# 1.4 scrapy框架与Spider类

pip install pillow

from PIL import Image

新建一个单独的文件夹保存项目文件，在这个项目文件夹中新建scrapy项目

scrapy startproject mySpider

使用pycharm打开mySpider项目

scrapy.cfg是项目的设置文件，并发数，管道文件等设置。

/mySpider中\_\_init\_\_.py

item.py

pip install --upgrade pip

pip install Scrapy

## 入门案例

**学习目标**

* 创建一个Scrapy项目
* 定义提取的结构化数据(Item)
* 编写爬取网站的 Spider 并提取出结构化数据(Item)
* 编写 Item Pipelines 来存储提取到的Item(即结构化数据)

### 一、新建项目(scrapy startproject)

* 在开始爬取之前，必须创建一个新的Scrapy项目。进入自定义的项目目录中，运行下列命令：

**scrapy startproject mySpider**

* 其中， mySpider 为项目名称，可以看到将会创建一个 mySpider 文件夹，目录结构大致如下：

下面来简单介绍一下各个主要文件的作用：

scrapy.cfg ：项目的配置文件

mySpider/ ：项目的Python模块，将会从这里引用代码

mySpider/items.py ：项目的目标文件

mySpider/pipelines.py ：项目的管道文件

mySpider/settings.py ：项目的设置文件

BOT\_NAME = **'mySpider'**

SPIDER\_MODULES = [**'mySpider.spiders'**]

NEWSPIDER\_MODULE = **'mySpider.spiders'**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#USER\_AGENT = 'mySpider (+http://www.yourdomain.com)'*

*# Obey robots.txt rules*

*#是否遵循robots协议*

*# ROBOTSTXT\_OBEY = True*

*# Configure maximum concurrent requests performed by Scrapy (default: 16)*

*#爬虫的并发量，可以同时下载的数据*

*# CONCURRENT\_REQUESTS = 32*

*# Configure a delay for requests for the same website (default: 0)*

*# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay*

*# See also autothrottle settings and docs*

*#下载延迟，一般不启动，与time.sleep功能类似*

*#DOWNLOAD\_DELAY = 3*

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

*#是否启动cookie，模拟登陆时要打开，使用爬虫爬取数据时就不打开，服务器就不会根据cookie来检测我们爬取的内容了*

*#COOKIES\_ENABLED = False*

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# Override the default request headers:*

*#默认的请求报头，可以在这里设置user-agent，但如果网站对请求头进行了限制，就要在下载中间件中设置随机的请求头。*

*#DEFAULT\_REQUEST\_HEADERS = {*

*# 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8',*

*#只显示英文的选项关闭。*

*# 'Accept-Language': 'en',*

*#}*

*# Enable or disable spider middlewares*

*# See http://scrapy.readthedocs.org/en/latest/topics/spider-middleware.html*

*#爬虫的中间件一般不用，下载的中间件用的较多*

*#SPIDER\_MIDDLEWARES = {*

*# 'mySpider.middlewares.MyspiderSpiderMiddleware': 543,*

*#}*

*# Enable or disable downloader middlewares*

*# See http://scrapy.readthedocs.org/en/latest/topics/downloader-middleware.html*

*#下载中间件，543是优先级，1~1000，数字越小就优先级越高，可以设置多个，根据优先级进行选择*

*#DOWNLOADER\_MIDDLEWARES = {*

*# 'mySpider.middlewares.MyCustomDownloaderMiddleware': 543,*

*#}*

*# Enable or disable extensions*

*# See http://scrapy.readthedocs.org/en/latest/topics/extensions.html*

*#EXTENSIONS = {*

*# 'scrapy.extensions.telnet.TelnetConsole': None,*

*#}*

*# Configure item pipelines*

*# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html*

*#管道文件，决定了下载的数据如何进行处理*

*#ITEM\_PIPELINES = {*

*# 'mySpider.pipelines.MyspiderPipeline': 300,*

*#}*

mySpider/spiders/ ：存储爬虫代码目录

### 二、明确目标(mySpider/items.py)

我们打算抓取：<http://www.itcast.cn/channel/teacher.shtml> 网站里的所有讲师的姓名、职称和个人信息。

1. 打开mySpider目录下的items.py
2. Item 定义结构化数据字段，用来保存爬取到的数据，有点像Python中的dict，但是提供了一些额外的保护减少错误。
3. 可以通过创建一个 scrapy.Item 类， 并且定义类型为 scrapy.Field的类属性来定义一个Item（可以理解成类似于ORM的映射关系）。
4. 接下来，创建一个ItcastItem 类，和构建item模型（model）。

**import** scrapy

**class** ItcastItem(scrapy.Item):

*# define the fields for your item here like:*

*#姓名*

name = scrapy.Field()

*#职称*

title = scrapy.Field()

*#个人简介*

info = scrapy.Field()

### 三、制作爬虫（spiders/itcastSpider.py）

**爬虫功能要分两步：**

#### 1. 爬数据

在当前目录下输入命令，将在mySpider/spider目录下创建一个名为itcast的爬虫，并指定爬取域的范围：

**注意爬虫名和项目名不能相同**

scrapy genspider itcast "itcast.cn"

打开 mySpider/spider目录里的 itcast.py，默认增加了下列代码:

import scrapy

class ItcastSpider(scrapy.Spider):

name = "itcast"

allowed\_domains = ["itcast.cn"]

start\_urls = (

'http://www.itcast.cn/',

)

def parse(self, response):

pass

其实也可以由我们自行创建itcast.py并编写上面的代码，只不过使用命令可以免去编写固定代码的麻烦

要建立一个Spider， 你必须用scrapy.Spider类创建一个子类，并确定了三个强制的属性 和 一个方法。

* name = "" ：这个爬虫的识别名称，必须是唯一的，在不同的爬虫必须定义不同的名字。
* allow\_domains = [] 是搜索的域名范围，也就是爬虫的约束区域，规定爬虫只爬取这个域名下的网页，不存在的URL会被忽略。
* start\_urls = () ：爬取的URL元祖/列表。爬虫从这里开始抓取数据，所以，第一次下载的数据将会从这些urls开始。其他子URL将会从这些起始URL中继承性生成。可以使用元组也可以使用列表，使用元组时，要在url的后面加上一个逗号，不然就会出错。
* parse(self, response) ：解析的方法，每个初始URL完成下载后将被调用，调用的时候传入从每一个URL传回的Response对象来作为唯一参数，主要作用如下：
  1. 负责解析返回的网页数据(response.body)，提取结构化数据(生成item)
  2. 生成需要下一页的URL请求。

将start\_urls的值修改为需要爬取的第一个url

start\_urls = ("http://www.itcast.cn/channel/teacher.shtml",)

修改parse()方法

def parse(self, response):

filename = "teacher.html"

open(filename, 'w').write(response.body)

##### itcast各科教师爬虫

*# -\*- coding: utf-8 -\*-*

**import** scrapy

*#创建一个爬虫类*

**class** ItcastSpider(scrapy.Spider):

*#爬虫名，在执行时就是使用的这个名字*

name = **"itcast"**

*#允许爬虫爬取的范围，可以写多个值*

allowed\_domains = [**"itcast.cn"**]

*#爬虫起始的url地址，可以使用元组或列表。*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml#'**]

*#定义一个parse方法，名字不能修改*

*#处理返回的response*

**def** parse(self, response):

**with** open(**"teacher.html"**, **"w"**, encoding=**"utf-8"**) **as** f:

*#scrapy中的用法，不能使用response，要写body*

f.write(response.body.decode(**"utf-8"**))

然后运行一下看看，在mySpider目录下执行：

scrapy crawl itcast

运行出错：

ImportError: No module named 'win32api'

使用pip安装win32

pip install pypiwin32

再次运行，就能成功了。

是的，就是 itcast，看上面代码，它是 ItcastSpider 类的 name 属性，也就是使用 scrapy genspider命令的唯一爬虫名。

运行之后，如果打印的日志出现 [scrapy] INFO: Spider closed (finished)，代表执行完成。 之后当前文件夹中就出现了一个 teacher.html 文件，里面就是我们刚刚要爬取的网页的全部源代码信息。

# 注意，Python2.x默认编码环境是ASCII，当和取回的数据编码格式不一致时，可能会造成乱码；

# 我们可以指定保存内容的编码格式，一般情况下，我们可以在代码最上方添加：

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

# 这三行代码是Python2.x里解决中文编码的万能钥匙，经过这么多年的吐槽后Python3学乖了，默认编码是Unicode了...(祝大家早日拥抱Python3)

#### 2. 取数据

* 爬取整个网页完毕，接下来的就是的取过程了，首先观察页面源码：

<div class="li\_txt">

<h3> xxx </h3>

<h4> xxxxx </h4>

<p> xxxxxxxx </p>

是不是一目了然？直接上XPath开始提取数据吧。

父节点：

response.xpath('//div[@class=”li\_txt”]')

子节点：

xpath('./h3/text()')

xpath('./h4/text()')

xpath('./p/text()')

* 我们之前在mySpider/items.py 里定义了一个ItcastItem类。 这里引入进来

from mySpider.items import ItcastItem

* 然后将我们得到的数据封装到一个 ItcastItem 对象中，可以保存每个老师的属性：

##### itcast各科教师爬虫取数据

**import** scrapy

**from** mySpider.items **import** ItcastItem

*# -\*- coding: utf-8 -\*-*

*#创建一个爬虫类*

**class** ItcastSpider(scrapy.Spider):

*#爬虫名，在执行时就是使用的这人名字*

name = **"itcast"**

*#允许爬虫爬取的范围，可以写多个值*

allowed\_domains = [**"itcast.cn"**]

*#爬虫起始的url地址，可以使用元组或列表。*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml#'**]

start\_urls = [

**"http://www.itcast.cn/channel/teacher.shtml#aandroid"**,

**"http://www.itcast.cn/channel/teacher.shtml#ac"**,

**"http://www.itcast.cn/channel/teacher.shtml#acloud"**,

**"http://www.itcast.cn/channel/teacher.shtml#aios"**,

**"http://www.itcast.cn/channel/teacher.shtml#ajavaee"**,

**"http://www.itcast.cn/channel/teacher.shtml#anetmarket"**,

**"http://www.itcast.cn/channel/teacher.shtml#aphp"**,

**"http://www.itcast.cn/channel/teacher.shtml#apython"**,

**"http://www.itcast.cn/channel/teacher.shtml#astack"**,

**"http://www.itcast.cn/channel/teacher.shtml#aui"**,

**"http://www.itcast.cn/channel/teacher.shtml#aweb"**

]

*#定义一个parse方法，名字不能修改*

*#处理返回的response*

*# def parse(self, response):*

*# with open("teacher.html", "w", encoding="utf-8") as f:*

*# #scrapy中的用法，不能使用response，要使用response.body*

*# f.write(response.body.decode("utf-8"))*

**def** parse(self, response):

*#open("teacher.html","wb").write(response.body).close()*

*# 存放老师信息的集合*

teacherItems = []

*#通过scrapy自带的xpath匹配出所有老师的根节点*

teacher\_list = response.xpath(**"//div[@class='li\_txt']"**)

*#遍历根节点集合*

**for** each **in** teacher\_list:

*# 将我们得到的数据封装到一个 `ItcastItem` 对象*

*#实例化一个对象，用来保存数据*

item = ItcastItem()

*#extract()方法将匹配出来的结果转换成unicode字符串*

*#不加extract()结果为xpath匹配出来的对象*

name = each.xpath(**"./h3/text()"**).extract()

title = each.xpath(**"./h4/text()"**).extract()

info = each.xpath(**"./p/text()"**).extract()

*#xpath返回的是包含所有匹配对象的的列表*

*# print(name[0],title[0],info[0],sep="\n==========\n")*

item[**'name'**] = name[0]

item[**'title'**] = title[0]

item[**'info'**] = info[0]

*#*

teacherItems.append(item)

*# 直接返回最后数据*

**return** teacherItems

* 我们暂时先不处理管道，后面会详细介绍。

#### 3. 保存数据

scrapy保存信息的最简单的方法主要有四种，-o 输出指定格式的文件，，命令如下：

# json格式，默认为Unicode编码

scrapy crawl itcast -o teachers.json

或scrapy crawl itcast -o teacher -t json

# json lines格式，默认为Unicode编码

scrapy crawl itcast -o teachers.jsonl

# csv 逗号表达式，可用Excel打开

# windows的Excel是gbk格式的，要在itcast.py文件中把item的内容encode为gbk格式的。extract()提取出来的是unicode字符，用encode改为gbk格式。

scrapy crawl itcast -o teachers.csv

# xml格式

scrapy crawl itcast -o teachers.xml

##### itcast各科教师爬虫yield

如果将代码改成下面形式，结果完全一样。

请思考 yield 在这里的作用：

from mySpider.items import ItcastItem

def parse(self, response):

#open("teacher.html","wb").write(response.body).close()

# 存放老师信息的集合

#items = []

for each in response.xpath("//div[@class='li\_txt']"):

# 将我们得到的数据封装到一个 `ItcastItem` 对象

item = ItcastItem()

#extract()方法返回的都是unicode字符串

name = each.xpath("h3/text()").extract()

title = each.xpath("h4/text()").extract()

info = each.xpath("p/text()").extract()

#xpath返回的是包含一个元素的列表

item['name'] = name[0]

item['title'] = title[0]

item['info'] = info[0]

#items.append(item)

#将获取的数据交给pipelines进行处理

yield item

# 返回数据，不经过pipeline

#return items

## Spider类

Spider类定义了如何爬取某个(或某些)网站。包括了爬取的动作(例如:是否跟进链接)以及如何从网页的内容中提取结构化数据(爬取item)。 换句话说，Spider就是您定义爬取的动作及分析某个网页(或者是有些网页)的地方。

class scrapy.Spider是最基本的类，所有编写的爬虫必须继承这个类。

主要用到的函数及调用顺序为：

\_\_init\_\_() : 初始化爬虫名字和start\_urls列表

start\_requests() 调用make\_requests\_from url():生成Requests对象交给Scrapy下载并返回response

parse() : 解析response，并返回Item或Requests（需指定回调函数）。Item传给Item pipline持久化，而Requests交由Scrapy下载，并由指定的回调函数处理（默认parse())，一直进行循环，直到处理完所有的数据为止。

### 源码参考

#所有爬虫的基类，用户定义的爬虫必须从这个类继承

class Spider(object\_ref):

#定义spider名字的字符串(string)。spider的名字定义了Scrapy如何定位(并初始化)spider，所以其必须是唯一的。

#name是spider最重要的属性，而且是必须的。

#一般做法是以该网站(domain)(加或不加 后缀 )来命名spider。 例如，如果spider爬取 mywebsite.com ，该spider通常会被命名为 mywebsite

name = None

#初始化，提取爬虫名字，start\_ruls

def \_\_init\_\_(self, name=None, \*\*kwargs):

if name is not None:

self.name = name

# 如果爬虫没有名字，中断后续操作则报错

elif not getattr(self, 'name', None):

raise ValueError("%s must have a name" % type(self).\_\_name\_\_)

# python 对象或类型通过内置成员\_\_dict\_\_来存储成员信息

self.\_\_dict\_\_.update(kwargs)

#URL列表。当没有指定的URL时，spider将从该列表中开始进行爬取。 因此，第一个被获取到的页面的URL将是该列表之一。 后续的URL将会从获取到的数据中提取。

if not hasattr(self, 'start\_urls'):

self.start\_urls = []

# 打印Scrapy执行后的log信息

def log(self, message, level=log.DEBUG, \*\*kw):

log.msg(message, spider=self, level=level, \*\*kw)

# 判断对象object的属性是否存在，不存在做断言处理

def set\_crawler(self, crawler):

assert not hasattr(self, '\_crawler'), "Spider already bounded to %s" % crawler

self.\_crawler = crawler

@property

def crawler(self):

assert hasattr(self, '\_crawler'), "Spider not bounded to any crawler"

return self.\_crawler

@property

def settings(self):

return self.crawler.settings

**#该方法将读取start\_urls内的地址，并为每一个地址生成一个Request对象，交给Scrapy下载并返回Response**

**#该方法仅调用一次**

**def start\_requests(self):**

**for url in self.start\_urls:**

**yield self.make\_requests\_from\_url(url)**

**#start\_requests()中调用，实际生成Request的函数。**

**#Request对象默认的回调函数为parse()，提交的方式为get**

**#dont\_filter=True忽略域组的影响。**<http://www.baidu.com>**时忽略域组的错误。**

**def make\_requests\_from\_url(self, url):**

**return Request(url, dont\_filter=True)**

#默认的Request对象回调函数，处理返回的response。

#生成Item或者Request对象。用户必须实现这个类

def parse(self, response):

raise NotImplementedError

@classmethod

def handles\_request(cls, request):

return url\_is\_from\_spider(request.url, cls)

def \_\_str\_\_(self):

return "<%s %r at 0x%0x>" % (type(self).\_\_name\_\_, self.name, id(self))

\_\_repr\_\_ = \_\_str\_\_

### 主要属性和方法

* name

定义spider名字的字符串。

例如，如果spider爬取 mywebsite.com ，该spider通常会被命名为 mywebsite

* allowed\_domains

包含了spider允许爬取的域名(domain)的列表，可选。

* start\_urls

初始URL元祖/列表。当没有制定特定的URL时，spider将从该列表中开始进行爬取。

* start\_requests(self)

该方法必须返回一个可迭代对象(iterable)。该对象包含了spider用于爬取（默认实现是使用 start\_urls 的url）的第一个Request。

当spider启动爬取并且未指定start\_urls时，该方法被调用。

* parse(self, response)

当请求url返回网页没有指定回调函数时，默认的Request对象回调函数。用来处理网页返回的response，以及生成Item或者Request对象。

* log(self, message[, level, component])

使用 scrapy.log.msg() 方法记录(log)message。 更多数据请参见 [logging](file:///D:\David\Desktop\boxuegu%E5%8D%9A%E5%AD%A6%E8%B0%B7%E5%8F%82%E8%80%83%E8%B5%84%E6%96%99\%E7%AC%AC7%E7%AB%A0%20%E7%88%AC%E8%99%AB\Python%E7%88%AC%E8%99%AB%E8%AF%BE%E4%BB%B6\file\part04\4.7.html)

## IPython 6.0与yield

pip install ipython **--**upgrade

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

print(b)

fib(5)

1

2

3

5

8

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

yield(b)

m=fib(5)

m.\_\_next\_\_()

Out[12]: 1

m.\_\_next\_\_()

Out[13]: 2

m.\_\_next\_\_()

Out[14]: 3

m.\_\_next\_\_()

Out[15]: 5

m.\_\_next\_\_()

Out[16]: 8

m.\_\_next\_\_()

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

yield(b)

for i in fib(5):

print(i)

1

2

3

5

8

## Scrapy Shell

Scrapy终端是一个交互终端，我们可以在未启动spider的情况下尝试及调试代码，也可以用来测试XPath或CSS表达式，查看他们的工作方式，方便我们爬取的网页中提取的数据。

如果安装了 IPython ，Scrapy终端将使用 IPython (替代标准Python终端)。 IPython 终端与其他相比更为强大，提供智能的自动补全，高亮输出，及其他特性。（推荐安装IPython）

### 启动Scrapy Shell

进入项目的根目录，执行下列命令来启动shell:

scrapy shell "http://www.itcast.cn/channel/teacher.shtml"

Scrapy Shell根据下载的页面会自动创建一些方便使用的对象，例如 Response 对象，以及 Selector 对象 (对HTML及XML内容)。

* 当shell载入后，将得到一个包含response数据的本地 response 变量，输入 response.body将输出response的包体，输出 response.headers 可以看到response的包头。
* 输入 response.selector 时， 将获取到一个response 初始化的类 Selector 的对象，此时可以通过使用 response.selector.xpath()或response.selector.css() 来对 response 进行查询。
* Scrapy也提供了一些快捷方式, 例如 response.xpath()或response.css()同样可以生效（如之前的案例）。

### Selectors选择器

**Scrapy Selectors 内置 XPath 和 CSS Selector 表达式机制**

Selector有四个基本的方法，最常用的还是xpath:

* xpath(): 传入xpath表达式，返回该表达式所对应的所有节点的selector list列表
* extract(): 序列化该节点为Unicode字符串并返回list
* css(): 传入CSS表达式，返回该表达式所对应的所有节点的selector list列表，语法同 BeautifulSoup4
* re(): 根据传入的正则表达式对数据进行提取，返回Unicode字符串list列表

**XPath表达式的例子及对应的含义:**

/html/head/title: 选择<HTML>文档中 <head> 标签内的 <title> 元素

/html/head/title/text(): 选择上面提到的 <title> 元素的文字

//td: 选择所有的 <td> 元素

//div[@class="mine"]: 选择所有具有 class="mine" 属性的 div 元素

**尝试Selector**

我们用腾讯社招的网站<http://hr.tencent.com/position.php?&start=0#a>举例：

# 启动

scrapy shell "http://hr.tencent.com/position.php?&start=0#a"

# 返回 xpath选择器对象列表

response.xpath('//title')

[<Selector xpath='//title' data=u'<title>\u804c\u4f4d\u641c\u7d22 | \u793e\u4f1a\u62db\u8058 | Tencent \u817e\u8baf\u62db\u8058</title'>]

# 使用 extract()方法返回 Unicode字符串列表

response.xpath('//title').extract()

[u'<title>\u804c\u4f4d\u641c\u7d22 | \u793e\u4f1a\u62db\u8058 | Tencent \u817e\u8baf\u62db\u8058</title>']

# 打印列表第一个元素，终端编码格式显示

print response.xpath('//title').extract()[0]

<title>职位搜索 | 社会招聘 | Tencent 腾讯招聘</title>

# 返回 xpath选择器对象列表

response.xpath('//title/text()')

<Selector xpath='//title/text()' data=u'\u804c\u4f4d\u641c\u7d22 | \u793e\u4f1a\u62db\u8058 | Tencent \u817e\u8baf\u62db\u8058'>

# 返回列表第一个元素的Unicode字符串

response.xpath('//title/text()')[0].extract()

u'\u804c\u4f4d\u641c\u7d22 | \u793e\u4f1a\u62db\u8058 | Tencent \u817e\u8baf\u62db\u8058'

# 按终端编码格式显示

print response.xpath('//title/text()')[0].extract()

职位搜索 | 社会招聘 | Tencent 腾讯招聘

response.xpath('//\*[@class="even"]')

职位名称:

print site[0].xpath('./td[1]/a/text()').extract()[0]

TEG15-运营开发工程师（深圳）

职位名称详情页:

print site[0].xpath('./td[1]/a/@href').extract()[0]

position\_detail.php?id=20744&keywords=&tid=0&lid=0

职位类别:

print site[0].xpath('./td[2]/text()').extract()[0]

技术类

以后做数据提取的时候，可以把现在Scrapy Shell中测试，测试通过后再应用到代码中。

当然Scrapy Shell作用不仅仅如此，但是不属于我们课程重点，不做详细介绍。

官方文档：<http://scrapy-chs.readthedocs.io/zh_CN/latest/topics/shell.html>

## Item Pipeline

上面只是爬取了一个url的数据，如果有多个url的话，如何去处理，就要使用pipeline文件。如果spider得到的是数据，会把数据交给item pipline去处理。如果是请求，再经scrapy engine交给scheduler调度器，再次下载。

使用yield生成数据，一次拿到一个老师的数据，把一个老师的数据经yield交给pipeline管道文件进行处理。

当Item在Spider中被收集之后，它将会被传递到Item Pipeline，这些Item Pipeline组件按定义的顺序处理Item。

每个Item Pipeline都是实现了简单方法的Python类，比如决定此Item是丢弃而存储。以下是item pipeline的一些典型应用：

* 验证爬取的数据(检查item包含某些字段，比如说name字段)
* 查重(并丢弃)
* 将爬取结果保存到文件或者数据库中

### 编写item pipeline

编写item pipeline很简单，item pipiline组件是一个独立的Python类，其中process\_item()方法必须实现:

import something

class SomethingPipeline(object):

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

# 可选实现，做参数初始化等

# doing something

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

# item (Item 对象) – 被爬取的item

# spider (Spider 对象) – 爬取该item的spider

# 这个方法必须实现，每个item pipeline组件都需要调用该方法，

**# 这个方法必须返回一个 Item 对象，否则被丢弃的item将不会被之后的pipeline组件所处理。如果不returen item，下一次处理的时候是同一个item**

return item

def open\_spider(self, spider):

# spider (Spider 对象) – 被开启的spider

# 可选实现，当spider被开启时，这个方法被调用。

def close\_spider(self, spider):

# spider (Spider 对象) – 被关闭的spider

# 可选实现，当spider被关闭时，这个方法被调用

### 使用pipeline的itcast爬虫

#### ItcastSpider文件

**import** scrapy

**from** mySpider.items **import** ItcastItem

*# -\*- coding: utf-8 -\*-*

*#创建一个爬虫类*

**class** ItcastSpider(scrapy.Spider):

*#爬虫名，在执行时就是使用的这人名字*

name = **"itcast"**

*#允许爬虫爬取的范围，可以写多个值*

allowed\_domains = [**"itcast.cn"**]

*#爬虫起始的url地址，可以使用元组或列表。*

*# start\_urls = ['http://www.itcast.cn/channel/teacher.shtml#']*

start\_urls = [

**"http://www.itcast.cn/channel/teacher.shtml#aandroid"**,

**"http://www.itcast.cn/channel/teacher.shtml#ac"**,

**"http://www.itcast.cn/channel/teacher.shtml#acloud"**,

**"http://www.itcast.cn/channel/teacher.shtml#aios"**,

**"http://www.itcast.cn/channel/teacher.shtml#ajavaee"**,

**"http://www.itcast.cn/channel/teacher.shtml#anetmarket"**,

**"http://www.itcast.cn/channel/teacher.shtml#aphp"**,

**"http://www.itcast.cn/channel/teacher.shtml#apython"**,

**"http://www.itcast.cn/channel/teacher.shtml#astack"**,

**"http://www.itcast.cn/channel/teacher.shtml#aui"**,

**"http://www.itcast.cn/channel/teacher.shtml#aweb"**

]

*#定义一个parse方法，名字不能修改*

*#处理返回的response*

**def** parse(self, response):

*#open("teacher.html","wb").write(response.body).close()*

*# 存放老师信息的集合*

*# teacherItems = []*

*#通过scrapy自带的xpath匹配出所有老师的根节点*

teacher\_list = response.xpath(**"//div[@class='li\_txt']"**)

*#遍历根节点集合*

**for** each **in** teacher\_list:

*# 将我们得到的数据封装到一个 `ItcastItem` 对象*

*#实例化一个对象，用来保存数据*

item = ItcastItem()

*#extract()方法将匹配出来的结果转换成unicode字符串*

*#不加extract()结果为xpath匹配对象*

name = each.xpath(**"./h3/text()"**).extract()

title = each.xpath(**"./h4/text()"**).extract()

info = each.xpath(**"./p/text()"**).extract()

*#xpath返回的是包含一个元素的列表*

item[**'name'**] = name[0]

item[**'title'**] = title[0]

item[**'info'**] = info[0]

*#*

*# teacherItems.append(item)*

*# print(name[0],title[0],info[0],sep="\n==========\n")*

*# 直接返回最后数据*

*# return teacherItems*

*#通过yield的方式返回老师的数据，拿到一个老师的数据就交给管道文件去处理，处理完成后再返回来处理另外一个老师的数据*

**yield** item

#### pipeline文件

item写入JSON文件

以下pipeline将所有(从所有'spider'中)爬取到的item，存储到一个独立地items.json 文件，每行包含一个序列化为'JSON'格式的'item':

**import** json

**class** ItcastPipeline(object):

*# \_\_init\_\_方法是可选的，作为类的初始化方法*

**def** \_\_init\_\_(self):

*# 创建了一个文件*

self.filename = open(**"teacher.json"**, **"w"**)

*# process\_item方法是必须写的，用来处理item数据*

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

*#*保存为json文件，如果有中文时，不使用ascii编码，使用unicode编码

jsontext = json.dumps(dict(item), ensure\_ascii = **False**) + **"\n"**

self.filename.write(jsontext.encode(**"utf-8"**))

**return** item

*# close\_spider方法是可选的，结束时调用这个方法*

*初始化方法和结束 方法可以都写在process\_item方法中*

**def** close\_spider(self, spider):

self.filename.close()

#### 启用一个Item Pipeline组件

为了启用Item Pipeline组件，必须将它的类添加到 settings.py文件ITEM\_PIPELINES 配置，就像下面这个例子:

BOT\_NAME = **'mySpider'**

*#爬虫存放的位置*

SPIDER\_MODULES = [**'mySpider.spiders'**]

NEWSPIDER\_MODULE = **'mySpider.spiders'**

*#Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

**'User-Agent'** : **'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;'**,

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8'**,

*#'Accept-Language': 'en',*

}

#下载延迟，一般不启动

DOWNLOAD\_DELAY = 2

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

*#是否启动cookie，模拟登陆时要打开，爬虫时就不打开，服务器就不会根据cookie来检测我们爬取的内容了*

COOKIES\_ENABLED = **False**

# Configure item pipelines

# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html

ITEM\_PIPELINES = {

#'mySpider.pipelines.SomePipeline': 300,

"mySpider.pipelines.ItcastJsonPipeline":300

}

分配给每个类的整型值，确定了他们运行的顺序，item按数字从低到高的顺序，通过pipeline，通常将这些数字定义在0-1000范围内（0-1000随意设置，数值越低，组件的优先级越高）

#### 重新启动爬虫

将parse()方法改为4.2中最后思考中的代码，然后执行下面的命令：

scrapy crawl itcast

查看当前目录是否生成teacher.json

或者在项目根目录下创建start.py

**from** scrapy **import** cmdline

cmdline.execute(**"scrapy crawl itcast"**.split())

执行这个文件即可

## 腾讯招聘网自动翻页采集

### scrapy项目步骤

1、创建项目

scrapy startproject xxx

2、编写item文件

设置需要保存的数据字段

3、进入xxx/spiders

编写爬虫文件，文件里的name就是爬虫名

4、运行

scrapy crawl itcast

scrapy crawl itcast -o teacher.json/csv/xml

### 使用scrapy shell来进行测试

scrapy shell <http://hr.tencent.com/position.php?&start=0#a>

print(response.body)

response.xpath(“//title”)

返回一个包含匹配选择规则的选择器的列表

response.xpath(“//title”).extract()

转换为unicode字符串

response.xpath('//title').extract()[0]

取出来选择器对象

print(response.xpath('//title').extract()[0])

以终端默认的编码格式打印出来

print(response.xpath('//title/text()').extract()[0])

text()取出来标签中的内容

list\_1 = response.xpath(“//tr[@class='even']”)

list\_2 = response.xpath(“//tr[@class='odd']”)

list = list\_1 + list\_2

detailLink = each.xpath('./td[1]/a/@href').extract()[0]

### 创建一个新的爬虫：

scrapy genspider tencent "tencent.com"

### 编写items.py

获取职位名称、详细信息、

**import** scrapy

**class** TencentspiderItem(scrapy.Item):

*#职位名称*

position\_name = scrapy.Field()

*#详情链接*

position\_link = scrapy.Field()

*#职位类别*

position\_type = scrapy.Field()

*#招聘人数*

people\_num = scrapy.Field()

*#工作地点*

work\_location = scrapy.Field()

*#发布时间*

publish\_time = scrapy.Field()

### 编写tencentZhaopin.py

**方法一：使用自增的方法处理分页**

**import** scrapy

**from** tencentSpider.items **import** TencentspiderItem

**class** TencentzhaopinSpider(scrapy.Spider):

name = **"tencentZhaopin"**

allowed\_domains = [**"tencent.com"**]

url = **"http://hr.tencent.com/position.php?&start="**

*#代表每页的url地址，每爬完一页，就使offset的值加10，就到了下一页。*

offset = 0

start\_urls = [url + str(offset)]

**def** parse(self, response):

*# 网页中有两种不同的样式*

**for** each **in** response.xpath(**"//tr[@class='even'] | //tr[@class='odd']"**):

*#初始化模型对象，类似于python中的字典*

item = TencentspiderItem

*# 职位名称，直接保存到item的对象中了*

item[**'position\_name'**] = each.xpath(**"./td[1]/a/text()"**).extract()[0]*#.encode("utf-8")*

*# 详情连接*

item[**'position\_link'**] = each.xpath(**"./td[1]/a/@href"**).extract()[0]*#.encode("utf-8")*

*# 职位类别*

item[**'position\_type'**] = each.xpath(**"./td[2]/text()"**).extract()[0]*#.encode("utf-8")*

*# 招聘人数*

item[**'people\_num'**] = each.xpath(**"./td[3]/text()"**).extract()[0]*#.encode("utf-8")*

*# 工作地点*

item[**'work\_location'**] = each.xpath(**"./td[4]/text()"**).extract()[0]*#.encode("utf-8")*

*# 发布时间*

item[**'publish\_time'**] = each.xpath(**"./td[5]/text()"**).extract()[0]*#.encode("utf-8")*

*#如果得到的是数据，将数据交给管道文件处理*

**yield** item

*#当一页中所有的数据都处理完成后，接着处理下一页中的数据*

*#上面定义的offset相当于\_init\_\_方法中定义的初始化的值，要使用self.offset*

**if** self.offset < 2111:

self.offset += 10

*# else:*

*# break*

*# raise "结束工作"*

*# 如果得到的是数据，将数据交给管道文件处理，如果是请求，将请求重新发送给调度器入队列，出队列，交给下载 器下载yield scrapy.Request(url, callback=self.parse)*

*# callback回调函数*

*#每次处理完一页的数据之后，重新发送下一页的请求*

*#self.off自增10，同时拼接为新的url，并调用回调函数self.parse处理Response*

**yield** scrapy.Request(self.url + str(self.offset), callback=self.parse)

**方法二：使用re匹配的方法处理分页**

**import** scrapy, re

**from** tencentSpider.items **import** TencentspiderItem

**class** TencentzhaopinSpider(scrapy.Spider):

name = **"tencentZhaopin"**

allowed\_domains = [**"tencent.com"**]

url = **"http://hr.tencent.com/position.php?&start="**

*#代表每页的url地址，每爬完一页，就使offset的值加10，就到了下一页。*

offset = 0

start\_urls = [url + str(offset)]

**def** parse(self, response):

*# 网页中有两种不同的样式*

**for** each **in** response.xpath(**"//tr[@class='even'] | //tr[@class='odd']"**):

*#初始化模型对象，类似于python中的字典*

item = TencentspiderItem

*# 职位名称，直接保存到item的对象中了*

item[**'position\_name'**] = each.xpath(**"./td[1]/a/text()"**).extract()[0]*#.encode("utf-8")*

*# 详情连接*

item[**'position\_link'**] = each.xpath(**"./td[1]/a/@href"**).extract()[0]*#.encode("utf-8")*

*# 职位类别*

item[**'position\_type'**] = each.xpath(**"./td[2]/text()"**).extract()[0]*#.encode("utf-8")*

*# 招聘人数*

item[**'people\_num'**] = each.xpath(**"./td[3]/text()"**).extract()[0]*#.encode("utf-8")*

*# 工作地点*

item[**'work\_location'**] = each.xpath(**"./td[4]/text()"**).extract()[0]*#.encode("utf-8")*

*# 发布时间*

item[**'publish\_time'**] = each.xpath(**"./td[5]/text()"**).extract()[0]*#.encode("utf-8")*

*#如果得到的是数据，将数据交给管道文件处理*

**yield** item

*# 从任意地方开始查找匹配一次，匹配url。*

curpage = re.search(**'(\d+)'**, response.url).group(1)

page = int(curpage) + 10

*# 第一个参数，匹配的数字，第二个参数生成的新的页码，第三个参数，提交的url。把response字符串中的所有数字替换为str(page)。re.sub(s1,s2,s3), 将s3中所有的s1替换为s2.*

url = re.sub(**'\d+'**, str(page), response.url)

*# 把yield放到for循环中并不会影响程序的执行和最终的结果，但是这样是每获取一个数据就会发一个url请求到调度器中，入队列。把yield scrapy.Request放在for循环外面的话是一个循环结束后才去发送下一个页面的request请求。*

*# 发送新的url请求加入待爬队列，并调用回调函数 self.parse*

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

### 编写pipeline.py文件

* 初始化和关闭文件都只执行一次，初始化是管道文件在接收到第一个数据时就执行的，会创建一个文件，而关闭文件则是在爬虫执行结束后才执行的，而不是在管道文件关闭时执行。管道文件在打开之后，在整个爬虫执行的过程中都是打开的。所以文件的写入是用的w写入，而不是a追加。因为不是写入一次就关闭一次，而是始终打开的。中间的处理部分执行很多次，来一个数据就执行一次。

import json

#class ItcastJsonPipeline(object):

class TencentJsonPipeline(object):

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

#self.file = open('teacher.json', 'wb')

self.file = open('tencent.json', 'wb')

#如果数据中有中文的话，就要禁用ascii，使用unicode格式，这样就更具有通用性，如果需要，再把unicode转换为utf-8或gbk的编码形式。否则中文会使用ascii编码，在linux中是无法显示的。

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

content = json.dumps(dict(item), ensure\_ascii=False) + "\n"

self.file.write(content)

return item

def close\_spider(self, spider):

self.file.close()

### middleware.py

**from** scrapy **import** signals

**class** TencentspiderSpiderMiddleware(object):

*# Not all methods need to be defined. If a method is not defined,*

*# scrapy acts as if the spider middleware does not modify the*

*# passed objects.*

@classmethod

**def** from\_crawler(cls, crawler):

*# This method is used by Scrapy to create your spiders.*

s = cls()

crawler.signals.connect(s.spider\_opened, signal=signals.spider\_opened)

**return** s

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

*# Called for each response that goes through the spider*

*# middleware and into the spider.*

*# Should return None or raise an exception.*

**return None**

**def** process\_spider\_output(response, result, spider):

*# Called with the results returned from the Spider, after*

*# it has processed the response.*

*# Must return an iterable of Request, dict or Item objects.*

**for** i **in** result:

**yield** i

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

*# Called when a spider or process\_spider\_input() method*

*# (from other spider middleware) raises an exception.*

*# Should return either None or an iterable of Response, dict*

*# or Item objects.*

**pass**

**def** process\_start\_requests(start\_requests, spider):

*# Called with the start requests of the spider, and works*

*# similarly to the process\_spider\_output() method, except*

*# that it doesn't have a response associated.*

*# Must return only requests (not items).*

**for** r **in** start\_requests:

**yield** r

**def** spider\_opened(self, spider):

spider.logger.info(**'Spider opened: %s'** % spider.name)

* 在 setting.py 里设置ITEM\_PIPELINES

ITEM\_PIPELINES = {

#'mySpider.pipelines.SomePipeline': 300,

#"mySpider.pipelines.ItcastJsonPipeline":300

"mySpider.pipelines.TencentJsonPipeline":300

}

### 在setting中设置请求头

DEFAULT\_REQUEST\_HEADERS = {

**'User-Agent'**: **'Mozilla/5.0 (Windows NT 10.0; Win64; IA64; Trident/7.0; rv:11.0) like Gecko'**,

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8'**,

#**'Accept-Language'**: **'en'**,

}

### 执行爬虫：scrapy crawl tencent

### 思考parse()方法的工作机制：

1. 因为使用的yield，而不是return。parse函数将会被当做一个生成器使用。scrapy会逐一获取parse方法中生成的结果，并判断该结果是一个什么样的类型；

2. 如果是request则加入爬取队列，如果是item类型则使用pipeline处理，其他类型则返回错误信息。

3. scrapy取到第一部分的request不会立马就去发送这个request，只是把这个request放到队列里，然后接着从生成器里获取；

4. 取尽第一部分的request，然后再获取第二部分的item，取到item了，就会放到对应的pipeline里处理；

5. parse()方法作为回调函数(callback)赋值给了Request，指定parse()方法来处理这些请求 scrapy.Request(url, callback=self.parse)

6. Request对象经过调度，执行生成 scrapy.http.response()的响应对象，并送回给parse()方法，直到调度器中没有Request（递归的思路）

7. 取尽之后，parse()工作结束，引擎再根据队列和pipelines中的内容去执行相应的操作；

8. 程序在取得各个页面的items前，会先处理完之前所有的request队列里的请求，然后再提取items。

7. 这一切的一切，Scrapy引擎和调度器将负责到底。

## CrawlSpiders

通过下面的命令可以快速创建 CrawlSpider模板 的代码：

**scrapy genspider -t crawl tencent tencent.com**

**-t指定一个模板**

上一个案例中，我们通过正则表达式，制作了新的url作为Request请求参数，现在我们可以换个花样...

class scrapy.spiders.CrawlSpider

它是Spider的派生类，Spider类的设计原则是只爬取start\_url列表中的网页，而CrawlSpider类定义了一些规则(rule)来提供跟进link的方便的机制，从爬取的网页中获取link并继续爬取的工作更适合。

### 源码参考

class CrawlSpider(Spider):

rules = ()

def \_\_init\_\_(self, \*a, \*\*kw):

super(CrawlSpider, self).\_\_init\_\_(\*a, \*\*kw)

self.\_compile\_rules()

#首先调用parse()来处理start\_urls中返回的response对象

#parse()则将这些response对象传递给了\_parse\_response()函数处理，并设置回调函数为parse\_start\_url()

#设置了跟进标志位True

#parse将返回item和跟进了的Request对象

def parse(self, response):

return self.\_parse\_response(response, self.parse\_start\_url, cb\_kwargs={}, follow=True)

#处理start\_url中返回的response，需要重写

def parse\_start\_url(self, response):

return []

def process\_results(self, response, results):

return results

#从response中抽取符合任一用户定义'规则'的链接，并构造成Resquest对象返回

def \_requests\_to\_follow(self, response):

if not isinstance(response, HtmlResponse):

return

seen = set()

#抽取之内的所有链接，只要通过任意一个'规则'，即表示合法

for n, rule in enumerate(self.\_rules):

links = [l for l in rule.link\_extractor.extract\_links(response) if l not in seen]

#使用用户指定的process\_links处理每个连接

if links and rule.process\_links:

links = rule.process\_links(links)

#将链接加入seen集合，为每个链接生成Request对象，并设置回调函数为\_repsonse\_downloaded()

for link in links:

seen.add(link)

#构造Request对象，并将Rule规则中定义的回调函数作为这个Request对象的回调函数

r = Request(url=link.url, callback=self.\_response\_downloaded)

r.meta.update(rule=n, link\_text=link.text)

#对每个Request调用process\_request()函数。该函数默认为indentify，即不做任何处理，直接返回该Request.

yield rule.process\_request(r)

#处理通过rule提取出的连接，并返回item以及request

def \_response\_downloaded(self, response):

rule = self.\_rules[response.meta['rule']]

return self.\_parse\_response(response, rule.callback, rule.cb\_kwargs, rule.follow)

#解析response对象，会用callback解析处理他，并返回request或Item对象

def \_parse\_response(self, response, callback, cb\_kwargs, follow=True):

#首先判断是否设置了回调函数。（该回调函数可能是rule中的解析函数，也可能是 parse\_start\_url函数）

#如果设置了回调函数（parse\_start\_url()），那么首先用parse\_start\_url()处理response对象，

#然后再交给process\_results处理。返回cb\_res的一个列表

if callback:

#如果是parse调用的，则会解析成Request对象

#如果是rule callback，则会解析成Item

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

#如果需要跟进，那么使用定义的Rule规则提取并返回这些Request对象

if follow and self.\_follow\_links:

#返回每个Request对象

for request\_or\_item in self.\_requests\_to\_follow(response):

yield request\_or\_item

def \_compile\_rules(self):

def get\_method(method):

if callable(method):

return method

elif isinstance(method, basestring):

return getattr(self, method, None)

self.\_rules = [copy.copy(r) for r in self.rules]

for rule in self.\_rules:

rule.callback = get\_method(rule.callback)

rule.process\_links = get\_method(rule.process\_links)

rule.process\_request = get\_method(rule.process\_request)

def set\_crawler(self, crawler):

super(CrawlSpider, self).set\_crawler(crawler)

self.\_follow\_links = crawler.settings.getbool('CRAWLSPIDER\_FOLLOW\_LINKS', True)

CrawlSpider继承于Spider类，除了继承过来的属性外（name、allow\_domains），还提供了新的属性和方法:

### LinkExtractors

class scrapy.linkextractors.LinkExtractor

Link Extractors 的目的很简单: 提取链接｡

每个LinkExtractor有唯一的公共方法是 extract\_links()，它接收一个 Response 对象，并返回一个 scrapy.link.Link 对象。

Link Extractors要实例化一次，并且 extract\_links 方法会根据不同的 response 调用多次提取链接。如腾讯案例中，也可以把职位的详情页的链接提取出来。只要设定其匹配规则即可。

class scrapy.linkextractors.LinkExtractor(

allow = (),

deny = (),

allow\_domains = (),

deny\_domains = (),

deny\_extensions = None,

restrict\_xpaths = (),

tags = ('a','area'),

attrs = ('href'),

canonicalize = True,

unique = True,

process\_value = None

)

主要参数：

* allow：满足括号中“正则表达式”的值会被提取，如果为空，则全部匹配。
* deny：与这个正则表达式(或正则表达式列表)不匹配的URL一定不提取。
* allow\_domains：会被提取的链接的domains。
* deny\_domains：一定不会被提取链接的domains。
* restrict\_xpaths：使用xpath表达式，和allow共同作用过滤链接。

### rules

在rules中包含一个或多个Rule对象，可以是正则匹配规则对象，每个Rule对爬取网站的动作定义了特定操作。如果多个rule匹配了相同的链接，则根据规则在本集合中被定义的顺序，第一个会被使用。

class scrapy.spiders.Rule(

link\_extractor,

callback = None,

cb\_kwargs = None,

follow = None,

process\_links = None,

process\_request = None

)

* link\_extractor：是一个Link Extractor对象，用于定义需要提取的链接。如page\_lx = LinkExtractor(allow=('position.php?&start=\d+'))
* callback： 从link\_extractor中每获取到链接时，参数所指定的值作为回调函数，该回调函数接受一个response作为其第一个参数。

注意：当编写爬虫规则时，避免使用parse作为回调函数。由于CrawlSpider使用parse方法来实现其逻辑，如果覆盖了 parse方法，crawl spider将会运行失败。所以要手动写一个解析函数作为回调函数

* follow：是一个布尔(boolean)值，指定了根据该规则从response提取的链接是否需要跟进。 如果callback为None，follow 默认设置为True ，否则默认为False。是否继续对链接进行跟进处理。深度爬虫。
* process\_links：指定该spider中哪个的函数将会被调用，从link\_extractor中获取到链接列表时将会调用该函数。该方法主要用来过滤。
* process\_request：指定该spider中哪个的函数将会被调用， 该规则提取到每个request时都会调用该函数。 (用来过滤request)

### 使用scrapy shell找到爬取规则(Crawling rules)

继续用腾讯招聘为例，给出配合rule使用CrawlSpider的例子:

1. 首先运行

scrapy shell <http://hr.tencent.com/position.php?&start=0#a>

获取响应内容，打印内容的时候用body

response.body

获取unicode格式的响应内容，解析的时候用text

response.text

1. 打开网页，查看页码导航的url地址，是类似于如下的形式：

<a href="position.php?&start=10#a">2</a>

1. 导入LinkExtractor，创建LinkExtractor实例对象。：

from scrapy.linkextractors import LinkExtractor

LinkExtractor是匹配整个html文档中的链接

1. page\_lx = LinkExtractor(allow=('position.php?&start=\d+'))

创建一个正则匹配规则对象。

allow : LinkExtractor对象最重要的参数之一，这是一个正则表达式，必须要匹配这个正则表达式(或正则表达式列表)的URL才会被提取，如果没有给出(或为空), 它会匹配所有的链接。

deny : 用法同allow，只不过与这个正则表达式匹配的URL不会被提取) 。它的优先级高于 allow 的参数，如果没有给出(或None)，将不排除任何链接。

1. 调用LinkExtractor实例的extract\_links()方法查询匹配结果：

使用正则匹配对象从响应文件中取出来所有匹配的links。

page\_lx.extract\_links(response)

1. 没有查到：

[]

1. 注意转义字符的问题，继续重新匹配：

page\_lx = LinkExtractor(allow=('position\.php\?&start=\d+'))

或者

page\_lx = LinkExtractor(allow=(r'position.php?&start=\d+'))

或者

# page\_lx = LinkExtractor(allow = ('start=\d+'))

1. page\_lx.extract\_links(response)

### tencent爬虫的CrawlSpider 版本

那么，scrapy shell测试完成之后，修改以下代码

#提取匹配 'http://hr.tencent.com/position.php?&start=\d+'的链接

page\_lx = LinkExtractor(allow = ('start=\d+'))

rules = [

#提取匹配,并使用spider的parse方法进行分析;并跟进链接(没有callback意味着follow默认为True)

Rule(page\_lx, callback = 'parse', follow = True)

]

**这么写对吗？**

**不对！千万记住 callback 千万不能写 parse，再次强调：由于CrawlSpider使用parse方法来实现其逻辑，如果覆盖了 parse方法，crawl spider将会运行失败。**

**scrapy startproject TencentSpider**

**scrapy genspider -t crawl tencent tencent.com**

#### item文件

**import** scrapy

**class** TencentItem(scrapy.Item):

*# define the fields for your item here like:*

*# 职位名*

positionname = scrapy.Field()

*# 详情连接*

positionlink = scrapy.Field()

*# 职位类别*

positionType = scrapy.Field()

*# 招聘人数*

peopleNum = scrapy.Field()

*# 工作地点*

workLocation = scrapy.Field()

*# 发布时间*

publishTime = scrapy.Field()

#### Spider

#tencent.py

**import** scrapy

*# 导入CrawlSpider类和Rule*

**from** scrapy.spiders **import** CrawlSpider, Rule

*# 导入链接规则匹配类，用来提取符合规则的连接*

**from** scrapy.linkextractors **import** LinkExtractor

**from** TencentSpider.items **import** TencentItem

**class** TencentSpider(CrawlSpider):

name = **"tencent"**

allow\_domains = [**"hr.tencent.com"**]

start\_urls = [**"http://hr.tencent.com/position.php?&start=0#a"**]

*# Response里链接的提取规则，返回的符合匹配规则的链接匹配对象的列表*

pagelink = LinkExtractor(allow=(**"start=\d+"**))

*#职位的详情页匹配规则*

*#position\_link = LinkExtractor(allow=("position.php?"))*

rules = [

*# 获取这个列表里的链接，依次发送请求，并且继续跟进，调用指定回调函数处理*

*# 拿到了页码对应的链接，并且要进入到对应的链接中，即要继续对链接进行跟进。*

Rule(pagelink, callback = **"parseTencent"**, follow = **True**),

*#如果有多个匹配规则，在列表中继续写*

*#Rule(new\_link, callback = "parsePosition", follow = True)*

]

*# 指定的回调函数，不需要写parse()方法*

**def** parseTencent(self, response):

*#evenlist = response.xpath("//tr[@class='even'] | //tr[@class='odd']")*

*#oddlist = response.xpath("//tr[@class='even'] | //tr[@class='odd']")*

*#fulllist = evenlist + oddlist*

*#for each in fulllist:*

**for** each **in** response.xpath(**"//tr[@class='even'] | //tr[@class='odd']"**):

item = TencentItem()

*# 职位名称*

item[**'positionname'**] = each.xpath(**"./td[1]/a/text()"**).extract()[0]

*# 详情连接*

item[**'positionlink'**] = each.xpath(**"./td[1]/a/@href"**).extract()[0]

*# 职位类别*

item[**'positionType'**] = each.xpath(**"./td[2]/text()"**).extract()[0]

*# 招聘人数*

item[**'peopleNum'**] = each.xpath(**"./td[3]/text()"**).extract()[0]

*# 工作地点*

item[**'workLocation'**] = each.xpath(**"./td[4]/text()"**).extract()[0]

*# 发布时间*

item[**'publishTime'**] = each.xpath(**"./td[5]/text()"**).extract()[0]

**yield** item

#### pipeline文件

**import** json

**class** TencentPipeline(object):

**def** \_\_init\_\_(self):

self.filename = open(**"tencent.json"**, **"w"**)

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

text = json.dumps(dict(item), ensure\_ascii = **False**) + **",\n"**

self.filename.write(text.encode(**"utf-8"**))

**return** item

**def** close\_spider(self, spider):

self.filename.close()

#### settings文件

BOT\_NAME = **'TencentSpider'**

SPIDER\_MODULES = [**'TencentSpider.spiders'**]

NEWSPIDER\_MODULE = **'TencentSpider.spiders'**

*# 保存日志信息的文件名*

LOG\_FILE = **"tencentlog.log"**

*# 保存日志等级，低于|等于此等级的信息都被保存*

LOG\_LEVEL = **"DEBUG"**

ITEM\_PIPELINES = {

**'TencentSpider.pipelines.TencentPipeline'**: 300,

}

#### 运行： scrapy crawl tencent

### Logging日志文件

Scrapy提供了log功能，可以通过 logging 模块使用。

可以修改配置文件settings.py，任意位置添加下面两行，效果会清爽很多。

LOG\_FILE = "TencentSpider.log"

LOG\_LEVEL = "INFO"

#### Log levels

* Scrapy提供5层logging级别:
* CRITICAL - 严重错误(critical)
* ERROR - 一般错误(regular errors)
* WARNING - 警告信息(warning messages)
* INFO - 一般信息(informational messages)
* DEBUG - 调试信息(debugging messages)

#### logging设置

通过在setting.py中进行以下设置可以被用来配置logging:

1. LOG\_ENABLED 默认: True，启用logging
2. LOG\_ENCODING 默认: 'utf-8'，logging使用的编码
3. LOG\_FILE 默认: None，在当前目录里创建logging输出文件的文件名
4. LOG\_LEVEL 默认: 'DEBUG'，log的最低级别
5. 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。

## Request/Response与模拟登陆

### Request

Request 部分源码：

# 部分代码

class Request(object\_ref):

def \_\_init\_\_(self, url, callback=None, method='GET', headers=None, body=None,

cookies=None, meta=None, encoding='utf-8', priority=0,

dont\_filter=False, errback=None):

self.\_encoding = encoding # this one has to be set first

self.method = str(method).upper()

self.\_set\_url(url)

self.\_set\_body(body)

assert isinstance(priority, int), "Request priority not an integer: %r" % priority

self.priority = priority

assert callback or not errback, "Cannot use errback without a callback"

self.callback = callback

self.errback = errback

self.cookies = cookies or {}

self.headers = Headers(headers or {}, encoding=encoding)

self.dont\_filter = dont\_filter

self.\_meta = dict(meta) if meta else None

@property

def meta(self):

if self.\_meta is None:

self.\_meta = {}

return self.\_meta

其中，比较常用的参数：

url: 就是需要请求，并进行下一步处理的url

callback: 指定该请求返回的Response，由那个函数来处理。

method: 请求一般不需要指定，默认GET方法，可设置为"GET", "POST", "PUT"等，且保证字符串大写

headers: 请求时，包含的头文件。一般不需要。内容一般如下：

# 自己写过爬虫的肯定知道

Host: media.readthedocs.org

User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0

Accept: text/css,\*/\*;q=0.1

Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Referer: http://scrapy-chs.readthedocs.org/zh\_CN/0.24/

Cookie: \_ga=GA1.2.1612165614.1415584110;

Connection: keep-alive

If-Modified-Since: Mon, 25 Aug 2014 21:59:35 GMT

Cache-Control: max-age=0

meta: 比较常用，在不同的请求之间传递数据使用的。字典dict型

request\_with\_cookies = Request(

url="http://www.example.com",

cookies={'currency': 'USD', 'country': 'UY'},

meta={'dont\_merge\_cookies': True}

)

encoding: 使用默认的 'utf-8' 就行。

dont\_filter: 表明该请求不由调度器过滤。这是当你想使用多次执行相同的请求,忽略重复的过滤器。默认为False。不进行地址指纹的过滤。在模拟登陆的时候设置为True，第一次登陆时向主页发送一个请求，登陆上去后还要再次向这个主页发送数据，如果执行过滤的话，第二次登陆上去后的请求就无法发送了。

errback: 指定错误处理函数

### Response

# 部分代码

class Response(object\_ref):

def \_\_init\_\_(self, url, status=200, headers=None, body='', flags=None, request=None):

self.headers = Headers(headers or {})

self.status = int(status)

self.\_set\_body(body)

self.\_set\_url(url)

self.request = request

self.flags = [] if flags is None else list(flags)

@property

def meta(self):

try:

return self.request.meta

except AttributeError:

raise AttributeError("Response.meta not available, this response " \

"is not tied to any request")

大部分参数和上面的差不多：

status: 响应码

\_set\_body(body)： 响应体

\_set\_url(url)：响应url

self.request = request

#### 发送POST请求

* 可以使用 yield scrapy.FormRequest(url, formdata, callback)方法发送POST请求。
* 如果希望程序执行一开始就发送POST请求，可以重写Spider类的start\_requests(self) 方法，并且不再调用start\_urls里的url。

class mySpider(scrapy.Spider):

# start\_urls = ["http://www.example.com/"]

def start\_requests(self):

url = 'http://www.renren.com/PLogin.do'

# FormRequest 是Scrapy发送POST请求的方法

yield scrapy.FormRequest(

url = url,

formdata = {"email" : "mr\_mao\_hacker@163.com", "password" : "axxxxxxxe"},

callback = self.parse\_page

)

def parse\_page(self, response):

# do something

#### 模拟登陆

使用FormRequest.from\_response()方法[模拟用户登录](http://docs.pythontab.com/scrapy/scrapy0.24/topics/request-response.html#topics-request-response-ref-request-userlogin)

通常网站通过 实现对某些表单字段（如数据或是登录界面中的认证令牌等）的预填充。

使用Scrapy抓取网页时，如果想要预填充或重写像用户名、用户密码这些表单字段， 可以使用 FormRequest.from\_response() 方法实现。

下面是使用这种方法的爬虫例子:

import scrapy

class LoginSpider(scrapy.Spider):

name = 'example.com'

start\_urls = ['http://www.example.com/users/login.php']

def parse(self, response):

return scrapy.FormRequest.from\_response(

response,

formdata={'username': 'john', 'password': 'secret'},

callback=self.after\_login

)

def after\_login(self, response):

# check login succeed before going on

if "authentication failed" in response.body:

self.log("Login failed", level=log.ERROR)

return

# continue scraping with authenticated session...

### 人人网三种scrapy模拟登陆策略

注意：模拟登陆时，必须保证settings.py里的 COOKIES\_ENABLED(Cookies中间件) 处于开启状态。如果是不需要登陆就能采集数据的网站，要禁用cookie。

COOKIES\_ENABLED = True 或 # COOKIES\_ENABLED = False

#### 策略一：直接POST数据（比如需要登陆的账户信息)

只要是需要提供post数据的，就可以用这种方法。下面示例里post的数据是账户密码：

# -\*- coding: utf-8 -\*-

import scrapy

*# 只需要是提供post数据的，就可以用这种方法，*

*# 下面示例：post数据是账户密码*

class Renren1Spider(scrapy.Spider):

name = "renren1"

allowed\_domains = ["renren.com"]

def start\_requests(self):

url = 'http://www.renren.com/PLogin.do'

# FormRequest 是Scrapy发送POST请求的方法

yield scrapy.FormRequest(

url = url,

formdata = {"email" : "mr\_mao\_hacker@163.com", "password" : "axxxxxxxe"},

callback = self.parse\_page)

def parse\_page(self, response):

with open("mao2.html", "w") as filename:

filename.write(response.body)

#### 策略二：标准的模拟登陆步骤

正统模拟登录方法：

1. 首先发送登录页面的get请求，获取到页面里的登录必须的参数（比如说zhihu登陆界面的 \_xsrf）
2. 然后和账户密码一起post到服务器，登录成功

# -\*- coding: utf-8 -\*-

import scrapy

class Renren2Spider(scrapy.Spider):

name = "renren2"

allowed\_domains = ["renren.com"]

start\_urls = (

"http://www.renren.com/PLogin.do",

)

# 处理start\_urls里的登录url的响应内容，提取登陆需要的参数（如果需要的话)

def parse(self, response):

# 提取登陆需要的参数

#\_xsrf = response.xpath("//\_xsrf").extract()[0]

# 发送post请求参数，把账户密码和其它相关的参数发送到服务器，并调用指定回调函数处理

yield scrapy.FormRequest.from\_response(

response,

formdata = {"email" : "mr\_mao\_hacker@163.com", "password" : "axxxxxxxe"},#, "\_xsrf" = \_xsrf},

*#调用回调函数来处理登陆成功后的页面*

callback = self.parse\_page

)

# 获取登录成功后的页面，访问需要登录后才能访问的页面，这里的response是登陆之后的response

def parse\_page(self, response):

print(**"=========1==="** + response.url)

*#with open("mao.html", "w") as filename:*

*# filename.write(response.body)*

*#可以定义多个url，多次迭代来发送请求*

url = **"http://www.renren.com/422167102/profile"**

*#发送一个get的请求，这时是已经登陆后发送的请求。*

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

# 处理响应内容

def parse\_newpage(self, response):

print(**"===========2===="** + response.url)

with open("xiao.html", "w") as filename:

filename.write(response.body)

#### 策略三：直接使用保存登陆状态的Cookie模拟登陆

如果实在没办法了，可以用这种方法模拟登录，虽然麻烦一点，但是成功率100%

**取登陆后的单个url**

**import** scrapy

*# 实在没办法了，可以用这种方法模拟登录，麻烦一点，成功率100%*

**class** RenrenSpider(scrapy.Spider):

name = **"renren"**

allowed\_domains = [**"renren.com"**]

*#抓包，在raw中查看登陆后的cookie值。使用正则替换，查找(.\*?)=(.\*?); 替换为"\1": "\2",\n*

cookies = {

**"anonymid"** : **"ixrna3fysufnwv"**,

**"\_r01\_"** : **"1"**,

**"ap"** : **"327550029"**,

**"JSESSIONID"** : **"abciwg61A\_RvtaRS3GjOv"**,

**"depovince"** : **"GW"**,

**"springskin"** : **"set"**,

**"jebe\_key"** : **"f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198628950"**,

**"jebe\_key"** : **"f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198619601"**,

**"ver"** : **"7.0"**,

**"XNESSESSIONID"** : **"e703b11f8809"**,

**"jebecookies"** : **"98c7c881-779f-4da8-a57c-7464175cd469|||||"**,

**"ick\_login"** : **"4b4a254a-9f25-4d4a-b686-a41fda73e173"**,

**"\_de"** : **"BF09EE3A28DED52E6B65F6A4705D973F1383380866D39FF5"**,

**"p"** : **"ea5541736f993365a23d04c0946c10e29"**,

**"first\_login\_flag"** : **"1"**,

**"ln\_uact"** : **"mr\_mao\_hacker@163.com"**,

**"ln\_hurl"** : **"http://hdn.xnimg.cn/photos/hdn521/20140529/1055/h\_main\_9A3Z\_e0c300019f6a195a.jpg"**,

**"t"** : **"691808127750a83d33704a565d8340ae9"**,

**"societyguester"** : **"691808127750a83d33704a565d8340ae9"**,

**"id"** : **"327550029"**,

**"xnsid"** : **"f42b25cf"**,

**"loginfrom"** : **"syshome"**

}

*#重写系统中的方法。定义一个与系统中同名的方法，就会覆盖系统中的方法。*

**def** start\_requests(self):

*#get请求*

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

*#不使用starturl，而是使用自定义的url*

url = **"http://www.renren.com/410043129/profile"**

*#post请求*

**yield** scrapy.FormRequest(url, cookies = self.cookies, callback = self.parse\_page)

**def** parse\_page(self, response):

print(**"==========="** + response.url)

**with** open(**"deng.html"**, **"w"**) **as** filename:

filename.write(response.body)

**取登陆后的多个url**

# -\*- coding: utf-8 -\*-

import scrapy

class RenrenSpider(scrapy.Spider):

name = "renren"

allowed\_domains = ["renren.com"]

*#虽然没有被scrapy自己的start request方法调用，但我们可以手动去调用它。*

*#定义一系列登陆后才能查看的页面，例如是好友的主页。*

start\_urls = (

'http://www.renren.com/111111',

'http://www.renren.com/222222',

'http://www.renren.com/333333',

)

*#抓包，在raw中查看登陆后的cookie值。使用正则替换，查找(.\*?)=(.\*?); 替换为"\1": "\2",\n*

cookies = {

"anonymid" : "ixrna3fysufnwv",

"\_r01\_" : "1",

"ap" : "327550029",

"JSESSIONID" : "abciwg61A\_RvtaRS3GjOv",

"depovince" : "GW",

"springskin" : "set",

"jebe\_key" : "f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198628950",

"t" : "691808127750a83d33704a565d8340ae9",

"societyguester" : "691808127750a83d33704a565d8340ae9",

"id" : "327550029",

"xnsid" : "f42b25cf",

"loginfrom" : "syshome"

}

# 可以重写Spider类的start\_requests方法，附带Cookie值，发送POST请求

*#重写系统中的方法。定义一个与系统中同名的方法，就会覆盖系统中的方法。*

def start\_requests(self):

for url in self.start\_urls:

*#get请求*

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

*#post请求*

yield scrapy.FormRequest(url, cookies = self.cookies, callback = self.parse\_page)

# 处理响应内容

def parse\_page(self, response):

print "===========" + response.url

with open("deng.html", "w") as filename:

filename.write(response.body)

### 知乎爬虫模拟登陆案例

#### zhihuSpider.py爬虫代码

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

from scrapy.spiders import CrawlSpider, Rule

from scrapy.selector import Selector

from scrapy.linkextractors import LinkExtractor

from scrapy import Request, FormRequest

from zhihu.items import ZhihuItem

class ZhihuSipder(CrawlSpider) :

name = "zhihu"

allowed\_domains = ["www.zhihu.com"]

start\_urls = [

"http://www.zhihu.com"

]

rules = (

Rule(LinkExtractor(allow = ('/question/\d+#.\*?', )), callback = 'parse\_page', follow = True),

Rule(LinkExtractor(allow = ('/question/\d+', )), callback = 'parse\_page', follow = True),

)

headers = {

"Accept": "\*/\*",

"Accept-Encoding": "gzip,deflate",

"Accept-Language": "en-US,en;q=0.8,zh-TW;q=0.6,zh;q=0.4",

"Connection": "keep-alive",

"Content-Type":" application/x-www-form-urlencoded; charset=UTF-8",

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.111 Safari/537.36",

"Referer": "http://www.zhihu.com/"

}

#重写了爬虫类的方法, 实现了自定义请求, 运行成功后会调用callback回调函数

def start\_requests(self):

return [Request("https://www.zhihu.com/login", meta = {'cookiejar' : 1}, callback = self.post\_login)]

def post\_login(self, response):

print 'Preparing login'

#下面这句话用于抓取请求网页后返回网页中的\_xsrf字段的文字, 用于成功提交表单

xsrf = Selector(response).xpath('//input[@name="\_xsrf"]/@value').extract()[0]

print xsrf

#FormRequeset.from\_response是Scrapy提供的一个函数, 用于post表单

#登陆成功后, 会调用after\_login回调函数

return [FormRequest.from\_response(response, #"http://www.zhihu.com/login",

meta = {'cookiejar' : response.meta['cookiejar']},

headers = self.headers, #注意此处的headers

formdata = {

'\_xsrf': xsrf,

'email': '1095511864@qq.com',

'password': '123456'

},

callback = self.after\_login,

dont\_filter = True

)]

#重写start\_request之后，就不会自动调用start\_url了，必须要手写一个方法调用start\_url

def after\_login(self, response) :

for url in self.start\_urls :

yield self.make\_requests\_from\_url(url)

def parse\_page(self, response):

problem = Selector(response)

item = ZhihuItem()

item['url'] = response.url

item['name'] = problem.xpath('//span[@class="name"]/text()').extract()

print item['name']

item['title'] = problem.xpath('//h2[@class="zm-item-title zm-editable-content"]/text()').extract()

item['description'] = problem.xpath('//div[@class="zm-editable-content"]/text()').extract()

item['answer']= problem.xpath('//div[@class=" zm-editable-content clearfix"]/text()').extract()

return item

#### Item类设置

from scrapy.item import Item, Field

class ZhihuItem(Item):

# define the fields for your item here like:

# name = scrapy.Field()

url = Field() #保存抓取问题的url

title = Field() #抓取问题的标题

description = Field() #抓取问题的描述

answer = Field() #抓取问题的答案

name = Field() #个人用户的名称

#### setting.py 设置抓取间隔

BOT\_NAME = 'zhihu'

SPIDER\_MODULES = ['zhihu.spiders']

NEWSPIDER\_MODULE = 'zhihu.spiders'

DOWNLOAD\_DELAY = 0.25 #设置下载间隔为250ms

## 爬取豆瓣电影top250并保存到mongodb中

爬取豆瓣电影top250http://[movie.douban.com/top250](file:///D:\David\Desktop\boxuegu\%E7%88%AC%E8%99%AB\file\part05\movie.douban.com\top250)的电影数据，并保存在MongoDB中。

### items.py

**import** scrapy

**class** DoubanItem(scrapy.Item):

*# 电影标题*

title = scrapy.Field()

*# 电影评分*

score = scrapy.Field()

*# 电影信息*

content = scrapy.Field()

*# 简介*

info = scrapy.Field()

### spiders/douban.py

**import** scrapy

**from** douban.items **import** DoubanItem

**class** DoubamovieSpider(scrapy.Spider):

name = **"doubanmovie"**

allowed\_domains = [**"movie.douban.com"**]

offset = 0

url = **"https://movie.douban.com/top250?start="**

start\_urls = (

url+str(offset),

)

**def** parse(self, response):

item = DoubanItem()

*#包含每一页的25个元素*

movies = response.xpath(**"//div[@class='info']"**)

**for** each **in** movies:

*# 标题*

item[**'title'**] = each.xpath(**".//span[@class='title'][1]/text()"**).extract()[0]

*# 信息*

content = each.xpath(**".//div[@class='bd']/p/text()"**).extract()

*# 以;作为分隔，将content列表里所有元素合并成一个新的字符串*

item[**'content'**] = **';'**.join(content)

*# 评分*

item[**'score'**] = each.xpath(**".//div[@class='star']/span[@class='rating\_num']/text()"**).extract()[0]

*# 简介*

info = each.xpath(**".//p[@class='quote']/span/text()"**).extract()

**if** len(info) != 0:

item[**'quote'**] = info[0]

*#发送到管道文件进行处理*

**yield** item

**if** self.offset < 225:

self.offset += 25

**yield** scrapy.Request(self.url + str(self.offset), callback = self.parse)

### settings.py

BOT\_NAME = **'douban'**

SPIDER\_MODULES = [**'douban.spiders'**]

NEWSPIDER\_MODULE = **'douban.spiders'**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

USER\_AGENT = **"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"**

DOWNLOAD\_DELAY = 2.5

COOKIES\_ENABLED = **False**

ITEM\_PIPELINES = {

**'douban.pipelines.DoubanPipeline'**: 300,

}

*# MONGODB 主机名*

MONGODB\_HOST = **"127.0.0.1"**

*# MONGODB 端口号*

MONGODB\_PORT = 27017

*# 数据库名称*

MONGODB\_DBNAME = **"DouBan"**

*# 存放数据的表名称*

MONGODB\_SHEETNAME = **'DouBanMovies'**

MONGODB\_DOCNAME = **'DouBanMovies'**

### pipelines.py

**import** pymongo

**from** scrapy.conf **import** settings

**class** DoubanPipeline(object):

**def** \_\_init\_\_(self):

*# 获取setting主机名、端口号和数据库名*

host = settings[**"MONGODB\_HOST"**]

port = settings[**"MONGODB\_PORT"**]

dbname = settings[**"MONGODB\_DBNAME"**]

sheetname= settings[**"MONGODB\_SHEETNAME"**]

*# pymongo.MongoClient(host, port)创建MONGODB数据库链接*

client = pymongo.MongoClient(host = host, port = port)

*# 指定数据库*

mydb = client[dbname]

*# 存放数据的数据库表名*

self.sheet = mydb[sheetname]

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

data = dict(item)

*# 向指定的表里添加数据*

self.sheet.insert(data)

**return** item

### 运行

### 附：MongoDB数据库操作

启动MongoDB数据库需要两个命令：

mongod：是mongoDB数据库进程本身，启动mongodb的服务

mongo：是命令行shell客户端

sudo mongod # 首先启动数据库服务，再执行Scrapy

#启动mongodb数据库服务之后，保持服务窗口不关闭，要再打开一个窗口来执行命令

sudo mongo # 启动数据库shell

在mongo shell下使用命令:

# 查看当前数据库

> db

# 列出所有的数据库

> show dbs

# 连接DouBan数据库

> use DouBan

# 列出所有表

> show collections

# 查看表里的数据

> db.DouBanMoives.find()

> db.DouBanMoives.findone()

# 删除当前数据库

> db.dropDatabase()

## [Downloader Middlewares](file:///D:\David\Desktop\boxuegu\%E7%88%AC%E8%99%AB\file\part04\4.8.html)

### 反反爬虫相关机制

Some websites implement certain measures to prevent bots from crawling them, with varying degrees of sophistication. Getting around those measures can be difficult and tricky, and may sometimes require special infrastructure. Please consider contacting commercial support if in doubt.

(有些些网站使用特定的不同程度的复杂性规则防止爬虫访问，绕过这些规则是困难和复杂的，有时可能需要特殊的基础设施，如果有疑问，请联系商业支持。)

来自于Scrapy官方文档描述：<http://doc.scrapy.org/en/master/topics/practices.html#avoiding-getting-banned>

### 通常防止爬虫被反主要有以下几个策略：

* 动态设置User-Agent（随机切换User-Agent，模拟不同用户的浏览器信息）
* 禁用Cookies（也就是不启用cookies middleware，不向Server发送cookies，有些网站通过cookie的使用发现爬虫行为）
  + 可以通过COOKIES\_ENABLED 控制 CookiesMiddleware 开启或关闭
* 设置延迟下载（防止访问过于频繁，设置为 2秒 或更高）
* Google Cache 和 Baidu Cache：如果可能的话，使用谷歌/百度等搜索引擎服务器页面缓存获取页面数据。
* 使用IP地址池：VPN和代理IP，现在大部分网站都是根据IP来ban的。
* 使用 [Crawlera](https://scrapinghub.com/crawlera)（专用于爬虫的代理组件），正确配置和设置下载中间件后，项目所有的request都是通过crawlera发出。

DOWNLOADER\_MIDDLEWARES = {

'scrapy\_crawlera.CrawleraMiddleware': 600

}

CRAWLERA\_ENABLED = True

CRAWLERA\_USER = '注册/购买的UserKey'

CRAWLERA\_PASS = '注册/购买的Password'

### 设置下载中间件（Downloader Middlewares）

下载中间件是处于引擎(crawler.engine)和下载器(crawler.engine.download())之间的一层组件，可以有多个下载中间件被加载运行。

下载中间件一般写在settings同级目录下。

1. 当引擎传递请求给下载器的过程中，下载中间件可以对请求进行处理 （例如增加http header信息，增加proxy信息等）；
2. 在下载器完成http请求，传递响应给引擎的过程中，下载中间件可以对响应进行处理（例如进行gzip的解压等）

要激活下载器中间件组件，将其加入到 DOWNLOADER\_MIDDLEWARES 设置中。该设置是一个字典(dict)，键为中间件类的路径，值为其中间件的顺序(order)。

这里是一个例子:

myspider为目录，middlewares为管道文件名，MyDownloaderMiddleware为类名。

DOWNLOADER\_MIDDLEWARES = {

'mySpider.middlewares.MyDownloaderMiddleware': 543,

}

编写下载器中间件十分简单。每个中间件组件是一个定义了以下一个或多个方法的Python类:

class scrapy.contrib.downloadermiddleware.DownloaderMiddleware

#### process\_request(self, request, spider)

* 当每个request通过下载中间件时，该方法被调用。
* 此方法必须要写
* process\_request()必须返回以下其中之一：一个 None、一个Response对象、一个Request对象或raise IgnoreRequest:
  + 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)。
  + 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。
  + 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。
  + 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)。
* 参数:
  + request (Request 对象) – 处理的request
  + spider (Spider 对象) – 该request对应的spider

#### process\_response(self, request, response, spider)

当下载器完成http请求，传递响应给引擎的时候调用

* process\_request() 必须返回以下其中之一: 返回一个 Response 对象、 返回一个 Request 对象或raise一个 IgnoreRequest 异常。
  + 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。
  + 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。
  + 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)。
* 参数:
  + request (Request 对象) – response所对应的request
  + response (Response 对象) – 被处理的response
  + spider (Spider 对象) – response所对应的spider

### 使用案例：给豆瓣电影爬虫添加下载中间件

#### 1. 创建middlewares.py文件。

Scrapy代理IP、Uesr-Agent的切换都是通过DOWNLOADER\_MIDDLEWARES进行控制，我们在settings.py同级目录下创建middlewares.py文件，包装所有请求。

# middlewares.py

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

**import** random

**import** base64

**from** douban.settings **import** USER\_AGENTS

**from** douban.settings **import** PROXIES

*# 随机的User-Agent*

**class** RandomUserAgent(object):

*#此方法是必须要调用的，且是固定的，不能修改名称*

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

useragent = random.choice(USER\_AGENTS)

*#print(useragent)*

request.headers.setdefault(**"User-Agent"**, useragent)

**class** RandomProxy(object):

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

proxy = random.choice(PROXIES)

*# print(proxy)*

**if** proxy[**'user\_passwd'**] **is None**:

*# 没有代理账户验证的代理使用方式*

request.meta[**'proxy'**] = **"http://"** + proxy[**'ip\_port'**]

**else**:

*# 对账户密码进行base64编码转换*

*#base64编码转换www1.tc711.com/tool/BASE64.htm*

base64\_userpasswd = base64.b64encode(proxy[**'user\_passwd'**])

*# 对应到代理服务器的信令格式里*

request.headers[**'Proxy-Authorization'**] = **'Basic '** + base64\_userpasswd

request.meta[**'proxy'**] = **"http://"** + proxy[**'ip\_port'**]

为什么HTTP代理要使用base64编码：

HTTP代理的原理很简单，就是通过HTTP协议与代理服务器建立连接，协议信令中包含要连接到的远程主机的IP和端口号，如果有需要身份验证的话还需要加上授权信息，服务器收到信令后首先进行身份验证，通过后便与远程主机建立连接，连接成功之后会返回给客户端200，表示验证通过，就这么简单，下面是具体的信令格式：

CONNECT 59.64.128.198:21 HTTP/1.1

Host: 59.64.128.198:21

Proxy-Authorization: Basic bGV2I1TU5OTIz

User-Agent: OpenFetion

其中Proxy-Authorization是身份验证信息，Basic后面的字符串是用户名和密码组合后进行base64编码的结果，也就是对username:password进行base64编码。

HTTP/1.0 200 Connection established

OK，客户端收到收面的信令后表示成功建立连接，接下来要发送给远程主机的数据就可以发送给代理服务器了，代理服务器建立连接后会在根据IP地址和端口号对应的连接放入缓存，收到信令后再根据IP地址和端口号从缓存中找到对应的连接，将数据通过该连接转发出去。

#### 2. 修改settings.py配置USER\_AGENTS和PROXIES

* 添加USER\_AGENTS：

USER\_AGENTS = [

**"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 2.0.50727; Media Center PC 6.0)"**,

**"Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 1.0.3705; .NET CLR 1.1.4322)"**,

**"Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.2; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.2; .NET CLR 3.0.04506.30)"**,

**"Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN) AppleWebKit/523.15 (KHTML, like Gecko, Safari/419.3) Arora/0.3 (Change: 287 c9dfb30)"**,

**"Mozilla/5.0 (X11; U; Linux; en-US) AppleWebKit/527+ (KHTML, like Gecko, Safari/419.3) Arora/0.6"**,

**"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.2pre) Gecko/20070215 K-Ninja/2.1.1"**,

**"Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9) Gecko/20080705 Firefox/3.0 Kapiko/3.0"**,

**"Mozilla/5.0 (X11; Linux i686; U;) Gecko/20070322 Kazehakase/0.4.5"**

]

* 添加代理IP设置PROXIES：

免费代理IP可以网上搜索，或者付费购买一批可用的私密代理IP：

PROXIES = [

*#有用户名和密码验证的代理*

{**"ip\_port"** :**"121.42.140.113:16816"**, **"user\_passwd"** : **"mr\_mao\_hacker:sffqry9r"**},

{**'ip\_port'**: **'111.8.60.9:8123'**, **'user\_passwd'**: **'user1:pass1'**},

{**'ip\_port'**: **'101.71.27.120:80'**, **'user\_passwd'**: **'user2:pass2'**},

*#没有用户名和密码验证的代理*

*#{"ip\_prot" :"121.42.140.113:16816", "user\_passwd" : ""}*

*#{"ip\_prot" :"121.42.140.113:16816", "user\_passwd" : ""}*

*#{"ip\_prot" :"121.42.140.113:16816", "user\_passwd" : ""}*

]

* 除非特殊需要，禁用cookies，防止某些网站根据Cookie来封锁爬虫。

COOKIES\_ENABLED = **False**

* 设置下载延迟

DOWNLOAD\_DELAY = 2.5

* 最后设置setting.py里的DOWNLOADER\_MIDDLEWARES，添加自己编写的下载中间件类。

DOWNLOADER\_MIDDLEWARES = {

**'douban.middlewares.RandomUserAgent'**: 100,

**'douban.middlewares.RandomProxy'**: 200,

}

## Settings

Scrapy设置(settings)提供了定制Scrapy组件的方法。可以控制包括核心(core)，插件(extension)，pipeline及spider组件。比如 设置Json Pipeliine、LOG\_LEVEL等。

参考文档：<http://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/settings.html#topics-settings-ref>

### 内置设置参考手册

* BOT\_NAME
  + 默认: 'scrapybot'
  + 当您使用 startproject 命令创建项目时其也被自动赋值。
* CONCURRENT\_ITEMS
  + 默认: 100
  + Item Processor(即 Item Pipeline) 同时处理(每个response的)item的最大值。

**CONCURRENT\_REQUESTS**

* + 默认: 16
  + Scrapy downloader 并发请求(concurrent requests)的最大值。

**DEFAULT\_REQUEST\_HEADERS**

* + 默认: 如下
  + {
  + 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8',
  + 'Accept-Language': 'en',
  + }

Scrapy HTTP Request使用的默认header。

其它地方都没有写的时候才会使用settings中的设置，如果其它地方定义了，则使用另外自定义的。

* DEPTH\_LIMIT
  + 默认: 0
  + 爬取网站最大允许的深度(depth)值。如果为0，则没有限制。

**DOWNLOAD\_DELAY**

* + 默认: 0
  + 下载器在下载同一个网站下一个页面前需要等待的时间。该选项可以用来限制爬取速度， 减轻服务器压力。同时也支持小数:

DOWNLOAD\_DELAY = 0.25 # 250 ms of delay

* + 默认情况下，Scrapy在两个请求间不等待一个固定的值， 而是使用0.5到1.5之间的一个随机值 \* DOWNLOAD\_DELAY 的结果作为等待间隔。
* DOWNLOAD\_TIMEOUT
  + 默认: 180
  + 下载器超时时间(单位: 秒)。

**ITEM\_PIPELINES**

* + 默认: {}
  + 保存项目中启用的pipeline及其顺序的字典。该字典默认为空，值(value)任意，不过值(value)习惯设置在0-1000范围内，值越小优先级越高。
  + ITEM\_PIPELINES = {
  + 'mySpider.pipelines.SomethingPipeline': 300,
  + 'mySpider.pipelines.ItcastJsonPipeline': 800,
  + }
* LOG\_ENABLED
  + 默认: True
  + 是否启用logging。
* LOG\_ENCODING
  + 默认: 'utf-8'
  + logging使用的编码。
* LOG\_LEVEL
  + 默认: 'DEBUG'
  + log的最低级别。可选的级别有: CRITICAL、ERROR、WARNING、INFO、DEBUG。调试的时候一般写debug信息。后期运维的话一般设置为info信息。

**USER\_AGENT**

* + 默认: "Scrapy/VERSION (+<http://scrapy.org>)"
  + 爬取的默认User-Agent，除非被覆盖。

**PROXIES：代理设置**

* + 示例：
  + PROXIES = [
  + {'ip\_port': '111.11.228.75:80', 'password': ''},
  + {'ip\_port': '120.198.243.22:80', 'password': ''},
  + {'ip\_port': '111.8.60.9:8123', 'password': ''},
  + {'ip\_port': '101.71.27.120:80', 'password': ''},
  + {'ip\_port': '122.96.59.104:80', 'password': ''},
  + {'ip\_port': '122.224.249.122:8088', 'password':''},
  + ]

**COOKIES\_ENABLED = False**

* + 禁用Cookies

# Scrapy实战项目

## 手机App抓包爬虫

### 通过Fiddler进行手机抓包

通过Fiddler抓包工具，可以抓取手机的网络通信，但前提是手机和电脑处于同一局域网内（WI-FI或热点），然后进行以下设置：

保证电脑和手机在同一个网段，使用同一个路由器，或者电脑设置为热点，手机连接电脑热点

设置手机wifi代理地址为电脑的地址，端口号为8888

就能够抓取到手机app的流量请求了。

在其中找到发送数据对应的url地址

找到传输数据的js文件，获得其url地址即可。

#### 用Fiddler对Android应用进行抓包

1. 打开Fiddler设置
2. 在Connections里设置允许连接远程计算机，确认后重新启动Fiddler
3. 在命令提示符下输入ipconfig查看本机IP
4. 打开Android设备的“设置”->“WLAN”，找到你要连接的网络，在上面长按，然后选择“修改网络”，弹出网络设置对话框，然后勾选“显示高级选项”。
5. 在“代理”后面的输入框选择“手动”，在“代理服务器主机名”后面的输入框输入电脑的ip地址，在“代理服务器端口”后面的输入框输入8888，然后点击“保存”按钮。
6. 启动Android设备中的浏览器，访问网页即可在Fiddler中可以看到完成的请求和响应数据。

#### 用Fiddler对iPhone手机应用进行抓包

基本流程差不多，只是手机设置不太一样：

iPhone手机：点击设置 > 无线局域网 > 无线网络 > HTTP代理 > 手动：

代理地址(电脑IP)：192.168.xx.xxx

端口号：8888

### 模拟斗鱼手机客户端爬虫

"""

limit是一页加载的数据个数，offset是偏移量，以<http://capi.douyucdn.cn/api/v1/getVerticalRoom?limit=20&offset=0>为例来进行说明，offset是从所有数据跳过的个数。offset=1则是跳过所有数据的第1个。在爬取数据的时候，只需要修改offset的值即可。offset=20，即抓取所有数据第二页的数据。

"""

#### 1. items.py

**import** scrapy

**class** DouyuItem(scrapy.Item):

*# define the fields for your item here like:*

nickname = scrapy.Field()*# 存储照片的名字*

imagelink = scrapy.Field()*# 照片的url路径*

imagePath = scrapy.Field()*# 照片保存在本地的路径*

#### 2. spiders/douyu.py

**import** scrapy, json

**from** douyu.items **import** DouyuItem

**class** DouyumeinvSpider(scrapy.Spider):

name = **"douyumeinv"**

allowed\_domains = [**"capi.douyucdn.cn"**]

offset = 0

url = **"http://capi.douyucdn.cn/api/v1/getVerticalRoom?limit=20&offset="**

start\_urls = [url + str(offset)]

**def** parse(self, response):

*# 返回从json里获取 data段数据集合*

*# 把json格式的数据转换为python格式，data段是列表*

data = json.loads(response.text)[**"data"**]

**for** each **in** data:

item = DouyuItem()

item[**"nickname"**] = each[**"nickname"**]

item[**"imagelink"**] = each[**"vertical\_src"**]

**yield** item

self.offset += 20

**yield** scrapy.Request(self.url + str(self.offset), callback = self.parse)

#### 3. 设置setting.py

BOT\_NAME = **'douyu'**

SPIDER\_MODULES = [**'douyu.spiders'**]

NEWSPIDER\_MODULE = **'douyu.spiders'**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#在抓包软件中查看*

USER\_AGENT = **'DYZB/2.290 (iPhone; iOS 9.3.4; Scale/2.00)'**

ITEM\_PIPELINES = {

**'douyu.pipelines.ImagesPipeline'**: 300,

}

*# Images 的存放位置，之后会在pipelines.py里调用*

IMAGES\_STORE = **"/Users/Power/lesson\_python/1103/day04/douyu/Images"**

#### 4. pipelines.py

**import** scrapy, os

*#获取设置文件，上面设置了图片的保存路径*

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

*#scrapy根目录的pipeline文件夹中有专门处理images的文件images.py，导入到我们的文件中使用*

**from** scrapy.pipelines.images **import** ImagesPipeline

**class** ImagesPipeline(ImagesPipeline):

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

*# return item*

*# 获取settings文件里设置的变量值*

IMAGES\_STORE = get\_project\_settings().get(**"IMAGES\_STORE"**)

*#在scarpy安装目录中的pipelines/images.py文件中，查找到这两个函数*

*#get\_media\_requests的作用就是为每一个图片链接生成一个Request对象，这个方法的输出将作为item\_completed的输入中的results，results是一个元组，每个元组包括(success, imageinfoorfailure)。如果success=true，imageinfoor\_failure是一个字典，包括url/path/checksum三个key。*

**def** get\_media\_requests(self, item, info):

image\_url = item[**"imagelink"**]

*#发送图片的请求*

**yield** scrapy.Request(image\_url)

*#通过此函数对获得的图片进行处理*

**def** item\_completed(self, result, item, info):

*# 固定写法，获取图片路径，同时判断这个路径是否正确，如果正确，就放到 image\_path里，ImagesPipeline源码剖析可见*

image\_path = [x[**"path"**] **for** ok, x **in** result **if** ok]

*#使用用户的昵称来重命名下载的图片*

os.rename(self.IMAGES\_STORE + **"/"** + image\_path[0], self.IMAGES\_STORE + **"/"** + item[**"nickname"**] + **".jpg"**)

item[**"imagePath"**] = self.IMAGES\_STORE + **"/"** + item[**"nickname"**]

**return** item

#### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl douyu'.split())

#### 执行程序

py2 main.py

## 阳光热线问政平台

<http://wz.sun0769.com/index.php/question/questionType?type=4>

爬取投诉帖子的编号、帖子的url、帖子的标题，和帖子里的内容。

response.url获取响应的url地址

### items.py

**import** scrapy

**class** DongguanItem(scrapy.Item):

*# 每个帖子的标题*

title = scrapy.Field()

*# 每个帖子的编号*

number = scrapy.Field()

*# 每个帖子的文字内容*

content = scrapy.Field()

*# 每个帖子的url*

url = scrapy.Field()

### Spiders/Crawl Spider

#### Spider 版本

# -\*- coding: utf-8 -\*-

**import** scrapy

**from** newdongguan.items **import** DongguanItem

**class** DongdongSpider(scrapy.Spider):

name = **'dongguan'**

allowed\_domains = [**'wz.sun0769.com'**]

url = **'http://wz.sun0769.com/index.php/question/questionType?type=4&page='**

offset = 0

start\_urls = [url + str(offset)]

**def** parse(self, response):

*# 取出每一页里的所有帖子的链接，返回一个列表*

links = response.xpath(**'//div[@class="greyframe"]/table//td/a[@class="news14"]/@href'**).extract()

*# 迭代取出集合里的链接*

**for** link **in** links:

*# 提取列表里每个帖子的链接，发送的请求并不是立即执行，而是把请求放到请求队列里,并调用self.parse\_item来处理*

**yield** scrapy.Request(link, callback = self.parse\_item)

*# 页面终止条件成立前，会一直自增offset的值，并发送新的页面请求，调用parse方法处理*

**if** self.offset <= 71160:

self.offset += 30

*# 发送请求放到请求队列里，调用self.parse处理response*

**yield** scrapy.Request(self.url + str(self.offset), callback = self.parse)

*# 处理每个帖子的response内容*

**def** parse\_item(self, response):

item = DongguanItem()

*# 标题*

item[**'title'**] = response.xpath(**'//div[contains(@class, "pagecenter p3")]//strong/text()'**).extract()[0]

*# 编号*

item[**'number'**] = item[**'title'**].split(**' '**)[-1].split(**":"**)[-1]

*# 链接*

item[**'url'**] = response.url

*# 有图片和无图片时内容所在的位置是不同的，先使用有图片情况下的匹配规则，如果有内容，返回所有内容的列表集合*

content = response.xpath(**'//div[@class="contentext"]/text()'**).extract()

*# 如果没有内容，则返回空列表，则使用无图片情况下的匹配规则*

**if** len(content) == 0:

content = response.xpath(**'//div[@class="c1 text14\_2"]/text()'**).extract()

*# content为列表，通过join方法拼接为字符串，并去除首尾空格*

item[**'content'**] = **""**.join(content).strip()

**else**:

item[**'content'**] = **""**.join(content).strip()

*# 把数据交给管道文件进行处理*

**yield** item

#### CrawlSpider 版本

# -\*- coding: utf-8 -\*-

**import** scrapy

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

**from** newdongguan.items **import** DongguanItem

**class** DongdongSpider(CrawlSpider):

name = **'dongdong'**

allowed\_domains = [**'wz.sun0769.com'**]

start\_urls = [**'http://wz.sun0769.com/index.php/question/questionType?type=4&page='**]

*# 每一页的匹配规则*

pagelink = LinkExtractor(allow=(**"type=4"**))

*# 每一页里的每个帖子的匹配规则*

contentlink = LinkExtractor(allow=(**r"/html/question/\d+/\d+.shtml"**))

rules = (

*# 本案例中返回的pagelink被web服务器篡改，需要调用process\_links来处理提取出来的url，这也是反爬虫的方式之一，短时间内大量的访问会返回一些假的url地址。*

*#如果没有回调函数，follow默认为True*

Rule(pagelink, process\_links = **"deal\_links"**),

*#CrawlSpider中的callback函数必须用''括起来。*

Rule(contentlink, callback = **"parse\_item"**)

)

*# 需要重新处理每个页面里的链接，将链接里的'Type&type=4?page=xxx'替换为'Type?type=4&page=xxx'（或者是Type&page=xxx?type=4'替换为'Type?page=xxx&type=4'），否则无法发送这个链接*

*# links 是当前response里提取出来的链接列表*

**def** deal\_links(self, links):

**for** each **in** links:

each.url = each.url.replace(**"?"**,**"&"**).replace(**"Type&"**,**"Type?"**)

*#或使用re.sub(s1,s2,s3)*

*#返回修改后的links链接列表*

**return** links

**def** parse\_item(self, response):

item = DongguanItem()

*# 标题*

item[**'title'**] = response.xpath(**'//div[contains(@class, "pagecenter p3")]//strong/text()'**).extract()[0]

*# 编号*

item[**'number'**] = item[**'title'**].split(**' '**)[-1].split(**":"**)[-1]

*# 链接，使用response.url来获取链接*

item[**'url'**] = response.url

*# 有图片和无图片时内容所在的位置是不同的，先使用有图片情况下的匹配规则，如果有内容，返回所有内容的列表集合*

content = response.xpath(**'//div[@class="contentext"]/text()'**).extract()

*# 如果没有内容，则返回空列表，则使用无图片情况下的匹配规则*

**if** len(content) == 0:

content = response.xpath(**'//div[@class="c1 text14\_2"]/text()'**).extract()

item[**'content'**] = **""**.join(content).strip()

**else**:

item[**'content'**] = **""**.join(content).strip()

*# 把数据交给管道文件进行处理*

**yield** item

### pipelines.py

# -\*- coding: utf-8 -\*-

**import** codecs, json

*# codecs是文件处理类库，可以指定编码格式*

**class** NewdongguanPipeline(object):

**def** \_\_init\_\_(self):

*# 创建一个文件，指定文本存储的格式为utf-8*

*#在整个爬虫运行期间，管道文件是始终打开的，所以这里不用使用追加写入的方式。*

self.filename = codecs.open(**"donggguan.json"**, **"w"**, encoding = **"utf-8"**)

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

*# 中文默认使用ascii码来存储，禁用后默认为Unicode字符串*

content = json.dumps(dict(item), ensure\_ascii=**False**) + **",\n"**

self.filename.write(content)

*#如果在\_\_init\_\_方法中不指定创建的文件的格式，就要在这里指定写入的内容的格式*

*# self.filename.write(content.encode("utf-8"))*

**return** item

**def** close\_spider(self, spider):

self.filename.close()

### settings.py

BOT\_NAME = **'newdongguan'**

SPIDER\_MODULES = [**'newdongguan.spiders'**]

NEWSPIDER\_MODULE = **'newdongguan.spiders'**

ITEM\_PIPELINES = {

'dongguan.pipelines.DongguanPipeline': 300,

}

#设置请求头

#提供了一个单独的USER-AGENT

#可以在DEFAULT\_REQUEST\_HEADERS中设置User-Agent，也可以直接在USER-AGENT中设置。

USER\_AGENT = **"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"**

ITEM\_PIPELINES = {

**'newdongguan.pipelines.NewdongguanPipeline'**: 1,

}

# 日志文件名和处理等级

LOG\_FILE = **"dongguan.log"**

LOG\_LEVEL = **"DEBUG"**

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl sunwz'.split())

### 执行程序

py2 main.py

## 新浪网分类资讯爬虫

爬取新浪网导航页所有下所有大类、小类、小类里的子链接，以及子链接页面的新闻内容。

### items.py

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### spiders/sina.py

# -\*- coding: utf-8 -\*-

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

import scrapy

import os

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaSpider(scrapy.Spider):

name= "sina"

allowed\_domains= ["sina.com.cn"]

start\_urls= [

"http://news.sina.com.cn/guide/"

]

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类目录的路径和目录名

parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在，则创建目录

if(not os.path.exists(parentFilename)):

os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头，如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类，将存储目录放在本大类目录下

if(if\_belong):

subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在，则创建目录

if(not os.path.exists(subFilename)):

os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求，得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url，再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾，如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类，获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] = meta\_1['subUrls']

item['subTitle'] = meta\_1['subTitle']

item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求，得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法，获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()')

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head

item['content']= content

yield item

### pipelines.py

from scrapy import signals

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaPipeline(object):

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

sonUrls = item['sonUrls']

# 文件名为子链接url中间部分，并将 / 替换为 \_，保存为 .txt格式

filename = sonUrls[7:-6].replace('/','\_')

filename += ".txt"

fp = open(item['subFilename']+'/'+filename, 'w')

fp.write(item['content'])

fp.close()

return item

### settings.py

BOT\_NAME = 'Sina'

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

ITEM\_PIPELINES = {

'Sina.pipelines.SinaPipeline': 300,

}

LOG\_LEVEL = 'DEBUG'

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl sina'.split())

### 执行程序

py2 main.py

## [图片下载器爬虫](file:///D:\David\Desktop\boxuegu\%E7%88%AC%E8%99%AB\file\part05\5.4.html)

### items.py

class CoserItem(scrapy.Item):

url = scrapy.Field()

name = scrapy.Field()

info = scrapy.Field()

image\_urls = scrapy.Field()

images = scrapy.Field()

### spiders/coser.py

# -\*- coding: utf-8 -\*-

**from** scrapy.selector **import** Selector

**import** scrapy

**from** scrapy.contrib.loader **import** ItemLoader

**from** Cosplay.items **import** CoserItem

**class** CoserSpider(scrapy.Spider):

name = **"coser"**

allowed\_domains = [**"bcy.net"**]

start\_urls = (

**'http://bcy.net/cn125101'**,

**'http://bcy.net/cn126487'**,

**'http://bcy.net/cn126173'**

)

**def** parse(self, response):

sel = Selector(response)

**for** link **in** sel.xpath(**"//ul[@class='js-articles l-works']/li[@class='l-work--big']/article[@class='work work--second-created']/h2[@class='work\_\_title']/a/@href"**).extract():

link = **'http://bcy.net%s'** % link

request = scrapy.Request(link, callback=self.parse\_item)

**yield** request

**def** parse\_item(self, response):

*#使用ItemLoader创建一个实例对象*

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

*#指定使用的解析方法，name为字段名，后面所取的值直接放在前面的字段中。*

l.add\_xpath(**'name'**, **"//h1[@class='js-post-title']/text()"**)

l.add\_xpath(**'info'**, **"//div[@class='post\_\_info']/div[@class='post\_\_type post\_\_info-group']/span/text()"**)

*#从l这个实例对象中直接取值，并赋值给urls*

urls = l.get\_xpath(**'//img[@class="detail\_std detail\_clickable"]/@src'**)

urls = [url.replace(**'/w650'**, **''**) **for** url **in** urls]

*#把urls的值放到image\_urls字段中。*

l.add\_value(**'image\_urls'**, urls)

l.add\_value(**'url'**, response.url)

**return** l.load\_item()

### pipelines.py

import requests

from Cosplay import settings

import os

class ImageDownloadPipeline(object):

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

if 'image\_urls' in item:

images = []

dir\_path = '%s/%s' % (settings.IMAGES\_STORE, spider.name)

if not os.path.exists(dir\_path):

os.makedirs(dir\_path)

for image\_url in item['image\_urls']:

us = image\_url.split('/')[3:]

image\_file\_name = '\_'.join(us)

file\_path = '%s/%s' % (dir\_path, image\_file\_name)

images.append(file\_path)

if os.path.exists(file\_path):

continue

with open(file\_path, 'wb') as handle:

response = requests.get(image\_url, stream=True)

for block in response.iter\_content(1024):

if not block:

break

handle.write(block)

item['images'] = images

return item

### settings.py

ITEM\_PIPELINES = {'Cosplay.pipelines.ImageDownloadPipeline': 1}

IMAGES\_STORE = '../Images'

DOWNLOAD\_DELAY = 0.25 # 250 ms of delay

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl coser'.split())

### 执行程序

py2 main.py

使用在sublime中使用 :vsp打开一个水平方向并排的窗口

# scrapy-redis分布式组件

## Scrapy 和 scrapy-redis的区别

Scrapy 是一个通用的爬虫框架，但是不支持分布式，Scrapy-redis是为了更方便地实现Scrapy分布式爬取，而提供了一些以redis为基础的组件(仅有组件)。

数据存储在redis数据库中，是一个基于内存的数据库。

pip install scrapy-redis

Scrapy-redis提供了下面四种组件（components）：(四种组件意味着这四个模块都要做相应的修改)

* Scheduler
* Duplication Filter
* Item Pipeline
* Base Spider

## scrapy-redis架构

如上图所⽰示，scrapy-redis在scrapy的架构上增加了redis，基于redis的特性拓展了如下组件：

#### Scheduler：

Scrapy改造了python本来的collection.deque(双向队列)形成了自己的Scrapy queue([https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py)](https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py))，但是Scrapy多个spider不能共享待爬取队列Scrapy queue， 即Scrapy本身不支持爬虫分布式，scrapy-redis 的解决是把这个Scrapy queue换成redis数据库（也是指redis队列），从同一个redis-server存放要爬取的request，便能让多个spider去同一个数据库里读取。

Scrapy中跟“待爬队列”直接相关的就是调度器Scheduler，它负责对新的request进行入列操作（加入Scrapy queue），取出下一个要爬取的request（从Scrapy queue中取出）等操作。它把待爬队列按照优先级建立了一个字典结构，比如：

{

优先级0 : 队列0

优先级1 : 队列1

优先级2 : 队列2

}

然后根据request中的优先级，来决定该入哪个队列，出列时则按优先级较小的优先出列。为了管理这个比较高级的队列字典，Scheduler需要提供一系列的方法。但是原来的Scheduler已经无法使用，所以使用Scrapy-redis的scheduler组件。

#### Duplication Filter

Scrapy中用集合实现这个request去重功能，Scrapy中把已经发送的request指纹放入到一个集合中，把下一个request的指纹拿到集合中比对，如果该指纹存在于集合中，说明这个request发送过了，如果没有则继续操作。这个核心的判重功能是这样实现的：

def request\_seen(self, request):

# self.request\_figerprints就是一个指纹集合

fp = self.request\_fingerprint(request)

# 这就是判重的核心操作

if fp in self.fingerprints:

return True

self.fingerprints.add(fp)

if self.file:

self.file.write(fp + os.linesep)

在scrapy-redis中去重是由Duplication Filter组件来实现的，它通过redis的set 不重复的特性，巧妙的实现了Duplication Filter去重。scrapy-redis调度器从引擎接受request，将request的指纹存⼊redis的set检查是否重复，并将不重复的request push写⼊redis的 request queue。

引擎请求request(Spider发出的）时，调度器从redis的request queue队列⾥里根据优先级pop 出⼀个request 返回给引擎，引擎将此request发给spider处理。

#### Item Pipeline：

引擎将(Spider返回的)爬取到的Item给Item Pipeline，scrapy-redis 的Item Pipeline将爬取到的 Item 存⼊redis的 items queue。

修改过Item Pipeline可以很方便的根据 key 从 items queue 提取item，从⽽实现 items processes集群。

#### Base Spider

不在使用scrapy原有的Spider类，重写的RedisSpider继承了Spider和RedisMixin这两个类，RedisMixin是用来从redis读取url的类。

当我们生成一个Spider继承RedisSpider时，调用setup\_redis函数，这个函数会去连接redis数据库，然后会设置signals(信号)：

* 一个是当spider空闲时候的signal，会调用spider\_idle函数，这个函数调用schedule\_next\_request函数，保证spider是一直活着的状态，并且抛出DontCloseSpider异常。
* 一个是当抓到一个item时的signal，会调用item\_scraped函数，这个函数会调用schedule\_next\_request函数，获取下一个request。

## 源码分析参考：Connection

官方站点：<https://github.com/rolando/scrapy-redis>

scrapy-redis的官方文档写的比较简洁，没有提及其运行原理，所以如果想全面的理解分布式爬虫的运行原理，还是得看scrapy-redis的源代码才行。

scrapy-redis工程的主体还是是redis和scrapy两个库，工程本身实现的东西不是很多，这个工程就像胶水一样，把这两个插件粘结了起来。下面我们来看看，scrapy-redis的每一个源代码文件都实现了什么功能，最后如何实现分布式的爬虫系统：

1. [connection.py](https://github.com/rolando/scrapy-redis/blob/master/src/scrapy_redis/connection.py)

负责根据setting中配置实例化redis连接。被dupefilter和scheduler调用，总之涉及到redis存取的都要使用到这个模块。

# 这里引入了redis模块，这个是redis-python库的接口，用于通过python访问redis数据库，

# 这个文件主要是实现连接redis数据库的功能，这些连接接口在其他文件中经常被用到

import redis

import six

from scrapy.utils.misc import load\_object

DEFAULT\_REDIS\_CLS = redis.StrictRedis

# 可以在settings文件中配置套接字的超时时间、等待时间等

# Sane connection defaults.

DEFAULT\_PARAMS = {

'socket\_timeout': 30,

'socket\_connect\_timeout': 30,

'retry\_on\_timeout': True,

}

# 要想连接到redis数据库，和其他数据库差不多，需要一个ip地址、端口号、用户名密码（可选）和一个整形的数据库编号

# Shortcut maps 'setting name' -> 'parmater name'.

SETTINGS\_PARAMS\_MAP = {

'REDIS\_URL': 'url',

'REDIS\_HOST': 'host',

'REDIS\_PORT': 'port',

}

def get\_redis\_from\_settings(settings):

"""Returns a redis client instance from given Scrapy settings object.

This function uses ``get\_client`` to instantiate the client and uses

``DEFAULT\_PARAMS`` global as defaults values for the parameters. You can

override them using the ``REDIS\_PARAMS`` setting.

Parameters

----------

settings : Settings

A scrapy settings object. See the supported settings below.

Returns

-------

server

Redis client instance.

Other Parameters

----------------

REDIS\_URL : str, optional

Server connection URL.

REDIS\_HOST : str, optional

Server host.

REDIS\_PORT : str, optional

Server port.

REDIS\_PARAMS : dict, optional

Additional client parameters.

"""

params = DEFAULT\_PARAMS.copy()

params.update(settings.getdict('REDIS\_PARAMS'))

# XXX: Deprecate REDIS\_\* settings.

for source, dest in SETTINGS\_PARAMS\_MAP.items():

val = settings.get(source)

if val:

params[dest] = val

# Allow ``redis\_cls`` to be a path to a class.

if isinstance(params.get('redis\_cls'), six.string\_types):

params['redis\_cls'] = load\_object(params['redis\_cls'])

# 返回的是redis库的Redis对象，可以直接用来进行数据操作的对象

return get\_redis(\*\*params)

# Backwards compatible alias.

from\_settings = get\_redis\_from\_settings

def get\_redis(\*\*kwargs):

"""Returns a redis client instance.

Parameters

----------

redis\_cls : class, optional

Defaults to ``redis.StrictRedis``.

url : str, optional

If given, ``redis\_cls.from\_url`` is used to instantiate the class.

\*\*kwargs

Extra parameters to be passed to the ``redis\_cls`` class.

Returns

-------

server

Redis client instance.

"""

redis\_cls = kwargs.pop('redis\_cls', DEFAULT\_REDIS\_CLS)

url = kwargs.pop('url', None)

if url:

return redis\_cls.from\_url(url, \*\*kwargs)

else:

return redis\_cls(\*\*kwargs)

Copyright © BigCat all right reserved，powered by Gitbook「Revision Time: 2016-12-13 03:01:26」

## 源码分析参考：Dupefilter

**dupefilter.py**

负责执行requst的去重，实现的很有技巧性，使用redis的set数据结构。但是注意scheduler并不使用其中用于在这个模块中实现的dupefilter键做request的调度，而是使用queue.py模块中实现的queue。

当request不重复时，将其存入到queue中，调度时将其弹出。

import logging

import time

from scrapy.dupefilters import BaseDupeFilter

from scrapy.utils.request import request\_fingerprint

from .connection import get\_redis\_from\_settings

DEFAULT\_DUPEFILTER\_KEY = "dupefilter:%(timestamp)s"

logger = logging.getLogger(\_\_name\_\_)

# TODO: Rename class to RedisDupeFilter.

class RFPDupeFilter(BaseDupeFilter):

"""Redis-based request duplicates filter.

This class can also be used with default Scrapy's scheduler.

"""

logger = logger

def \_\_init\_\_(self, server, key, debug=False):

"""Initialize the duplicates filter.

Parameters

----------

server : redis.StrictRedis

The redis server instance.

key : str

Redis key Where to store fingerprints.

debug : bool, optional

Whether to log filtered requests.

"""

self.server = server

self.key = key

self.debug = debug

self.logdupes = True

@classmethod

def from\_settings(cls, settings):

"""Returns an instance from given settings.

This uses by default the key ``dupefilter:<timestamp>``. When using the

``scrapy\_redis.scheduler.Scheduler`` class, this method is not used as

it needs to pass the spider name in the key.

Parameters

----------

settings : scrapy.settings.Settings

Returns

-------

RFPDupeFilter

A RFPDupeFilter instance.

"""

server = get\_redis\_from\_settings(settings)

# XXX: This creates one-time key. needed to support to use this

# class as standalone dupefilter with scrapy's default scheduler

# if scrapy passes spider on open() method this wouldn't be needed

# TODO: Use SCRAPY\_JOB env as default and fallback to timestamp.

key = DEFAULT\_DUPEFILTER\_KEY % {'timestamp': int(time.time())}

debug = settings.getbool('DUPEFILTER\_DEBUG')

return cls(server, key=key, debug=debug)

@classmethod

def from\_crawler(cls, crawler):

"""Returns instance from crawler.

Parameters

----------

crawler : scrapy.crawler.Crawler

Returns

-------

RFPDupeFilter

Instance of RFPDupeFilter.

"""

return cls.from\_settings(crawler.settings)

def request\_seen(self, request):

"""Returns True if request was already seen.

Parameters

----------

request : scrapy.http.Request

Returns

-------

bool

"""

fp = self.request\_fingerprint(request)

# This returns the number of values added, zero if already exists.

added = self.server.sadd(self.key, fp)

return added == 0

def request\_fingerprint(self, request):

"""Returns a fingerprint for a given request.

Parameters

----------

request : scrapy.http.Request

Returns

-------

str

"""

return request\_fingerprint(request)

def close(self, reason=''):

"""Delete data on close. Called by Scrapy's scheduler.

Parameters

----------

reason : str, optional

"""

self.clear()

def clear(self):

"""Clears fingerprints data."""

self.server.delete(self.key)

def log(self, request, spider):

"""Logs given request.

Parameters

----------

request : scrapy.http.Request

spider : scrapy.spiders.Spider

"""

if self.debug:

msg = "Filtered duplicate request: %(request)s"

self.logger.debug(msg, {'request': request}, extra={'spider': spider})

elif self.logdupes:

msg = ("Filtered duplicate request %(request)s"

" - no more duplicates will be shown"

" (see DUPEFILTER\_DEBUG to show all duplicates)")

msg = "Filtered duplicate request: %(request)s"

self.logger.debug(msg, {'request': request}, extra={'spider': spider})

self.logdupes = False

这个文件看起来比较复杂，重写了scrapy本身已经实现的request判重功能。因为本身scrapy单机跑的话，只需要读取内存中的request队列或者持久化的request队列（scrapy默认的持久化似乎是json格式的文件，不是数据库）就能判断这次要发出的request url是否已经请求过或者正在调度（本地读就行了）。而分布式跑的话，就需要各个主机上的scheduler都连接同一个数据库的同一个request池来判断这次的请求是否是重复的了。

在这个文件中，通过继承BaseDupeFilter重写他的方法，实现了基于redis的判重。根据源代码来看，scrapy-redis使用了scrapy本身的一个fingerprint接request\_fingerprint，这个接口很有趣，根据scrapy文档所说，他通过hash来判断两个url是否相同（相同的url会生成相同的hash结果），但是当两个url的地址相同，get型参数相同但是顺序不同时，也会生成相同的hash结果（这个真的比较神奇。。。）所以scrapy-redis依旧使用url的fingerprint来判断request请求是否已经出现过。

这个类通过连接redis，使用一个key来向redis的一个set中插入fingerprint（这个key对于同一种spider是相同的，redis是一个key-value的数据库，如果key是相同的，访问到的值就是相同的，这里使用spider名字+DupeFilter的key就是为了在不同主机上的不同爬虫实例，只要属于同一种spider，就会访问到同一个set，而这个set就是他们的url判重池），如果返回值为0，说明该set中该fingerprint已经存在（因为集合是没有重复值的），则返回False，如果返回值为1，说明添加了一个fingerprint到set中，则说明这个request没有重复，于是返回True，还顺便把新fingerprint加入到数据库中了。 DupeFilter判重会在scheduler类中用到，每一个request在进入调度之前都要进行判重，如果重复就不需要参加调度，直接舍弃就好了，不然就是白白浪费资源。

## 源码分析参考：Picklecompat

**picklecompat.py**

"""A pickle wrapper module with protocol=-1 by default."""

try:

import cPickle as pickle # PY2

except ImportError:

import pickle

def loads(s):

return pickle.loads(s)

def dumps(obj):

return pickle.dumps(obj, protocol=-1)

这里实现了loads和dumps两个函数，其实就是实现了一个序列化器。

因为redis数据库不能存储复杂对象（key部分只能是字符串，value部分只能是字符串，字符串列表，字符串集合和hash），所以我们存啥都要先串行化成文本才行。

这里使用的就是python的pickle模块，一个兼容py2和py3的串行化工具。这个serializer主要用于一会的scheduler存reuqest对象。

## 源码分析参考：Pipelines

**pipelines.py**

这是是用来实现分布式处理的作用。它将Item存储在redis中以实现分布式处理。由于在这里需要读取配置，所以就用到了from\_crawler()函数。

from scrapy.utils.misc import load\_object

from scrapy.utils.serialize import ScrapyJSONEncoder

from twisted.internet.threads import deferToThread

from . import connection

default\_serialize = ScrapyJSONEncoder().encode

class RedisPipeline(object):

"""Pushes serialized item into a redis list/queue"""

def \_\_init\_\_(self, server,

key='%(spider)s:items',

serialize\_func=default\_serialize):

self.server = server

self.key = key

self.serialize = serialize\_func

@classmethod

def from\_settings(cls, settings):

params = {

'server': connection.from\_settings(settings),

}

if settings.get('REDIS\_ITEMS\_KEY'):

params['key'] = settings['REDIS\_ITEMS\_KEY']

if settings.get('REDIS\_ITEMS\_SERIALIZER'):

params['serialize\_func'] = load\_object(

settings['REDIS\_ITEMS\_SERIALIZER']

)

return cls(\*\*params)

@classmethod

def from\_crawler(cls, crawler):

return cls.from\_settings(crawler.settings)

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

return deferToThread(self.\_process\_item, item, spider)

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

key = self.item\_key(item, spider)

data = self.serialize(item)

self.server.rpush(key, data)

return item

def item\_key(self, item, spider):

"""Returns redis key based on given spider.

Override this function to use a different key depending on the item

and/or spider.

"""

return self.key % {'spider': spider.name}

pipelines文件实现了一个item pipieline类，和scrapy的item pipeline是同一个对象，通过从settings中拿到我们配置的REDIS\_ITEMS\_KEY作为key，把item串行化之后存入redis数据库对应的value中（这个value可以看出出是个list，我们的每个item是这个list中的一个结点），这个pipeline把提取出的item存起来，主要是为了方便我们延后处理数据。

## 源码分析参考：Queue

**queue.py**

该文件实现了几个容器类，可以看这些容器和redis交互频繁，同时使用了我们上边picklecompat中定义的序列化器。这个文件实现的几个容器大体相同，只不过一个是队列，一个是栈，一个是优先级队列，这三个容器到时候会被scheduler对象实例化，来实现request的调度。比如我们使用SpiderQueue最为调度队列的类型，到时候request的调度方法就是先进先出，而实用SpiderStack就是先进后出了。

从SpiderQueue的实现看出来，他的push函数就和其他容器的一样，只不过push进去的request请求先被scrapy的接口request\_to\_dict变成了一个dict对象（因为request对象实在是比较复杂，有方法有属性不好串行化），之后使用picklecompat中的serializer串行化为字符串，然后使用一个特定的key存入redis中（该key在同一种spider中是相同的）。而调用pop时，其实就是从redis用那个特定的key去读其值（一个list），从list中读取最早进去的那个，于是就先进先出了。 这些容器类都会作为scheduler调度request的容器，scheduler在每个主机上都会实例化一个，并且和spider一一对应，所以分布式运行时会有一个spider的多个实例和一个scheduler的多个实例存在于不同的主机上，但是，因为scheduler都是用相同的容器，而这些容器都连接同一个redis服务器，又都使用spider名加queue来作为key读写数据，所以不同主机上的不同爬虫实例公用一个request调度池，实现了分布式爬虫之间的统一调度。

from scrapy.utils.reqser import request\_to\_dict, request\_from\_dict

from . import picklecompat

class Base(object):

"""Per-spider queue/stack base class"""

def \_\_init\_\_(self, server, spider, key, serializer=None):

"""Initialize per-spider redis queue.

Parameters:

server -- redis connection

spider -- spider instance

key -- key for this queue (e.g. "%(spider)s:queue")

"""

if serializer is None:

# Backward compatibility.

# TODO: deprecate pickle.

serializer = picklecompat

if not hasattr(serializer, 'loads'):

raise TypeError("serializer does not implement 'loads' function: %r"

% serializer)

if not hasattr(serializer, 'dumps'):

raise TypeError("serializer '%s' does not implement 'dumps' function: %r"

% serializer)

self.server = server

self.spider = spider

self.key = key % {'spider': spider.name}

self.serializer = serializer

def \_encode\_request(self, request):

"""Encode a request object"""

obj = request\_to\_dict(request, self.spider)

return self.serializer.dumps(obj)

def \_decode\_request(self, encoded\_request):

"""Decode an request previously encoded"""

obj = self.serializer.loads(encoded\_request)

return request\_from\_dict(obj, self.spider)

def \_\_len\_\_(self):

"""Return the length of the queue"""

raise NotImplementedError

def push(self, request):

"""Push a request"""

raise NotImplementedError

def pop(self, timeout=0):

"""Pop a request"""

raise NotImplementedError

def clear(self):

"""Clear queue/stack"""

self.server.delete(self.key)

class SpiderQueue(Base):

"""Per-spider FIFO queue"""

def \_\_len\_\_(self):

"""Return the length of the queue"""

return self.server.llen(self.key)

def push(self, request):

"""Push a request"""

self.server.lpush(self.key, self.\_encode\_request(request))

def pop(self, timeout=0):

"""Pop a request"""

if timeout > 0:

data = self.server.brpop(self.key, timeout)

if isinstance(data, tuple):

data = data[1]

else:

data = self.server.rpop(self.key)

if data:

return self.\_decode\_request(data)

class SpiderPriorityQueue(Base):

"""Per-spider priority queue abstraction using redis' sorted set"""

def \_\_len\_\_(self):

"""Return the length of the queue"""

return self.server.zcard(self.key)

def push(self, request):

"""Push a request"""

data = self.\_encode\_request(request)

score = -request.priority

# We don't use zadd method as the order of arguments change depending on

# whether the class is Redis or StrictRedis, and the option of using

# kwargs only accepts strings, not bytes.

self.server.execute\_command('ZADD', self.key, score, data)

def pop(self, timeout=0):

"""

Pop a request

timeout not support in this queue class

"""

# use atomic range/remove using multi/exec

pipe = self.server.pipeline()

pipe.multi()

pipe.zrange(self.key, 0, 0).zremrangebyrank(self.key, 0, 0)

results, count = pipe.execute()

if results:

return self.\_decode\_request(results[0])

class SpiderStack(Base):

"""Per-spider stack"""

def \_\_len\_\_(self):

"""Return the length of the stack"""

return self.server.llen(self.key)

def push(self, request):

"""Push a request"""

self.server.lpush(self.key, self.\_encode\_request(request))

def pop(self, timeout=0):

"""Pop a request"""

if timeout > 0:

data = self.server.blpop(self.key, timeout)

if isinstance(data, tuple):

data = data[1]

else:

data = self.server.lpop(self.key)

if data:

return self.\_decode\_request(data)

\_\_all\_\_ = ['SpiderQueue', 'SpiderPriorityQueue', 'SpiderStack']

## 源码分析参考：Scheduler

**scheduler.py**

此扩展是对scrapy中自带的scheduler的替代（在settings的SCHEDULER变量中指出），正是利用此扩展实现crawler的分布式调度。其利用的数据结构来自于queue中实现的数据结构。

scrapy-redis所实现的两种分布式：爬虫分布式以及item处理分布式就是由模块scheduler和模块pipelines实现。上述其它模块作为为二者辅助的功能模块

import importlib

import six

from scrapy.utils.misc import load\_object

from . import connection

# TODO: add SCRAPY\_JOB support.

class Scheduler(object):

"""Redis-based scheduler"""

def \_\_init\_\_(self, server,

persist=False,

flush\_on\_start=False,

queue\_key='%(spider)s:requests',

queue\_cls='scrapy\_redis.queue.SpiderPriorityQueue',

dupefilter\_key='%(spider)s:dupefilter',

dupefilter\_cls='scrapy\_redis.dupefilter.RFPDupeFilter',

idle\_before\_close=0,

serializer=None):

"""Initialize scheduler.

Parameters

----------

server : Redis

The redis server instance.

persist : bool

Whether to flush requests when closing. Default is False.

flush\_on\_start : bool

Whether to flush requests on start. Default is False.

queue\_key : str

Requests queue key.

queue\_cls : str

Importable path to the queue class.

dupefilter\_key : str

Duplicates filter key.

dupefilter\_cls : str

Importable path to the dupefilter class.

idle\_before\_close : int

Timeout before giving up.

"""

if idle\_before\_close < 0:

raise TypeError("idle\_before\_close cannot be negative")

self.server = server

self.persist = persist

self.flush\_on\_start = flush\_on\_start

self.queue\_key = queue\_key

self.queue\_cls = queue\_cls

self.dupefilter\_cls = dupefilter\_cls

self.dupefilter\_key = dupefilter\_key

self.idle\_before\_close = idle\_before\_close

self.serializer = serializer

self.stats = None

def \_\_len\_\_(self):

return len(self.queue)

@classmethod

def from\_settings(cls, settings):

kwargs = {

'persist': settings.getbool('SCHEDULER\_PERSIST'),

'flush\_on\_start': settings.getbool('SCHEDULER\_FLUSH\_ON\_START'),

'idle\_before\_close': settings.getint('SCHEDULER\_IDLE\_BEFORE\_CLOSE'),

}

# If these values are missing, it means we want to use the defaults.

optional = {

# TODO: Use custom prefixes for this settings to note that are

# specific to scrapy-redis.

'queue\_key': 'SCHEDULER\_QUEUE\_KEY',

'queue\_cls': 'SCHEDULER\_QUEUE\_CLASS',

'dupefilter\_key': 'SCHEDULER\_DUPEFILTER\_KEY',

# We use the default setting name to keep compatibility.

'dupefilter\_cls': 'DUPEFILTER\_CLASS',

'serializer': 'SCHEDULER\_SERIALIZER',

}

for name, setting\_name in optional.items():

val = settings.get(setting\_name)

if val:

kwargs[name] = val

# Support serializer as a path to a module.

if isinstance(kwargs.get('serializer'), six.string\_types):

kwargs['serializer'] = importlib.import\_module(kwargs['serializer'])

server = connection.from\_settings(settings)

# Ensure the connection is working.

server.ping()

return cls(server=server, \*\*kwargs)

@classmethod

def from\_crawler(cls, crawler):

instance = cls.from\_settings(crawler.settings)

# FIXME: for now, stats are only supported from this constructor

instance.stats = crawler.stats

return instance

def open(self, spider):

self.spider = spider

try:

self.queue = load\_object(self.queue\_cls)(

server=self.server,

spider=spider,

key=self.queue\_key % {'spider': spider.name},

serializer=self.serializer,

)

except TypeError as e:

raise ValueError("Failed to instantiate queue class '%s': %s",

self.queue\_cls, e)

try:

self.df = load\_object(self.dupefilter\_cls)(

server=self.server,

key=self.dupefilter\_key % {'spider': spider.name},

debug=spider.settings.getbool('DUPEFILTER\_DEBUG'),

)

except TypeError as e:

raise ValueError("Failed to instantiate dupefilter class '%s': %s",

self.dupefilter\_cls, e)

if self.flush\_on\_start:

self.flush()

# notice if there are requests already in the queue to resume the crawl

if len(self.queue):

spider.log("Resuming crawl (%d requests scheduled)" % len(self.queue))

def close(self, reason):

if not self.persist:

self.flush()

def flush(self):

self.df.clear()

self.queue.clear()

def enqueue\_request(self, request):

if not request.dont\_filter and self.df.request\_seen(request):

self.df.log(request, self.spider)

return False

if self.stats:

self.stats.inc\_value('scheduler/enqueued/redis', spider=self.spider)

self.queue.push(request)

return True

def next\_request(self):

block\_pop\_timeout = self.idle\_before\_close

request = self.queue.pop(block\_pop\_timeout)

if request and self.stats:

self.stats.inc\_value('scheduler/dequeued/redis', spider=self.spider)

return request

def has\_pending\_requests(self):

return len(self) > 0

这个文件重写了scheduler类，用来代替scrapy.core.scheduler的原有调度器。其实对原有调度器的逻辑没有很大的改变，主要是使用了redis作为数据存储的媒介，以达到各个爬虫之间的统一调度。 scheduler负责调度各个spider的request请求，scheduler初始化时，通过settings文件读取queue和dupefilters的类型（一般就用上边默认的），配置queue和dupefilters使用的key（一般就是spider name加上queue或者dupefilters，这样对于同一种spider的不同实例，就会使用相同的数据块了）。每当一个request要被调度时，enqueue\_request被调用，scheduler使用dupefilters来判断这个url是否重复，如果不重复，就添加到queue的容器中（先进先出，先进后出和优先级都可以，可以在settings中配置）。当调度完成时，next\_request被调用，scheduler就通过queue容器的接口，取出一个request，把他发送给相应的spider，让spider进行爬取工作。

## 源码分析参考：Spider

**spider.py**

设计的这个spider从redis中读取要爬的url,然后执行爬取，若爬取过程中返回更多的url，那么继续进行直至所有的request完成。之后继续从redis中读取url，循环这个过程。

分析：在这个spider中通过connect signals.spider\_idle信号实现对crawler状态的监视。当idle时，返回新的make\_requests\_from\_url(url)给引擎，进而交给调度器调度。

from scrapy import signals

from scrapy.exceptions import DontCloseSpider

from scrapy.spiders import Spider, CrawlSpider

from . import connection

# Default batch size matches default concurrent requests setting.

DEFAULT\_START\_URLS\_BATCH\_SIZE = 16

DEFAULT\_START\_URLS\_KEY = '%(name)s:start\_urls'

class RedisMixin(object):

"""Mixin class to implement reading urls from a redis queue."""

# Per spider redis key, default to DEFAULT\_START\_URLS\_KEY.

redis\_key = None

# Fetch this amount of start urls when idle. Default to DEFAULT\_START\_URLS\_BATCH\_SIZE.

redis\_batch\_size = None

# Redis client instance.

server = None

def start\_requests(self):

"""Returns a batch of start requests from redis."""

return self.next\_requests()

def setup\_redis(self, crawler=None):

"""Setup redis connection and idle signal.

This should be called after the spider has set its crawler object.

"""

if self.server is not None:

return

if crawler is None:

# We allow optional crawler argument to keep backwards

# compatibility.

# XXX: Raise a deprecation warning.

crawler = getattr(self, 'crawler', None)

if crawler is None:

raise ValueError("crawler is required")

settings = crawler.settings

if self.redis\_key is None:

self.redis\_key = settings.get(

'REDIS\_START\_URLS\_KEY', DEFAULT\_START\_URLS\_KEY,

)

self.redis\_key = self.redis\_key % {'name': self.name}

if not self.redis\_key.strip():

raise ValueError("redis\_key must not be empty")

if self.redis\_batch\_size is None:

self.redis\_batch\_size = settings.getint(

'REDIS\_START\_URLS\_BATCH\_SIZE', DEFAULT\_START\_URLS\_BATCH\_SIZE,

)

try:

self.redis\_batch\_size = int(self.redis\_batch\_size)

except (TypeError, ValueError):

raise ValueError("redis\_batch\_size must be an integer")

self.logger.info("Reading start URLs from redis key '%(redis\_key)s' "

"(batch size: %(redis\_batch\_size)s)", self.\_\_dict\_\_)

self.server = connection.from\_settings(crawler.settings)

# The idle signal is called when the spider has no requests left,

# that's when we will schedule new requests from redis queue

crawler.signals.connect(self.spider\_idle, signal=signals.spider\_idle)

def next\_requests(self):

"""Returns a request to be scheduled or none."""

use\_set = self.settings.getbool('REDIS\_START\_URLS\_AS\_SET')

fetch\_one = self.server.spop if use\_set else self.server.lpop

# XXX: Do we need to use a timeout here?

found = 0

while found < self.redis\_batch\_size:

data = fetch\_one(self.redis\_key)

if not data:

# Queue empty.

break

req = self.make\_request\_from\_data(data)

if req:

yield req

found += 1

else:

self.logger.debug("Request not made from data: %r", data)

if found:

self.logger.debug("Read %s requests from '%s'", found, self.redis\_key)

def make\_request\_from\_data(self, data):

# By default, data is an URL.

if '://' in data:

return self.make\_requests\_from\_url(data)

else:

self.logger.error("Unexpected URL from '%s': %r", self.redis\_key, data)

def schedule\_next\_requests(self):

"""Schedules a request if available"""

for req in self.next\_requests():

self.crawler.engine.crawl(req, spider=self)

def spider\_idle(self):

"""Schedules a request if available, otherwise waits."""

# XXX: Handle a sentinel to close the spider.

self.schedule\_next\_requests()

raise DontCloseSpider

class RedisSpider(RedisMixin, Spider):

"""Spider that reads urls from redis queue when idle."""

@classmethod

def from\_crawler(self, crawler, \*args, \*\*kwargs):

obj = super(RedisSpider, self).from\_crawler(crawler, \*args, \*\*kwargs)

obj.setup\_redis(crawler)

return obj

class RedisCrawlSpider(RedisMixin, CrawlSpider):

"""Spider that reads urls from redis queue when idle."""

@classmethod

def from\_crawler(self, crawler, \*args, \*\*kwargs):

obj = super(RedisCrawlSpider, self).from\_crawler(crawler, \*args, \*\*kwargs)

obj.setup\_redis(crawler)

return obj

spider的改动也不是很大，主要是通过connect接口，给spider绑定了spider\_idle信号，spider初始化时，通过setup\_redis函数初始化好和redis的连接，之后通过next\_requests函数从redis中取出strat url，使用的key是settings中REDIS\_START\_URLS\_AS\_SET定义的（注意了这里的初始化url池和我们上边的queue的url池不是一个东西，queue的池是用于调度的，初始化url池是存放入口url的，他们都存在redis中，但是使用不同的key来区分，就当成是不同的表吧），spider使用少量的start url，可以发展出很多新的url，这些url会进入scheduler进行判重和调度。直到spider跑到调度池内没有url的时候，会触发spider\_idle信号，从而触发spider的next\_requests函数，再次从redis的start url池中读取一些url。

## 总结

最后总结一下scrapy-redis的总体思路：这个工程通过重写scheduler和spider类，实现了调度、spider启动和redis的交互。实现新的dupefilter和queue类，达到了判重和调度容器和redis的交互，因为每个主机上的爬虫进程都访问同一个redis数据库，所以调度和判重都统一进行统一管理，达到了分布式爬虫的目的。 当spider被初始化时，同时会初始化一个对应的scheduler对象，这个调度器对象通过读取settings，配置好自己的调度容器queue和判重工具dupefilter。每当一个spider产出一个request的时候，scrapy内核会把这个reuqest递交给这个spider对应的scheduler对象进行调度，scheduler对象通过访问redis对request进行判重，如果不重复就把他添加进redis中的调度池。当调度条件满足时，scheduler对象就从redis的调度池中取出一个request发送给spider，让他爬取。当spider爬取的所有暂时可用url之后，scheduler发现这个spider对应的redis的调度池空了，于是触发信号spider\_idle，spider收到这个信号之后，直接连接redis读取strart url池，拿去新的一批url入口，然后再次重复上边的工作。

# scrapy-redis实战

## 从零搭建Redis-Scrapy分布式爬虫

### Scrapy-Redis分布式策略：

假设有四台电脑：Windows 10、Mac OS X、Ubuntu 16.04、CentOS 7.2，任意一台电脑都可以作为 Master端 或 Slaver端，比如：

* Master端(核心服务器) ：使用 Windows 10，搭建一个Redis数据库，不负责爬取，只负责url指纹判重、Request的分配，以及数据的存储
* Slaver端(爬虫程序执行端) ：使用 Mac OS X 、Ubuntu 16.04、CentOS 7.2，负责执行爬虫程序，运行过程中提交新的Request给Master

1. 首先Slaver端从Master端拿任务（Request、url）进行数据抓取，Slaver抓取数据的同时，产生新任务的Request便提交给 Master 处理；
2. Master端只有一个Redis数据库，负责将未处理的Request去重和任务分配，将处理后的Request加入待爬队列，并且存储爬取的数据。

Scrapy-Redis默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作Scrapy-Redis都已经帮我们做好了，我们只需要继承RedisSpider、指定redis\_key就行了。

redis\_key是统一管理slaver端爬取的指令。

缺点是，Scrapy-Redis调度的任务是Request对象，里面信息量比较大（不仅包含url，还有callback函数、headers等信息），可能导致的结果就是会降低爬虫速度、而且会占用Redis大量的存储空间，所以如果要保证效率，那么就需要一定硬件水平。

### 一、安装Redis

安装Redis：<http://redis.io/download>

安装完成后，拷贝一份Redis安装目录下的redis.conf到任意目录，建议保存到：/etc/redis/redis.conf（Windows系统可以无需变动)

## Installation

Download, extract and compile Redis with:

$ wget http://download.redis.io/releases/redis-3.2.9.tar.gz

$ tar xzf redis-3.2.9.tar.gz

$ cd redis-3.2.9

$ make

The binaries that are now compiled are available in the src directory. Run Redis with:

$ src/redis-server

You can interact with Redis using the built-in client:

$ src/redis-cli

redis> set foo bar

OK

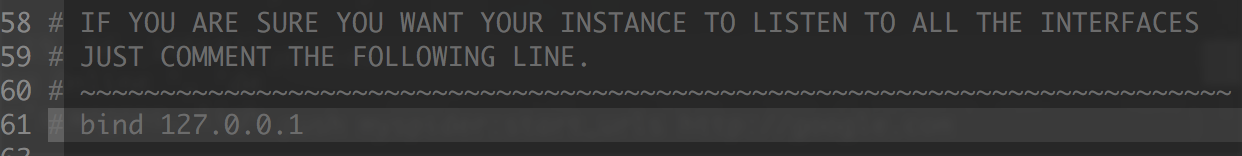
redis> get foo

### 二、修改配置文件 redis.conf

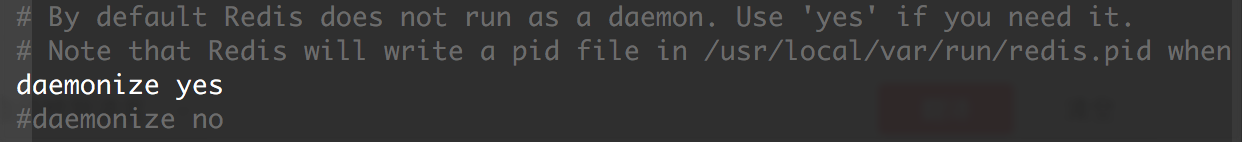
打开你的redis.conf配置文件，示例:

* 非Windows系统: sudo vi /etc/redis/redis.conf
* Windows系统：C:\Intel\Redis\conf\redis.conf

1. Master端redis.conf里注释bind 127.0.0.1，Slave端才能远程连接到Master端的Redis数据库。



* + daemonize no表示Redis默认不作为守护进程运行，即在运行redis-server /etc/redis/redis.conf时，将显示Redis启动提示画面；
    - daemonize yes则默认后台运行，不必重新启动新的终端窗口执行其他命令，看个人喜好和实际需要。



windows用户必须要以管理员模式运行命令行

redis-server C:\Intel\Redis\conf\redis.conf

redis-cli

如果显示redis 127.0.0.1:6379>，则表示服务已经启动了。

set key1 "hello world"

get key1

想要在其它电脑上连接本地的redis数据库，首先要知道本地的IP地址，ipconfig，只要两台电脑之间能够ping通，就能够连接。假设本机ip为192.168.21.64，在远程电脑上使用以下命令

sudo redis-cl -h 192.168.21.64

显示

192.168.21.64:6379>

get key1

此时就能连接到本机数据库，显示hello world

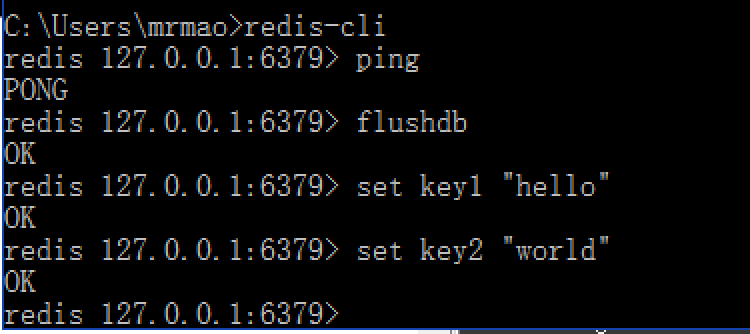
flushdb

清空数据库

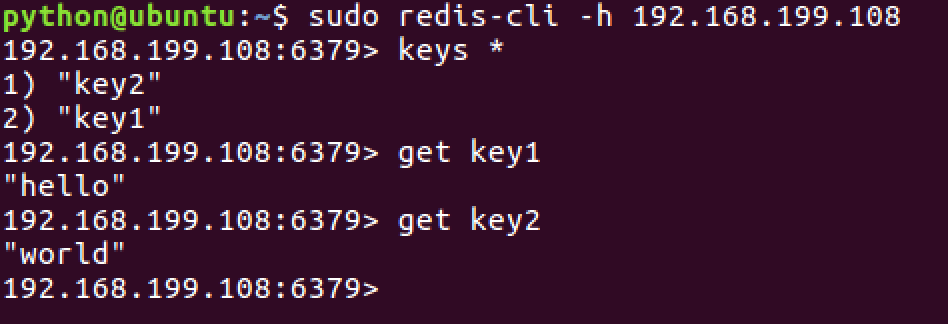
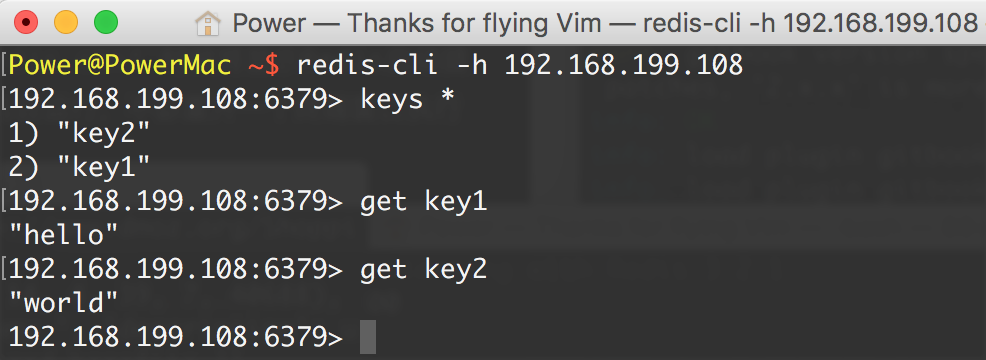
### 三、测试Slave端远程连接Master端

测试中，Master端Windows 10 的IP地址为：192.168.199.108

1. Master端按指定配置文件启动 redis-server，示例：
   * 非Windows系统：sudo redis-server /etc/redis/redis/conf
   * Windows系统：命令提示符(管理员)模式下执行 redis-server C:\Intel\Redis\conf\redis.conf读取默认配置即可。
2. Master端启动本地redis-cli：



1. slave端启动redis-cli -h 192.168.199.108，-h 参数表示连接到指定主机的redis数据库



注意：Slave端无需启动redis-server，Master端启动即可。只要 Slave 端读取到了 Master 端的 Redis 数据库，则表示能够连接成功，可以实施分布式。

### 四、Redis数据库桌面管理工具

这里推荐 Redis Desktop Manager，支持 Windows、Mac OS X、Linux 等平台：

下载地址：<https://redisdesktop.com/download>

### 五、安装scrapy-redis

pip install scrapy-redis

### 源码自带项目说明：

#### 使用scrapy-redis的example来修改

先从github上拿到scrapy-redis的示例，然后将里面的example-project目录移到指定的地址：

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例，改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapyredis-project

我们clone到的 scrapy-redis 源码中有自带一个example-project项目，这个项目包含3个spider，分别是dmoz, myspider\_redis，mycrawler\_redis。

注意example-project的名字可以任意修改，但是其中example文件夹不能随意修改，一旦修改了example文件夹，就要在example-project文件夹下的scrapy.cfg文件中修改对应的文件夹名称。example/settings.py中的SPIDER\_MODULES的名称也要修改。

#### 一、dmoz (class DmozSpider(CrawlSpider))

这个爬虫继承的是CrawlSpider，它是用来说明Redis的持续性，当我们第一次运行dmoz爬虫，然后Ctrl + C停掉之后，再运行dmoz爬虫，之前的爬取记录是保留在Redis里的。

分析起来，其实这就是一个 scrapy-redis 版 CrawlSpider 类，需要设置Rule规则，以及callback不能写parse()方法。

实际上就是一个Crawl Scrapy的项目，爬取<http://www.dmoz.org/>网站上的分类目录

##### items文件

**from** scrapy.item **import** Item, Field

**from** scrapy.loader **import** ItemLoader

**from** scrapy.loader.processors **import** MapCompose, TakeFirst, Join

*#定义了多个字段*

**class** ExampleItem(Item):

name = Field()

description = Field()

link = Field()

crawled = Field()

spider = Field()

url = Field()

*#使用ItemLoader方法，与item方法有些许的不同*

**class** ExampleLoader(ItemLoader):

default\_item\_class = ExampleItem

default\_input\_processor = MapCompose(**lambda** s: s.strip())

default\_output\_processor = TakeFirst()

description\_out = Join()

##### pipeline文件

**from** datetime **import** datetime

**class** ExamplePipeline(object):

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

*#数据给过管道后会添加以下两个字段*

*#添加了linux的时间戳，即数据获取的时间，UTC时间，全球标准时间。英国格林威治的时间。中国时间为utc时间+8*

item[**"crawled"**] = datetime.utcnow()

*#添加了一个爬虫名的字段，有很多个爬虫，给数据添加一个爬虫名的字段后就知道是哪个爬虫爬取的了。*

item[**"spider"**] = spider.name

**return** item

##### settings文件

*#爬虫位置*

SPIDER\_MODULES = [**'example.spiders'**]

NEWSPIDER\_MODULE = **'example.spiders'**

*#默认的user\_agent，要进行修改*

USER\_AGENT = **'scrapy-redis (+https://github.com/rolando/scrapy-redis)'**

*#使用了scrapy\_redis中的去重方法*

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

*#使用了scrapy-redis中的调度组件*

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

*#允许执行项目时可以暂停，暂停后不会清空redis数据库的记录*

SCHEDULER\_PERSIST = **True**

*#如果三个都注释掉，就使用默认的优先级策略。默认的是scrapy请求队列形式，是按优先级顺序排列的队列形式，一般来说使用把三个都注释掉，使用默认的即可。出队列的顺序不影响我们的结果。*

*#在使用redis分布式的爬虫时，如果使用scrapy的请求队列形式，虽然不影响请求的发送，但在redis数据库中却不会显示出请求队列。必须使用redis的三种请求队列之一才能在redis数据库中显示出请求队列。*

*#按sorted排序顺序出队列*

SCHEDULER\_QUEUE\_CLASS = **"scrapy\_redis.queue.SpiderPriorityQueue"**

*#队列形式，规则是请求先进先出*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"*

*#栈形式，规则是请求先进后出*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"*

ITEM\_PIPELINES = {

*#自定义的pipelines，先经过我们自定义的pipeline，添加了utc时间和爬虫名。可以自定义多个管道文件，经过一系列的处理，最后再放到redis数据库中。*

**'example.pipelines.ExamplePipeline'**: 300,

*#scrapy\_redis.pipelines.RedisPipeline将数据存储在Redis数据库中，必须要启动，如果不想把数据存储在Redis数据库中，也可以使用自定义的pipeline进行数据的存储。此时把放在redis数据库的优先级设置到最后。*

**'scrapy\_redis.pipelines.RedisPipeline'**: 400,

}

*#定义数据存储的位置，如果不写这两个参数，则默认是保存在本地的。*

*#指定数据库的主机IP*

REDIS\_HOST = **"111.186.110.33"**

*#指定远程数据库的端口号*

REDIS\_PORT = 6379

*#LOG日志*

LOG\_LEVEL = **'DEBUG'**

*# Introduce an artifical delay to make use of parallelism. to speed up the*

*# crawl.*

*#下载延迟*

DOWNLOAD\_DELAY = 2

##### dmoz爬虫文件

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

*#一个简单的CrawlSpider*

*#使用的是CralSpider的爬虫，不是redis的爬虫，故不支持分布式。*

**class** DmozSpider(CrawlSpider):

*"""Follow categories and extract links."""*

name = **'dmoz'**

allowed\_domains = [**'dmoz.org'**]

start\_urls = [**'http://www.dmoz.org/'**]

rules = [

Rule(LinkExtractor(

restrict\_css=(**'.top-cat'**, **'.sub-cat'**, **'.cat-item'**)

), callback=**'parse\_directory'**, follow=**True**),

]

**def** parse\_directory(self, response):

**for** div **in** response.css(**'.title-and-desc'**):

**yield** {

**'name'**: div.css(**'.site-title::text'**).extract\_first(),

**'description'**: div.css(**'.site-descr::text'**).extract\_first().strip(),

**'link'**: div.css(**'a::attr(href)'**).extract\_first(),

}

**main.py**

from scrapy import cmdline

cmdline.execute('scrapy crawl dmoz'.split())

##### 执行dmoz爬虫

**执行方式：scrapy crawl dmoz**

#### 二、myspider\_redis (class MySpider(RedisSpider))

这个爬虫继承了RedisSpider， 它能够支持分布式的抓取，采用的是basic spider，需要写parse函数。

其次就是不再有start\_urls了，取而代之的是redis\_key，scrapy-redis将key从Redis里pop出来，成为请求的url地址。

**from** scrapy\_redis.spiders **import** RedisSpider

**class** MySpider(RedisSpider):

*"""Spider that reads urls from redis queue (myspider:start\_urls)."""*

name = **'myspider\_redis'**

*#启动所有slaver端爬虫的指令，下面的格式是参考格式，建议采用这种格式*

*#myspider为当前爬虫类的名字*

*#使用lpush myspider:start\_urls http://www.dmoz.org/命令来启动爬虫*

redis\_key = **'myspider:start\_urls'**

*#指定爬取的域范围*

*# allowd\_domains = ['dmoz.org']*

*# 可选：等效于allowd\_domains()，\_\_init\_\_方法按规定格式写，使用时只需要修改super()里的类名参数即可*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

*#通过执行时输入的start\_urls动态的获取爬取的域的范围*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

*# 修改这里的类名为当前类名，super相当于调用了父类的初始化方法*

super(MySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

**def** parse(self, response):

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

**main.py**

from scrapy import cmdline

cmdline.execute('scrapy runspider myspider\_redis.py'.split())

**注意：**

RedisSpider类 不需要写allowd\_domains和start\_urls：

1. scrapy-redis将从在构造方法\_\_init\_\_()里动态定义爬虫爬取域范围，也可以选择直接写allowd\_domains。
2. 必须指定redis\_key，即启动爬虫的命令，参考格式：redis\_key = 'myspider:start\_urls'
3. 根据指定的格式，start\_urls将在 Master端的 redis-cli 里 lpush 到 Redis数据库里，RedisSpider 将在数据库里获取start\_urls。

**执行方式：**

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条），爬虫（们）将处于等待准备状态：

**scrapy runspider myspider\_redis.py**

这里是执行爬虫的文件名了。

1. 在Master端的redis-cli输入push指令，参考格式：

$redis > lpush myspider:start\_urls <http://www.dmoz.org/>

只要是连接到这个redis数据库上的slaver端，都会开始分配url并且执行爬取任务。是由master端通过redis数据库进行任务的分配的。

1. Slaver端爬虫获取到请求，开始爬取。

#### 三、mycrawler\_redis (class MyCrawler(RedisCrawlSpider))

这个RedisCrawlSpider类爬虫继承了RedisCrawlSpider，能够支持分布式的抓取。因为采用的是crawlSpider，所以需要遵守Rule规则，以及callback不能写parse()方法。

同样也不再有start\_urls了，取而代之的是redis\_key，scrapy-redis将key从Redis里pop出来，成为请求的url地址。

**from** scrapy.spiders **import** Rule

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

**class** MyCrawler(RedisCrawlSpider):

*"""Spider that reads urls from redis queue (myspider:start\_urls)."""*

name = **'mycrawler\_redis'**

redis\_key = **'mycrawler:start\_urls'**

rules = (

*# follow all links,可以写多个Rule，callback的名字不能是parse*

Rule(LinkExtractor(), callback=**'parse\_page'**, follow=**True**),

)

*# \_\_init\_\_方法必须按规定写，使用时只需要修改super()里的类名参数即可*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

*# 修改这里的类名为当前类名*

super(MyCrawler, self).\_\_init\_\_(\*args, \*\*kwargs)

**def** parse\_page(self, response):

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

**注意：**

同样的，RedisCrawlSpider类不需要写allowd\_domains和start\_urls：

1. scrapy-redis将从在构造方法\_\_init\_\_()里动态定义爬虫爬取域范围，也可以选择直接写allowd\_domains。
2. 必须指定redis\_key，即启动爬虫的命令，参考格式：redis\_key = 'myspider:start\_urls'
3. 根据指定的格式，start\_urls将在 Master端的 redis-cli 里 lpush 到 Redis数据库里，RedisSpider 将在数据库里获取start\_urls。

**执行方式：**

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条），爬虫（们）将处于等待准备状态：

**scrapy runspider mycrawler\_redis.py**

1. 在Master端的redis-cli输入push指令，参考格式：

**$redis > lpush mycrawler:start\_urls http://www.dmoz.org/**

1. 爬虫获取url，开始执行。

### 总结：

1. 如果只是用到Redis的去重和保存功能，就选第一种；
2. 如果要写分布式，则根据情况，选择第二种、第三种；
3. 通常情况下，会选择用第三种方式编写深度聚焦爬虫。

**windows下的清屏是cls**

## 第1步：有缘网爬虫CrawlSpider版写入到本地文件

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例，改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapy-youyuan

或者使用命令从头开始创建项目

scrapy startproject youyuan

cd youyuan/spiders

scrapy genspider -t crawl yy “youyuan.com”

### 修改settings.py

下面列举了修改后的配置文件中与scrapy-redis有关的部分，middleware、proxy等内容在此就省略了。

非分布式的版本可以使用redis的方法和redis数据库，但是没有分布式的功能。

# -\*- coding: utf-8 -\*-

BOT\_NAME = **'youyuan'**

SPIDER\_MODULES = [**'youyuan.spiders'**]

NEWSPIDER\_MODULE = **'youyuan.spiders'**

*#允许执行项目时可以暂停，暂停后不会清空redis数据库的记录*

*# 在redis中保持scrapy-redis用到的各个队列，从而允许暂停和暂停后恢复，也就是不清理redis queues*

SCHEDULER\_PERSIST = **True**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#USER\_AGENT = 'youyuan (+http://www.yourdomain.com)'*

*# Obey robots.txt rules*

#ROBOTSTXT\_OBEY = **True**

*# Configure a delay for requests for the same website (default: 0)*

*# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay*

*# See also autothrottle settings and docs*

DOWNLOAD\_DELAY = 3

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

COOKIES\_ENABLED = **False**

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# 覆盖默认请求头，可以自己编写Downloader Middlewares设置代理和UserAgent*

DEFAULT\_REQUEST\_HEADERS = {

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8'**,

**'Accept-Language'**: **'zh-CN,zh;q=0.8'**,

**'Connection'**: **'keep-alive'**,

**'Accept-Encoding'**: **'gzip, deflate, sdch'**

}

*# Configure item pipelines*

*# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html*

*# 通过配置RedisPipeline将item写入key为 spider.name : items 的redis的list中，供后面的分布式处理item*

*# 这个已经由 scrapy-redis 实现，不需要我们写代码*

ITEM\_PIPELINES = {

**'youyuan.pipelines.YouyuanPipeline'**: 300,

}

*# LOG等级*

#LOG\_LEVEL = **'DEBUG'**

*#默认情况下,RFPDupeFilter只记录第一个重复请求。将DUPEFILTER\_DEBUG设置为True会记录所有重复的请求。*

#DUPEFILTER\_DEBUG =**True**

### 查看pipeline.py

**写入到本地文件**

**import** json

**class** YouyuanPipeline(object):

**def** \_\_init\_\_(self):

*#self.lists = []*

self.filename = open(**"youyuan.json"**, **"w"**)

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

content = json.dumps(dict(item), ensure\_ascii=**False**) + **"\n"**

self.filename.write(content.encode(**"utf-8"**))

*#self.lists.append(content.encode("utf-8"))*

**return** item

**def** close\_spider(self, spider):

self.filename.close()

### 修改items.py

增加我们最后要保存的youyuanItem项，这里只写出来一个非常简单的版本

# -\*- coding: utf-8 -\*-

from scrapy.item import Item, Field

class youyuanItem(Item):

# 个人头像链接

header\_url = Field()

# 用户名

username = Field()

# 内心独白

monologue = Field()

# 相册图片链接

pic\_urls = Field()

# 年龄

age = Field()

# 网站来源 youyuan

source = Field()

# 个人主页源url

source\_url = Field()

### 编写 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫，之后就可以运行爬虫了。 这里的提供一个简单的版本：

# -\*- coding:utf-8 -\*-

**import** scrapy, re

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

**from** youyuan.items **import** YouyuanItem

**class** YySpider(CrawlSpider):

name = **'yy'**

allowed\_domains = [**'youyuan.com'**]

*# 有缘网的列表页*

start\_urls = [**'http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/'**]

*# 第一级匹配规则：北京市18~25岁女性的每一页链接匹配规则*

page\_links = LinkExtractor(allow = (**r"youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/"**))

*# 第二级匹配规则：每个女性个人主页的匹配规则*

profile\_links = LinkExtractor(allow = (**r"youyuan.com/\d+-profile/"**))

*# 没有callback函数时默认follow=True*

rules = (

Rule(page\_links),

Rule(profile\_links, callback = **"parse\_item"**),

)

**def** parse\_item(self, response):

item = YouyuanItem()

item[**'username'**] = self.get\_username(response)

*# 年龄*

item[**'age'**] = self.get\_age(response)

*# 头像图片的链接*

item[**'header\_url'**] = self.get\_header\_url(response)

*# 相册图片的链接*

item[**'images\_url'**] = self.get\_images\_url(response)

*# 内心独白*

item[**'content'**] = self.get\_content(response)

*# 籍贯*

item[**'place\_from'**] = self.get\_place\_from(response)

*# 学历*

item[**'education'**] = self.get\_education(response)

*# 　兴趣爱好*

item[**'hobby'**] = self.get\_hobby(response)

*# 个人主页*

item[**'source\_url'**] = response.url

*# 数据来源网站*

item[**'sourec'**] = **"youyuan"**

**yield** item

*#用户名*

**def** get\_username(self, response):

username = response.xpath(**"//dl[@class='personal\_cen']//div[@class='main']/strong/text()"**).extract()

**if** len(username):

username = username[0]

**else**:

username = **"NULL"**

**return** username.strip()

*#年龄*

**def** get\_age(self, response):

age = response.xpath(**"//dl[@class='personal\_cen']//dd/p/text()"**).extract()

**if** len(age):

*#不能使用split(' ')的方法，因为有的一个空格，有的两个空格。*

*#这里的u表示什么？？*

age = re.findall(**u"\d+岁"**, age[0])[0]

*#使用r会报错*

*# age = re.findall(r"\d+岁", age[0])[0]*

**else**:

age = **"NULL"**

**return** age.strip()

*#头像图片链接*

**def** get\_header\_url(self, response):

header\_url = response.xpath(**"//dl[@class='personal\_cen']/dt/img/@src"**).extract()

**if** len(header\_url):

header\_url = header\_url[0]

**else**:

header\_url = **"NULL"**

**return** header\_url.strip()

*#相册链接*

**def** get\_images\_url(self, response):

images\_url = response.xpath(**"//div[@class='ph\_show']/ul/li/a/img/@src"**).extract()

**if** len(images\_url):

images\_url = images\_url

**else**:

images\_url = **"NULL"**

**return** images\_url

*#内心独白*

**def** get\_content(self, response):

content = response.xpath(**"//div[@class='pre\_data']/ul/li/p/text()"**).extract()

**if** len(content):

content = content[0]

**else**:

content = **"NULL"**

**return** content.strip()

*#籍贯*

**def** get\_place\_from(self, response):

place\_from = response.xpath(**"//div[@class='pre\_data']/ul/li[2]//ol[1]/li[1]/span/text()"**).extract()

**if** len(place\_from):

place\_from = place\_from[0]

**else**:

place\_from = **"NULL"**

**return** place\_from.strip()

*#学历*

**def** get\_education(self, response):

education = response.xpath(**"//div[@class='pre\_data']/ul/li[3]//ol[2]/li[2]/span/text()"**).extract()

**if** len(education):

education = education[0]

**else**:

education = **"NULL"**

**return** education.strip()

*#爱好*

**def** get\_hobby(self, response):

hobby = response.xpath(**"//dl[@class='personal\_cen']//ol/li/text()"**).extract()

**if** len(hobby):

*#使用逗号把列表中各个元素合并为一个元素*

hobby = **","**.join(hobby).replace(**" "**,**""**)

**else**:

hobby = **"NULL"**

**return** hobby.strip()

### 运行程序：

scrapy crawl youyuan

## 第2步：有缘网爬虫CrawlSpider版写入到redis数据库

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例，改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapy-youyuan

或者使用命令从头开始创建项目

scrapy startproject youyuan

cd youyuan/spiders

scrapy genspider -t crawl yy “youyuan.com”

想要把数据写入到redis数据库中，而不是写入到本地文件，只需要在settings中添加redis的去重方法、指定redis的调度器、指定redis的队列方式，添加redis的pipeline即可。这时也可以使用scrapy的队列方法，但是在爬虫运行期间，redis数据库中就不会显示出队列这一项了。

写分布式爬虫项目，就按正常的scrapy项目写就行，写完了在settings中添加以上三个内容，修改pipeline文件即可成为分布式的代码。

### 修改settings.py

下面列举了修改后的配置文件中与scrapy-redis有关的部分，middleware、proxy等内容在此就省略了。

非分布式的版本可以使用redis的方法和redis数据库，但是没有分布式的功能。

# -\*- coding: utf-8 -\*-

BOT\_NAME = **'youyuan'**

SPIDER\_MODULES = [**'youyuan.spiders'**]

NEWSPIDER\_MODULE = **'youyuan.spiders'**

*#可以使用redis的方法和redis数据库，但是没有分布式的功能*

*# 指定使用scrapy-redis的去重方法*

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

*## 指定使用scrapy-redis的调度器*

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

*# 指定排序爬取地址时使用的队列，默认的 按优先级排序(Scrapy默认)，由sorted set实现的一种非FIFO、LIFO方式。*

*#scrapy-redis请求队列形式，是按优先级顺序，先进的优先级就高*

SCHEDULER\_QUEUE\_CLASS = **"scrapy\_redis.queue.SpiderPriorityQueue"**

*#队列形式，可选的，按先进先出排序（FIFO）*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"*

*#栈形式，可选的，按后进先出排序（LIFO）*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"*

*#允许执行项目时可以暂停，暂停后不会清空redis数据库的记录*

*# 在redis中保持scrapy-redis用到的各个队列，从而允许暂停和暂停后恢复，也就是不清理redis queues*

SCHEDULER\_PERSIST = **True**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#USER\_AGENT = 'youyuan (+http://www.yourdomain.com)'*

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = **True**

*# Configure a delay for requests for the same website (default: 0)*

*# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay*

*# See also autothrottle settings and docs*

DOWNLOAD\_DELAY = 3

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

COOKIES\_ENABLED = **False**

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# 覆盖默认请求头，可以自己编写Downloader Middlewares设置代理和UserAgent*

DEFAULT\_REQUEST\_HEADERS = {

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8'**,

**'Accept-Language'**: **'zh-CN,zh;q=0.8'**,

**'Connection'**: **'keep-alive'**,

**'Accept-Encoding'**: **'gzip, deflate, sdch'**

}

*# Configure item pipelines*

*# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html*

*# 通过配置RedisPipeline将item写入key为 spider.name : items 的redis的list中，供后面的分布式处理item*

*# 这个已经由 scrapy-redis 实现，不需要我们写代码*

*# scrapy\_redis.pipelines.RedisPipeline支持将数据存储到Redis数据库里，必须启动。*

ITEM\_PIPELINES = {

**'youyuan.pipelines.YouyuanPipeline'**: 300,

**'scrapy\_redis.pipelines.RedisPipeline'** : 400,

}

*# LOG等级*

#LOG\_LEVEL = **'DEBUG'**

*#默认情况下,RFPDupeFilter只记录第一个重复请求。将DUPEFILTER\_DEBUG设置为True会记录所有重复的请求。*

#DUPEFILTER\_DEBUG =**True**

### 修改pipeline.py

添加两个字段，并写入到redis数据库中

# -\*- coding: utf-8 -\*-

from datetime import datetime

class ExamplePipeline(object):

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

#utcnow() 是获取UTC时间

item["crawled"] = datetime.utcnow()

# 爬虫名

item["spider"] = spider.name

return item

### 修改items.py

增加我们最后要保存的youyuanItem项，这里只写出来一个非常简单的版本

# -\*- coding: utf-8 -\*-

**import** scrapy

**class** YouyuanItem(scrapy.Item):

*# 用户名*

username = scrapy.Field()

*# 年龄*

age = scrapy.Field()

*# 头像图片的链接*

header\_url = scrapy.Field()

*# 相册图片的链接*

images\_url = scrapy.Field()

*# 内心独白*

content = scrapy.Field()

*# 籍贯*

place\_from = scrapy.Field()

*# 学历*

education = scrapy.Field()

*#　兴趣爱好*

hobby = scrapy.Field()

*# 个人主页*

source\_url = scrapy.Field()

*# 数据来源网站*

sourec = scrapy.Field()

*# 在pipeline中添加两个字段并存入redis数据库中时使用*

*#UTC时间*

crawled = scrapy.Field()

*# 爬虫名*

spider = scrapy.Field()

### 修改pipeline文件

**添加两个字段**

**from** datetime **import** datetime

**class** YouyuanPipeline(object):

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

*# 格林威治时间，距离中国 +8 时区*

item[**'time'**] = datetime.utcnow()

*# 爬虫名*

item[**'spidername'**] = spider.name

**return** item

### 运行程序：

1. Master端打开 Redis： redis-server
2. Slave端直接运行爬虫： scrapy crawl youyuan
3. 多个Slave端运行爬虫顺序没有限制。

## 第3步：有缘网分布式爬虫RedisCrawlSpider

将项目修改成 RedisCrawlSpider 类的分布式爬虫，并尝试在多个Slave端运行。

在第2步中，因为没有用到分布式的爬虫，redis数据是存储在本地的，故不需要指定redis数据库的地址和端口号。想要使用分布式执行，还要在第2步的基础上，修改slave端的redis数据地址和端口号。

### 修改 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫，使其具有分布式：

此时引入的spider类应该是RedisCralSpider

**from** scrapy.linkextractors **import** LinkExtractor

*#from scrapy.spiders import CrawlSpider, Rule*

**from** scrapy.spiders **import** Rule

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

**from** youyuan.items **import** YouyuanItem

**import** re

*#class YySpider(CrawlSpider):*

**class** YySpider(RedisCrawlSpider):

name = **'yy'**

*#allowed\_domains = ['youyuan.com']*

*#start\_urls = ['http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/']*

redis\_key = **"yyspider:start\_urls"**

*# 动态域范围获取*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

super(YySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

*# 第一级匹配规则：北京市18~25岁女性的每一页链接匹配规则*

page\_links = LinkExtractor(allow = (**r"youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/"**))

*# 第二级匹配规则：每个女性个人主页的匹配规则*

profile\_links = LinkExtractor(allow = (**r"youyuan.com/\d+-profile/"**))

rules = (

Rule(page\_links),

Rule(profile\_links, callback = **"parse\_item"**),

)

**def** parse\_item(self, response):

item = YouyuanItem()

item[**'username'**] = self.get\_username(response)

*# 年龄*

item[**'age'**] = self.get\_age(response)

*# 头像图片的链接*

item[**'header\_url'**] = self.get\_header\_url(response)

*# 相册图片的链接*

item[**'images\_url'**] = self.get\_images\_url(response)

*# 内心独白*

item[**'content'**] = self.get\_content(response)

*# 籍贯*

item[**'place\_from'**] = self.get\_place\_from(response)

*# 学历*

item[**'education'**] = self.get\_education(response)

*# 　兴趣爱好*

item[**'hobby'**] = self.get\_hobby(response)

*# 个人主页*

item[**'source\_url'**] = response.url

*# 数据来源网站*

item[**'sourec'**] = **"youyuan"**

**yield** item

*#用户名*

**def** get\_username(self, response):

username = response.xpath(**"//dl[@class='personal\_cen']//div[@class='main']/strong/text()"**).extract()

**if** len(username):

username = username[0]

**else**:

username = **"NULL"**

**return** username.strip()

*#年龄*

**def** get\_age(self, response):

age = response.xpath(**"//dl[@class='personal\_cen']//dd/p/text()"**).extract()

**if** len(age):

*#不能使用split(' ')的方法，因为有的一个空格，有的两个空格。*

*#这里的u表示什么？？*

age = re.findall(**u"\d+岁"**, age[0])[0]

*#使用r会报错*

*# age = re.findall(r"\d+岁", age[0])[0]*

**else**:

age = **"NULL"**

**return** age.strip()

*#头像图片链接*

**def** get\_header\_url(self, response):

header\_url = response.xpath(**"//dl[@class='personal\_cen']/dt/img/@src"**).extract()

**if** len(header\_url):

header\_url = header\_url[0]

**else**:

header\_url = **"NULL"**

**return** header\_url.strip()

*#相册链接*

**def** get\_images\_url(self, response):

images\_url = response.xpath(**"//div[@class='ph\_show']/ul/li/a/img/@src"**).extract()

**if** len(images\_url):

images\_url = images\_url

**else**:

images\_url = **"NULL"**

**return** images\_url

*#内心独白*

**def** get\_content(self, response):

content = response.xpath(**"//div[@class='pre\_data']/ul/li/p/text()"**).extract()

**if** len(content):

content = content[0]

**else**:

content = **"NULL"**

**return** content.strip()

*#籍贯*

**def** get\_place\_from(self, response):

place\_from = response.xpath(**"//div[@class='pre\_data']/ul/li[2]//ol[1]/li[1]/span/text()"**).extract()

**if** len(place\_from):

place\_from = place\_from[0]

**else**:

place\_from = **"NULL"**

**return** place\_from.strip()

*#学历*

**def** get\_education(self, response):

education = response.xpath(**"//div[@class='pre\_data']/ul/li[3]//ol[2]/li[2]/span/text()"**).extract()

**if** len(education):

education = education[0]

**else**:

education = **"NULL"**

**return** education.strip()

*#爱好*

**def** get\_hobby(self, response):

hobby = response.xpath(**"//dl[@class='personal\_cen']//ol/li/text()"**).extract()

**if** len(hobby):

*#使用逗号把列表中各个元素合并为一个元素*

hobby = **","**.join(hobby).replace(**" "**,**""**)

**else**:

hobby = **"NULL"**

**return** hobby.strip()

### 把代码上传到所有slave端

tar -cvf youyuan.tar youyuan

sftp 远程的用户名@ip

sftp python@1921682184

ls显示远程的文件和目录

lls显示要地的文件和目录

put youyun.tar

tar -xvf youyuan.tar

c是打包collection

x是解包

z是压缩，对应gz

### 分布式爬虫执行方式：

#### 6. 在Master端启动redis-server：

redis-server

#### 7. 在Slave端分别启动爬虫，不分先后：

修改slave端settings中的REDIS\_HOST为Master端的ip

REDIS\_HOST = “192.168.21.64”

REDIS\_PORT = 6379

scrapy runspider youyuan.py

#### 8. 在Master端的redis-cli里push一个start\_urls

redis-cli> lpush youyuan:start\_urls http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/

#### 9. 爬虫启动，查看redis数据库数据。

sudo redis-cli

keys \* 显示所有数据库

yy是数据库名，0是从第一个开始，-1表示最后一个。即取数据库中的所有数据。

lrange “yy:items” 0 -1

## 处理Redis里的数据

有缘网的数据爬回来了，但是放在Redis里没有处理。之前我们配置文件里面没有定制自己的ITEM\_PIPELINES，而是使用了RedisPipeline，所以现在这些数据都被保存在redis的youyuan:items键中，所以我们需要另外做处理。

在scrapy-youyuan目录下可以看到一个process\_items.py文件，这个文件就是scrapy-redis的example提供的从redis读取item进行处理的模版。

假设我们要把youyuan:items中保存的数据读出来写进MongoDB或者MySQL，那么我们可以自己写一个process\_youyuan\_profile.py文件，然后保持后台运行就可以不停地将爬回来的数据入库了。

### 存入MongoDB

1. 启动MongoDB数据库：sudo mongod
2. 执行下面程序：py2 process\_youyuan\_mongodb.py

# process\_youyuan\_mongodb.py

# -\*- coding: utf-8 -\*-

**import** redis, pymongo, json

*#操作redis数据库和mongo数据库，之前向redis数据库存储数据其实是一个json对象，现在保存为MongoDB数据，还要转换回去。*

**def** process\_item():

*# 创建redis数据库连接*

rediscli = redis.Redis(host = **"127.0.0.1"**, port = 6379, db = **"0"**)

*# 创建MongoDB数据库连接*

mongocli = pymongo.MongoClient(host = **"127.0.0.1"**, port = 27017)

*# 创建mongodb数据库名称*

dbname = mongocli[**"youyuan"**]

*# 创建mongodb数据库youyuan的表名称*

sheetname = dbname[**"beijing\_18\_25\_mm"**]

*#查看有多少条记录*

offset = 0

*#使用循环向MongoDB中添加多个数据*

**while True**:

*# redis 数据表名 和 数据*

source, data = rediscli.blpop(**"yy:items"**)

offset += 1

*# 将json对象转换为Python对象*

data = json.loads(data)

*# 将数据插入到sheetname表里*

sheetname.insert(data)

print(offset)

**if** \_\_name\_\_ == **"\_\_main\_\_"**:

process\_item()

mongod启动mongodb服务

执行python process\_item\_for\_mongodb.py

mongo启动mongo shell环境

db

show dbs

use youyuan

show collections

db.beijing\_18\_25\_mm.find()

### 存入 MySQL

1. 启动mysql：mysql.server start（更平台不一样）
2. 登录到root用户：mysql -uroot -p
3. 创建数据库youyuan:create database youyuan;
4. 切换到指定数据库：use youyuan
5. 创建表beijing\_18\_25以及所有字段的列名和数据类型。
6. 执行下面程序：py2 process\_youyuan\_mysql.py

#process\_youyuan\_mysql.py

# -\*- coding: utf-8 -\*-

import json

import redis

import MySQLdb

def main():

# 指定redis数据库信息

rediscli = redis.StrictRedis(host='192.168.199.108', port = 6379, db = 0)

# 指定mysql数据库

mysqlcli = MySQLdb.connect(host='127.0.0.1', user='power', passwd='xxxxxxx', db = 'youyuan', port=3306, use\_unicode=True)

while True:

# FIFO模式为 blpop，LIFO模式为 brpop，获取键值

source, data = rediscli.blpop(["youyuan:items"])

item = json.loads(data)

try:

# 使用cursor()方法获取操作游标

cur = mysqlcli.cursor()

# 使用execute方法执行SQL INSERT语句

cur.execute("INSERT INTO beijing\_18\_25 (username, crawled, age, spider, header\_url, source, pic\_urls, monologue, source\_url) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s )", [item['username'], item['crawled'], item['age'], item['spider'], item['header\_url'], item['source'], item['pic\_urls'], item['monologue'], item['source\_url']])

# 提交sql事务

mysqlcli.commit()

#关闭本次操作

cur.close()

print "inserted %s" % item['source\_url']

except MySQLdb.Error,e:

print "Mysql Error %d: %s" % (e.args[0], e.args[1])

if \_\_name\_\_ == '\_\_main\_\_':

main()

## 新浪网分类资讯爬虫1

思考：如何将已有的Scrapy爬虫项目，改写成scrapy-redis分布式爬虫。

要求：将所有对应的大类的 标题和urls、小类的 标题和urls、子链接url、文章名以及文章内容，存入Redis数据库。

### 以下为原Scrapy爬虫项目源码：

### items.py

# -\*- coding: utf-8 -\*-

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### pipelines.py

# -\*- coding: utf-8 -\*-

from scrapy import signals

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaPipeline(object):

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

sonUrls = item['sonUrls']

# 文件名为子链接url中间部分，并将 / 替换为 \_，保存为 .txt格式

filename = sonUrls[7:-6].replace('/','\_')

filename += ".txt"

fp = open(item['subFilename']+'/'+filename, 'w')

fp.write(item['content'])

fp.close()

return item

### settings.py

# -\*- coding: utf-8 -\*-

BOT\_NAME = 'Sina'

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

ITEM\_PIPELINES = {

'Sina.pipelines.SinaPipeline': 300,

}

LOG\_LEVEL = 'DEBUG'

### spiders/sina.py

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

import scrapy

import os

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaSpider(scrapy.Spider):

name= "sina"

allowed\_domains= ["sina.com.cn"]

start\_urls= [

"http://news.sina.com.cn/guide/"

]

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类目录的路径和目录名

parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在，则创建目录

if(not os.path.exists(parentFilename)):

os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头，如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类，将存储目录放在本大类目录下

if(if\_belong):

subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在，则创建目录

if(not os.path.exists(subFilename)):

os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求，得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url，再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾，如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类，获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] = meta\_1['subUrls']

item['subTitle'] = meta\_1['subTitle']

item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求，得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法，获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()')

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head

item['content']= content

yield item

### 执行：

scrapy crawl sina

## 新浪网分类资讯爬虫2

将已有的新浪网分类资讯Scrapy爬虫项目，修改为基于RedisSpider类的scrapy-redis分布式爬虫项目

注：items数据直接存储在Redis数据库中，这个功能已经由scrapy-redis自行实现。除非单独做额外处理(比如直接存入本地数据库等)，否则不用编写pipelines.py代码。

### items.py文件

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

# subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### settings.py文件

# settings.py

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

USER\_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

SCHEDULER\_PERSIST = True

SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderPriorityQueue"

#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"

#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"

ITEM\_PIPELINES = {

# 'Sina.pipelines.SinaPipeline': 300,

'scrapy\_redis.pipelines.RedisPipeline': 400,

}

LOG\_LEVEL = 'DEBUG'

# Introduce an artifical delay to make use of parallelism. to speed up the

# crawl.

DOWNLOAD\_DELAY = 1

REDIS\_HOST = "192.168.13.26"

REDIS\_PORT = 6379

### spiders/sina.py

# sina.py

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

from scrapy\_redis.spiders import RedisSpider

#from scrapy.spiders import Spider

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

#class SinaSpider(Spider):

class SinaSpider(RedisSpider):

name= "sina"

redis\_key = "sinaspider:start\_urls"

#allowed\_domains= ["sina.com.cn"]

#start\_urls= [

# "http://news.sina.com.cn/guide/"

#]#起始urls列表

def \_\_init\_\_(self, \*args, \*\*kwargs):

domain = kwargs.pop('domain', '')

self.allowed\_domains = filter(None, domain.split(','))

super(SinaSpider, self).\_\_init\_\_(\*args, \*\*kwargs)

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类的路径和目录名

#parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在，则创建目录

#if(not os.path.exists(parentFilename)):

# os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头，如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类，将存储目录放在本大类目录下

if(if\_belong):

#subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在，则创建目录

#if(not os.path.exists(subFilename)):

# os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

#item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求，得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url，再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾，如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类，获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] =meta\_1['subUrls']

item['subTitle'] =meta\_1['subTitle']

#item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求，得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法，获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()').extract()

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head[0] if len(head) > 0 else "NULL"

item['content']= content

yield item

### 执行：

slave端：

scrapy runspider sina.py

Master端：

redis-cli> lpush sinaspider:start\_urls http://news.sina.com.cn/guide/

## IT桔子分布式项目1

IT桔子是关注IT互联网行业的结构化的公司数据库和商业信息服务提供商，于2013年5月21日上线。

IT桔子致力于通过信息和数据的生产、聚合、挖掘、加工、处理，帮助目标用户和客户节约时间和金钱、提高效率，以辅助其各类商业行为，包括风险投资、收购、竞争情报、细分行业信息、国外公司产品信息数据服务等。

用于需自行对所发表或采集的内容负责,因所发表或采集的内容引发的一切纠纷、损失，由该内容的发表或采集者承担全部直接或间接(连带)法律责任，IT桔子不承担任何法律责任。

项目采集地址：[http://www.itjuzi.com/company](http://www.itjuzi.com/company/)

要求：采集页面下所有创业公司的公司信息，包括以下但不限于：

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

class CompanyItem(scrapy.Item):

# 公司id (url数字部分)

info\_id = scrapy.Field()

# 公司名称

company\_name = scrapy.Field()

# 公司口号

slogan = scrapy.Field()

# 分类

scope = scrapy.Field()

# 子分类

sub\_scope = scrapy.Field()

# 所在城市

city = scrapy.Field()

# 所在区域

area = scrapy.Field()

# 公司主页

home\_page = scrapy.Field()

# 公司标签

tags = scrapy.Field()

# 公司简介

company\_intro = scrapy.Field()

# 公司全称：

company\_full\_name = scrapy.Field()

# 成立时间：

found\_time = scrapy.Field()

# 公司规模：

company\_size = scrapy.Field()

# 运营状态

company\_status = scrapy.Field()

# 投资情况列表：包含获投时间、融资阶段、融资金额、投资公司

tz\_info = scrapy.Field()

# 团队信息列表：包含成员姓名、成员职称、成员介绍

tm\_info = scrapy.Field()

# 产品信息列表：包含产品名称、产品类型、产品介绍

pdt\_info = scrapy.Field()

## IT桔子分布式项目2

### 项目实现：

### items.py

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

class CompanyItem(scrapy.Item):

# 公司id (url数字部分)

info\_id = scrapy.Field()

# 公司名称

company\_name = scrapy.Field()

# 公司口号

slogan = scrapy.Field()

# 分类

scope = scrapy.Field()

# 子分类

sub\_scope = scrapy.Field()

# 所在城市

city = scrapy.Field()

# 所在区域

area = scrapy.Field()

# 公司主页

home\_page = scrapy.Field()

# 公司标签

tags = scrapy.Field()

# 公司简介

company\_intro = scrapy.Field()

# 公司全称：

company\_full\_name = scrapy.Field()

# 成立时间：

found\_time = scrapy.Field()

# 公司规模：已经删除

company\_size = scrapy.Field()

# 运营状态

company\_status = scrapy.Field()

# 投资情况列表：包含获投时间、融资阶段、融资金额、投资公司

tz\_info = scrapy.Field()

# 团队信息列表：包含成员姓名、成员职称、成员介绍

tm\_info = scrapy.Field()

# 产品信息列表：包含产品名称、产品类型、产品介绍

pdt\_info = scrapy.Field()

### settings.py

# -\*- coding: utf-8 -\*-

BOT\_NAME = 'itjuzi'

SPIDER\_MODULES = ['itjuzi.spiders']

NEWSPIDER\_MODULE = 'itjuzi.spiders'

# Enables scheduling storing requests queue in redis.

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# Ensure all spiders share same duplicates filter through redis.

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

# REDIS\_START\_URLS\_AS\_SET = True

COOKIES\_ENABLED = False

DOWNLOAD\_DELAY = 1.5

# 支持随机下载延迟0-1.5随机取值

RANDOMIZE\_DOWNLOAD\_DELAY = True

# Obey robots.txt rules

ROBOTSTXT\_OBEY = False

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

DOWNLOADER\_MIDDLEWARES = {

# 该中间件将会收集失败的页面，并在爬虫完成后重新调度。（失败情况可能由于临时的问题，例如连接超时或者HTTP 500错误导致失败的页面）

'scrapy.downloadermiddlewares.retry.RetryMiddleware': 80,

# 该中间件提供了对request设置HTTP代理的支持。您可以通过在 Request 对象中设置 proxy 元数据来开启代理。

'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 100,

'itjuzi.middlewares.RotateUserAgentMiddleware': 200,

}

REDIS\_HOST = "192.168.199.108"

REDIS\_PORT = 6379

### middlewares.py

#没有写代理中间件，因为要使用api接口

# -\*- coding: utf-8 -\*-

from scrapy.contrib.downloadermiddleware.useragent import UserAgentMiddleware

import random

# User-Agetn 下载中间件

class RotateUserAgentMiddleware(UserAgentMiddleware):

def \_\_init\_\_(self, user\_agent=''):

self.user\_agent = user\_agent

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

# 这句话用于随机选择user-agent

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

request.headers.setdefault('User-Agent', ua)

user\_agent\_list = [

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/22.0.1207.1 Safari/537.1",

"Mozilla/5.0 (X11; CrOS i686 2268.111.0) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.57 Safari/536.11",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.6 (KHTML, like Gecko) Chrome/20.0.1092.0 Safari/536.6",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.6 (KHTML, like Gecko) Chrome/20.0.1090.0 Safari/536.6",

"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/19.77.34.5 Safari/537.1",

"Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.9 Safari/536.5",

"Mozilla/5.0 (Windows NT 6.0) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.36 Safari/536.5",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_8\_0) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1062.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1062.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.0 Safari/536.3",

"Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/535.24 (KHTML, like Gecko) Chrome/19.0.1055.1 Safari/535.24",

"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/535.24 (KHTML, like Gecko) Chrome/19.0.1055.1 Safari/535.24",

"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/531.21.8 (KHTML, like Gecko) Version/4.0.4 Safari/531.21.10",

"Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US) AppleWebKit/533.17.8 (KHTML, like Gecko) Version/5.0.1 Safari/533.17.8",

"Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/533.19.4 (KHTML, like Gecko) Version/5.0.2 Safari/533.18.5",

"Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.1.17) Gecko/20110123 (like Firefox/3.x) SeaMonkey/2.0.12",

"Mozilla/5.0 (Windows NT 5.2; rv:10.0.1) Gecko/20100101 Firefox/10.0.1 SeaMonkey/2.7.1",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_5\_8; en-US) AppleWebKit/532.8 (KHTML, like Gecko) Chrome/4.0.302.2 Safari/532.8",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_4; en-US) AppleWebKit/534.3 (KHTML, like Gecko) Chrome/6.0.464.0 Safari/534.3",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_5; en-US) AppleWebKit/534.13 (KHTML, like Gecko) Chrome/9.0.597.15 Safari/534.13",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_2) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.835.186 Safari/535.1",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_6\_8) AppleWebKit/535.2 (KHTML, like Gecko) Chrome/15.0.874.54 Safari/535.2",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_6\_8) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.36 Safari/535.7",

"Mozilla/5.0 (Macintosh; U; Mac OS X Mach-O; en-US; rv:2.0a) Gecko/20040614 Firefox/3.0.0 ",

"Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5; en-US; rv:1.9.0.3) Gecko/2008092414 Firefox/3.0.3",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1) Gecko/20090624 Firefox/3.5",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.14) Gecko/20110218 AlexaToolbar/alxf-2.0 Firefox/3.6.14",

"Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5; en-US; rv:1.9.2.15) Gecko/20110303 Firefox/3.6.15",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:2.0.1) Gecko/20100101 Firefox/4.0.1"

]

### spiders/juzi.py

# -\*- coding: utf-8 -\*-

from bs4 import BeautifulSoup

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

from scrapy\_redis.spiders import RedisCrawlSpider

from itjuzi.items import CompanyItem

class ITjuziSpider(RedisCrawlSpider):

name = 'itjuzi'

allowed\_domains = ['www.itjuzi.com']

# start\_urls = ['http://www.itjuzi.com/company']

redis\_key = 'itjuzispider:start\_urls'

rules = [

# 获取每一页的链接

Rule(link\_extractor=LinkExtractor(allow=('/company\?page=\d+'))),

# 获取每一个公司的详情

Rule(link\_extractor=LinkExtractor(allow=('/company/\d+')), callback='parse\_item')

]

def parse\_item(self, response):

soup = BeautifulSoup(response.body, 'lxml')

# 开头部分： //div[@class="infoheadrow-v2 ugc-block-item"]

cpy1 = soup.find('div', class\_='infoheadrow-v2')

if cpy1:

# 公司名称：//span[@class="title"]/b/text()[1]

company\_name = cpy1.find(class\_='title').b.contents[0].strip().replace('\t', '').replace('\n', '')

# 口号： //div[@class="info-line"]/p

slogan = cpy1.find(class\_='info-line').p.get\_text()

# 分类：子分类//span[@class="scope c-gray-aset"]/a[1]

scope\_a = cpy1.find(class\_='scope c-gray-aset').find\_all('a')

# 分类：//span[@class="scope c-gray-aset"]/a[1]

scope = scope\_a[0].get\_text().strip() if len(scope\_a) > 0 else ''

# 子分类：# //span[@class="scope c-gray-aset"]/a[2]

sub\_scope = scope\_a[1].get\_text().strip() if len(scope\_a) > 1 else ''

# 城市+区域：//span[@class="loca c-gray-aset"]/a

city\_a = cpy1.find(class\_='loca c-gray-aset').find\_all('a')

# 城市：//span[@class="loca c-gray-aset"]/a[1]

city = city\_a[0].get\_text().strip() if len(city\_a) > 0 else ''

# 区域：//span[@class="loca c-gray-aset"]/a[2]

area = city\_a[1].get\_text().strip() if len(city\_a) > 1 else ''

# 主页：//a[@class="weblink marl10"]/@href

home\_page = cpy1.find(class\_='weblink marl10')['href']

# 标签：//div[@class="tagset dbi c-gray-aset"]/a

tags = cpy1.find(class\_='tagset dbi c-gray-aset').get\_text().strip().strip().replace('\n', ',')

#基本信息：//div[@class="block-inc-info on-edit-hide"]

cpy2 = soup.find('div', class\_='block-inc-info on-edit-hide')

if cpy2:

# 公司简介：//div[@class="block-inc-info on-edit-hide"]//div[@class="des"]

company\_intro = cpy2.find(class\_='des').get\_text().strip()

# 公司全称：成立时间：公司规模：运行状态：//div[@class="des-more"]

cpy2\_content = cpy2.find(class\_='des-more').contents

# 公司全称：//div[@class="des-more"]/div[1]

company\_full\_name = cpy2\_content[1].get\_text().strip()[len('公司全称：'):] if cpy2\_content[1] else ''

# 成立时间：//div[@class="des-more"]/div[2]/span[1]

found\_time = cpy2\_content[3].contents[1].get\_text().strip()[len('成立时间：'):] if cpy2\_content[3] else ''

# 公司规模：//div[@class="des-more"]/div[2]/span[2]

company\_size = cpy2\_content[3].contents[3].get\_text().strip()[len('公司规模：'):] if cpy2\_content[3] else ''

#运营状态：//div[@class="des-more"]/div[3]

company\_status = cpy2\_content[5].get\_text().strip() if cpy2\_content[5] else ''

# 主体信息：

main = soup.find('div', class\_='main')

# 投资情况：//table[@class="list-round-v2 need2login"]

# 投资情况，包含获投时间、融资阶段、融资金额、投资公司

tz = main.find('table', 'list-round-v2')

tz\_list = []

if tz:

all\_tr = tz.find\_all('tr')

for tr in all\_tr:

tz\_dict = {}

all\_td = tr.find\_all('td')

tz\_dict['tz\_time'] = all\_td[0].span.get\_text().strip()

tz\_dict['tz\_round'] = all\_td[1].get\_text().strip()

tz\_dict['tz\_finades'] = all\_td[2].get\_text().strip()

tz\_dict['tz\_capital'] = all\_td[3].get\_text().strip().replace('\n', ',')

tz\_list.append(tz\_dict)

# 团队信息：成员姓名、成员职称、成员介绍

tm = main.find('ul', class\_='list-prodcase limited-itemnum')

tm\_list = []

if tm:

for li in tm.find\_all('li'):

tm\_dict = {}

tm\_dict['tm\_m\_name'] = li.find('span', class\_='c').get\_text().strip()

tm\_dict['tm\_m\_title'] = li.find('span', class\_='c-gray').get\_text().strip()

tm\_dict['tm\_m\_intro'] = li.find('p', class\_='mart10 person-des').get\_text().strip()

tm\_list.append(tm\_dict)

# 产品信息：产品名称、产品类型、产品介绍

pdt = main.find('ul', class\_='list-prod limited-itemnum')

pdt\_list = []

if pdt:

for li in pdt.find\_all('li'):

pdt\_dict = {}

pdt\_dict['pdt\_name'] = li.find('h4').b.get\_text().strip()

pdt\_dict['pdt\_type'] = li.find('span', class\_='tag yellow').get\_text().strip()

pdt\_dict['pdt\_intro'] = li.find(class\_='on-edit-hide').p.get\_text().strip()

pdt\_list.append(pdt\_dict)

item = CompanyItem()

item['info\_id'] = response.url.split('/')[-1:][0]

item['company\_name'] = company\_name

item['slogan'] = slogan

item['scope'] = scope

item['sub\_scope'] = sub\_scope

item['city'] = city

item['area'] = area

item['home\_page'] = home\_page

item['tags'] = tags

item['company\_intro'] = company\_intro

item['company\_full\_name'] = company\_full\_name

item['found\_time'] = found\_time

item['company\_size'] = company\_size

item['company\_status'] = company\_status

item['tz\_info'] = tz\_list

item['tm\_info'] = tm\_list

item['pdt\_info'] = pdt\_list

return item

### scrapy.cfg

# Automatically created by: scrapy startproject

#

# For more information about the [deploy] section see:

# https://scrapyd.readthedocs.org/en/latest/deploy.html

[settings]

default = itjuzi.settings

[deploy]

#url = http://localhost:6800/

project = itjuzi

### 运行

Slave端：

scrapy runspider juzi.py

Master端：

redis-cli > lpush itjuzispider:start\_urls http://www.itjuzi.com/company

### 演示效果：