离下载器更近。换句话说，每个中间件的process_request()方法将以递增的顺序被调用，而每个中间件的process_response()方法将以递减的顺序调用。

要决定分配给中间件的顺序，请查看 DOWNLOADER_MIDDLEWARES_BASE设置并根据您要插入中间件的位置选择一个值。顺序很重要，因为每个中间件执行不同的操作，并且您的中间件可能取决于所应用的某些先前（或后续）中间件。

如果要禁用内置中间件（DOWNLOADER_MIDDLEWARES_BASE默认情况下在定义和启用的中间件），则必须在项目的DOWNLOADER_MIDDLEWARES设置为None。

上面的一些话中有一句很重要：**每个中间件的process_request()方法将以递增的顺序被调用，而每个中间件的process_response()方法将以递减的顺序调用。**这应该就很清楚值怎么选吧，不过一般中间件并没有确定的顺序，只要不产生矛盾就行了。

举个例子吧：你已经写了一个中间件newspider.middlewares.UAMiddleware，但是你在启用的时候没有禁止内置的UserAgentMiddleware，怎样才能使你的中间件生效呢？代码如下：

```
DOWNLOADER_MIDDLEWARES = {
    #'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
    'newspider.middlewares.UAMiddleware': 501 #newspider为scrapy项目名称,也就是
    settings.py中BOT_NAME的值, UAMiddleware为定义的中间件的名字
}
```

因为默认的用户AgentMiddleware值为500，因为修改请求头是在process_request，而process_request是以递增的形式被调用，所以会先调用内置UserAgentMiddleware，在调用你定义的UAMiddleware，则内置的会被覆盖。不过因为很多人修改请求头都是以字典访问的形式修改(request.headers["User-Agent"] = ""),这样的话即使你设置值为499，结果还是你设定的User-Agent，而不是默认内置的。

其实看一下内置的代码就可以理解了：源代码

代码使用的是request.headers.setdefault(b'User-Agent', self.user_agent)，setdefault是当不存在键时设置，存在则不设置。

内置下载中间件

```
DOWNLOADER_MIDDLEWARES_BASE = {
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware': 300,
    'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware':
350,
    'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware': 400,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': 500,
    'scrapy.downloadermiddlewares.retry.RetryMiddleware': 550,
    'scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware': 560,
    'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware': 580,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware':
590,
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': 600,
    'scrapy.downloadermiddlewares.cookies.CookiesMiddleware': 700,
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.downloadermiddlewares.stats.DownloaderStats': 850,
    'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware': 900,
}
```

下载中间件的编写

```
process_request(request, spider): 所有请求都会调用此方法
process_response(request, response, spider): 这里的参数比上面的多了response，肯定是用
来处理response的
process_exception(request, exception, spider): 处理异常
from_crawler(cls, crawler): 从settings.py获取配置
```

假设我们需要一个自动管理cookie的中间件，应该怎么编写？首先管理cookie一般就是将服务器返回的Set-cookie更新一下已保存的cookie。伪代码如下：

```
from scrapy.exceptions import NotConfigured

class CookieMiddleware(object):
    def __init__(self):
        # 实际上cookies并不能使用字典来保存，因为cookies是可以包含相同的键的
        # 这里只做简单的演示，正常可以使用scrapy.http.cookies.CookieJar
        self.cookies = {}
    def process_request(request, spider):
        self.addcookies(request, self.cookies)

    def process_response(request, response, spider):
        newcookie = response.headers.getlist('Set-Cookie')
        self.updatecookies(self.cookies, newcookie)
        return response
```

```

def process_exception(request, exception, spider):
    pass

def addcookies(self, request, cookies):
    '''
    给请求加cookie
    '''
    pass

def updatecookie(self, cookies, cookie):
    '''
    更新cookie字典(对象)
    '''
    pass

@classmethod
def from_crawler(cls, crawler):
    if not crawler.settings.get('是否开启cookie'):
        #这个方法在__init__之前，抛出异常的话，类中的所有方法就都不执行了
        raise NotConfigured

```

其实scrapy已经内置了cookies管理的中间件，可以参考一下源代码：<https://docs.scrapy.org/en/latest/modules/scrapy/downloadermiddlewares/cookies.html#CookiesMiddleware>

想要一个随机User-Agent的中间件也很简单，只需要在process_request方法中改变请求头就行了。其实process_request和process_response这两个方法是有返回值的，在上面的例子中只是对请求头和响应头做了修改，所以没有涉及到返回值。如果我们需要对请求和响应整个做更改的话就需要重新构造请求和响应。这里最典型的例子就是selenium中间件了。

spider示例代码：

```

import scrapy
from selenium import webdriver

class SeleniumSpider(scrapy.Spider):
    name = 'seleniumspider'
    start_urls = ['https://example.com']

    # 实例化浏览器对象，因为浏览器对象只需要实例化一次，所以不在中间件中进行实例化
    bro =
    webdriver.Chrome(executable_path=r'E:\spiderman\day13\wynews\wynews\wynews\chromedriver.exe')

    def parse(self, response):
        '''
        和正常一样，该怎么解析怎么解析
        '''
        pass

```

selenium中间件示例代码（只是简单演示，为了更好理解中间件怎么写）：

```

from selenium.common.exceptions import TimeoutException
from scrapy.http import HtmlResponse

class SeleniumMiddleware(object):

```

```

def __init__(self, timeout=None, options=None):
    pass

def process_request(self, request, spider):
    try:
        url = request.url
        spider.bro.get(url)
        return HtmlResponse(url=url, body=spider.bro.page_source,
request=request, encoding='utf-8', status=200)
    except TimeoutException:
        return HtmlResponse(url=url, status=500, request=request)

@classmethod
def from_crawler(cls, crawler):
    pass

```

在代码中HtmlResponse类就是我们新构建的响应，这样前面的spider处理的response就是这个响应了。这里有个疑问？为什么要在process_request这个方法中返回响应，就不能在process_response中吗？其实也可以，区别在于process_request返回的话，会忽略其他中间件中的process_request和process_exception方法，只执行其他中间件的process_response方法。试想一下如果你还有一个cookie管理的中间件，process_request的操作是不是就和SeleniumMiddleware中的矛盾了，所以我们要在process_request返回response来使其他中间件的process_request方法失效。

返回值

process_request方法返回值

None: 如果返回None，则一切正常传递
request对象: 如果返回request对象，scrapy会停止调用接下来的中间件而重新处理request对象(也就是交由引擎重新管理)
response对象: ..., Scrapy不再调用其他中间件的process_request()或process_exception()方法，它将直接调用其他中间件的process_response方法
IgnoreRequest异常: 依次调用所有中间件的process_exception方法

process_response方法返回值 (注意这个方法必须返回一个对象，不能是None)

response对象: 传递给下一个中间件的process_response方法
request对象: scrapy会停止调用接下来的中间件而重新处理request对象(也就是交由引擎重新管理)
IgnoreRequest异常: 依次调用所有中间件的process_exception方法

process_exception方法返回值

None: 继续传递
request对象: scrapy会停止调用接下来的中间件的process_exception方法而重新处理request对象(也就是交由引擎重新管理)
response对象: 停止传递接下来的中间件中的process_exception，而开始调用所有process_response方法

蜘蛛中间件

```
process_spider_input(response, spider): 所有请求都会调用这个方法
process_spider_output(response, result, spider): spider解析完response之后调用该方法, result就是解析的结果(是一个可迭代对象), 其中可能是items也可能是request对象
process_spider_exception(response, exception, spider): 处理异常
process_start_requests(start_requests, spider): 同process_spider_output, 不过只处理spider中start_requests方法返回的结果
from_crawler(cls, crawler): ...
```

同样是拦截请求和响应, 那么和下载中间件有什么区别呢? 因为下载中间件是连通引擎和下载器的, 所以如果修改请求只会影响下载器返回的结果。如果修改响应, 会影响蜘蛛处理。而蜘蛛中间件是连通引擎和蜘蛛的, 如果修改请求则会影响整个scrapy的请求, 因为scrapy的所有请求都来自于蜘蛛, 当然包括调度器和下载器。如果修改响应, 则只会影响蜘蛛的解析, 因为响应是由引擎传递给蜘蛛的。

同样是拦截请求和响应, 那么和下载中间件有什么区别呢? 因为下载中间件是连通引擎和下载器的, 所以如果修改请求只会影响下载器返回的结果。如果修改响应, 会影响蜘蛛处理。而蜘蛛中间件是连通引擎和蜘蛛的, 如果修改请求则会影响整个scrapy的请求, 因为scrapy的所有请求都来自于蜘蛛, 当然包括调度器和下载器。如果修改响应, 则只会影响蜘蛛的解析, 因为响应是由引擎传递给蜘蛛的。

上面说的有点乱, 整理一下这两个的功能:

蜘蛛中间件: 记录深度、丢弃非200状态码响应、丢弃非指定域名请求等

下载中间件: 修改请求头、修改响应、管理cookies、丢弃非200状态码响应、丢弃非指定域名请求等

丢弃非指定域名请求一般使用蜘蛛中间件, 如果使用下载中间件会导致引擎和调度器的无效任务变多, 丢弃非200状态码响应我感觉应该用下载中间件, 但是scrapy内置的使用的是蜘蛛中间件, 可能是我理解不够透彻吧。

scrapy已经提供了很多内置的中间件, 只需要启用即可。另外, 蜘蛛中间件一般用于操作蜘蛛返回的request, 而下载中间件用于操作向互联网发起请求的request和返回的response。更多的示例可以看看一些内置中间件的源码, 蜘蛛中间件一般不需要自己编写, 使用内置的几个也足够了

Scrapy对接Selenium

Scrapy框架的使用之Scrapy对接Selenium

Scrapy抓取页面的方式和requests库类似, 都是直接模拟HTTP请求, 而Scrapy也不能抓取JavaScript动态渲染的页面。抓取JavaScript渲染的页面有两种方式。一种是分析Ajax请求, 找到其对应的接口抓取, Scrapy同样可以用此种方式抓取。另一种是直接用Selenium或Splash模拟浏览器进行抓取, 我们不需要关心页面后台发生的请求, 也不需要分析渲染过程, 只需要关心页面最终结果即可, 可见即可爬。那么, 如果Scrapy可以对接Selenium, 那Scrapy就可以处理任何网站的抓取了。

一、本目标

本节我们来看看Scrapy框架如何对接Selenium, 以PhantomJS进行演示。我们依然抓取淘宝商品信息, 抓取逻辑和前文中用Selenium抓取淘宝商品完全相同。

二、准备工作

请确保PhantomJS和MongoDB已经安装好并可以正常运行, 安装好Scrapy、Selenium、PyMongo库。

三、新建项目

首先新建项目, 名为scrapyseleniumtest, 命令如下所示:

```
scrapy startproject scrapyseleniumtest
```

新建一个Spider, 命令如下所示:

```
scrapy genspider taobao www.taobao.com
```


修改 `ROBOTSTXT_OBEY` 为 `False`，如下所示：

`ROBOTSTXT_OBEY = False`

四、定义 Item

首先定义 `Item` 对象，名为 `ProductItem`，代码如下所示：

```
from scrapy import Item, Field
class ProductItem(Item):
    collection = 'products'
    image = Field()
    price = Field()
    deal = Field()
    title = Field()
    shop = Field()
    location = Field()
```

这里我们定义了6个Field，也就是6个字段，跟之前的案例完全相同。然后定义了一个 `collection` 属性，即此Item保存的MongoDB的Collection名称。

初步实现Spider的 `start_requests()` 方法，如下所示：

```
from scrapy import Request, Spider
from urllib.parse import quote
from scrapy.seleniumtest.items import ProductItem
class TaobaoSpider(Spider):
    name = 'taobao'
    allowed_domains = ['www.taobao.com']
    base_url = 'https://s.taobao.com/search?q='
    def start_requests(self):
        for keyword in self.settings.get('KEYWORDS'):
            for page in range(1, self.settings.get('MAX_PAGE') + 1):
                url = self.base_url + quote(keyword)
                yield Request(url=url, callback=self.parse, meta={'page': page},
                    dont_filter=True)
```

首先定义了一个 `base_url`，即商品列表的URL，其后拼接一个搜索关键字就是该关键字在淘宝的搜索结果商品列表页面。

关键字用 `KEYWORDS` 标识，定义为一个列表。最大翻页页码用 `MAX_PAGE` 表示。它们统一定义在 `settings.py` 里面，如下所示：

```
KEYWORDS = ['iPad']
MAX_PAGE = 100
```

在 `start_requests()` 方法里，我们首先遍历了关键字，遍历了分页页码，构造并生成Request。由于每次搜索的URL是相同的，所以分页页码用 `meta` 参数来传递，同时设置 `dont_filter` 不去重。这样爬虫启动的时候，就会生成每个关键字对应的商品列表的每一页的请求了。

五、对接 Selenium

接下来我们需要处理这些请求的抓取。这次我们对接Selenium进行抓取，采用Downloader Middleware来实现。在Middleware里面的 `process_request()` 方法里对每个抓取请求进行处理，启动浏览器并进行页面渲染，再将渲染后的结果构造一个 `HtmlResponse` 对象返回。代码实现如下所示：

```

from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from scrapy.http import HtmlResponse
from logging import getLogger
class SeleniumMiddleware():
    def __init__(self, timeout=None, service_args=[]):
        self.logger = getLogger(__name__)
        self.timeout = timeout
        self.browser = webdriver.PhantomJS(service_args=service_args)
        self.browser.set_window_size(1400, 700)
        self.browser.set_page_load_timeout(self.timeout)
        self.wait = WebDriverWait(self.browser, self.timeout)
    def __del__(self):
        self.browser.close()
    def process_request(self, request, spider):
        """
        用PhantomJS抓取页面
        :param request: Request对象
        :param spider: Spider对象
        :return: HtmlResponse
        """
        self.logger.debug('PhantomJS is Starting')
        page = request.meta.get('page', 1)
        try:
            self.browser.get(request.url)
            if page > 1:
                input = self.wait.until(
                    EC.presence_of_element_located((By.CSS_SELECTOR, '#mainsrp-
pager div.form > input')))
                submit = self.wait.until(
                    EC.element_to_be_clickable((By.CSS_SELECTOR, '#mainsrp-pager
div.form > span.btn.J_Submit')))
                input.clear()
                input.send_keys(page)
                submit.click()
                self.wait.until(EC.text_to_be_present_in_element((By.CSS_SELECTOR,
'#mainsrp-pager li.item.active > span'), str(page)))
                self.wait.until(EC.presence_of_element_located((By.CSS_SELECTOR, '.m-
itemlist .items .item')))
                return HtmlResponse(url=request.url, body=self.browser.page_source,
request=request, encoding='utf-8', status=200)
            except TimeoutException:
                return HtmlResponse(url=request.url, status=500, request=request)
        @classmethod
        def from_crawler(cls, crawler):
            return cls(timeout=crawler.settings.get('SELENIUM_TIMEOUT'),
                service_args=crawler.settings.get('PHANTOMJS_SERVICE_ARGS'))

```

首先我们在 `__init__()` 里对一些对象进行初始化，包括 `PhantomJS`、`WebDriverWait` 等对象，同时设置页面大小和页面加载超时时间。在 `process_request()` 方法中，我们通过 `Request` 的 `meta` 属性获取当前需要爬取的页码，调用 `PhantomJS` 对象的 `get()` 方法访问 `Request` 的对应的 URL。这就相当于从 `Request` 对象里获取请求链接，然后再用 `PhantomJS` 加载，而不再使用 `Scrapy` 里的 `Downloader`。

随后的处理等待和翻页的方法在此不再赘述，和前文的原理完全相同。最后，页面加载完成之后，我们调用PhantomJS的 `page_source` 属性即可获取当前页面的源代码，然后用它来直接构造并返回一个 `HtmlResponse` 对象。构造这个对象的时候需要传入多个参数，如 `url`、`body` 等，这些参数实际上就是它的基础属性。可以在官方文档查看 `HtmlResponse` 对象的结构：<https://doc.scrapy.org/en/latest/topics/request-response.html>。这样我们就成功利用PhantomJS来代替Scrapy完成了页面的加载，最后将Response返回即可。

有人可能会纳闷：为什么实现这么一个Downloader Middleware就可以了？之前的Request对象怎么办？Scrapy不再处理了吗？Response返回后又传递给了谁？

是的，Request对象到这里就不会再处理了，也不会再像以前一样交给Downloader下载。Response会直接传给Spider进行解析。

我们需要回顾一下Downloader Middleware的 `process_request()` 方法的处理逻辑，内容如下所示：

当 `process_request()` 方法返回Response对象的时候，更低优先级的Downloader Middleware的 `process_request()` 和 `process_exception()` 方法就不会被继续调用了，转而开始执行每个Downloader Middleware的 `process_response()` 方法，调用完毕之后直接将Response对象发送给Spider来处理。

这里直接返回了一个 `HtmlResponse` 对象，它是Response的子类，返回之后便顺次调用每个Downloader Middleware的 `process_response()` 方法。而在 `process_response()` 中我们没有对其做特殊处理，它会被发送给Spider，传给Request的回调函数进行解析。

到现在，我们应该能了解Downloader Middleware实现Selenium对接的原理了。

在settings.py里，我们设置调用刚才定义的 `SeleniumMiddleware`，如下所示：

```
DOWNLOADER_MIDDLEWARES = {
    'scrapyseleniumtest.middlewares.SeleniumMiddleware': 543,
}
```

六、解析页面

Response对象就会回传给Spider内的回调函数进行解析。所以下一步我们就实现其回调函数，对网页来进行解析，代码如下所示：

```

def parse(self, response):
    products = response.xpath(
        '//div[@id="main-srp-item-list"]//div[@class="items"]
        [1]//div[contains(@class, "item")]')
    for product in products:
        item = ProductItem()
        item['price'] = ''.join(product.xpath('..//div[contains(@class,
            "price")]//text()').extract()).strip()
        item['title'] = ''.join(product.xpath('..//div[contains(@class,
            "title")]//text()').extract()).strip()
        item['shop'] = ''.join(product.xpath('..//div[contains(@class,
            "shop")]//text()').extract()).strip()
        item['image'] =
            ''.join(product.xpath('..//div[@class="pic"]//img[contains(@class, "img")]/@data-
            src').extract()).strip()
        item['deal'] = product.xpath('..//div[contains(@class, "deal-
            cnt")]//text()').extract_first()
        item['location'] = product.xpath('..//div[contains(@class,
            "location")]//text()').extract_first()
        yield item

```

在这里我们使用XPath进行解析，调用 response 变量的 xpath() 方法即可。首先我们传递选取所有商品对应的XPath，可以匹配所有商品，随后对结果进行遍历，依次选取每个商品的名称、价格、图片等内容，构造并返回一个 ProductItem 对象。

七、存储结果

最后我们实现一个Item Pipeline，将结果保存到MongoDB，如下所示：

```

import pymongo
class MongoPipeline(object):
    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_DB'))
    def open_spider(self, spider):
        self.client = pymongo.MongoClient(self.mongo_uri)
        self.db = self.client[self.mongo_db]
    def process_item(self, item, spider):
        self.db[item.collection].insert(dict(item))
        return item
    def close_spider(self, spider):
        self.client.close()

```

此实现和前文中存储到MongoDB的方法完全一致，原理不再赘述。记得在settings.py中开启它的调用，如下所示：

```

ITEM_PIPELINES = {
    'scrapyseleniumtest.pipelines.MongoPipeline': 300,
}

```

其中，MONGO_URI 和 MONGO_DB 的定义如下所示：

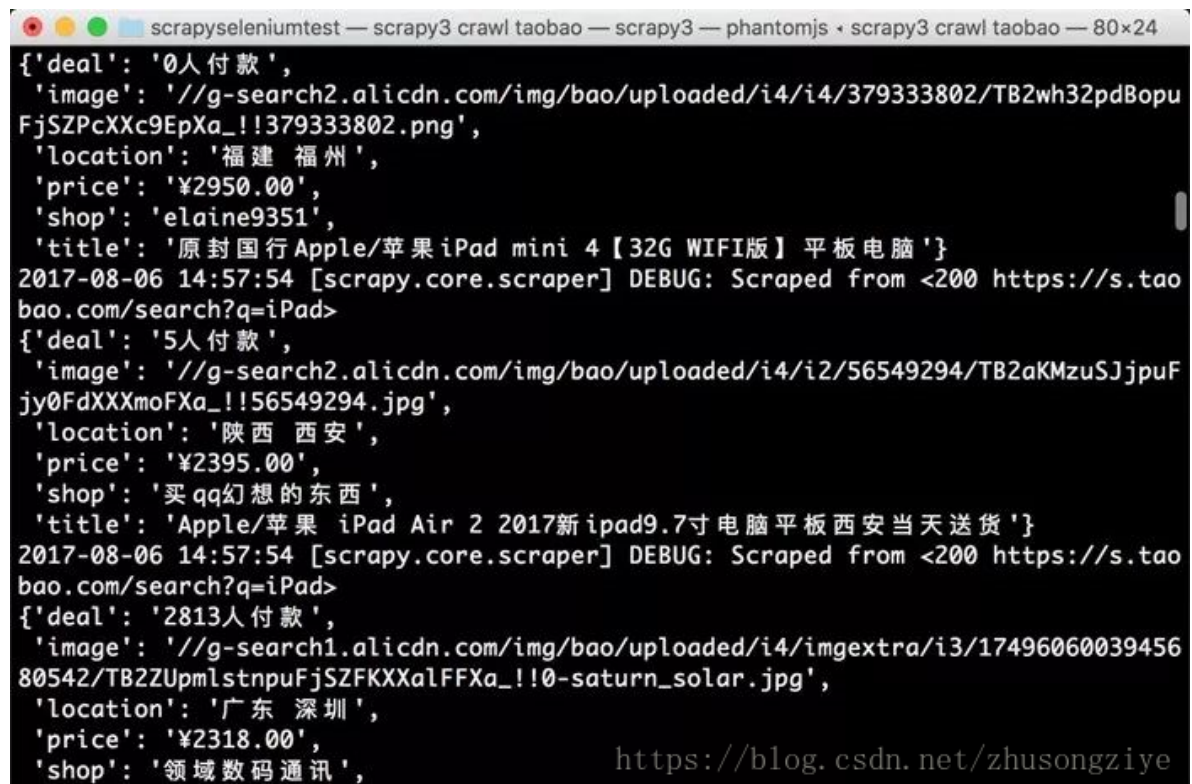
```
MONGO_URI = 'localhost'
MONGO_DB = 'taobao'
```

八、运行

整个项目就完成了，执行如下命令启动抓取即可：

```
scrapy crawl taobao
```

运行结果如下图所示。

A terminal window titled 'scrapyseleniumtest — scrapy3 crawl taobao — scrapy3 — phantomjs • scrapy3 crawl taobao — 80x24' displays the output of the 'scrapy crawl taobao' command. The output shows three JSON objects representing scraped items from Taobao, each containing fields like 'deal', 'image', 'location', 'price', 'shop', and 'title'. The items are for Apple iPad mini 4, Apple iPad Air 2, and another Apple iPad. Debug messages from scrapy.core.scrapers are also visible, indicating the source of the scraped data. A URL 'https://blog.csdn.net/zhusongziye' is visible in the bottom right corner of the terminal window.

```
scrapyseleniumtest — scrapy3 crawl taobao — scrapy3 — phantomjs • scrapy3 crawl taobao — 80x24
{'deal': '0人付款',
 'image': '//g-search2.alicdn.com/img/bao/uploaded/i4/i4/379333802/TB2wh32pdBopuFjSZPcXXc9EpXa_!!379333802.png',
 'location': '福建 福州',
 'price': '¥2950.00',
 'shop': 'elaine9351',
 'title': '原封国行 Apple/苹果 iPad mini 4【32G WIFI版】平板电脑'}
2017-08-06 14:57:54 [scrapy.core.scrapers] DEBUG: Scraped from <200 https://s.taobao.com/search?q=iPad>
{'deal': '5人付款',
 'image': '//g-search2.alicdn.com/img/bao/uploaded/i4/i2/56549294/TB2aKMzuSJjpuFjy0FdXXXmoFXa_!!56549294.jpg',
 'location': '陕西 西安',
 'price': '¥2395.00',
 'shop': '买qq幻想的东西',
 'title': 'Apple/苹果 iPad Air 2 2017新ipad9.7寸电脑平板西安当天送货'}
2017-08-06 14:57:54 [scrapy.core.scrapers] DEBUG: Scraped from <200 https://s.taobao.com/search?q=iPad>
{'deal': '2813人付款',
 'image': '//g-search1.alicdn.com/img/bao/uploaded/i4/imgextra/i3/1749606003945680542/TB2ZUpmlstnpuFjSZFKXXalFFXa_!!0-saturn_solar.jpg',
 'location': '广东 深圳',
 'price': '¥2318.00',
 'shop': '领域数码通讯',
 https://blog.csdn.net/zhusongziye
```

查看MongoDB，结果如下图所示。

db.getCollection('products').find({})

localhost localhost:27017 taobao

db.getCollection('products').find({})

products 0.002 sec. 2

Key	Value	Type
(1) ObjectId("5984a9f215c260883641a725")	{ 7 fields }	Object
_id	ObjectId("5984a9f215c260883641a725")	ObjectId
price	¥5480.00	String
title	Apple/苹果iPad Pro9.7寸32/128g wifi+4G大平板电脑Pr...	String
shop	我是单纯的呀	String
image	//g-search3.alicdn.com/img/bao/uploaded/i4/i2/9232...	String
deal	7人付款	String
location	上海	String
(2) ObjectId("5984a9f215c260883641a726")	{ 7 fields }	Object
_id	ObjectId("5984a9f215c260883641a726")	ObjectId
price	¥3020.00	String
title	Apple/苹果 iPad mini 4 WIFI 4G 32 64 128G 原封港版...	String
shop	静房王树楷	String
image	//g-search2.alicdn.com/img/bao/uploaded/i4/i4/13415...	String
deal	26人付款	String
location	上海	String
(3) ObjectId("5984a9f215c260883641a727")	{ 7 fields }	Object
(4) ObjectId("5984a9f215c260883641a728")	{ 7 fields }	Object
(5) ObjectId("5984a9f215c260883641a729")	{ 7 fields }	Object
(6) ObjectId("5984a9f215c260883641a72a")	{ 7 fields }	Object
(7) ObjectId("5984a9f215c260883641a72b")	{ 7 fields }	Object
(8) ObjectId("5984a9f215c260883641a72c")	{ 7 fields }	Object
(9) ObjectId("5984a9f215c260883641a72d")	{ 7 fields }	Object

<https://blog.csdn.net/zhongziye>

这样我们便成功在Scrapy中对接Selenium并实现了淘宝商品的抓取。

九、本节代码

本节代码地址为: <https://github.com/Python3WebSpider/ScrapySeleniumTest>。

十、结语

我们通过实现Downloader Middleware的方式实现了Selenium的对接。但这种方法其实是阻塞式的，也就是说这样就破坏了Scrapy异步处理的逻辑，速度会受到影响。为了不破坏其异步加载逻辑，我们可以使用Splash实现。

Scrapy 对接selenium

```
# 在爬虫启动后，就只打开一个chrom浏览器，以后都用这单独一个浏览器来爬数据

# 1 在爬虫中创建bro对象
from selenium import webdriver
bro = webdriver.Chrome()

# 2 中间件中使用：
from scrapy.http import HtmlResponse
spider.bro.get(request.url)
text=spider.bro.page_source
response=HtmlResponse(url=request.url,status=200,body=text.encode('utf-8'))
return response

# 3 关闭，在爬虫中
def close(self, reason):
    self.bro.close()
```

Scrapy部署到Docker

Python架构师手把手教你Scrapy 对接 Docker
环境配置问题可能一直会让我们头疼，包括如下几种情况。

我们在本地写好了一个Scrapy爬虫项目，想要把它放到服务器上运行，但是服务器上没有安装Python环境。

其他人给了我们一个Scrapy爬虫项目，项目使用包的版本和本地环境版本不一致，项目无法直接运行。

我们需要同时管理不同版本的Scrapy项目，如早期的项目依赖于Scrapy 0.25，现在的项目依赖于Scrapy 1.4.0。

在这些情况下，我们需要解决的就是环境的安装配置、环境的版本冲突解决等问题。

对于Python来说，VirtualEnv的确可以解决版本冲突的问题。但是，VirtualEnv不太方便做项目部署，我们还是需要安装Python环境，

如何解决上述问题呢？答案是用Docker。Docker可以提供操作系统级别的虚拟环境，一个Docker镜像一般都包含一个完整的操作系统，而这些系统内也有已经配置好的开发环境，如Python 3.6环境等。

我们可以直接使用此Docker的Python 3镜像运行一个容器，将项目直接放到容器里运行，就不用再额外配置Python 3环境。这样就解决了环境配置的问题。

我们也可以进一步将Scrapy项目制作成一个新的Docker镜像，镜像里只包含适用于本项目的Python环境。如果要部署到其他平台，只需要下载该镜像并运行就好了，因为Docker运行时采用虚拟环境，和宿主主机是完全隔离的，所以也不需要担心环境冲突问题。

如果我们能够把Scrapy项目制作成一个Docker镜像，只要其他主机安装了Docker，那么只要将镜像下载并运行即可，而不必再担心环境配置问题或版本冲突问题。

接下来，我们尝试把一个Scrapy项目制作成一个Docker镜像。

一、本节目标

我们要实现把前文Scrapy的入门项目打包成一个Docker镜像的过程。项目爬取的网址为：<http://quotes.toscrape.com/>。本章Scrapy入门一节已经实现了Scrapy对此站点的爬取过程，项目代码为：<https://github.com/Python3WebSpider/ScrapyTutorial>。如果本地不存在的话可以将代码Clone下来。

二、准备工作

请确保已经安装好Docker和MongoDB并可以正常运行。

三、创建Dockerfile

首先在项目的根目录下新建一个requirements.txt文件，将整个项目依赖的Python环境包都列出来，如下所示：

```
scrapy
pymongo
```

如果库需要特定的版本，我们还可以指定版本号，如下所示：

```
scrapy>=1.4.0
pymongo>=3.4.0
```

在项目根目录下新建一个Dockerfile文件，文件不加任何后缀名，修改内容如下所示：

```
FROM python:3.6
ENV PATH /usr/local/bin:$PATH
ADD . /code
WORKDIR /code
RUN pip3 install -r requirements.txt
CMD scrapy crawl quotes
```


第一行的FROM代表使用的Docker基础镜像，在这里我们直接使用python:3.6的镜像，在此基础上运行Scrapy项目。

第二行ENV是环境变量设置，将/usr/local/bin:\$PATH赋值给PATH，即增加/usr/local/bin这个环境变量路径。

第三行ADD是将本地的代码放置到虚拟容器中。它有两个参数：第一个参数是.，代表本地当前路径；第二个参数是/code，代表虚拟容器中的路径，也就是将本地项目所有内容放置到虚拟容器的/code目录下，以便于在虚拟容器中运行代码。

第四行WORKDIR是指定工作目录，这里将刚才添加的代码路径设成工作路径。这个路径下的目录结构和当前本地目录结构是相同的，所以我们可以直接执行库安装命令、爬虫运行命令等。

第五行RUN是执行某些命令来做一些环境准备工作。由于Docker虚拟容器内只有Python 3环境，而没有所需要的Python库，所以我们运行此命令来在虚拟容器中安装相应的Python库如Scrapy，这样就可以在虚拟容器中执行Scrapy命令了。

第六行CMD是容器启动命令。在容器运行时，此命令会被执行。在这里我们直接用scrapy crawl quotes来启动爬虫。

四、修改MongoDB连接

接下来我们需要修改MongoDB的连接信息。如果我们继续用localhost是无法找到MongoDB的，因为在Docker虚拟容器里localhost实际指向容器本身的运行IP，而容器内部并没有安装MongoDB，所以爬虫无法连接MongoDB。

这里的MongoDB地址可以有如下两种选择。

如果只想在本机测试，我们可以将地址修改为宿主机的IP，也就是容器外部的本机IP，一般是一个局域网IP，使用ifconfig命令即可查看。

如果要部署到远程主机运行，一般MongoDB都是可公网访问的地址，修改为此地址即可。

在本节中，我们的目标是将项目打包成一个镜像，让其他远程主机也可运行这个项目。所以我们直接将此处MongoDB地址修改为某个公网可访问的远程数据库地址，修改MONGO_URI如下所示：

```
MONGO_URI = 'mongodb://admin:admin123@120.27.34.25:27017'
```

此处地址可以修改为自己的远程MongoDB数据库地址。

这样项目的配置就完成了。

五、构建镜像

接下来我们构建Docker镜像，执行如下命令：

```
docker build -t quotes:latest .
```

执行过程中的输出如下所示：

```
Sending build context to Docker daemon 191.5 kB
Step 1/6 : FROM python:3.6
----> 968120d8cbe8
Step 2/6 : ENV PATH /usr/local/bin:$PATH
----> Using cache
----> 387abbba1189
Step 3/6 : ADD . /code
----> a844ee0db9c6
Removing intermediate container 4dc41779c573
Step 4/6 : WORKDIR /code
```



```
---> 619b2c064ae9
Removing intermediate container bcd7cd7f7337
Step 5/6 : RUN pip3 install -r requirements.txt
---> Running in 9452c83a12c5
...
Removing intermediate container 9452c83a12c5
Step 6/6 : CMD scrapy crawl quotes
---> Running in c092b5557ab8
---> c8101aca6e2a
Removing intermediate container c092b5557ab8
Successfully built c8101aca6e2a
```

这样的输出就说明镜像构建成功。这时我们查看一下构建的镜像，如下所示：

```
docker images
```

返回结果中的一行代码如下所示：

```
quotes   latest   41c8499ce210   2 minutes ago   769 MB
```

这就是我们新构建的镜像。

六、运行

镜像可以先在本地测试运行，我们执行如下命令：

```
docker run quotes
```

这样我们就利用此镜像新建并运行了一个Docker容器，运行效果完全一致，如下图所示略。

如果出现类似图上的运行结果，这就证明构建的镜像没有问题。

七、推送至Docker Hub

构建完成之后，我们可以将镜像Push到Docker镜像托管平台，如Docker Hub或者私有的Docker Registry等，这样我们就可以从远程服务器下拉镜像并运行了。

以Docker Hub为例，如果项目包含一些私有的连接信息（如数据库），我们最好将Repository设为私有或者直接放到私有的Docker Registry。

首先在<https://hub.docker.com>注册一个账号，新建一个Repository，名为quotes。比如，我的用户名为germey，新建的Repository名为quotes，那么此Repository的地址就可以用germey/quotes来表示。

为新建的镜像打一个标签，命令如下所示：

```
docker tag quotes:latest germey/quotes:latest
```

Push镜像到Docker Hub即可，命令如下所示：

```
docker push germey/quotes
```

Docker Hub便会出现新Push的Docker镜像了，如下图所示略。

如果我们想在其他的主机上运行这个镜像，主机上装好Docker后，可以直接执行如下命令：

```
docker run germey/quotes
```

这样就会自动下载镜像，启动容器运行。不需要配置Python环境，不需要关心版本冲突问题。

运行效果如下图所示略。

整个项目爬取完成后，数据就可以存储到指定的数据库中。

八、结语

我们讲解了将Scrapy项目制作成Docker镜像并部署到远程服务器运行的过程。使用此种方式，我们在本节开头所列出的问题都迎刃而解。