**scrapy框架学习**

# 介绍

scrapy 是一个开源和协作的框架，用于爬虫。

以快速、简单、可扩展的方式。

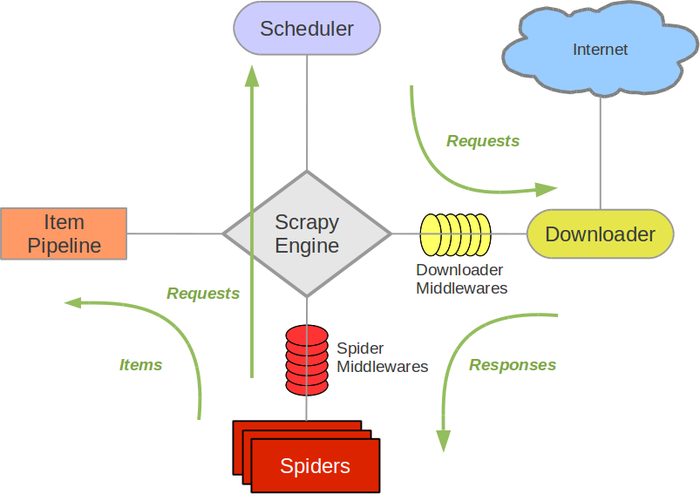
项目地址：https://github.com/scrapy/scrapy

官方网站：https://scrapy.org/

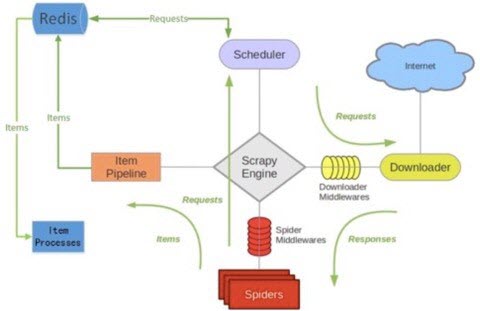
官方文档：https://docs.scrapy.org/en/latest/

官方教程：https://docs.scrapy.org/en/latest/intro/tutorial.html

架构：



scrapy-redis架构：



# 安装

基于 1.5.0

使用 pip install scrapy==1.5.0

# 各个模块

## settings，配置文件

所有的key 都是全部大写！

BOT\_NAME = 'firstscrapy' # 项目名

ROBOTSTXT\_OBEY = Ture，是否遵守 robots.txt， 修改为False

DEFAULT\_REQUEST\_HEADERS ： 默认的请求headers

PROXIES = [

{'ip\_port': '111.8.60.9:8123', 'user\_pass': ''},

{'ip\_port': '101.71.27.120:80', 'user\_pass': ''},

] # 代理 ，在downloder middleware 的 代理中间件中 proxy = random.choice(PROXIES) request.meta['proxy'] = "http://%s" % proxy['ip\_port']

SPIDER\_MIDDLEWARES：爬虫中间层

DOWNLOADER\_MIDDLEWARES：下载中间层

ITEM\_PIPELINES = {'项目名.pipelines.PipeLine类名': 300,}

# 开发模式时，把下面的注释取消掉，启用缓存，可以提高调试效率

# 同样的请求，如果 缓存 当中有保存内容的话，不会去进行网络请求，直接从缓存中返回

# 记住：开发环境下启用！！！！部署时一定要注释掉！！！

#HTTPCACHE\_ENABLED = True

#HTTPCACHE\_EXPIRATION\_SECS = 0

#HTTPCACHE\_DIR = 'httpcache'

#HTTPCACHE\_IGNORE\_HTTP\_CODES = [

#HTTPCACHE\_STORAGE = 'scrapy.extensions.httpcache.FilesystemCacheStorage'

# 日志管理

LOG\_ENABLED 默认: True，启用logging

LOG\_ENCODING 默认: 'utf-8'，logging使用的编码

LOG\_FILE 默认: None，在当前目录里创建logging输出文件的文件名，例如：LOG\_FILE = 'log.txt'

配置了这个文件，就不会在控制台输出日志了

LOG\_LEVEL 默认: 'DEBUG'，log的最低级别，共五级：

CRITICAL - 严重错误

ERROR - 一般错误

WARNING - 警告信息

INFO - 一般信息 print 属于这个 info

DEBUG - 调试信息

LOG\_STDOUT 默认: False 如果为 True，进程所有的标准输出(及错误)将会被重定向到log中。

例如，执行 print("hello") ，其将会在Scrapy log中显示

# 并发 ,现在 = 号右边的 value 就是默认值

CONCURRENT\_ITEMS = 100 # 并发处理 items 的最大数量

CONCURRENT\_REQUESTS = 16 # 并发下载request页面的最大数量

CONCURRENT\_REQUESTS\_PER\_DOMAIN = 8 # 并发下载任何单域的最大数量, baidu.com , sina.cn 各 8 个

CONCURRENT\_REQUESTS\_PER\_IP = 0 # 并发 每个IP 请求的最大数量，

DOWNLOAD\_DELAY = 0.25 # 单位秒，支持小数，一般都是随机范围：

0.5\*DOWNLOAD\_DELAY 到 1.5\*DOWNLOAD\_DELAY 之间

CONCURRENT\_REQUESTS\_PER\_IP 不为0时，这个延时是针对每个IP，而不是每个域

## 爬虫类

继承自scrapy.Spider，用于构造Request对象给Scheduler

属性

name：爬虫的名字，必须唯一 ，必须写！

start\_urls：爬虫初始爬取的链接列表

custom\_settings = {} # 自定义的setting配置

方法

start\_requests：启动爬虫的时候调用，爬取urls的链接，可以省略

parse：response到达spider的时候默认调用，如果在Request对象配置了callback函数，则不会调用，

parse方法可以迭代返回Item或Request对象，如果返回Request对象，则会进行增量爬取

## items

定义：

class Product(scrapy.Item):

name = scrapy.Field()

title = scrapy.Field()

调用：

和dict一样的调用

product = Product( , title='pc title')

# 像字典一样的使用：

print(product['name'])

print(product.get('name'))

product['title'] = 'new title'

可以这样转换为字典：dict(product)

## pipelines

必须在settings中，添加

ITEM\_PIPELINES = {

'first\_scrapy.pipelines.FirstScrapyPipeline': 300, # 优先级，数字越小，

优先级越高，越早调用范围 0-1000

}

对象如下：

class FirstScrapyPipeline(object):

def process\_item(self, item, spider):

return item

方法：

process\_item(self, item, spider)： 处理item的方法， 必须有的！！！

参数：

item (Item object or a dict) ： 获取到的item

spider (Spider object) ： 获取到item的spider

返回 一个dict或者item

open\_spider(self, spider) ： 当spider启动时，调用这个方法

参数：

spider (Spider object) – 启动的spider

close\_spider(self, spider)： 当spider关闭时，调用这个方法

参数：

spider (Spider object) – 关闭的spider

@classmethod

from\_crawler(cls, crawler)

参数：

crawler (Crawler object) – 使用这个pipe的爬虫crawler

4.1 返回 item 的例子：

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)

4.2 写入文件的例子：

import json

class JsonWriterPipeline(object):

def open\_spider(self, spider):

self.file = open('items.jl', 'w')

def close\_spider(self, spider):

self.file.close()

def process\_item(self, item, spider):

line = json.dumps(dict(item)) + "\n"

self.file.write(line)

return item

4.3 写入 mongodb 的例子：

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):

# 必须在settings中 配置 MONGO\_URI 和 MONGO\_DATABASE

return cls(

mongo\_uri=crawler.settings.get('MONGO\_URI'),

# items 是默认值，如果settings当中没有配置 MONGO\_DATABASE ，那么 mongo\_db = 'items'

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\_one(dict(item))

return item

4.4 网页快照的例子

import scrapy

import hashlib

from urllib.parse import quote

class ScreenshotPipeline(object):

"""Pipeline that uses Splash to render screenshot of

every Scrapy item."""

SPLASH\_URL = "http://localhost:8050/render.png?url={}"

def process\_item(self, item, spider):

encoded\_item\_url = quote(item["url"])

screenshot\_url = self.SPLASH\_URL.format(encoded\_item\_url)

request = scrapy.Request(screenshot\_url)

dfd = spider.crawler.engine.download(request, spider)

dfd.addBoth(self.return\_item, item)

return dfd

def return\_item(self, response, item):

if response.status != 200:

# Error happened, return item.

return item

# Save screenshot to file, filename will be hash of url.

url = item["url"]

url\_hash = hashlib.md5(url.encode("utf8")).hexdigest()

filename = "{}.png".format(url\_hash)

with open(filename, "wb") as f:

f.write(response.body)

# Store filename in item.

item["screenshot\_filename"] = filename

return item

4.5 去重复值的pipe

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

# 第一个项目

1、在命令行中，切换到 项目目录：譬如，f:\py\_study>

执行命令：scrapy startproject first\_scrapy

将在 f:\py\_study 路径下建立一个 first\_scrapy 项目文件夹

文件夹结构如下：

scrapy.cfg # 部署的配置文件，不需要修改

first\_scrapy/

\_\_init\_\_.py

items.py # 类一定要继承scrapy.Item，定义我们需要的结构化数据，和ORM有点类似，使用相当于dict

first\_item = FirstscrapyItem()

name = first\_item['name']

name1 = first\_item.get('name')

first\_item['name'] = 'lucy'

# 可以转换为字典

dict\_item = dict(first\_item)

# 格式是： dict\_item = {name:'terry', age:10, sex:'1'}

middlewares.py # 中间件，相当于钩子，可以对爬取前后做预处理，如修改请求header，url过滤等

分 downloadermiddleware、spidermiddleware

pipelines.py # 数据处理，将items中结构化的数据进行处理

settings.py # 项目配置文件，key=value 的方式，key必须全部大写！

包括所有的配置，一些公用的常量也可以写在里面

spiders/ # 爬虫模块的目录，负责配置需要爬取的数据和爬取规则，以及解析数据，

并且把结构化数据，return 到 pipelines 模块处理

\_\_init\_\_.py

2、在 spiders 中新建：

quotes.py：

import scrapy

# 必须继承 scrapy.Spider

class QuotesSpider(scrapy.Spider):

# 用于区别Spider。 该名字必须是唯一的，您不可以为不同的Spider设定相同的名字。

name = "quotes"

def start\_requests(self):

# 包含了Spider在启动时进行爬取的url列表。 因此，第一个被获取到的页面将是其中之一。

# 后续的URL则从初始的URL获取到的数据中提取。

urls = [

'http://quotes.toscrape.com/page/1/',

'http://quotes.toscrape.com/page/2/',

]

for url in urls:

# 必须使用 yield

# Request 是 scrapy 自定义的类

# callback， 获取到response 之后的 回调函数

yield scrapy.Request(url=url, callback=self.parse)

# 是spider的一个方法。 被调用时，每个初始URL完成下载后生成的 Response 对象将会作为唯一的参数传递给该函数。

def parse(self, response):

page = response.url.split("/")[-2]

filename = 'quotes-%s.html' % page

with open(filename, 'wb') as f:

f.write(response.body)

self.log('Saved file %s' % filename)

或

class BlogSpider(scrapy.Spider):

name = 'blogspider'

# 允许访问的域名，可以不写

allowed\_domains = ['scrapinghub.com']

start\_urls = ['https://blog.scrapinghub.com']

def parse(self, response):

for title in response.css('h2.entry-title'):

yield {'title': title.css('a ::text').extract\_first()}

for next\_page in response.css('div.prev-post > a'):

yield response.follow(next\_page, self.parse)

补充：

数据流，所有中间件都启动：

1、启动 spiders

2、spiders 包装一个request， 发送到 scheduler

2.1 request进入 scheduler 之前，会先到 scheduler middleware 进行处理，

处理后，再发送给 scheduler

3、scheduler 接收到 request 后，将之放入到一个 队列

4、engine 向 scheduler 申请 request，得到后，将request

发送给 downloader

4.1 request 从 scheduler 出来后，先到 scheduler middeware 进行处理，

再传给 scheduler middeware

4.2 request 从 scheduler middeware 出来后，需要到 downloader middleware

进行处理，再传给 downloader

5、downloader 接收到 request 之后，访问对应的http资源，接收 response

6、downloader 接收到 response 之后，将 response 发送给 spider

6.1 response 从 downloader 出来后，先经过 downloader middleware 处理

6.2 response 从 ownloader middleware 处理后，经过 spider middleware 处理，

再 传给 spider

7、spider 接收到reponse之后，解析内容，

7.1 url 资源，需要再次请求的，继续包装成 request ，继续第二个步骤

7.2 非 url 资源，就是 数据，结构化成 item ，向后传

8、item 从 spider 出来后，经过 spider middleware ，处理后 ，发送给

item pipeline

9、pipeline 接收到 item 后，读取其中的数据，进行数据持久化

# 运行

1、命令行中运行：

命令行 中 进入到 first\_scrapy 目录中，执行：

scrapy crawl quotes

quotes是spider类中的 name 属性

2、pycharm 运行

在 项目 根目录 添加 run.py 文件：

from first\_scrapy.spiders.quotes import QuotesSpider

from scrapy.crawler import CrawlerProcess

from scrapy.utils.project import get\_project\_settings

# 获取settings.py模块的设置

settings = get\_project\_settings()

process = CrawlerProcess(settings=settings)

# 可以添加多个spider

# process.crawl(Spider1)

# process.crawl(Spider2)

process.crawl(QuotesSpider)

# 启动爬虫，会阻塞，直到爬取完成

process.start()

或者：

from scrapy.cmdline import execute

#设置工程命令

import sys

import os

#设置工程路径，在cmd 命令更改路径而执行scrapy命令调试

#获取run文件的父目录，os.path.abspath(\_\_file\_\_) 为\_\_file\_\_文件目录

sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))

execute(["scrapy","crawl","quotes" ])

# spider的 parse

1、ItemLoader

def parse(self, response):

# text = response.text

# tree = etree.HTML(text)

# product = Product()

# product['name'] = tree.xpath('//div[@class="product\_name"]')[0].text + tree.xpath('//div[@class="product\_title"]')[0].text

# product['price'] = tree.xpath('//p[@id="price"]')[0].text

# product['stock'] = tree.xpath('//p#stock')[0].text

# product['last\_updated'] = 'today'

# return product

l = ItemLoader(item=Product(), response=response)

l.add\_xpath('name', '//div[@class="product\_name"]')

l.add\_xpath('title', '//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()

2、selector

response.selector.xpath('//span/text()').extract()

# 结果是 list，但是一般情况我们都是只取到一个值，如 ['想要的值']， extract\_first 取第一个

response.css('title::text')

response.css('img').xpath('@src').extract()

response.xpath('//div[@id="not-exists"]/text()').extract\_first()

response.xpath('//div[@id="not-exists"]/text()').extract\_first(default='not-found')

response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s\*(.\*)')

3、meta

yield Request(novel\_url, self.parse\_next, meta={'name':name, 'age':age})

在 parse\_next 中 通过 response.meta['name'] 获取参数

传递参数到下一个 parse\_next 函数

# middleware

1、settings中配置

DOWNLOADER\_MIDDLEWARES = {

'myproject.middlewares.CustomDownloaderMiddleware': 543,

'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None, # 不启用

}

2、process\_request(request, spider)

2.1 如果其返回 None ，Scrapy将继续处理该request，执行其他的中间件的相应方法，

直到合适的下载器处理函数(download handler)被调用， 该request被执行(其response被下载)。

2.2 如果其返回 Response 对象，Scrapy将不会调用 任何 其他的 process\_request() 或 process\_exception() 方法，

或相应地下载函数； 其将返回该response。 已安装的中间件的 process\_response() 方法则会在每个response返回时被调用。

2.3 如果其返回 Request 对象，Scrapy则停止调用 process\_request方法并重新调度返回的request。

当新返回的request被执行后， 相应地中间件链将会根据下载的response被调用。

2.4 如果其raise一个 IgnoreRequest 异常，则安装的下载中间件的 process\_exception() 方法会被调用。

如果没有任何一个方法处理该异常， 则request的errback(Request.errback)方法会被调用。

如果没有代码处理抛出的异常， 则该异常被忽略且不记录(不同于其他异常那样)。

3、process\_response(request, response, spider)

3.1 如果其返回一个 Response (可以与传入的response相同，也可以是全新的对象)，

该response会被在链中的其他中间件的 process\_response() 方法处理。

3.2 如果其返回一个 Request 对象，则中间件链停止， 返回的request会被重新调度下载。

处理类似于 process\_request() 返回request所做的那样。

3.3 如果其抛出一个 IgnoreRequest 异常，则调用request的errback(Request.errback)。

如果没有代码处理抛出的异常，则该异常被忽略且不记录(不同于其他异常那样)。

4、process\_exception(request, exception, spider)

5、from\_crawler(cls, crawler)