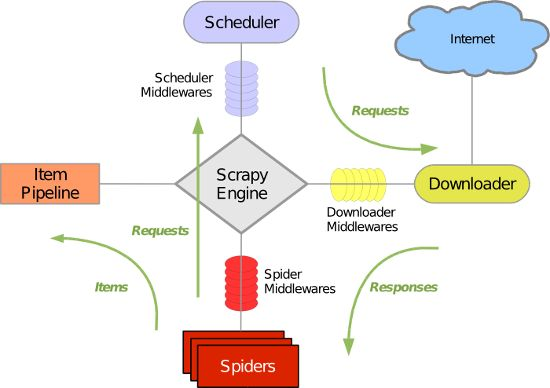
大家好，之前我们学习的爬虫程序都是通过requests库来实现的，爬取的数据量都很小，一旦要爬取的数据量很大时，单个requests的爬取速度就不能满足我们的需要了，这个时候就需要用到多线程或多进程或协程了，亦或者是Scrapy框架。

今天我们就来宏观讨论下Scrapy的原理，只有了解了Scrapy的运行原理，那么我们在学习的时候就更加有方向，在处理报错的时候也能更加得心应手，以及明确在哪些地方设置User-Agent、设置IP代理等等来应对反爬问题。

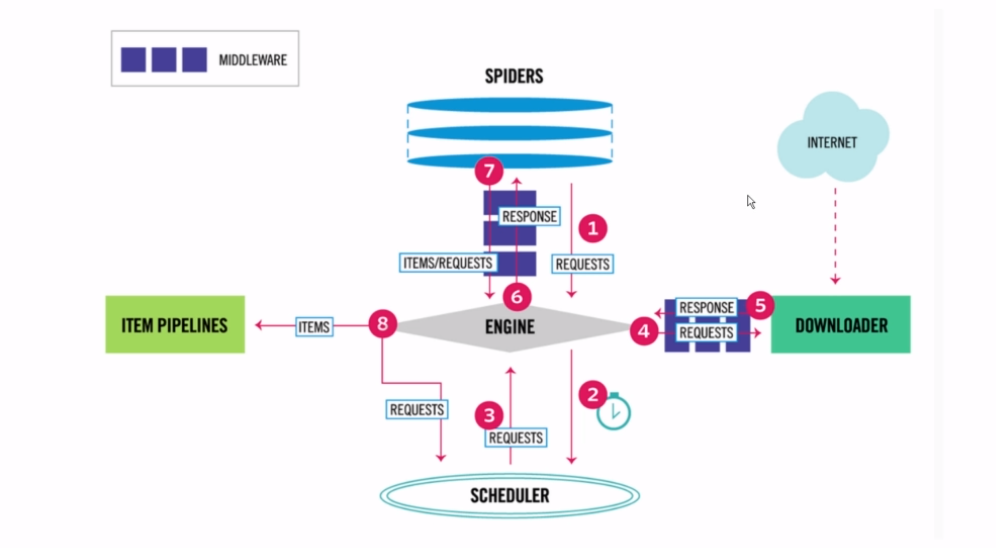
绝大多数讲解Scrapy的教程中，都能看到下面这张Scrapy的经典图：



这张图很好地指出了Scrapy的数据流向是怎么样的，使得我们能够站在一个宏观的位置上去了解Scrapy的运作原理。

遗憾地是，这张图没有指明数据流向的顺序是怎么样的，使得刚接触Scrapy的童鞋看得云里雾里。

我们再来看下面这张图，其中指明了Scrapy的数据流向顺序，接下来，我们将一步步来讨论Scrapy是怎么运作起来的。



第一步：最开始，我们书写了一个Spider，这个Spider会将Requests给yield出去，并发送给Engine

第二步：Engine获取到Spiders yield出来的Requests之后，什么都不做，直接发送给Scheduler（调度器）

第三步：在Scheduler中会对Engine传过来的Requests进行URL去重处理（昨天已经讲过了URL去重的几种方法，Scrapy用的是第三种），并且按照一定的优先级（例如先进先出，后进先出等等）将Requests再发送给Engine

**第四步：（重点）**Engine拿到Scheduler传过来的Requests之后，会将Requests传递给Downloader，而在这中间Requests会通过若干个Middleware的处理，我们把这里的Middleware称作**Downloader Middleware**。**这些Downloader Middleware的作用就是对Requests进行层层处理，例如设置Use-Agent、设置IP代理、集成selenium模拟登录等等一些应对反爬的措施，**我们可以来定义自己的Downloader Middleware。

举例：我们来定义一个随机产生User-Agent的Downloader Middleware，这样我们就能随机地给每个Requests的headers加上User-Agent了

# 实现功能：给每个Requests的请求头中添加随机User-Agent，代码看不懂没有关系，只是做个示例演示一下可以这么做来应对反爬

# settings.py中要修改的配置

USER\_AGETN\_LIST = ['Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3359.139 Safari/537.36']

DOWNLOADER\_MIDDLEWARES = {

'Douban.middlewares.DoubanDownloaderMiddleware': 543,

'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,

'Douban.middlewares.RandomUserAgentMiddleware': 1,

}

# middlewares.py中要书写的代码

import random

class RandomUserAgentMiddleware(object):

def \_\_init\_\_(self, crawler):

super(RandomUserAgentMiddleware, self).\_\_init\_\_()

self.user\_agent\_list = crawler.settings.get('USER\_AGETN\_LIST', [])

@classmethod

def from\_crawler(cls, crawler):

return cls(crawler)

def process\_request(self, request, spider):

user\_agent = random.choice(self.user\_agent\_list)

request.headers.setdefault('User-Agent', user\_agent)

# 当然，这种方式在实际编程中并不是一个好的选择，因为这需要我们自己维护一个USER\_AGENT\_LIST，而且这个USER\_AGENT\_LIST还是写死了的，一旦改变了其中的值，要使其生效，又得重新运行Scrapy项目。

# 一般我们会用到一个第三方库，叫做fake-useragent，来帮助我们随机生成User-Agent，具体安装和用法如下：

# https://github.com/hellysmile/fake-userag

# 实现功能：对于随机生成User-Agent的一个更好地选择

# settings.py中要修改的配置

DOWNLOADER\_MIDDLEWARES = {

'Douban.middlewares.DoubanDownloaderMiddleware': 543,

'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,

'Douban.middlewares.RandomUserAgentMiddleware': 1,

# middlewares.py中要书写的代码

from fake\_useragent import UserAgent

class RandomUserAgentMiddleware(object):

def \_\_init\_\_(self, crawler):

super(RandomUserAgentMiddleware, self).\_\_init\_\_()

self.ua = UserAgent()

@classmethod

def from\_crawler(cls, crawler):

return cls(crawler)

def process\_request(self, request, spider):

request.headers.setdefault('User-Agent', self.ua.rand

**第五步：（重点）**Downloader拿到经过了层层处理之后的Requests后，就会进行Download（发起请求），从而获取到响应，也就是Response对象，并将Response对象发送给Engine，而在这中间Response同样会通过若干个**Downloader Middleware**的处理，这些**Downloader Middleware**的作用就是对Response进行层层处理，同样，我们可以定义自己的**Downloader Middleware**来对这些Response进行处理

**第六步：（重点）**Engine拿到经过了层层处理之后的Response后，会发送给Spiders，而这个过程中，Response又会经过层层的Middleware，我们将这些Middleware称作**Spider Middleware**。同样的，我们可以定义自己的**Spider Middleware**来对这些Response进行处理

举例：Scrapy源码中已经提供了一些Spider Middleware，我们可以按照Scrapy提供的例子来书写我们自己的Spider Middleware，这里，我们只分析下Scrapy源码中的一个Spider Middleware，叫做HttpErrorMiddleware

# 实现功能：自定义对Response的处理，针对不同的Response状态码对Response进行过滤

# 这部分源码在scrapy.spidermiddlewares.httperror中

class HttpErrorMiddleware(object):

@classmethod

def from\_crawler(cls, crawler):

# 这个方法是必须要实现的，Middleware的manager会调用这个方法

return cls(crawler.settings)

def \_\_init\_\_(self, settings):

self.handle\_httpstatus\_all = settings.getbool('HTTPERROR\_ALLOW\_ALL')

self.handle\_httpstatus\_list = settings.getlist('HTTPERROR\_ALLOWED\_CODES')

def process\_spider\_input(self, response, spider):

# 真正的处理逻辑，判断是否将Response发送给Spider

if 200 <= response.status < 300: # common case

# 如果状态码在200-300之间就直接将Response发送给Spider

return

meta = response.meta

if 'handle\_httpstatus\_all' in meta:

# handle\_httpstatus\_all是我们可以自定义的一个配置，表示的内容为“是否选择自己处理所有状态码”，如果为True，那么无论Response的状态码是多少，都会返回给Spider

return

if 'handle\_httpstatus\_list' in meta:

# handle\_httpstatus\_list也是我们可以自定义的一个配置，表示的内容为“允许处理的状态码”，假设handle\_httpstatus\_list = [404, 403, 301]，那么状态码在这个列表里的Response都不会过滤掉，会返回给Spider

allowed\_statuses = meta['handle\_httpstatus\_list']

elif self.handle\_httpstatus\_list:

# 如果我们没有自定义handle\_httpstatus\_list，那么会获取Scrapy配置的默认值，为True

return

else:

allowed\_statuses = getattr(spider, 'handle\_httpstatus\_list', self.handle\_httpstatus\_list)

if response.status in allowed\_statuses:

return

raise HttpError(response, 'Ignoring non-200 response')

# 如果自定义的Response的过滤规则，那么就直接抛出一个异常，会被下面的process\_spider\_exception接收到，从而进行处理

def process\_spider\_exception(self, response, exception, spider):

# 处理抛出的异常

if isinstance(exception, HttpError):

spider.crawler.stats.inc\_value('httperror/response\_ignored\_count')

spider.crawler.stats.inc\_value(

'httperror/response\_ignored\_status\_count/%s' % response.status

)

logger.info(

"Ignoring response %(response)r: HTTP status code is not handled or not allowed",

{'response': response}, extra={'spider': spider},

)

return

第七步：这样Spiders就能对Response进行解析，也就是进入了Spider中的parse函数，对Response进行解析之后，可能会获取到两种结果，一种是yield出新的Requests，另一种是yield出items，Spiders会将获取到的结果发送给Engine，同样的，也会经过层层的 **Spider Middleware**，我们也可以自定义自己的**Spider Middleware**来对Response解析之后的结果进行处理

第八步：Engine拿到Spiders传递过来的结果后，会对结果进行判断，如果是Requests，那么就发送给Scheduler，如果是items，那么就发送给Item和Pipelines

按照上述步骤，往复循环就构成了一个完整的Scrapy项目。

我们在宏观地了解了Scrapy的完整数据流向之后，就能很明确地知道我们该在什么地方书写什么代码，该在什么地方自由地做出改动从而应对反爬，还有一个很重要的地方在于，方便我们去看Scrapy的源码。

在python的学习过程中，尤其是在学习某些框架的时候，要想获得进步，仅仅掌握框架的使用是不够的，而是得精通这个框架，而要精通这个框架，你就得去看框架的源码，通过看源码一方面能够使得自己对这个框架的认知更深刻，另一方面，优秀的框架有太多的地方值得我们去学习了，尤其是设计思想。

今天的分享到这里就结束了，大家加油。