使用Scrapy框架开发分布式爬虫

# Scrapy开发环境搭建

安装Python

安装 Twisted

从 http://www.lfd.uci.edu/~gohlke/pythonlibs/#twisted 下载wheel文件，利用pip安装即可。

如 Python3.6 、 Windows 64位系统，下载 Twisted-17.9.0-cp36-cp36m-win\_amd64.whl 然后pip安装。

pip install Twisted-17.9.0-cp36-cp36m-win\_amd64.whl

pip install -i https://pypi.doubanio.com/simple/ scrapy

pip install -i https://pypi.doubanio.com/simple/ pillow 图片下载依赖包

pip install -i https://pypi.doubanio.com/simple/ mysqlclient 安装 mysql 驱动

pip install -i https://pypi.doubanio.com/simple/ selenium 模拟浏览器

pip install requests 安装 requests 包

https://sourceforge.net/projects/pywin32/files/pywin32/Build%20221/ 安装 win32api 模块，

在虚拟环境中安装 win32api：

|  |
| --- |
| (Py3Scrapy) λ easy\_install D:\下载\pywin32-221.win-amd64-py3.6.exe |

https://github.com/MSOpenTech/redis/releases 安装 Redis

pip install redis 安装 redis

pip install mmh3 bloomfilter的依赖包

pip install scrapyd 安装 scrapyd 服务端

pip install scrapyd-client 安装 scrapyd-client 客户端

pip install -i https://pypi.doubanio.com/simple/ django==1.9.8

## Python包下载地址

<http://www.lfd.uci.edu/~gohlke/pythonlibs/>

在 pip 安装不成功的时候可以从这里下载对应的模块来安装,如 lxml 模块。

# 爬取 [www.jobbole.com](http://www.jobbole.com)

## 安装Scrapy

|  |
| --- |
| pip install scrapy |

## 创建爬虫项目

首先进入工作目录，然后按住Shift+右键，选择“在此处打开命令窗口”，这样就可以把项目创建在该工作目录下了。

创建Scrapy项目：

|  |
| --- |
| scrapy startproject ArticleSpider |

创建完项目之后，提示信息显示：

|  |
| --- |
| D:\workspace>scrapy startproject ArticleSpider  New Scrapy project 'ArticleSpider', using template directory 'C:\\Anaconda3\\lib\\site-packages\\scrapy\\templates\\project', created in:  D:\workspace\ArticleSpider  You can start your first spider with:  cd ArticleSpider  scrapy genspider example example.com |

意思是说该Scrap有项目创建使用的模板是 C:\\Anaconda3\\lib\\site-packages\\scrapy\\templates\\project 这个默认的模板，这个模板是可以自定义的，可以根据自己的需要进行定制。

然后创建完项目，You can start your first spider with 这个部分的内容是说，可以进入项目的目录，然后创建爬虫。以后创建新的爬虫的时候，需要先进入 ArticleSpider 这个项目目录，再通过 scrapy genspider example example.com 这样的方式来创建新的Scrapy爬虫。

打开PyCharm，通过File→Open打开我们新创建的Scrapy项目。此时只是创建了 Scrapy 项目，还没有创建 Scrapy 爬虫，通过以下命令创建爬虫：

|  |
| --- |
| scrapy genspider jobbole blob.jobbole.com |

刷新 Scrapy 项目，发现多了一个 jobbole.py 的文件，这个就是 Spider 文件：

|  |
| --- |
| jobbole.py |
| # -\*- coding: utf-8 -\*-  import scrapy  class JobboleSpider(scrapy.Spider):  name = "jobbole" # Spider的名称，是唯一的  allowed\_domains = ["blob.jobbole.com"] # 该蜘蛛允许抓取的域名  start\_urls = ['http://blob.jobbole.com/'] #  def parse(self, response):  pass |

## 编写 jobbole.py 代码

在上面一节我们创建了jobbole.py爬虫文件，并自动添加了部分代码。

为了方便调试代码，需要在 项目目录下创建一个 main.py 的文件，代码如下：

|  |
| --- |
| from scrapy.cmdline import execute  import sys  import os  sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))  execute(["scrapy","crawl","jobbole"]) # 要执行的 spider |

运行 main.py 就可以执行蜘蛛程序了。

修改 **settings.py** 文件，把下面的变量的值修改为 False ，表示不遵循 robots 协议：

|  |
| --- |
| # 设置为 False 表示 不遵循网站的 robots.txt 协议  ROBOTSTXT\_OBEY = False |

修改 **jobbole.py** 文件，在这里采用 xpath 和 css 这两种选择器来对内容页的字段进行提取，代码如下：

|  |
| --- |
| jobbole.py |
| # -\*- coding: utf-8 -\*-  import scrapy  import re  class JobboleSpider(scrapy.Spider):  name = "jobbole"  allowed\_domains = ["blog.jobbole.com"]  # start\_urls = ['http://blog.jobbole.com/']  start\_urls = ['http://blog.jobbole.com/110287/']  def parse(self, response):  #xpath操作方式  #extract\_first() 可以代替 extract()[0] 防止数组下标越界  title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()  create\_date = response.xpath("//div[@class='entry-meta']/p[@class='entry-meta-hide-on-mobile']/text()").extract()[0].strip().replace("·","").strip()  praise\_nums = int(response.xpath("//span[contains(@class,'vote-post-up')]/h10/text()").extract()[0])  fav\_nums = response.xpath("//span[contains(@class,'bookmark-btn')]/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", fav\_nums)  if match\_re:  fav\_nums = match\_re.group(1)  comment\_nums = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", comment\_nums)  if match\_re:  comment\_nums = match\_re.group(1)  content = response.xpath("//div[@class='entry']").extract()[0]  tag\_list = response.xpath("//p[@class='entry-meta-hide-on-mobile']/a/text()").extract()  tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]  tags = ",".join(tag\_list)  # 打印结果  print(title,create\_date,praise\_nums,fav\_nums,comment\_nums,content,tags)  #css操作方式  title = response.css(".entry-header h1::text").extract()[0]  create\_date = response.css("p.entry-meta-hide-on-mobile::text").extract()[0].strip().replace("·", "").strip()  praise\_nums = int(response.css(".vote-post-up h10::text").extract()[0])  fav\_nums = response.css("span.bookmark-btn::text").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", fav\_nums)  if match\_re:  fav\_nums = match\_re.group(1)  comment\_nums = response.css("a[href='#article-comment'] span::text").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", comment\_nums)  if match\_re:  comment\_nums = match\_re.group(1)  content = response.css("div.entry").extract()[0]  tag\_list = response.css("p.entry-meta-hide-on-mobile a::text").extract()  tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]  tags = ",".join(tag\_list)  print(title, create\_date, praise\_nums, fav\_nums, comment\_nums, content, tags)  pass |

## xpath 和 css 选择器

在上面我们使用了 xpath 和 css 这两种选择器来对内容页的字段进行提取，这里附带上 xpath 和 css 选择器的操作介绍。

在使用选择器提取内容页字段的时候，通过 scrapy shell 来测试更方便，比如要测试 <http://blog.jobbole.com/111422/> 这个页面的提取规则，可以在项目目录下打开 cmd 命令行，然后输入：

|  |
| --- |
| D:\workspace\ArticleSpider>scrapy shell http://blog.jobbole.com/111422/ |

这样就可以把选择器测试的结果直接写到 IDE 的 py 文件中去。具体查看[编写 jobbole.py 代码](#_编写_jobbole.py_代码)。

另外，在使用 scrapy shell 进行测试的时候，由于自带的 User-Agent 是 Python 的，会导致页面抓取测试失败，这个时候可以采用如下方式：

|  |
| --- |
| scrapy shell -s USER\_AGENT="Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36" http://blog.jobbole.com/111422/ |

### xpath 选择器

**xpath简介**

xpath 使用路径表达式在 xml 和 html 中进行导航。

xpath 包含标准的函数库。

xpath 是一个 w3c 的标准。

**xpath术语**

父节点：本节点的直接上级节点

子节点：本节点的直接下级节点

同胞节点：本节点同级的其他节点，也称为兄弟节点。

先辈节点：本节点的所有上级节点

子孙节点：本节点的所有下级节点

**xpath语法**

|  |  |
| --- | --- |
| 表达式 | 使用说明 |
| article | 选取所有 article 元素的所有子节点 |
| /article | 选取根元素 article |
| article/a | 选取所有属于 article 的子元素 a 元素 |
| //div | 选取所有 div 元素，不论出现在文档的任意地方 |
| article//div | 选取所有属于 article 元素的后代 div 元素，不管它出现在 article 之下的任何节点位置 |
| //@class | 选取所有带 class 属性的节点 |

举例：

|  |  |
| --- | --- |
| 表达式 | 说明 |
| /article/div[1] | 选取 article 节点的第一个 div 子节点 |
| /article/div[last()] | 选取 article 节点的最后一个 div 子节点 |
| /article/div[last()-1] | 选取 article 节点的倒数第二个 div 子节点 |
| //div[@lang] | 选取带 lang 属性的 div 节点 |
| //div[@lang='eng'] | 选取 lang属性值为 eng 的 div 节点 |
| /div/\* | 选取 div 节点的所有子节点 |
| //\* | 选取所有节点 |
| //div[@\*] | 选取所有带属性的 div 节点 |
| /div/a | //div/p | 选取 div 节点的 a 节点 和 p 节点 |
| //span | //ul | 选取所有 span 节点和 ul 节点 |
| article/div/p | //span | 选取 article 节点下的 div 节点下的 p 节点 和 所有 span 节点 |

### css 选择器

**常见用法**

|  |  |
| --- | --- |
| \* | 选取所有节点 |
| #container | 选取 id 为 container 的节点 |
| .container | 选取 class 为 container 的节点 |
| li a | 选取 li 节点下的所有 a 节点 |
| ul + p | 选取 ul 节点后面的第一个 p 兄弟节点 |
| div#container > ul | 选取 id 为 container 的 div 节点下的第一个 ul 子节点 |
| ul ~ p | 选取 ul 节点的所有相邻 p 节点 |
| a[title] | 选取包含 title 属性的所有 a 节点 |
| a[href="http://jobbole.com"] | 选取 href 属性值为 <http://jobbole.com> 的 a 节点 |
| a[href\*="jobbole"] | 选取 href 属性值包含 jobbole 的 a 节点 |
| a[href^="http"] | 选取 href 属性值以 http 开头的 a 节点 |
| a[href$=".jpg"] | 选取 href 属性值以 .jpg 结尾的 a 节点 |
| input[type=radio]:checked | 选取 type属性值为 radio且选中的 input 节点 |
| div:not(#container) | 选取 id 不等于 container 的 div 节点 |
| li:nth-child(3) | 选取 li 节点的第三个节点 |
| tr:nth-child(2n) | 选取 tr 节点的偶数节点 |

## 爬取 jobbole.com的文章列表和分页

本节在[编写 jobbole.py 代码](#_编写_jobbole.py_代码)的基础之上进行操作。

首先把解析内容页字段的函数 parse 修改为 parse\_detail ，同时对 fav\_nums 和 comment\_nums 这两个变量的值做个判断，如果没有值，则设置默认值为 0。

另外，xpath 和 css 选择器演示完成之后，我们在使用的时候只留下 xpath 选择器的代码。

然后增加 parse 函数，用来爬取列表分页的URL地址以及每个列表页的文章URL地址。

jobbole.py的完整代码如下：

|  |
| --- |
| jobbole.py |
| # -\*- coding: utf-8 -\*-  import scrapy  import re  from scrapy.http import Request  from urllib import parse  class JobboleSpider(scrapy.Spider):  name = "jobbole"  allowed\_domains = ["blog.jobbole.com"]  start\_urls = ['http://blog.jobbole.com/all-posts/']  def parse(self, response):  """  1. 获取文章列表页中的文章 url 并提交给 scrapy 下载并进行解析  2. 获取下一页的 url 并交给 scrapy 进行下载，下载完成后交给 parse  """  # 解析列表页中的所有文章的 url 并交给 scrapy 下载和解析  post\_urls = response.css("div#archive .floated-thumb .post-thumb a::attr(href)").extract()  for post\_url in post\_urls:  # parse.urljoin 用来把相对路径转换成绝对路径  print("文章URL地址",post\_url) # 把下面的 yield 语句注释掉，可以跳过具体内容采集，而测试网址采集结果  yield Request(url=parse.urljoin(response.url,post\_url) , callback=self.parse\_detail) # 回调 parse\_detail 函数  # 提取下一页并交给 scrapy 进行下载  # .next.page-numbers 对应 class="next page-numbers" 这些多个class的处理  next\_urls = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_urls:  print('分页链接地址：',next\_urls)  yield Request(url=parse.urljoin(response.url,next\_urls) , callback=self.parse) # 回调 parse 函数  def parse\_detail(self, response):  title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()  create\_date = response.xpath("//div[@class='entry-meta']/p[@class='entry-meta-hide-on-mobile']/text()").extract()[0].strip().replace("·","").strip()  praise\_nums = int(response.xpath("//span[contains(@class,'vote-post-up')]/h10/text()").extract()[0])  fav\_nums = response.xpath("//span[contains(@class,'bookmark-btn')]/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", fav\_nums)  if match\_re:  fav\_nums = int(match\_re.group(1)) # 这里要修改代码过滤掉没有 fav\_nums 的值并设置默认值  else:  fav\_nums = 0  comment\_nums = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", comment\_nums)  if match\_re:  comment\_nums = int(match\_re.group(1)) # 这里要修改代码过滤掉没有 comment\_nums 的值并设置默认值  else:  comment\_nums = 0  content = response.xpath("//div[@class='entry']").extract()[0]  tag\_list = response.xpath("//p[@class='entry-meta-hide-on-mobile']/a/text()").extract()  tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]  tags = ",".join(tag\_list)  #print(title,create\_date,praise\_nums,fav\_nums,comment\_nums,content,tags)#该语句仅作调试用 |

到这里，文章列表和内容页URL地址的抓取以及内容页字段的抓取的代码就完成了。

## items.py 数据封装

有了[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)这个基础，就可以把采集到的数据进行保存了。items 类似于 java 中的 Form，用于数据保存。

在[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)已经实现了数据的抓取，但是还没有对数据进行保存，接下来我们就编写 jobbole 对应的 item 对数据进行保存，这里我们定义的 item 为 JobBoleArticleItem ，JobBoleArticleItem 定义在 items.py 文件中，代码如下：

|  |
| --- |
| items.py |
| # -\*- coding: utf-8 -\*-  # Define here the models for your scraped items  #  # See documentation in:  # http://doc.scrapy.org/en/latest/topics/items.html  import scrapy  class ArticlespiderItem(scrapy.Item):  # define the fields for your item here like:  # name = scrapy.Field()  pass  # 该 Item 对应的是 jobbole.py 这个 spider 的 Item  class JobBoleArticleItem(scrapy.Item):  url = scrapy.Field() #文章的 URL 地址  url\_object\_id = scrapy.Field() #保存文章 URL 地址经过 md5 处理后的值  front\_image\_url = scrapy.Field() #文章的缩略图  front\_image\_path = scrapy.Field() #文章的缩略图的存放路径  title = scrapy.Field() #标题  create\_date = scrapy.Field() #创建时间  praise\_nums = scrapy.Field() #点赞数  fav\_nums = scrapy.Field() #收藏数  comment\_nums = scrapy.Field() #评论数  content = scrapy.Field() #内容  tags = scrapy.Field() #标签 |

在 JobBoleArticleItem 这个 item 里面有一个细节就是想要保存文章的缩略图，这个缩略图是我们希望在采集列表页里面的文章的URL地址的时候顺便把文章的封面图也采集下来作为文章的缩略图，如何实现呢？在这里我们需要对[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)这个文件的代码进行修改，修改里面的 **parse(self, response)** 函数，代码如下：

|  |
| --- |
| def parse(self, response):  # 这里把原来的 post\_urls 修改为 post\_nodes  post\_nodes = response.css("div#archive .floated-thumb .post-thumb a")  for post\_node in post\_nodes:  # 提取文章封面图的URL地址  image\_url = post\_node.css("img::attr(src)").extract\_first("")  # 提取文章内容页的URL地址  post\_url = post\_node.css("::attr(href)").extract\_first("")  print("文章URL地址", post\_url)  # 在这里通过 Request 对象的 meta 属性来定义保存 image\_url 的变量 front\_image\_url，  # 并把该变量 front\_image\_url 的值也传递给 parse\_detail 进行处理  yield Request(url=parse.urljoin(response.url,post\_url), meta={"front\_image\_url": image\_url},  callback=self.parse\_detail)  # 提取列表分页的URL地址的代码不变  next\_urls = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_urls:  print('分页链接地址：',next\_urls)  yield Request(url=parse.urljoin(response.url,next\_urls) , callback=self.parse) |

在[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)的 **parse\_detail** 如何获得传递过来的封面图 **front\_image\_url** 变量的值呢？代码如下：

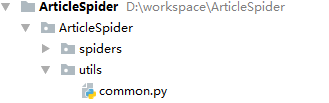
|  |
| --- |
| # 获取 meta={"front\_image\_url": image\_url} 传递过来的文章封面图url的值  front\_image\_url = response.meta.get("front\_image\_url","") |

那么在[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)中，如何在 jobbole.py 中使用我们这里定义的 JobBoleArticleItem 呢？

首先需要 import 该 JobBoleArticleItem 代码如下：

|  |
| --- |
| from ArticleSpider.items import JobBoleArticleItem |

另外，我们要对 JobBoleArticleItem 里面的文章的URL地址经过 md5 计算的，所以我们在 ArticleSpider 目录下创建一个目录 utils，然后在 utils 下创建一个 common.py 文件，如下图：



在 common.py 文件中编写如下代码：

|  |
| --- |
| import hashlib  def get\_md5(url):  if isinstance(url,str):  url = url.encode("utf-8")  m = hashlib.md5()  m.update(url)  return m.hexdigest()  if \_\_name\_\_ == "\_\_main\_\_":  print(get\_md5("http://www.jobbole.com".encode("utf-8"))) |

接下来我们需要对[爬取 jobbole.com的文章列表和分页](#_爬取_jobbole.com的文章列表和分页)的 parse\_detail 进行修改，将我们提取的数据都封装到 JobBoleArticleItem 当中去。在 parse\_detail 代码的后面增加如下代码：

|  |
| --- |
| article\_item = JobBoleArticleItem()  article\_item["title"] = title  article\_item["create\_date"] = create\_date  article\_item["praise\_nums"] = praise\_nums  article\_item["fav\_nums"] = fav\_nums  article\_item["comment\_nums"] = comment\_nums  article\_item["content"] = content  article\_item["tags"] = tags  article\_item["url"] = response.url  # 把 url 经过md5计算后保存到 url\_object\_id 变量中  article\_item["url\_object\_id"] = get\_md5(response.url)  article\_item["front\_image\_url"] = [front\_image\_url] |

就可以把数据封装到 JobBoleArticleItem 当中去了。接下来如何把封装好 JobBoleArticleItem 的数据进行保存呢？在 parse\_detail 中增加如下代码：

|  |
| --- |
| """  settings.py 文件中的  ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  }  需要打开,这样才能把 item 传递到 pipelines 中去  """  yield article\_item # 该语句会把封装好的 item 传递到 pipelines 中去处理 |

注意，要想把 item 传递到 pipeline 中，需要对 settings.py 中的配置项进行配置，这里主要是要把 ITEM\_PIPELINES 这个配置项的注释去掉。代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  } |

这样就会把 item 传递到 ITEM\_PIPELINES 这个配置项里面所配置的 pipeline 当中去进一步处理。但是这里 ITEM\_PIPELINES 的配置是通用的，也就是所有的 spider 共用的，并不是每个 spider 独有的。如果我们希望为每一个 spider 设置自己的 pipeline 怎么实现呢？很简单，示例代码如下：

|  |
| --- |
| class Spider1(scrapy.Spider):  name = 'spider1'  custom\_settings = {  'ITEM\_PIPELINES':{'pipelineClass1': 300,'pipelineClass2': 400},  }  class Spider2(scrapy.Spider):  name = 'spider2'  custom\_settings = {  'ITEM\_PIPELINES':{'pipelineClass3': 300},  } |

另外，在这里我们发现在进行数据封装的时候，并没有对 JobBoleArticleItem 中的 front\_image\_path 这个变量进行封装或者说如何把封面图进行下载并保存到本地呢？Scrapy已经封装了自动下载机制，我们需要在 settings.py 中配置一下即可。如下代码：（另外说明一下，图片下载需要安装 pil 库，pip install pillow，用豆瓣源安装更快）

|  |
| --- |
| # 注意：ITEM\_PIPELINES 这里的数字是指 Pipeline 处理的优先顺序，数字越小越优先  ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  'scrapy.pipelines.images.ImagesPipeline': 1,# scrapy提供的图片下载Pipeline  }  # 图片下载变量配置 开始  # 配置的变量名是固定的，变量用于 scrapy.pipelines.images.ImagesPipeline 的使用  # ImagesPipeline 位于 site-packages/scrapy/pipelines/images.py 文件的 class ImagesPipeline(FilesPipeline)  import os  IMAGES\_URLS\_FIELD = "front\_image\_url" # 这个是配置 item 中哪个字段为图片下载地址,它传递进去的值是一个数组,所以在封装 JobBoleArticleItem 的时候，使用 [front\_image\_url],封装成数组。  project\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))  # 下载图片之前，需要在 ArticleSpider 目录下先建立 images 目录  IMAGES\_STORE = os.path.join(project\_dir,'images') # 配置图片下载地址  # IMAGES\_MIN\_HEIGHT = 100 #设置图片的最小高度,需要的时候可以设置为打开  # IMAGES\_MIN\_WIDTH = 100 #设置图片的最小宽度,需要的时候可以设置为打开  # 图片下载变量配置 结束 |

图片下载已经配置完成了，那么如何把下载来的图片的路径保存到 JobBoleArticleItem 的 front\_image\_path 变量中去呢？这里，我们需要自己定义一个 pipeline。我们在 pipelines.py 文件中定义 ArticleImagePipeline 来保存图片在本地的路径，代码如下：

|  |
| --- |
| from scrapy.pipelines.images import ImagesPipeline  # ArticleImagePipeline 用来把下载下来的图片的本地路径保存到 front\_image\_path 变量中  # ArticleImagePipeline继承 scrapy.pipelines.images.ImagesPipeline  class ArticleImagePipeline(ImagesPipeline):  # 重载 item\_completed 函数  def item\_completed(self, results, item, info):  # ok 标志图片是否下载成功的变量  # value 中的 path 为图片下载后的本地路径  for ok,value in results:  image\_file\_path = value["path"]  item["front\_image\_path"] = image\_file\_path  # 每次重写 pipeline 之后，一定要把 item 返回去，方便下一个 pipeline 使用 item, 一般到最后一个的 pipeline 基本是用来做数据的最后保存的。  return item |

并把它配置到 settings.py 的 ITEM\_PIPELINES 配置项，由于我们自己定义的 pipeline 继承了 ImagesPipeline ，所以我们在 ITEM\_PIPELINES 中就不再需要配置 scrapy.pipelines.images.ImagesPipeline 了，修改配置后，代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  'ArticleSpider.pipelines.ArticleImagePipeline': 1,#定义自己的 ImagesPipeline 用来处理本地图片路径的保存  } |

这里，附上所有文件修改之后的代码，以后每节的代码的修改都是在前一节基础之上进行。

items.py 不变

common.py 不变

settings.py 代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  BOT\_NAME = 'ArticleSpider'  SPIDER\_MODULES = ['ArticleSpider.spiders']  NEWSPIDER\_MODULE = 'ArticleSpider.spiders'  # 设置为 False 表示 不遵循网站的 robots.txt 协议  ROBOTSTXT\_OBEY = False  # 注意：ITEM\_PIPELINES 这里的数字是指 Pipeline 处理的优先顺序，数字越小越优先  ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  #'scrapy.pipelines.images.ImagesPipeline': 1,  'ArticleSpider.pipelines.ArticleImagePipeline': 1,#定义自己的 ImagesPipeline  }  # 图片下载变量配置 开始  # 配置的变量名是固定的，变量用于 scrapy.pipelines.images.ImagesPipeline 的使用  # ImagesPipeline 位于 site-packages/scrapy/pipelines/images.py 文件中定义  import os  IMAGES\_URLS\_FIELD = "front\_image\_url"  project\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))  # 在 ArticleSpider 目录下建立 images 目录  IMAGES\_STORE = os.path.join(project\_dir,'images')  # IMAGES\_MIN\_HEIGHT = 100 #设置图片的最小高度,需要的时候可以设置为打开  # IMAGES\_MIN\_WIDTH = 100 #设置图片的最小宽度,需要的时候可以设置为打开  # 图片下载变量配置 结束 |

pipelines.py 代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  from scrapy.pipelines.images import ImagesPipeline  class ArticlespiderPipeline(object):  def process\_item(self, item, spider):  return item  # ArticleImagePipeline 用来把下载下来的图片的本地路径保存到 front\_image\_path 变量中  # ArticleImagePipeline继承 scrapy.pipelines.images.ImagesPipeline  class ArticleImagePipeline(ImagesPipeline):  # 重载 item\_completed 函数  def item\_completed(self, results, item, info):  # ok 标志图片是否下载成功的变量  # value 中的 path 为图片下载后的本地路径  for ok,value in results:  image\_file\_path = value["path"]  item["front\_image\_path"] = image\_file\_path  return item |

jobbole.py 代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import re  from scrapy.http import Request  from urllib import parse  from ArticleSpider.items import JobBoleArticleItem  from ArticleSpider.utils.common import get\_md5  class JobboleSpider(scrapy.Spider):  name = "jobbole"  allowed\_domains = ["blog.jobbole.com"]  start\_urls = ['http://blog.jobbole.com/all-posts/']  def parse(self, response):  # 这里把原来的 post\_urls 修改为 post\_nodes  post\_nodes = response.css("div#archive .floated-thumb .post-thumb a")  for post\_node in post\_nodes:  # 提取文章封面图的URL地址  image\_url = post\_node.css("img::attr(src)").extract\_first("")  # 提取文章内容页的URL地址  post\_url = post\_node.css("::attr(href)").extract\_first("")  print("文章URL地址", post\_url)  # 在这里通过 Request 对象的 meta 属性来定义保存 image\_url 的变量 front\_image\_url，  # 并把该变量 front\_image\_url 的值也传递给 parse\_detail 进行处理  yield Request(url=parse.urljoin(response.url,post\_url), meta={"front\_image\_url": image\_url},  callback=self.parse\_detail)  # 提取列表分页的URL地址的代码不变  next\_urls = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_urls:  print('分页链接地址：',next\_urls)  yield Request(url=parse.urljoin(response.url,next\_urls) , callback=self.parse)  def parse\_detail(self, response):  # 获取 meta={"front\_image\_url": image\_url} 传递过来的文章封面图url的值  front\_image\_url = response.meta.get("front\_image\_url","")  title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()  create\_date = response.xpath("//div[@class='entry-meta']/p[@class='entry-meta-hide-on-mobile']/text()").extract()[0].strip().replace("·","").strip()  praise\_nums = int(response.xpath("//span[contains(@class,'vote-post-up')]/h10/text()").extract()[0])  fav\_nums = response.xpath("//span[contains(@class,'bookmark-btn')]/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", fav\_nums)  if match\_re:  fav\_nums = int(match\_re.group(1))  else:  fav\_nums = 0  comment\_nums = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", comment\_nums)  if match\_re:  comment\_nums = int(match\_re.group(1))  else:  comment\_nums = 0  content = response.xpath("//div[@class='entry']").extract()[0]  tag\_list = response.xpath("//p[@class='entry-meta-hide-on-mobile']/a/text()").extract()  tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]  tags = ",".join(tag\_list)  # 数据封装  article\_item = JobBoleArticleItem()  article\_item["title"] = title  article\_item["create\_date"] = create\_date  article\_item["praise\_nums"] = praise\_nums  article\_item["fav\_nums"] = fav\_nums  article\_item["comment\_nums"] = comment\_nums  article\_item["content"] = content  article\_item["tags"] = tags  article\_item["url"] = response.url  # 把 url 经过md5计算后保存到 url\_object\_id 变量中  article\_item["url\_object\_id"] = get\_md5(response.url)  article\_item["front\_image\_url"] = [front\_image\_url]  """  settings.py 文件中的  ITEM\_PIPELINES = {  'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  }  需要打开,这样才能把 item 传递到 pipelines 中去  """  yield article\_item # 该语句会把封装好的 item 传递到 pipelines 中去处理 |

### Django 模块编写items

https://github.com/scrapy-plugins/scrapy-djangoitem 这个scrapy插件可以使用 Django models 来编写 items ， 这样操作数据库更方便。

## 数据保存

**如无特殊说明，本节代码均在上节代码的基础之上进行增加和修改**。

### JSON方式保存数据

方式一：使用自定义 JSON 文件保存格式

在 pipelines.py 文件自定义 JSON文件保存的函数 JsonWithEncodingPipeline(object) 函数，代码如下：

|  |
| --- |
| # 数据以 JSON 格式保存  class JsonWithEncodingPipeline(object):  # 自定义 json 文件的导出  def \_\_init\_\_(self):  # 把数据以 JSON 格式保存到 article.json 文件中  self.file = codecs.open('article.json','w',encoding='utf-8')  def process\_item(self, item, spider):  lines = json.dumps(dict(item),ensure\_ascii=False) + "\n"  self.file.write(lines)  return item  def spider\_closed(self,spider):  self.file.close() |

在 settings.py 配置 JsonWithEncodingPipeline ，代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.JsonWithEncodingPipeline': 2,  } |

方式二：使用 Scrapy 提供的类来保存数据到 JSON 文件

使用 Scrapy 自带的 JSON 文件保存函数 JsonItemExporter 来实现 JsonExporterPipeline(object) 函数，代码如下：

|  |
| --- |
| # 使用 scrapy 提供的 json export 导出数据到 json 文件  class JsonExporterPipeline(object):  def \_\_init\_\_(self):  self.file = open('articleexport.json','wb')  self.exporter = JsonItemExporter(self.file,encoding='utf-8',ensure\_ascii=False)  self.exporter.start\_exporting()  def process\_item(self, item, spider):  self.exporter.export\_item(item)  return item  def close\_spider(self,spider):  self.exporter.finish\_exporting()  self.file.close() |

在 settings.py 配置 JsonExporterPipeline ，代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.JsonExporterPipeline': 2,  } |

以上两种 JSON 保存的数据的不同点在于 JsonExporterPipeline 比 JsonWithEncodingPipeline 多了一对 [ ] 中括号。

### MySQL数据库保存数据

首先要设计我们的表结构，代码如下：

|  |
| --- |
| DROP TABLE IF EXISTS `jobbole\_article`;  CREATE TABLE `jobbole\_article` (  `title` varchar(200) NOT NULL,  `create\_date` date DEFAULT NULL,  `url` varchar(300) NOT NULL,  `url\_object\_id` varchar(50) NOT NULL,  `front\_image\_url` varchar(300) DEFAULT NULL,  `front\_image\_path` varchar(200) DEFAULT NULL,  `comment\_nums` int(11) NOT NULL DEFAULT '0',  `fav\_nums` int(11) NOT NULL DEFAULT '0',  `praise\_nums` int(11) NOT NULL DEFAULT '0',  `tags` varchar(200) DEFAULT NULL,  `content` longtext NOT NULL,  PRIMARY KEY (`url\_object\_id`)  ) ENGINE=MyISAM DEFAULT CHARSET=utf8; |

如下图所示：



由于我们需要在数据库里面把 create\_date 保存为 date 格式，所以需要修改 jobbole.py 中 parse\_detail(self,response) 函数中的 article\_item["create\_date"] = create\_date 语句，把它修改为：

|  |
| --- |
| # create\_date 进行日期转换并放到 try 语句中  try:  create\_date = datetime.datetime.strptime(create\_date,"%Y/%m/%d").date()  except Exception as e:  create\_date = datetime.datetime.now().date()  article\_item["create\_date"] = create\_date |

然后需要安装 mysql 驱动：

|  |
| --- |
| pip install -i https://pypi.doubanio.com/simple/ mysqlclient |

附带：

Ubuntu 安装 mysql 驱动： sudo apt-get install libmysqlclient-dev

CentOS 安装 mysql 驱动： sudo yum install python-devel mysql-devel

准备工作做好之后，我们就可以编写 Scrapy 保存数据到数据库的几种方式了。

#### 同步方式保存数据

在 pipelines.py 增加 MysqlPipeline(object) 函数，代码如下：

|  |
| --- |
| class MysqlPipeline(object):  #采用同步机制插入数据库，效率较低，容易阻塞  def \_\_init\_\_(self):  self.conn = MySQLdb.connect('127.0.0.1','root','root','article\_spider',charset='utf-8',use\_unicode=True)  self.cursor = self.conn.cursor()  def process\_item(self, item, spider):  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  self.cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums']))  self.conn.commit() |

在 settings.py 配置 MysqlPipeline ，代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.MysqlPipeline': 2,  } |

#### 使用Twisted异步保存

在 pipelines.py 增加 MysqlTwistedPipeline(object) 函数，代码如下：

|  |
| --- |
| # 使用 Twisted 提供的链接池操作数据库,可以把我们的数据异步化插入数据库，把插入操作变成异步操作  # 链接信息在 settings.py 中配置  # 目前 Twisted 只支持关系型数据库的异步操作，不支持非关系型数据库  class MysqlTwistedPipeline(object):  def \_\_init\_\_(self,dbpool):  self.dbpool = dbpool  # 该函数的定义是固定的，也就是说函数名 from\_settings 是固定的,用于读取 settings.py 配置文件  # 该方法在我们自定义组件和扩展的时候非常有用  # 这里的参数 settings 其实就是对应于 settings.py 这个文件  # from\_settings 由 scrapy 自动调用，在调用的时候会把 settings.py 的内容传递到 settings 变量  @classmethod  def from\_settings(cls,settings):  dbparms = dict(  host = settings['MYSQL\_HOST'],  db = settings['MYSQL\_DBNAME'],  user = settings['MYSQL\_USER'],  passwd = settings['MYSQL\_PASSWORD'],  charset = 'utf8',  cursorclass = MySQLdb.cursors.DictCursor,  use\_unicode = True,  )  dbpool = adbapi.ConnectionPool("MySQLdb",\*\*dbparms)  return cls(dbpool)  def process\_item(self, item, spider):  #使用 twisted 将 mysql 的 insert 操作变成异步执行  query = self.dbpool.runInteraction(self.do\_insert,item)  query.addErrback(self.handle\_error,item,spider) # 处理异常  def handle\_error(self,failure,item,spider):  #处理异步插入异常  print(failure)  def do\_insert(self,cursor,item):  #执行具体的 insert 操作语句  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums'])) |

在 settings.py 配置 MysqlTwistedPipeline 的 ITEM\_PIPELINES 和 数据库配置信息，代码如下：

|  |
| --- |
| ITEM\_PIPELINES = {  'ArticleSpider.pipelines.MysqlTwistedPipeline': 2,  }  MYSQL\_HOST = '127.0.0.1'  MYSQL\_DBNAME = 'article\_spider'  MYSQL\_USER = 'root'  MYSQL\_PASSWORD = 'root' |

这里附带上，以上 jobbole.py、pipelines.py、settings.py 这三个文件修改后的最后代码：

|  |
| --- |
| jobbole.py |
| # -\*- coding: utf-8 -\*-  import scrapy  import re  import datetime  from scrapy.http import Request  from urllib import parse  from ArticleSpider.items import JobBoleArticleItem  from ArticleSpider.utils.common import get\_md5  class JobboleSpider(scrapy.Spider):  name = "jobbole"  allowed\_domains = ["blog.jobbole.com"]  start\_urls = ['http://blog.jobbole.com/all-posts/']  def parse(self, response):  post\_nodes = response.css("div#archive .floated-thumb .post-thumb a")  for post\_node in post\_nodes:  image\_url = post\_node.css("img::attr(src)").extract\_first("")  post\_url = post\_node.css("::attr(href)").extract\_first("")  print("文章URL地址", post\_url)  yield Request(url=parse.urljoin(response.url,post\_url), meta={"front\_image\_url": image\_url},  callback=self.parse\_detail)  next\_urls = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_urls:  print('分页链接地址：',next\_urls)  yield Request(url=parse.urljoin(response.url,next\_urls) , callback=self.parse)  def parse\_detail(self, response):  front\_image\_url = response.meta.get("front\_image\_url","")  title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()  create\_date = response.xpath("//div[@class='entry-meta']/p[@class='entry-meta-hide-on-mobile']/text()").extract()[0].strip().replace("·","").strip()  praise\_nums = int(response.xpath("//span[contains(@class,'vote-post-up')]/h10/text()").extract()[0])  fav\_nums = response.xpath("//span[contains(@class,'bookmark-btn')]/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", fav\_nums)  if match\_re:  fav\_nums = int(match\_re.group(1))  else:  fav\_nums = 0  comment\_nums = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]  match\_re = re.match(r".\*?(\d+).\*", comment\_nums)  if match\_re:  comment\_nums = int(match\_re.group(1))  else:  comment\_nums = 0  content = response.xpath("//div[@class='entry']").extract()[0]  tag\_list = response.xpath("//p[@class='entry-meta-hide-on-mobile']/a/text()").extract()  tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]  tags = ",".join(tag\_list)  article\_item = JobBoleArticleItem()  article\_item["title"] = title  # create\_date 进行日期转换并放到 try 语句中  try:  create\_date = datetime.datetime.strptime(create\_date,"%Y/%m/%d").date()  except Exception as e:  create\_date = datetime.datetime.now().date()  article\_item["create\_date"] = create\_date  article\_item["praise\_nums"] = praise\_nums  article\_item["fav\_nums"] = fav\_nums  article\_item["comment\_nums"] = comment\_nums  article\_item["content"] = content  article\_item["tags"] = tags  article\_item["url"] = response.url  article\_item["url\_object\_id"] = get\_md5(response.url)  article\_item["front\_image\_url"] = [front\_image\_url]  yield article\_item |

|  |
| --- |
| pipelines.py |
| # -\*- coding: utf-8 -\*-  import codecs  import json  import MySQLdb  import MySQLdb.cursors  from scrapy.pipelines.images import ImagesPipeline  from scrapy.exporters import JsonItemExporter  from twisted.enterprise import adbapi  class ArticlespiderPipeline(object):  def process\_item(self, item, spider):  return item  # ArticleImagePipeline 用来把下载下来的图片的本地路径保存到 front\_image\_path 变量中  # ArticleImagePipeline继承 scrapy.pipelines.images.ImagesPipeline  class ArticleImagePipeline(ImagesPipeline):  # 重载 item\_completed 函数  def item\_completed(self, results, item, info):  # ok 标志图片是否下载成功的变量  # value 中的 path 为图片下载后的本地路径  for ok,value in results:  image\_file\_path = value["path"]  item["front\_image\_path"] = image\_file\_path  return item  # 数据以 JSON 格式保存  class JsonWithEncodingPipeline(object):  # 自定义 json 文件的导出  def \_\_init\_\_(self):  # 把数据以 JSON 格式保存到 article.json 文件中  self.file = codecs.open('article.json','w',encoding='utf-8')  def process\_item(self, item, spider):  lines = json.dumps(dict(item),ensure\_ascii=False) + "\n"  self.file.write(lines)  return item  def spider\_closed(self,spider):  self.file.close()  # 使用 scrapy 提供的 json export 导出数据到 json 文件  class JsonExporterPipeline(object):  def \_\_init\_\_(self):  self.file = open('articleexport.json','wb')  self.exporter = JsonItemExporter(self.file,encoding='utf-8',ensure\_ascii=False)  self.exporter.start\_exporting()  def process\_item(self, item, spider):  self.exporter.export\_item(item)  return item  def close\_spider(self,spider):  self.exporter.finish\_exporting()  self.file.close()  class MysqlPipeline(object):  #采用同步机制插入数据库，效率较低，容易阻塞  def \_\_init\_\_(self):  self.conn = MySQLdb.connect('127.0.0.1','root','root','article\_spider',charset='utf-8',use\_unicode=True)  self.cursor = self.conn.cursor()  def process\_item(self, item, spider):  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  self.cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums']))  self.conn.commit()  # 使用 Twisted 提供的链接池操作数据库,可以把我们的数据异步化插入数据库，把插入操作变成异步操作  # 链接信息在 settings.py 中配置  # 目前 Twisted 只支持关系型数据库的异步操作，不支持非关系型数据库  class MysqlTwistedPipeline(object):  def \_\_init\_\_(self,dbpool):  self.dbpool = dbpool  # 该函数的定义是固定的，也就是说函数名 from\_settings 是固定的,用于读取 settings.py 配置文件  # 该方法在我们自定义组件和扩展的时候非常有用  # 这里的参数 settings 其实就是对应于 settings.py 这个文件  # from\_settings 由 scrapy 自动调用，在调用的时候会把 settings.py 的内容传递到 settings 变量  @classmethod  def from\_settings(cls,settings):  dbparms = dict(  host = settings['MYSQL\_HOST'],  db = settings['MYSQL\_DBNAME'],  user = settings['MYSQL\_USER'],  passwd = settings['MYSQL\_PASSWORD'],  charset = 'utf8',  cursorclass = MySQLdb.cursors.DictCursor,  use\_unicode = True,  )  dbpool = adbapi.ConnectionPool("MySQLdb",\*\*dbparms)  return cls(dbpool)  def process\_item(self, item, spider):  #使用 twisted 将 mysql 的 insert 操作变成异步执行  query = self.dbpool.runInteraction(self.do\_insert,item)  query.addErrback(self.handle\_error,item,spider) # 处理异常  def handle\_error(self,failure,item,spider):  #处理异步插入异常  print(failure)  def do\_insert(self,cursor,item):  #执行具体的 insert 操作语句  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums'])) |

|  |
| --- |
| settings.py |
| # -\*- coding: utf-8 -\*-  BOT\_NAME = 'ArticleSpider'  SPIDER\_MODULES = ['ArticleSpider.spiders']  NEWSPIDER\_MODULE = 'ArticleSpider.spiders'  # 设置为 False 表示 不遵循网站的 robots.txt 协议  ROBOTSTXT\_OBEY = False  # 注意：ITEM\_PIPELINES 这里的数字是指 Pipeline 处理的优先顺序，数字越小越优先  ITEM\_PIPELINES = {  'ArticleSpider.pipelines.MysqlTwistedPipeline': 2,#异步插入数据,性能高  #'ArticleSpider.pipelines.MysqlPipeline': 2,#同步插入数据,性能低  #'ArticleSpider.pipelines.JsonExporterPipeline': 2,#使用Scrapy提供的JSON功能保存  #'ArticleSpider.pipelines.JsonWithEncodingPipeline': 2,# 自定义JSON保存  #'ArticleSpider.pipelines.ArticlespiderPipeline': 300,  #'scrapy.pipelines.images.ImagesPipeline': 1,  'ArticleSpider.pipelines.ArticleImagePipeline': 1,#定义自己的 ImagesPipeline  }  # 图片下载变量配置 开始  # 配置的变量名是固定的，变量用于 scrapy.pipelines.images.ImagesPipeline 的使用  # ImagesPipeline 位于 site-packages/scrapy/pipelines/images.py 文件中定义  import os  IMAGES\_URLS\_FIELD = "front\_image\_url"  project\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))  # 在 ArticleSpider 目录下建立 images 目录  IMAGES\_STORE = os.path.join(project\_dir,'images')  # IMAGES\_MIN\_HEIGHT = 100 #设置图片的最小高度,需要的时候可以设置为打开  # IMAGES\_MIN\_WIDTH = 100 #设置图片的最小宽度,需要的时候可以设置为打开  # 图片下载变量配置 结束  # 数据库连接配置  MYSQL\_HOST = '127.0.0.1'  MYSQL\_DBNAME = 'article\_spider'  MYSQL\_USER = 'root'  MYSQL\_PASSWORD = 'root' |

到这里数据的采集和保存就都完成了。

## ItemLoader 数据封装

使用 ItemLoader 可以使代码更简洁，另外，当我们多个 spider 的提取规则一样的时候，ItemLoader 可以达到代码的复用。

使用 ItemLoader 需要先导入，代码如下：

|  |
| --- |
| from scrapy.loader import ItemLoader |

这个代码在 items.py 文件中导入，为什么呢？

因为，ItemLoader 提取字段之后返回的都是 list 对象，而很多字段都是要提取 list 里面的第一个元素的值，会用到 ItemLoader 中的如下属性：

|  |
| --- |
| default\_output\_processor = TakeFirst() |

但是，如果每个字段都写这个代码的话就会造成代码的累赘，所以需要自定义一个 ItemLoader。我们自定义一个 ArticleItemLoader 该类继承 ItemLoader。然后在 spider 文件 jobbole.py 通过 ArticleItemLoader 来对 JobBoleArticleItem 进行 load 操作。

附带上 jobbole.py items.py pipelines.py 修改后的完整代码：在这些文件中有详细的解释：

|  |
| --- |
| jobbole.py |
| # -\*- coding: utf-8 -\*-  import scrapy  from scrapy.http import Request  from urllib import parse  from ArticleSpider.items import JobBoleArticleItem,ArticleItemLoader  from ArticleSpider.utils.common import get\_md5  class JobboleSpider(scrapy.Spider):  name = "jobbole"  allowed\_domains = ["blog.jobbole.com"]  start\_urls = ['http://blog.jobbole.com/all-posts/']  def parse(self, response):  post\_nodes = response.css("div#archive .floated-thumb .post-thumb a")  for post\_node in post\_nodes:  image\_url = post\_node.css("img::attr(src)").extract\_first("")  post\_url = post\_node.css("::attr(href)").extract\_first("")  print("文章URL地址", post\_url)  yield Request(url=parse.urljoin(response.url,post\_url), meta={"front\_image\_url": image\_url},  callback=self.parse\_detail)  next\_urls = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_urls:  print('分页链接地址：',next\_urls)  yield Request(url=parse.urljoin(response.url,next\_urls) , callback=self.parse)  def parse\_detail(self, response):  front\_image\_url = response.meta.get("front\_image\_url", "")  # 通过 item loader 加载 item  # 提取的每个字段都保存在各自的一个 list 当中  # 使用自定义 ItemLoader 的子类 ArticleItemLoader 来 load 操作 JobBoleArticleItem  item\_loader = ArticleItemLoader(item = JobBoleArticleItem(), response = response)  item\_loader.add\_css("title",".entry-header h1::text")  item\_loader.add\_value("url",response.url)  item\_loader.add\_value("url\_object\_id",get\_md5(response.url))  item\_loader.add\_css("create\_date","p.entry-meta-hide-on-mobile::text")  item\_loader.add\_value("front\_image\_url",[front\_image\_url])  item\_loader.add\_css("praise\_nums",".vote-post-up h10::text")  item\_loader.add\_css("comment\_nums","a[href='#article-comment'] span::text")  item\_loader.add\_css("fav\_nums","span.bookmark-btn::text")  item\_loader.add\_css("content", "div.entry")  item\_loader.add\_css("tags", "p.entry-meta-hide-on-mobile a::text")  article\_item = item\_loader.load\_item()  yield article\_item |

|  |
| --- |
| items.py |
| # -\*- coding: utf-8 -\*-  import scrapy  import datetime  import re  from scrapy.loader.processors import MapCompose,TakeFirst,Join  from scrapy.loader import ItemLoader  class ArticlespiderItem(scrapy.Item):  pass  def add\_jobbole(value):  return value + "-lhx"  def date\_convert(value):  try:  create\_date = datetime.datetime.strptime(value, "%Y/%m/%d").date()  except Exception as e:  create\_date = datetime.datetime.now().date()  return create\_date  def get\_nums(value):  match\_re = re.match(r".\*?(\d+).\*", value)  if match\_re:  nums = int(match\_re.group(1))  else:  nums = 0  return nums  #去掉 tag 中提取的评论  def remove\_comment\_tags(value):  if "评论" in value:  return ""  else:  return value  def return\_value(value):  return value  # 自定义 ItemLoader  class ArticleItemLoader(ItemLoader):  default\_output\_processor = TakeFirst()  # scrapy.Field() 实际上可以接收两个参数：input\_processor 和 output\_processor  class JobBoleArticleItem(scrapy.Item):  url = scrapy.Field()  url\_object\_id = scrapy.Field()  # 这里为什么要使用 return\_value 函数？  # 因为 front\_image\_url 是以 list 方式传递到 item 中的，  # 但是用了 default\_output\_processor 处理之后，它就变成了一个字符串  # 当我们把这个 item 交给 pipelines.py 中的 ArticleImagePipeline 进行图片下载的时候  # 由于 ArticleImagePipeline 需要传递一个 list 那这样就会抛出异常  # 所以这里要覆盖 default\_output\_processor 这个变量 ,  # 使 front\_image\_url 不会变成字符串，而保持 list 格式  # 所以定义了 return\_value 函数，让它直接返回原来的值即可  front\_image\_url = scrapy.Field(  output\_processor = MapCompose(return\_value)  )  front\_image\_path = scrapy.Field()  title = scrapy.Field(  # MapCompose 的作用是可以调用任意多个函数对数据进行处理  input\_processor = MapCompose(lambda x:x + '-zls',add\_jobbole)  )  create\_date = scrapy.Field(  input\_processor = MapCompose(date\_convert)  )  praise\_nums = scrapy.Field(  input\_processor = MapCompose(get\_nums)  )  fav\_nums = scrapy.Field(  input\_processor = MapCompose(get\_nums)  )  comment\_nums = scrapy.Field(  input\_processor = MapCompose(get\_nums)  )  content = scrapy.Field()  tags = scrapy.Field(  input\_processor = MapCompose(remove\_comment\_tags),  output\_processor = Join(',')  ) |

|  |
| --- |
| pipelines.py |
| # -\*- coding: utf-8 -\*-  import codecs  import json  import MySQLdb  import MySQLdb.cursors  from scrapy.pipelines.images import ImagesPipeline  from scrapy.exporters import JsonItemExporter  from twisted.enterprise import adbapi  class ArticlespiderPipeline(object):  def process\_item(self, item, spider):  return item  # ArticleImagePipeline 用来把下载下来的图片的本地路径保存到 front\_image\_path 变量中  # ArticleImagePipeline继承 scrapy.pipelines.images.ImagesPipeline  class ArticleImagePipeline(ImagesPipeline):  # 重载 item\_completed 函数  def item\_completed(self, results, item, info):  # 由于 pipeline 是共用的,有些 spider 采集的字段不包含 front\_image\_url 字段，  # 所以这里在上节代码的基础上加个判断，判断 item 是否包含 front\_image\_url 字段，否则会出错  if 'front\_image\_url' in item:  # ok 标志图片是否下载成功的变量  # value 中的 path 为图片下载后的本地路径  for ok,value in results:  image\_file\_path = value["path"]  item["front\_image\_path"] = image\_file\_path  return item  # 数据以 JSON 格式保存  class JsonWithEncodingPipeline(object):  # 自定义 json 文件的导出  def \_\_init\_\_(self):  # 把数据以 JSON 格式保存到 article.json 文件中  self.file = codecs.open('article.json','w',encoding='utf-8')  def process\_item(self, item, spider):  lines = json.dumps(dict(item),ensure\_ascii=False) + "\n"  self.file.write(lines)  return item  def spider\_closed(self,spider):  self.file.close()  # 使用 scrapy 提供的 json export 导出数据到 json 文件  class JsonExporterPipeline(object):  def \_\_init\_\_(self):  self.file = open('articleexport.json','wb')  self.exporter = JsonItemExporter(self.file,encoding='utf-8',ensure\_ascii=False)  self.exporter.start\_exporting()  def process\_item(self, item, spider):  self.exporter.export\_item(item)  return item  def close\_spider(self,spider):  self.exporter.finish\_exporting()  self.file.close()  class MysqlPipeline(object):  #采用同步机制插入数据库，效率较低，容易阻塞  def \_\_init\_\_(self):  self.conn = MySQLdb.connect('127.0.0.1','root','root','article\_spider',charset='utf-8',use\_unicode=True)  self.cursor = self.conn.cursor()  def process\_item(self, item, spider):  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  self.cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums']))  self.conn.commit()  # 使用 Twisted 提供的链接池操作数据库,可以把我们的数据异步化插入数据库，把插入操作变成异步操作  # 链接信息在 settings.py 中配置  # 目前 Twisted 只支持关系型数据库的异步操作，不支持非关系型数据库  class MysqlTwistedPipeline(object):  def \_\_init\_\_(self,dbpool):  self.dbpool = dbpool  # 该函数的定义是固定的，也就是说函数名 from\_settings 是固定的,用于读取 settings.py 配置文件  # 该方法在我们自定义组件和扩展的时候非常有用  # 这里的参数 settings 其实就是对应于 settings.py 这个文件  # from\_settings 由 scrapy 自动调用，在调用的时候会把 settings.py 的内容传递到 settings 变量  @classmethod  def from\_settings(cls,settings):  dbparms = dict(  host = settings['MYSQL\_HOST'],  db = settings['MYSQL\_DBNAME'],  user = settings['MYSQL\_USER'],  passwd = settings['MYSQL\_PASSWORD'],  charset = 'utf8',  cursorclass = MySQLdb.cursors.DictCursor,  use\_unicode = True,  )  dbpool = adbapi.ConnectionPool("MySQLdb",\*\*dbparms)  return cls(dbpool)  def process\_item(self, item, spider):  #使用 twisted 将 mysql 的 insert 操作变成异步执行  query = self.dbpool.runInteraction(self.do\_insert,item)  query.addErrback(self.handle\_error,item,spider) # 处理异常  def handle\_error(self,failure,item,spider):  #处理异步插入异常  print(failure)  def do\_insert(self,cursor,item):  #执行具体的 insert 操作语句  insert\_sql = """  insert into jobbole\_article(title,create\_date,url,fav\_nums)  values(%s,%s,%s,%s)  """  cursor.execute(insert\_sql,(item['title'],item['create\_date'],item['url'],item['fav\_nums'])) |

# 爬取 [www.zhihu.com](http://www.zhihu.com)

## 知乎模拟登陆

在进行知乎 <https://www.zhihu.com/> 模拟登陆之前，先准备好测试账号，我这里的测试账号为： 13459282910 a5s7sh4u 。有了账号才能进行测试。

在模拟登陆编码的过程中，我们可以输入一些错误的账号，以便查看账号的提交地址，因为如果输入正确的话，它登陆之后就跳转了，不方便查看账号提交地址。

### 使用 requests 的 session 模拟登陆

在 ArticleSpider.utils 目录下建立 requests.login.zhihu.py 文件，代码如下：

|  |
| --- |
| requests.login.zhihu.py |
| # \_\*\_ coding:utf-8 \_\*\_  import requests # pip install requests 安装 requests 库文件  import re  try:  import cookielib # python2 版本  except:  import http.cookiejar as cookielib # python3 版本  agent = "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:54.0) Gecko/20100101 Firefox/54.0"  headers = {  "HOST":"www.zhihu.com",  "Referer":"https://www.zhihu.com",  "User-Agent":agent  }  # 步骤：  # 1、先加载 cookies.txt 文件的内容，判断是否存在，如果存在且有效则读取 cookies 进行登陆  # 2、如果 cookies.txt 不存在或者 cookies 过期无效，则先登录再保存 cookies 到 cookies.txt 文件  session = requests.session()  session.cookies = cookielib.LWPCookieJar(filename = "cookies.txt")  def is\_session\_useful():  useful = False  try:  session.cookies.load(ignore\_discard = True)  print("cookies.txt 文件有效")  useful = True  except:  print("cookies.txt 文件无效")  return useful  def get\_index():  response = session.get("https://www.zhihu.com", headers = headers)  # 登陆成功后知乎首页的内容返回并保存到 index\_page.html 文件中,用来查看登陆成功后首页的成功标志  with open("index\_page.html","wb") as f:  f.write(response.text.encode("utf-8"))  print("登陆 OK!!!")  def is\_login():  #通过个人中心页面的返回的状态码来判断是否为登陆状态  inbox\_url = "https://www.zhihu.com/inbox"  response = session.get(inbox\_url,headers = headers,allow\_redirects = False)  with open("inbox\_page.html","wb") as f:  f.write(response.text.encode("utf-8"))  print("个人中心状态码",response.status\_code)  if response.status\_code != 200:  return False  else:  return True  # 获取 \_xsrf 的值  def get\_xsrf():  response = session.get("https://www.zhihu.com",headers = headers)  match\_obj = re.search('.\*name="\_xsrf" value="(.\*)"',response.text)  print(match\_obj)  if match\_obj:  print("\_xsrf 的值为",match\_obj.group(1))  return match\_obj.group(1)  else:  print("\_xsrf 的值为 NULL")  return ""  def zhihu\_login(account,password):  if re.match("^1\d{10}",account):  print("手机号码登陆")  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data = {  "\_xsrf":get\_xsrf(),  "phone\_num":account,  "password":password  }  else:  if "@" in account:  print("邮箱方式登陆")  post\_url = "https://www.zhihu.com/login/email"  post\_data = {  "\_xsrf": get\_xsrf(),  "email": account,  "password": password  }  response = session.post(post\_url, data = post\_data, headers = headers)  print(response.text)  print("登陆的返回值",response.text)  session.cookies.save()  # 登陆流程  is\_session\_useful = is\_session\_useful()  if is\_session\_useful :  print("session 有效，登陆成功")  get\_index()  is\_login = is\_login()  if is\_login:  print("正在个人中心")  else:  print("不在个人中心")  else:  print("session 无效，登陆失败，重新登陆")  account = "13459282910"  password = "a5s7sh4u"  zhihu\_login(account, password)  get\_index()  is\_login = is\_login()  if is\_login:  print("正在个人中心")  else:  print("不在个人中心") |

代码逻辑没问题，但是不知道是不是需要验证码，一直都显示“不在个人中心”。

### 使用 Scrapy 的 FormRequest 模拟登陆

我们先创建一个名为 zhihu 的 spider，代码如下：

|  |
| --- |
| scrapy genspider zhihu www.zhihu.com |

然后编写 zhihu.py 文件的代码，如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import json  import re  class ZhihuSpider(scrapy.Spider):  name = "zhihu"  allowed\_domains = ["www.zhihu.com"]  start\_urls = ['http://www.zhihu.com/']  user\_agent = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"  headers = {  "HOST": "www.zhihu.com",  "Referer": "https://www.zhihu.com",  "User-Agent": user\_agent  }  def parse(self, response):  pass  # start\_requests 是 spider 抓取数据的入口，由于在抓取数据之前需要先登录，  # 所以需要先改写 start\_requests 函数。  # 重写 start\_requests 函数,该函数也是爬虫的入口程序  # Scrapy 是基于 Twisted 框架的，它所有的操作都是异步的。  # 这里需要设置 callback 函数，否则会默认调用 parse 函数。  def start\_requests(self):  return [scrapy.Request('https://www.zhihu.com/#signin', headers = self.headers, callback = self.login)]  def login(self,response):  response\_text = response.text  # 如果不加 re.DOTALL 参数，那么把 re.match 修改为 re.search 也可以实现多行匹配  match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL) # re.DOTALL 表示开启多行匹配  xsrf = ''  if match\_obj:  xsrf = match\_obj.group(1)  print("\_xsrf：",xsrf)  if xsrf:  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data = {  "\_xsrf":xsrf,  "phone\_num":"13459282910",  "password":"a5s7sh4u"  }  return [scrapy.FormRequest(  url = post\_url,  formdata = post\_data,  headers = self.headers,  callback = self.check\_login # 不加()表示调用函数对象，加()表示函数调用执行  )]  # 与 requests.login.zhihu.py 不同的是，scrapy 不需要把 cookies 保存起来，  # 因为 scrapy 的所有后续的 request 请求，都会默认把 cookies 带进去  def check\_login(self,response):  #验证服务器返回的数据来验证是否登录成功  text\_json = json.loads(response.text)  print("msg",text\_json["msg"])  if "msg" in text\_json and text\_json["msg"] == "登录成功":  print("登录成功")  # 这里就是开始处理 start\_urls 了。  for url in self.start\_urls:  # 这里没有设置 callback 函数，所以默认调用 parse 函数  # dont\_filter  yield scrapy.Request(url, dont\_filter = True,headers = self.headers) |

然后测试该蜘蛛，需要在 main.py 中添加如下代码：

|  |
| --- |
| execute(["scrapy","crawl","zhihu"]) |

此时 main.py 的完整代码如下：

|  |
| --- |
| # \_\*\_ coding:utf-8 \_\*\_  from scrapy.cmdline import execute  import sys  import os  sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))  #execute(["scrapy","crawl","jobbole"])  execute(["scrapy","crawl","zhihu"]) |

zhihu.py 也是差一个验证码问题，导致测试失败。

## 知乎登陆验证码问题

我们在上一节[知乎模拟登陆](#_知乎模拟登陆)编写模拟登陆的时候，测试都是不能显示个人中心，这个是因为碰到了验证码的问题。

这里将在上一节[知乎模拟登陆](#_知乎模拟登陆)的基础之上加入验证码功能。

requests.login.zhihu.py 加入验证码后的代码如下：

|  |
| --- |
| # \_\*\_ coding:utf-8 \_\*\_  import requests # pip install requests 安装 requests 库文件  import re  try:  import cookielib # python2 版本  except:  import http.cookiejar as cookielib # python3 版本  agent = "Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:54.0) Gecko/20100101 Firefox/54.0"  headers = {  "HOST":"www.zhihu.com",  "Referer":"https://www.zhihu.com",  "User-Agent":agent  }  # 步骤：  # 1、先加载 cookies.txt 文件的内容，判断是否存在，如果存在且有效则读取 cookies 进行登陆  # 2、如果 cookies.txt 不存在或者 cookies 过期无效，则先登录再保存 cookies 到 cookies.txt 文件  session = requests.session()  session.cookies = cookielib.LWPCookieJar(filename = "cookies.txt")  def is\_session\_useful():  useful = False  try:  session.cookies.load(ignore\_discard = True)  print("cookies.txt 文件有效")  useful = True  except:  print("cookies.txt 文件无效")  return useful  def get\_index():  response = session.get("https://www.zhihu.com", headers = headers)  # 登陆成功后知乎首页的内容返回并保存到 index\_page.html 文件中,用来查看登陆成功后首页的成功标志  with open("index\_page.html","wb") as f:  f.write(response.text.encode("utf-8"))  print("登陆 OK!!!")  def is\_login():  #通过个人中心页面的返回的状态码来判断是否为登陆状态  inbox\_url = "https://www.zhihu.com/inbox"  response = session.get(inbox\_url,headers = headers,allow\_redirects = False)  with open("inbox\_page.html","wb") as f:  f.write(response.text.encode("utf-8"))  print("个人中心状态码",response.status\_code)  if response.status\_code != 200:  return False  else:  return True  # 获取 \_xsrf 的值  def get\_xsrf():  response = session.get("https://www.zhihu.com",headers = headers)  match\_obj = re.search('.\*name="\_xsrf" value="(.\*)"',response.text)  print(match\_obj)  if match\_obj:  print("\_xsrf 的值为",match\_obj.group(1))  return match\_obj.group(1)  else:  print("\_xsrf 的值为 NULL")  return ""  # 打开浏览器的调试功能，单击验证码图片，查看验证码图片的 url 地址  # 另外，在验证码图片打开之后,要记住验证码然后关闭验证码图片，才能输入验证码  def get\_captcha():  import time  t = str(int(time.time()\*1000))  captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login".format(t)  # 另外, 这里必须用 session , 而不能用 request 否则会返回图片验证码错误  # 因为 request 会单独再建立一个 session 这样会造成 \_xsrf 的值和图片不匹配, 因为它们不在同一个 session 中  t = session.get(captcha\_url, headers = headers)  with open("captcha.jpg","wb") as f:  f.write(t.content)  f.close()  from PIL import Image  try:  im = Image.open('captcha.jpg')  im.show()  im.close()  except:  print("图片打开失败")  pass  captcha = input("输入验证码：\n")  return captcha  def zhihu\_login(account,password):  if re.match("^1\d{10}",account):  print("手机号码登陆")  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data = {  "\_xsrf":get\_xsrf(),  "phone\_num":account,  "password":password,  "captcha":get\_captcha() # 加入验证码的获取和发送功能  }  else:  if "@" in account:  print("邮箱方式登陆")  post\_url = "https://www.zhihu.com/login/email"  post\_data = {  "\_xsrf": get\_xsrf(),  "email": account,  "password": password  }  response = session.post(post\_url, data = post\_data, headers = headers)  session.cookies.save()  # 登陆流程  is\_session\_useful = is\_session\_useful()  if is\_session\_useful :  print("session 有效，登陆成功")  get\_index()  is\_login = is\_login()  if is\_login:  print("正在个人中心")  else:  print("不在个人中心")  else:  print("session 无效，登陆失败，重新登陆")  account = "13459282910"  password = "a5s7sh4u"  zhihu\_login(account, password)  get\_index()  is\_login = is\_login()  if is\_login:  print("正在个人中心")  else:  print("不在个人中心") |

zhihu.py 加入验证码后的代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import json  import re  class ZhihuSpider(scrapy.Spider):  name = "zhihu"  allowed\_domains = ["www.zhihu.com"]  start\_urls = ['http://www.zhihu.com/']  user\_agent = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"  headers = {  "HOST": "www.zhihu.com",  "Referer": "https://www.zhihu.com",  "User-Agent": user\_agent  }    def parse(self, response):  pass    # start\_requests 是 spider 抓取数据的入口，由于在抓取数据之前需要先登录，  # 所以需要先改写 start\_requests 函数。  # 重写 start\_requests 函数,该函数也是爬虫的入口程序  # Scrapy 是基于 Twisted 框架的，它所有的操作都是异步的。  # 这里需要设置 callback 函数，否则会默认调用 parse 函数。  def start\_requests(self):  return [scrapy.Request('https://www.zhihu.com/#signin', headers = self.headers, callback = self.login)]    def login(self,response):  response\_text = response.text  # 如果不加 re.DOTALL 参数，那么把 re.match 修改为 re.search 也可以实现多行匹配  match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL) # re.DOTALL 表示开启多行匹配  xsrf = ''  if match\_obj:  xsrf = match\_obj.group(1)  print("\_xsrf：",xsrf)  if xsrf:  post\_data = {  "\_xsrf":xsrf,  "phone\_num":"13459282910",  "password":"a5s7sh4u",  "captcha":""  }  import time  t = str(int(time.time() \* 1000))  captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login".format(t)  # 该语句获取图片并使它与 scrapy 的 request 在同一个会话中  yield scrapy.Request(captcha\_url, headers = self.headers, meta = {"post\_data":post\_data}, callback = self.login\_after\_captcha)  def login\_after\_captcha(self, response):  with open("captcha.jpg", "wb") as f:  f.write(response.body) # 注意 这里是 response.body 不是 response.text  f.close()  from PIL import Image  try:  im = Image.open('captcha.jpg')  im.show()  im.close()  except:  print("图片打开失败")  pass  captcha = input("输入验证码：\n")  post\_data = response.meta.get("post\_data", {})  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data["captcha"] = captcha  return [scrapy.FormRequest(  url = post\_url,  formdata = post\_data,  headers = self.headers,  callback = self.check\_login # 不加()表示调用函数对象，加()表示函数调用执行  )]    # 与 requests.login.zhihu.py 不同的是，scrapy 不需要把 cookies 保存起来，  # 因为 scrapy 的所有后续的 request 请求，都会默认把 cookies 带进去  def check\_login(self,response):  #验证服务器返回的数据来验证是否登录成功  text\_json = json.loads(response.text)  print("msg",text\_json["msg"])  if "msg" in text\_json and text\_json["msg"] == "登录成功":  print("登录成功")  # 这里就是开始处理 start\_urls 了。  for url in self.start\_urls:  # 这里没有设置 callback 函数，所以默认调用 parse 函数  # dont\_filter  yield scrapy.Request(url, dont\_filter = True,headers = self.headers) |

到这里，验证码获取和输入功能也就实现了。至于验证码的识别问题，在后面学习中会碰到。

## 定义知乎表结构

我们抓取知乎的问答内容，也就是问题以及该问题的所有回答的内容。

知乎路径：

显示问题和部分回答

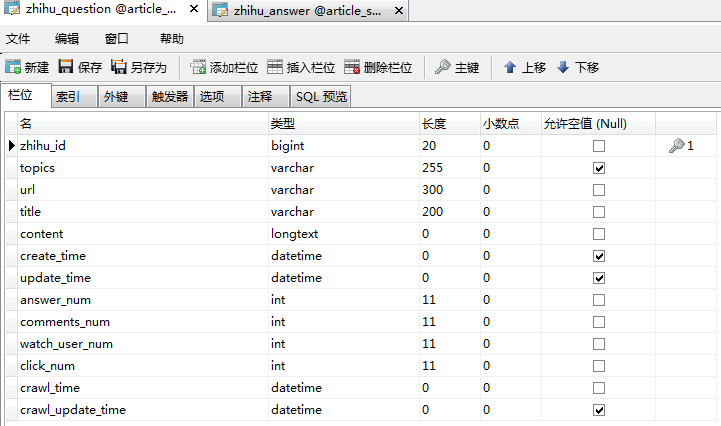
<https://www.zhihu.com/question/61098126/answer/185683826>

显示问题和所有回答

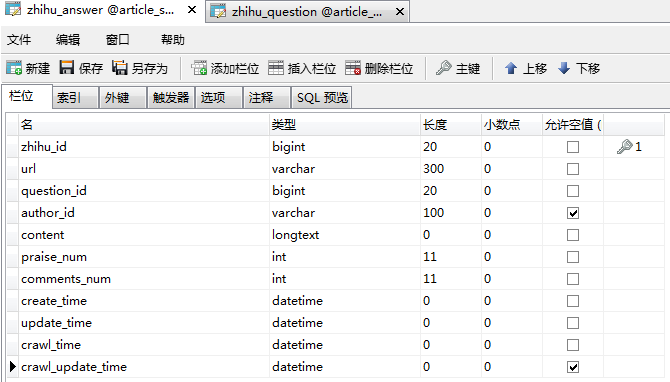
<https://www.zhihu.com/question/61098126>

我们在使用的时候需要把 https://www.zhihu.com/question/61098126/answer/185683826 转换成 https://www.zhihu.com/question/61098126 这样的路径。

这里需要定义两个表，一个用于存放问题，另一个用于存放回答，所以设计两个表：zhihu\_question 和 zhihu\_answer。这里带上两个表结构的截图和SQL代码。



|  |
| --- |
| DROP TABLE IF EXISTS `zhihu\_question`;  CREATE TABLE `zhihu\_question` (  `zhihu\_id` bigint(20) NOT NULL,  `topics` varchar(255) DEFAULT NULL,  `url` varchar(300) NOT NULL,  `title` varchar(200) NOT NULL,  `content` longtext NOT NULL,  `create\_time` datetime DEFAULT NULL,  `update\_time` datetime DEFAULT NULL,  `answer\_num` int(11) NOT NULL DEFAULT '0',  `comments\_num` int(11) NOT NULL DEFAULT '0',  `watch\_user\_num` int(11) NOT NULL DEFAULT '0',  `click\_num` int(11) NOT NULL DEFAULT '0',  `crawl\_time` datetime NOT NULL,  `crawl\_update\_time` datetime DEFAULT NULL,  PRIMARY KEY (`zhihu\_id`)  ) ENGINE=MyISAM DEFAULT CHARSET=utf8; |



|  |
| --- |
| DROP TABLE IF EXISTS `zhihu\_answer`;  CREATE TABLE `zhihu\_answer` (  `zhihu\_id` bigint(20) NOT NULL,  `url` varchar(300) NOT NULL,  `question\_id` bigint(20) NOT NULL,  `author\_id` varchar(100) DEFAULT NULL,  `content` longtext NOT NULL,  `praise\_num` int(11) NOT NULL DEFAULT '0',  `comments\_num` int(11) NOT NULL DEFAULT '0',  `create\_time` datetime NOT NULL,  `update\_time` datetime NOT NULL,  `crawl\_time` datetime NOT NULL,  `crawl\_update\_time` datetime DEFAULT NULL,  PRIMARY KEY (`zhihu\_id`)  ) ENGINE=MyISAM DEFAULT CHARSET=utf8; |

## 编写蜘蛛程序

知乎抓取与 jobbole 抓取的不同点：

1、jobbole 列表页的路径比较容易抓取，知乎不方便，所以知乎抓取按照深度优先的方式来抓取；

2、知乎比 jobbole 多了模拟登陆；

3、jobbole 只有一个表结构，而知乎需要两个表结构；

4、jobbole内容页没有分页，而知乎的回答是有分页的；

这里我们的代码将在[知乎模拟登陆](#_知乎模拟登陆)的 zhihu.py 文件的的基础上进行编写。这里要编写的文件包括 zhihu.py 和 items.py 。

在 zhihu.py 文件里面主要实现了问题和回答的 URL 的抓取以及字段的解析，代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import json  import re  import datetime  try:  import urlparse as parse # python2  except:  from urllib import parse # python3  from scrapy.loader import ItemLoader  from ArticleSpider.items import ZhihuAnswerItem,ZhihuQuestionItem  class ZhihuSpider(scrapy.Spider):  name = "zhihu"  allowed\_domains = ["www.zhihu.com"]  start\_urls = ['http://www.zhihu.com/']  user\_agent = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"  headers = {  "HOST": "www.zhihu.com",  "Referer": "https://www.zhihu.com",  "User-Agent": user\_agent  }  # question 的 answer 的 url 规则  start\_answer\_url = "https://www.zhihu.com/api/v4/questions/{0}/answers?sort\_by=default&include=data%5B%2A%5D.is\_normal%2Cis\_sticky%2Ccollapsed\_by%2Csuggest\_edit%2Ccomment\_count%2Ccollapsed\_counts%2Creviewing\_comments\_count%2Ccan\_comment%2Ccontent%2Ceditable\_content%2Cvoteup\_count%2Creshipment\_settings%2Ccomment\_permission%2Cmark\_infos%2Ccreated\_time%2Cupdated\_time%2Crelationship.is\_author%2Cvoting%2Cis\_thanked%2Cis\_nothelp%2Cupvoted\_followees%3Bdata%5B%2A%5D.author.is\_blocking%2Cis\_blocked%2Cis\_followed%2Cvoteup\_count%2Cmessage\_thread\_token%2Cbadge%5B%3F%28type%3Dbest\_answerer%29%5D.topics&limit={1}&offset={2}"  # 这里采用 深度优先 的方式来抓取 URL 地址  def parse(self, response):  """  提取 html 页面中的所有的 url 并跟踪这些 url 以便进一步爬取  如果提取的 url 满足格式 /question/60581534 则爬取该 url 之后直接进入解析函数 parse\_datail  """  all\_urls = response.css("a::attr(href)").extract()  # 对 all\_urls 进行遍历操作，把所有的相对路径修改为绝对路径  all\_urls = [parse.urljoin(response.url,url) for url in all\_urls]  # 过滤掉所有不以 https 开头的 url ，比如 href="javascript:" 这样的链接  all\_urls = filter(lambda url:True if url.startswith("https") else False,all\_urls)  for url in all\_urls:  #print(url)  match\_obj = re.match("(.\*zhihu.com/question/(\d+))($|/answer).\*",url)  if match\_obj:  request\_url = match\_obj.group(1)  # question\_id = match\_obj.group(2)  yield scrapy.Request(request\_url,headers = self.headers,callback = self.parse\_question)  else:  #如果不是 question 页面则直接进一步跟踪  yield scrapy.Request(url, headers = self.headers, callback = self.parse)  def parse\_question(self,response):  # 处理 question 页面，从页面中提取出具体的 question 的 item 字段  # 代码写到这里我们就到 items.py 把 question 和 answer 这两个 item 都定义好  # 知乎新版问答页的标题的用的是 h1 标签，class 为 QuestionHeader-title，而这个 class 的值在老版本是没有的  # 所以可以通过搜索 url 返回来的文本是否包含 QuestionHeader-title 来判断是新版本还是老版本  if "QuestionHeader-title" in response.text:  # 处理新版本  # 从 url 里面匹配出 question 的 id 值 即 parse 函数中的 question\_id = match\_obj.group(2)  # 当然也可以像 jobbole.py 中的 meta={"front\_image\_url": image\_url} 中 meta 方式传递 question 的 id 值  match\_obj = re.match("(.\*zhihu.com/question/(\d+))(/answer|$).\*", response.url)  if match\_obj:  question\_id = int(match\_obj.group(2)) # 转换成 int 类型  item\_loader = ItemLoader(item = ZhihuQuestionItem(), response = response)  item\_loader.add\_css("title", "h1.QuestionHeader-title::text")  item\_loader.add\_css("content", ".QuestionHeader-detail")  item\_loader.add\_value("url", response.url)  item\_loader.add\_value("zhihu\_id", question\_id)  item\_loader.add\_css("answer\_num", ".List-headerText span::text")  item\_loader.add\_css("comments\_num", ".QuestionHeader-actions button::text")  # 关注数 和 浏览数 是相同的提取规则，所以后面要通过数组操作把它们分别提取出来  item\_loader.add\_css("watch\_user\_num", ".NumberBoard-value::text")  item\_loader.add\_css("topics", ".QuestionHeader-topics .Popover div::text")  question\_item = item\_loader.load\_item()  else:  # 处理老版本  match\_obj = re.match("(.\*zhihu.com/question/(\d+))(/answer|$).\*", response.url)  if match\_obj:  question\_id = int(match\_obj.group(2))  item\_loader = ItemLoader(item = ZhihuQuestionItem(), response = response)  # item\_loader.add\_css("title", ".zh-question-title h2 a::text")  # 由于 css 选择器没有 | 这个或选择器，所以使用 xpath 选择器来提取  item\_loader.add\_xpath("title", "//\*[@id='zh-question-title']/h2/a/text()|//\*[@id='zh-question-title']/h2/span/text()")  item\_loader.add\_css("content", "#zh-question-detail")  item\_loader.add\_value("url", response.url)  item\_loader.add\_value("zhihu\_id", question\_id)  item\_loader.add\_css("answer\_num", "#zh-question-answer-num::text")  item\_loader.add\_css("comments\_num", "#zh-question-meta-wrap a[name='addcomment']::text")  # item\_loader.add\_css("watch\_user\_num", "#zh-question-side-header-wrap::text")  item\_loader.add\_xpath("watch\_user\_num", "//\*[@id='zh-question-side-header-wrap']/text()|//\*[@class='zh-question-followers-sidebar']/div/a/strong/text()")  item\_loader.add\_css("topics", ".zm-tag-editor-labels a::text")  question\_item = item\_loader.load\_item()  # format(question\_id, 20, 0) question\_id 对应的 answer 从 0 开始取，每次取 20 条  # 要注意与 start\_answer\_url 中的参数顺序一一对应  yield scrapy.Request(self.start\_answer\_url.format(question\_id, 20, 0), headers = self.headers, callback = self.parse\_answer)  yield question\_item # 到这里就把 question\_item 交给 pipeline 处理了  """  # 在 parse\_question 也可以加入以下代码来对 question 页面的 url 进行提取  # 这里没加上是想让代码的逻辑更清晰  all\_urls = response.css("a::attr(href)").extract()  all\_urls = [parse.urljoin(response.url,url) for url in all\_urls]  all\_urls = filter(lambda url:True if url.startswith("https") else False,all\_urls)  for url in all\_urls:  match\_obj = re.match("(.\*zhihu.com/question/(\d+))($|/answer).\*",url)  if match\_obj:  request\_url = match\_obj.group(1)  yield scrapy.Request(request\_url,headers = self.headers,callback = self.parse\_question)  else:  yield scrapy.Request(url, headers = self.headers, callback = self.parse)  """  def parse\_answer(self,response):  # 处理 question 的 answer  answer\_json = json.loads(response.text)  is\_end = answer\_json["paging"]["is\_end"]  next\_url = answer\_json["paging"]["next"]  # 提取 answer 的具体字段  for answer in answer\_json["data"]:  answer\_item = ZhihuAnswerItem()  answer\_item["zhihu\_id"] = answer["id"]  answer\_item["url"] = answer["url"]  answer\_item["question\_id"] = answer["question"]["id"]  # answer["author"]["id"] if "id" in answer["author"] else None 这句话的意思是：  # 如果在 answer["author"] 中包含 "id" 则返回 answer["author"]["id"] 否则返回 None  # 因为有些是匿名评论的，匿名评论没有 "id" 值  answer\_item["author\_id"] = answer["author"]["id"] if "id" in answer["author"] else None  answer\_item["content"] = answer["content"] if "content" in answer else None  answer\_item["parise\_num"] = answer["voteup\_count"]  answer\_item["comments\_num"] = answer["comment\_count"]  answer\_item["create\_time"] = answer["created\_time"]  answer\_item["update\_time"] = answer["updated\_time"]  answer\_item["crawl\_time"] = datetime.datetime.now()  yield answer\_item # 到这里就把 answer\_item 交给 pipeline 处理了  if not is\_end:  yield scrapy.Request(next\_url, headers = self.headers, callback = self.parse\_answer)  # 在 Scrapy 中 start\_requests 函数是整个爬虫的入口，也就是说爬虫程序是从 start\_requests 开始执行的。  # 这里的 start\_requests 是对 Scrapy 中的 start\_requests 进行了重写  def start\_requests(self):  return [scrapy.Request('https://www.zhihu.com/#signin', headers = self.headers, callback = self.login)]  def login(self,response):  response\_text = response.text  match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL)  xsrf = ''  if match\_obj:  xsrf = match\_obj.group(1)  print("\_xsrf：",xsrf)  if xsrf:  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data = {  "\_xsrf":xsrf,  "phone\_num":"13459282910",  "password":"a5s7sh4u"  }  return [scrapy.FormRequest(  url = post\_url,  formdata = post\_data,  headers = self.headers,  callback = self.check\_login  )]  def check\_login(self,response):  text\_json = json.loads(response.text)  print("msg",text\_json["msg"])  if "msg" in text\_json and text\_json["msg"] == "登录成功":  for url in self.start\_urls:  yield scrapy.Request(url, dont\_filter = True,headers = self.headers) |

在 items.py 中增加 ZhihuQuestionItem 和 ZhihuAnswerItem 这两个 item 代码如下：

|  |
| --- |
| # 知乎 问题对应的 item  class ZhihuQuestionItem(scrapy.Item):  zhihu\_id = scrapy.Field()  topics = scrapy.Field() # 主题  url = scrapy.Field()  title = scrapy.Field()  content = scrapy.Field()  answer\_num = scrapy.Field() # 回答数  comments\_num = scrapy.Field() # 评论数  watch\_user\_num = scrapy.Field() # 关注数  click\_num = scrapy.Field() # 浏览数  crawl\_time = scrapy.Field()  # 知乎 问题的回答对应的 item  class ZhihuAnswerItem(scrapy.Item):  zhihu\_id = scrapy.Field()  url = scrapy.Field()  question\_id = scrapy.Field()  author\_id = scrapy.Field()  content = scrapy.Field()  praise\_num = scrapy.Field()  comments\_num = scrapy.Field()  create\_time = scrapy.Field()  update\_time = scrapy.Field()  crawl\_time = scrapy.Field() |

## 知乎数据保存

本节没有对 zhihu.py 进行大的修改，这里仍然把 zhihu.py 的代码粘贴上，因为可能有增加一些注释，或者一些细节代码：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import json  import re  import datetime  try:  import urlparse as parse # python2  except:  from urllib import parse # python3  from scrapy.loader import ItemLoader  from ArticleSpider.items import ZhihuAnswerItem,ZhihuQuestionItem  class ZhihuSpider(scrapy.Spider):  name = "zhihu"  allowed\_domains = ["www.zhihu.com"]  start\_urls = ['http://www.zhihu.com/']  user\_agent = "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"  headers = {  "HOST": "www.zhihu.com",  "Referer": "https://www.zhihu.com",  "User-Agent": user\_agent  }  # question 的 answer 的 url 规则  start\_answer\_url = "https://www.zhihu.com/api/v4/questions/{0}/answers?sort\_by=default&include=data%5B%2A%5D.is\_normal%2Cis\_sticky%2Ccollapsed\_by%2Csuggest\_edit%2Ccomment\_count%2Ccollapsed\_counts%2Creviewing\_comments\_count%2Ccan\_comment%2Ccontent%2Ceditable\_content%2Cvoteup\_count%2Creshipment\_settings%2Ccomment\_permission%2Cmark\_infos%2Ccreated\_time%2Cupdated\_time%2Crelationship.is\_author%2Cvoting%2Cis\_thanked%2Cis\_nothelp%2Cupvoted\_followees%3Bdata%5B%2A%5D.author.is\_blocking%2Cis\_blocked%2Cis\_followed%2Cvoteup\_count%2Cmessage\_thread\_token%2Cbadge%5B%3F%28type%3Dbest\_answerer%29%5D.topics&limit={1}&offset={2}"  # 这里采用 深度优先 的方式来抓取 URL 地址  def parse(self, response):  """  提取 html 页面中的所有的 url 并跟踪这些 url 以便进一步爬取  如果提取的 url 满足格式 /question/60581534 则爬取该 url 之后直接进入解析函数 parse\_datail  """  all\_urls = response.css("a::attr(href)").extract()  # 对 all\_urls 进行遍历操作，把所有的相对路径修改为绝对路径  all\_urls = [parse.urljoin(response.url,url) for url in all\_urls]  # 过滤掉所有不以 https 开头的 url ，比如 href="javascript:" 这样的链接  all\_urls = filter(lambda url:True if url.startswith("https") else False,all\_urls)  for url in all\_urls:  #print(url)  match\_obj = re.match("(.\*zhihu.com/question/(\d+))($|/answer).\*",url)  if match\_obj:  request\_url = match\_obj.group(1)  # question\_id = match\_obj.group(2)  yield scrapy.Request(request\_url,headers = self.headers,callback = self.parse\_question)  # break # 代码调试的时候可使用  else:  #如果不是 question 页面则直接进一步跟踪  yield scrapy.Request(url, headers = self.headers, callback = self.parse)  # pass # 代码调试的时候可使用  def parse\_question(self,response):  # 处理 question 页面，从页面中提取出具体的 question 的 item 字段  # 代码写到这里我们就到 items.py 把 question 和 answer 这两个 item 都定义好  # 知乎新版问答页的标题的用的是 h1 标签，class 为 QuestionHeader-title，而这个 class 的值在老版本是没有的  # 所以可以通过搜索 url 返回来的文本是否包含 QuestionHeader-title 来判断是新版本还是老版本  if "QuestionHeader-title" in response.text:  # 处理新版本  # 从 url 里面匹配出 question 的 id 值 即 parse 函数中的 question\_id = match\_obj.group(2)  # 当然也可以像 jobbole.py 中的 meta={"front\_image\_url": image\_url} 中 meta 方式传递 question 的 id 值  match\_obj = re.match("(.\*zhihu.com/question/(\d+))(/answer|$).\*", response.url)  if match\_obj:  question\_id = int(match\_obj.group(2)) # 转换成 int 类型  item\_loader = ItemLoader(item = ZhihuQuestionItem(), response = response)  item\_loader.add\_css("title", "h1.QuestionHeader-title::text")  item\_loader.add\_css("content", ".QuestionHeader-detail")  item\_loader.add\_value("url", response.url)  item\_loader.add\_value("zhihu\_id", question\_id)  item\_loader.add\_css("answer\_num", ".List-headerText span::text")  item\_loader.add\_css("comments\_num", ".QuestionHeader-actions button::text")  # 关注数 和 浏览数 是相同的提取规则，所以后面要通过数组操作把它们分别提取出来  item\_loader.add\_css("watch\_user\_num", ".NumberBoard-value::text")  item\_loader.add\_css("topics", ".QuestionHeader-topics .Popover div::text")  question\_item = item\_loader.load\_item()  else:  # 处理老版本  match\_obj = re.match("(.\*zhihu.com/question/(\d+))(/answer|$).\*", response.url)  if match\_obj:  question\_id = int(match\_obj.group(2))  item\_loader = ItemLoader(item = ZhihuQuestionItem(), response = response)  # item\_loader.add\_css("title", ".zh-question-title h2 a::text")  # 由于 css 选择器没有 | 这个或选择器，所以使用 xpath 选择器来提取  item\_loader.add\_xpath("title", "//\*[@id='zh-question-title']/h2/a/text()|//\*[@id='zh-question-title']/h2/span/text()")  item\_loader.add\_css("content", "#zh-question-detail")  item\_loader.add\_value("url", response.url)  item\_loader.add\_value("zhihu\_id", question\_id)  item\_loader.add\_css("answer\_num", "#zh-question-answer-num::text")  item\_loader.add\_css("comments\_num", "#zh-question-meta-wrap a[name='addcomment']::text")  # item\_loader.add\_css("watch\_user\_num", "#zh-question-side-header-wrap::text")  item\_loader.add\_xpath("watch\_user\_num", "//\*[@id='zh-question-side-header-wrap']/text()|//\*[@class='zh-question-followers-sidebar']/div/a/strong/text()")  item\_loader.add\_css("topics", ".zm-tag-editor-labels a::text")  question\_item = item\_loader.load\_item()  # format(question\_id, 20, 0) question\_id 对应的 answer 从 0 开始取，每次取 20 条  # 要注意与 start\_answer\_url 中的参数顺序一一对应  yield scrapy.Request(self.start\_answer\_url.format(question\_id, 20, 0), headers = self.headers, callback = self.parse\_answer)  yield question\_item # 到这里就把 question\_item 交给 pipeline 处理了  # 代码调试不同的 item 的时候可以把没调试的 yield item 注释掉  """  # 在 parse\_question 也可以加入以下代码来对 question 页面的 url 进行提取  # 这里没加上是想让代码的逻辑更清晰  all\_urls = response.css("a::attr(href)").extract()  all\_urls = [parse.urljoin(response.url,url) for url in all\_urls]  all\_urls = filter(lambda url:True if url.startswith("https") else False,all\_urls)  for url in all\_urls:  match\_obj = re.match("(.\*zhihu.com/question/(\d+))($|/answer).\*",url)  if match\_obj:  request\_url = match\_obj.group(1)  yield scrapy.Request(request\_url,headers = self.headers,callback = self.parse\_question)  else:  yield scrapy.Request(url, headers = self.headers, callback = self.parse)  """  def parse\_answer(self,response):  # 处理 question 的 answer  answer\_json = json.loads(response.text)  is\_end = answer\_json["paging"]["is\_end"]  next\_url = answer\_json["paging"]["next"]  # 提取 answer 的具体字段  for answer in answer\_json["data"]:  answer\_item = ZhihuAnswerItem()  answer\_item["zhihu\_id"] = answer["id"]  answer\_item["url"] = answer["url"]  answer\_item["question\_id"] = answer["question"]["id"]  # answer["author"]["id"] if "id" in answer["author"] else None 这句话的意思是：  # 如果在 answer["author"] 中包含 "id" 则返回 answer["author"]["id"] 否则返回 None  # 因为有些是匿名评论的，匿名评论没有 "id" 值  answer\_item["author\_id"] = answer["author"]["id"] if "id" in answer["author"] else None  answer\_item["content"] = answer["content"] if "content" in answer else None  answer\_item["parise\_num"] = answer["voteup\_count"]  answer\_item["comments\_num"] = answer["comment\_count"]  answer\_item["create\_time"] = answer["created\_time"]  answer\_item["update\_time"] = answer["updated\_time"]  answer\_item["crawl\_time"] = datetime.datetime.now()  yield answer\_item # 到这里就把 answer\_item 交给 pipeline 处理了  if not is\_end:  yield scrapy.Request(next\_url, headers = self.headers, callback = self.parse\_answer)  # 在 Scrapy 中 start\_requests 函数是整个爬虫的入口，也就是说爬虫程序是从 start\_requests 开始执行的。  # 这里的 start\_requests 是对 Scrapy 中的 start\_requests 进行了重写  def start\_requests(self):  return [scrapy.Request('https://www.zhihu.com/#signin', headers = self.headers, callback = self.login)]  def login(self,response):  response\_text = response.text  match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL)  xsrf = ''  if match\_obj:  xsrf = match\_obj.group(1)  print("\_xsrf：",xsrf)  if xsrf:  post\_url = "https://www.zhihu.com/login/phone\_num"  post\_data = {  "\_xsrf":xsrf,  "phone\_num":"13459282910",  "password":"a5s7sh4u"  }  return [scrapy.FormRequest(  url = post\_url,  formdata = post\_data,  headers = self.headers,  callback = self.check\_login  )]  def check\_login(self,response):  text\_json = json.loads(response.text)  print("msg",text\_json["msg"])  if "msg" in text\_json and text\_json["msg"] == "登录成功":  for url in self.start\_urls:  yield scrapy.Request(url, dont\_filter = True,headers = self.headers) |

**代码调试的时候**：

1、只需要一条 url 的，则可以加入 break 来方便调试

2、else 的部分再不需要执行的时候 可以加一个 pass

3、不调试的 item ，可以把 yield item 注释掉

详细见 zhihu.py 的代码中的 红色 部分注释。

然后，我们对 items.py 中的相关 item 进行修改，为什么呢？

首先相同数据库的 pipeline 达到复用的目的，那么必须把 sql 语句和参数 转移到 item 进行处理；另外，我们在 zhihu.py 抓取的字段也需要放在 item 中进行处理。

知乎的 items.py 中的代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  import datetime  from ArticleSpider.utils.common import extract\_num  from ArticleSpider.settings import SQL\_DATETIME\_FORMAT, SQL\_DATE\_FORMAT  class ArticlespiderItem(scrapy.Item):  pass  # 知乎 问题对应的 item  class ZhihuQuestionItem(scrapy.Item):  zhihu\_id = scrapy.Field()  topics = scrapy.Field() # 主题  url = scrapy.Field()  title = scrapy.Field()  content = scrapy.Field()  answer\_num = scrapy.Field() # 回答数  comments\_num = scrapy.Field() # 评论数  watch\_user\_num = scrapy.Field() # 关注数  click\_num = scrapy.Field() # 浏览数  crawl\_time = scrapy.Field()  def get\_insert\_sql(self):  # 插入知乎 question 表的 sql 语句  insert\_sql = """  insert into zhihu\_question(zhihu\_id, topics, url, title, content, answer\_num, comments\_num,  watch\_user\_num, click\_num, crawl\_time)  VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)  ON DUPLICATE KEY UPDATE  content = VALUES(content),  answer\_num = VALUES(answer\_num),  comments\_num = VALUES(comments\_num),  watch\_user\_num = VALUES(watch\_user\_num),  click\_num = VALUES(click\_num)  """  zhihu\_id = self["zhihu\_id"][0] # 由于在 zhihu.py 中已经转换成 int 了，所以这里直接取 list 的第一个元素  # 或者 zhihu\_id = int("".join(self["zhihu\_id"]))  topics = ",".join(self["topics"]) # 把 list 转换成 字符串  url = self["url"][0]  title = "".join(self["title"])  content = "".join(self["content"])  # 在 common.py 中定义一个 extract\_num 函数  answer\_num = extract\_num("".join(self["answer\_num"]))  comments\_num = extract\_num("".join(self["comments\_num"]))  # 我们之前在 zhihu.py 中提取 watch\_user\_num 和 click\_num 的时候  # 新版中 watch\_user\_num 和 click\_num 的提取规则是一样的，且提取之后是保存在 watch\_user\_num 中的  # 旧版中只有 watch\_user\_num 没有 click\_num，所以只提取 watch\_user\_num 且保存在 watch\_user\_num 中  # 所以这里要做判断，判断 watch\_user\_num 的长度是 1 还是 2  if len(self["watch\_user\_num"]) == 2:  watch\_user\_num = int(self["watch\_user\_num"][0])  click\_num = int(self["watch\_user\_num"][1])  else:  watch\_user\_num = int(self["watch\_user\_num"][0])  click\_num = 0  # 在 settings.py 中定义 SQL\_DATETIME\_FORMAT 和 SQL\_DATE\_FORMAT 方便日期以及日期时间的格式化  # crawl\_time 直接获取当前系统时间  crawl\_time = datetime.datetime.now().strftime(SQL\_DATETIME\_FORMAT)  # 把字段的值按照 insert\_sql 语句中的顺序，封装成一个 tuple  params = (zhihu\_id, topics, url, title, content, answer\_num, comments\_num,  watch\_user\_num, click\_num, crawl\_time)  return insert\_sql, params  # 知乎 问题的回答对应的 item  class ZhihuAnswerItem(scrapy.Item):  zhihu\_id = scrapy.Field()  url = scrapy.Field()  question\_id = scrapy.Field()  author\_id = scrapy.Field()  content = scrapy.Field()  praise\_num = scrapy.Field()  comments\_num = scrapy.Field()  create\_time = scrapy.Field()  update\_time = scrapy.Field()  crawl\_time = scrapy.Field()  def get\_insert\_sql(self):  # 插入知乎 question 表的sql语句  # ON DUPLICATE KEY UPDATE 是 MySQL 数据库特有的  # ON 后面的作用是，如果没有数据则执行 insert，有数据则进行部分字段的 update 操作  # ON DUPLICATE KEY 意思是主键冲突  # 意思就是说当 ON DUPLICATE KEY 主键冲突时，就执行 UPDATE 语句，对 UPDATE 后面的字段进行更新操作  # 那对于 知乎 来说，content,parise\_num,comments\_num,update\_time 这几个字段可能是经常更新的  # 所以把这几个字段放在 UPDATE 后面  insert\_sql = """  insert into zhihu\_answer(zhihu\_id, url, question\_id, author\_id, content, parise\_num, comments\_num,  create\_time, update\_time, crawl\_time)  VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)  ON DUPLICATE KEY UPDATE  content = VALUES(content),  comments\_num = VALUES(comments\_num),  parise\_num = VALUES(parise\_num),  update\_time = VALUES(update\_time)  """  # create\_time 和 update\_time 提取出来的时候是一个时间戳，我们这里要把它按照 SQL\_DATETIME\_FORMAT 进行格式化  # 把 create\_time 和 update\_time 从 int 类型转换到 datetime 类型，再转换成 字符串类型  create\_time = datetime.datetime.fromtimestamp(self["create\_time"]).strftime(SQL\_DATETIME\_FORMAT)  update\_time = datetime.datetime.fromtimestamp(self["update\_time"]).strftime(SQL\_DATETIME\_FORMAT)  params = (  self["zhihu\_id"], self["url"], self["question\_id"],  self["author\_id"], self["content"], self["parise\_num"],  self["comments\_num"], create\_time, update\_time,  self["crawl\_time"].strftime(SQL\_DATETIME\_FORMAT)  )  return insert\_sql, params |

在知乎的 item 中，我们用到了 extract\_num 函数和 SQL\_DATETIME\_FORMAT, SQL\_DATE\_FORMAT 这两个常量，所以我们需要在 common.py 文件中增加 extract\_num 函数，代码如下：

|  |
| --- |
| # \_\*\_ coding:utf-8 \_\*\_  import re  #从字符串中提取出数字，或者说把字符串中的数字提取出来并返回  def extract\_num(text):  match\_re = re.match(".\*?(\d+).\*", text)  if match\_re:  nums = int(match\_re.group(1))  else:  nums = 0  return nums |

注意：为了达到 相同数据库 的 pipeline 的复用，在 item 中定义的 sql 语句的方法名要统一，比如这里统一为 get\_insert\_sql 函数。

然后在 settings.py 中定义 SQL\_DATETIME\_FORMAT, SQL\_DATE\_FORMAT 这两个常量，代码如下：

|  |
| --- |
| SQL\_DATETIME\_FORMAT = "%Y-%m-%d %H:%M:%S" # 即 xxxx-xx-xx xx:xx:xx  SQL\_DATE\_FORMAT = "%Y-%m-%d" # 即 xxxx-xx-xx |

保存数据的 pipelines.py 的代码修改后如下：

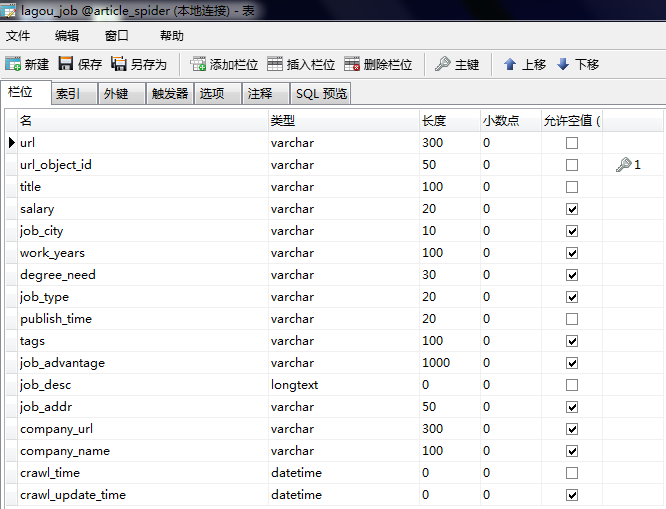
|  |
| --- |
| # -\*- coding: utf-8 -\*-  import MySQLdb  import MySQLdb.cursors  from twisted.enterprise import adbapi  class ArticlespiderPipeline(object):  def process\_item(self, item, spider):  return item  # pipeline 应该是在不同的数据库的时候才定义不同的 pipeline  # 相同的数据库则使用相同的 pipeline，因此数据的插入操作代码可以转移到 item 去编写  class MysqlTwistedPipeline(object):  def \_\_init\_\_(self,dbpool):  self.dbpool = dbpool  @classmethod  def from\_settings(cls,settings):  dbparms = dict(  host = settings['MYSQL\_HOST'],  db = settings['MYSQL\_DBNAME'],  user = settings['MYSQL\_USER'],  passwd = settings['MYSQL\_PASSWORD'],  charset = 'utf8',  cursorclass = MySQLdb.cursors.DictCursor,  use\_unicode = True,  )  dbpool = adbapi.ConnectionPool("MySQLdb",\*\*dbparms)  return cls(dbpool)  def process\_item(self, item, spider):  query = self.dbpool.runInteraction(self.do\_insert,item)  query.addErrback(self.handle\_error,item,spider)  def handle\_error(self,failure,item,spider):  print(failure)  def do\_insert(self,cursor,item):  # 每个 item 都定义一个 get\_insert\_sql 函数，由该函数返回 sql 语句和插入的字段值  # 这样就可以达到同一个数据库的 item 的 pipeline 的复用  insert\_sql, params = item.get\_insert\_sql()  cursor.execute(insert\_sql, params) |

遗留的问题：同一个 User-Agent 快速抓取知乎内容的时候，知乎会返回 403 所以在后面要加上 User-Agent 的随机更换操作。

# 爬取 [www.lagou.com](http://www.lagou.com)

## 拉勾网数据表结构

我们先设计**拉勾网**的数据库表结构：



|  |
| --- |
| DROP TABLE IF EXISTS `lagou\_job`;  CREATE TABLE `lagou\_job` (  `url` varchar(300) NOT NULL,  `url\_object\_id` varchar(50) NOT NULL,  `title` varchar(100) NOT NULL,  `salary` varchar(20) DEFAULT NULL COMMENT 'salary 有最小值和最大值 且应该是 int 类型,这里暂时不这么处理',  `job\_city` varchar(10) DEFAULT NULL,  `work\_years` varchar(100) DEFAULT NULL COMMENT 'work\_years 有最小值和最大值 且应该是 int 类型,这里暂时不这么处理',  `degree\_need` varchar(30) DEFAULT NULL,  `job\_type` varchar(20) DEFAULT NULL,  `publish\_time` varchar(20) NOT NULL,  `tags` varchar(100) DEFAULT NULL,  `job\_advantage` varchar(1000) DEFAULT NULL,  `job\_desc` longtext NOT NULL,  `job\_addr` varchar(50) DEFAULT NULL,  `company\_url` varchar(300) DEFAULT NULL,  `company\_name` varchar(100) DEFAULT NULL,  `crawl\_time` datetime NOT NULL,  `crawl\_update\_time` datetime DEFAULT NULL,  PRIMARY KEY (`url\_object\_id`)  ) ENGINE=MyISAM DEFAULT CHARSET=utf8; |

## CrawlSpider 整站爬取

在上节我们已经把 拉勾网 的数据表结构设计好了。

**附带知识**：

在 import 我们自己定义的库文件的时候，可以在 settings.py 中加入如下代码：

|  |
| --- |
| import sys  BASE\_DIR = os.path.dirname(os.path.abspath(os.path.dirname(\_\_file\_\_)))  # 把 ArticleSpider 路径加入到 PYTHONPATH 中，这样在 import的时候 就不需要从 ArticleSpider 开始了  sys.path.insert(0, os.path.join(BASE\_DIR, 'ArticleSpider')) |

这样在 import 的时候就不用从 ArticleSpider 开始了，而可以从它的下一级目录，比如 utils 开始。当然，我们不在 settings.py 中加入那些代码，然后在 import 的时候，从 ArticleSpider 开始也是没问题的。

这里，我自己使用的时候，就没有在 settings.py 中加入那些代码了。

使用 Scrapy 提供的全站爬取类 CrawlSpider 实现对 拉勾网 www.lagou.com 的整站爬取实现。

在使用 CrawlSpider 之前，我们先回忆一下之前创建蜘蛛的命令：

|  |
| --- |
| scrapy genspider zhihu www.zhihu.com |

这里创建蜘蛛默认使用 scrapy genspider 的 basic 模板来创建的，实际上 genspider 后面可以带参数：

|  |
| --- |
| D:\workspace\ArticleSpider>scrapy genspider --list  Available templates:  basic  crawl  csvfeed  xmlfeed |

通过参数 --list 可以查看 genspider 给我们提供了多少种模板，而 CrawlSpider 实际上是这些模板当中的一种，也就是 crawl 模板。

因此，创建 CrawlSpider 蜘蛛可以通过如下命令行创建：

|  |
| --- |
| scrapy genspider -t crawl lagou www.lagou.com |

运行结果如下：

|  |
| --- |
| D:\workspace\ArticleSpider>scrapy genspider -t crawl lagou www.lagou.com  Created spider 'lagou' using template 'crawl' in module:  ArticleSpider.spiders.lagou |

创建之后，lagou.py 的代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import datetime  from scrapy.linkextractors import LinkExtractor  from scrapy.spiders import CrawlSpider, Rule  from ArticleSpider.items import LagouJobItemLoader, LagouJobItem  from ArticleSpider.utils.common import get\_md5  # 这里不再是继承 scrapy.Spider 基类  # CrawlSpider 其实也是继承了 scrapy.Spider 对 scrapy.Spider 进行了包装  # 因为 CrawlSpider 继承自 scrapy.Spider 所以程序入口仍然是 start\_requests  # CrawlSpider 的回调函数 parse 在 CrawlSpider 的子类 LagouSpider 不能重载  # 为什么不能重载 parse 呢？  # 首先要知道 \_parse\_response 是 CrawlSpider 的核心函数  # 原因是 parse 的函数已经实现了代码且用到了 CrawlSpider 的核心函数 \_parse\_response 重载的话会出错  # 在采集之前的预处理可以重载 CrawlSpider 的 parse\_start\_url 函数  # 在这里除了写出 lagou 的爬虫之外，最重要的是自己去分析 CrawlSpider 的源码  class LagouSpider(CrawlSpider):  name = 'lagou'  allowed\_domains = ['www.lagou.com']  start\_urls = ['https://www.lagou.com/']  rules = (  # 回调函数不是通过 self 来调用的，而是写在 字符串 里面  Rule(LinkExtractor(allow = ("zhaopin/.\*",)), follow = True),  Rule(LinkExtractor(allow = ("gongsi/j\d+.html",)), follow=True),  # follow= True 表示对 allow=r'jobs/\d+.html' 抓取的页面里面的 url 地址进行爬取  Rule(LinkExtractor(allow = r'jobs/\d+.html'), callback='parse\_job', follow= True),  )  # def parse\_start\_url(self, response):  # print("进入 parse\_start\_url 函数")  # return []  #  # def process\_results(self, response, results):  # print("进入 process\_results 函数")  # return results  # 解析拉勾网的职位  def parse\_job(self, response):  item\_loader = LagouJobItemLoader(item= LagouJobItem(), response = response)  item\_loader.add\_css("title", ".job-name::attr(title)")  item\_loader.add\_value("url", response.url)  item\_loader.add\_value("url\_object\_id", get\_md5(response.url))  item\_loader.add\_css("salary", ".job\_request .salary::text")  item\_loader.add\_xpath("job\_city", "//\*[@class='job\_request']/p/span[2]/text()")  item\_loader.add\_xpath("work\_years", "//\*[@class='job\_request']/p/span[3]/text()")  item\_loader.add\_xpath("degree\_need", "//\*[@class='job\_request']/p/span[4]/text()")  item\_loader.add\_xpath("job\_type", "//\*[@class='job\_request']/p/span[5]/text()")  item\_loader.add\_css("tags", '.position-label li::text')  item\_loader.add\_css("publish\_time", ".publish\_time::text")  item\_loader.add\_css("job\_advantage", ".job-advantage p::text")  item\_loader.add\_css("job\_desc", ".job\_bt div")  item\_loader.add\_css("job\_addr", ".work\_addr")  item\_loader.add\_css("company\_name", "#job\_company dt a img::attr(alt)")  item\_loader.add\_css("company\_url", "#job\_company dt a::attr(href)")  item\_loader.add\_value("crawl\_time", datetime.now())  job\_item = item\_loader.load\_item()  return job\_item |

在这里用到了 get\_md5 函数，该函数定义在 common.py 这个文件，这里再次把代码附上：

|  |
| --- |
| def get\_md5(url):  if isinstance(url, str):  url = url.encode("utf-8")  m = hashlib.md5()  m.update(url)  return m.hexdigest() |

拉钩网的 item 代码如下：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import scrapy  from w3lib.html import remove\_tags # 用来过滤掉 html 标签  from scrapy.loader import ItemLoader  from scrapy.loader.processors import MapCompose, TakeFirst, Join  from ArticleSpider.settings import SQL\_DATETIME\_FORMAT  class ArticlespiderItem(scrapy.Item):  pass  def remove\_splash(value):  #去掉工作城市的斜线  return value.replace("/","")  def handle\_jobaddr(value):  addr\_list = value.split("\n")  addr\_list = [item.strip() for item in addr\_list if item.strip()!="查看地图"]  return "".join(addr\_list)  class LagouJobItemLoader(ItemLoader):  #自定义itemloader  default\_output\_processor = TakeFirst()  class LagouJobItem(scrapy.Item):  #拉勾网职位信息  title = scrapy.Field()  url = scrapy.Field()  url\_object\_id = scrapy.Field()  salary = scrapy.Field()  job\_city = scrapy.Field(  input\_processor=MapCompose(remove\_splash),  )  work\_years = scrapy.Field(  input\_processor = MapCompose(remove\_splash),  )  degree\_need = scrapy.Field(  input\_processor = MapCompose(remove\_splash),  )  job\_type = scrapy.Field()  publish\_time = scrapy.Field()  job\_advantage = scrapy.Field()  job\_desc = scrapy.Field()  job\_addr = scrapy.Field(  input\_processor = MapCompose(remove\_tags, handle\_jobaddr),  )  company\_name = scrapy.Field()  company\_url = scrapy.Field()  tags = scrapy.Field(  input\_processor = Join(",")  )  crawl\_time = scrapy.Field()  def get\_insert\_sql(self):  insert\_sql = """  insert into lagou\_job(title, url, url\_object\_id, salary, job\_city, work\_years, degree\_need,  job\_type, publish\_time, job\_advantage, job\_desc, job\_addr, company\_name, company\_url,  tags, crawl\_time)  VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)  ON DUPLICATE KEY UPDATE  salary = VALUES(salary),  job\_desc = VALUES(job\_desc)  """  params = (  self["title"], self["url"], self["url\_object\_id"], self["salary"], self["job\_city"],  self["work\_years"], self["degree\_need"], self["job\_type"],  self["publish\_time"], self["job\_advantage"], self["job\_desc"],  self["job\_addr"], self["company\_name"], self["company\_url"],  self["job\_addr"], self["crawl\_time"].strftime(SQL\_DATETIME\_FORMAT),  )  return insert\_sql, params |

这里用到的 SQL\_DATETIME\_FORMAT 之前用过，是在 settings.py 中定义的，这里再次把代码附上：

|  |
| --- |
| SQL\_DATETIME\_FORMAT = "%Y-%m-%d %H:%M:%S" # 即 xxxx-xx-xx xx:xx:xx |

pipeline 仍然是采用异步处理方式，代码如下：跟之前的代码一样

|  |
| --- |
| # pipeline 应该是在不同的数据库的时候才定义不同的 pipeline  # 相同的数据库则使用相同的 pipeline，因此数据的插入操作代码可以转移到 item 去编写  class MysqlTwistedPipeline(object):  def \_\_init\_\_(self,dbpool):  self.dbpool = dbpool  @classmethod  def from\_settings(cls,settings):  dbparms = dict(  host = settings['MYSQL\_HOST'],  db = settings['MYSQL\_DBNAME'],  user = settings['MYSQL\_USER'],  passwd = settings['MYSQL\_PASSWORD'],  charset = 'utf8',  cursorclass = MySQLdb.cursors.DictCursor,  use\_unicode = True,  )  dbpool = adbapi.ConnectionPool("MySQLdb",\*\*dbparms)  return cls(dbpool)  def process\_item(self, item, spider):  query = self.dbpool.runInteraction(self.do\_insert,item)  query.addErrback(self.handle\_error,item,spider)  def handle\_error(self,failure,item,spider):  print(failure)  def do\_insert(self,cursor,item):  # 每个 item 都定义一个 get\_insert\_sql 函数，由该函数返回 sql 语句和插入的字段值  # 这样就可以达到同一个数据库的 item 的 pipeline 的复用  insert\_sql, params = item.get\_insert\_sql()  cursor.execute(insert\_sql, params) |

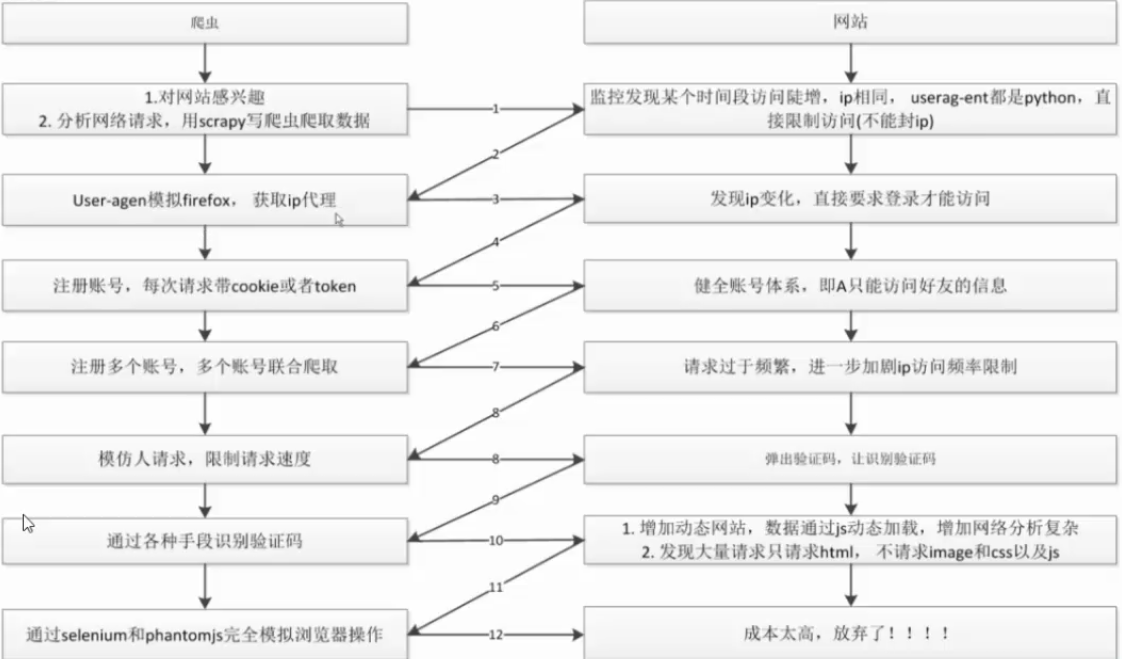
到这里，拉勾网的整站爬取就开发完成了。

# Scrapy反爬虫配置

## 爬虫与反爬虫介绍

在上节中，如果长时间对拉勾网进行数据抓取，它会把抓取的 url 重定向到一个 forbidden 页面，这就需要我们进行反爬虫处理。

爬虫与反爬虫的斗争历史如下示意图：



## 使用中间件随机更换 User-Agent

设置 spider 的 User-Agent 的方式很多，这里我们一步一步做介绍。

方式一：

在 settings.py 中定义一个 User-Agent 的 list，代码如下：

|  |
| --- |
| user\_agent\_list = [  'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36',  'Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:54.0) Gecko/20100101 Firefox/54.0',  'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0)'  ] |

然后在 spider 文件导入 user\_agent\_list 代码如下：

|  |
| --- |
| from ArticleSpider.settings import user\_agent\_list |

然后在 yield scrapy.Request 之前添加如下代码：

|  |
| --- |
| import random  random\_index = random.randint(0,len(user\_agent\_list) - 1)  random\_agent = user\_agent\_list[random\_index]  self.headers["User-Agent"] = random\_agent |

比如在 zhihu.py 中增加上面的代码，如下：

|  |
| --- |
| import random  random\_index = random.randint(0, len(user\_agent\_list) - 1)  random\_agent = user\_agent\_list[random\_index]  self.headers["User-Agent"] = random\_agent  yield scrapy.Request(request\_url,headers = self.headers,callback = self.parse\_question) |

但是这种做法在每个 yield scrapy.Request 的地方都要写，就会造成代码的累赘; 另外，每个 spider 文件都要写这样的一段代码，耦合太紧。

所以我们要把它修改成松的耦合，同时又不会造成代码的累赘。于是有了下面的另外一种方式。

方式二：

在这里我们采用中间件的方式来处理 User-Agent 的设置。首先我们把 settings.py 配置文件中的 DOWNLOADER\_MIDDLEWARES 这个配置打开，代码如下：

|  |
| --- |
| DOWNLOADER\_MIDDLEWARES = {  # 这个是创建项目时自动配置的 Middleware 我们不需要直接注释掉  # 'ArticleSpider.middlewares.MyCustomDownloaderMiddleware': 543,  # 这个是 scrapy 自带的设置 User-Agent 的中间件,我们要把它设置为 None  'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,  # 自定义 Middleware  'ArticleSpider.middlewares.RandomUserAgentMiddleware': 400,  } |

在上面的代码中 ArticleSpider.middlewares.MyCustomDownloaderMiddleware 是创建项目时就加上的，我们不需要所以直接注释掉，scrapy.downloadermiddlewares.useragent.UserAgentMiddleware 这个是 scrapy 自带的设置 User-Agent 的中间件，我们要把它配置上，且设置为 None 来达到取消使用默认的 UserAgentMiddleware的目的，那我们为什么不使用这个 scrapy 自带的 scrapy.downloadermiddlewares.useragent.UserAgentMiddleware 中间件呢？我们先对 UserAgentMiddleware 介绍一下如何使用，然后再解释为什么不用这个中间件。

**UserAgentMiddleware** 定义在 /site-packages/scrapy/downloadermiddlewares/useragent.py 文件中。在该类中有一个静态方法 from\_crawler(cls, crawler)，通过 UserAgentMiddleware.from\_crawler 可直接调用。另外，在 UserAgentMiddleware 中的 def process\_request(self, request, spider) 是用来处理每一个 request 的，这个函数是一个固定的写法。我们自己在写 Middleware 的时候，必须带上 def process\_request(self, request, spider) 然后重写它的函数体内容。

通过阅读 useragent.py 这个文件我们可以了解到， 在静态方法 from\_crawler 中的 crawler.settings['USER\_AGENT'] 这个语句会从 settings.py 中取得 USER\_AGENT 的值，否则会使用默认的 User-Agent 会设置为 Scrapy ，这个默认值的设置定义在 \_\_init\_\_ 方法中。

所以我们需要在 settings.py 中设置 USER\_AGENT 这个变量的值，比如把 USER\_AGENT 设置为如下：

|  |
| --- |
| USER\_AGENT = 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0)' |

到这里，我们解释为什么不用这个默认的中间件，因为 UserAgentMiddleware 使用 USER\_AGENT 并不是随机的，且每次修改完之后，都要重启 spider 程序，所以我们不使用这个默认的中间件。所以我们需要自己来定义一个中间件来处理随机更换 User-Agent 的目的。

方式三：

我们在 middlewares.py 中定义一个类 RandomUserAgentMiddleware 代码如下：

|  |
| --- |
| class RandomUserAgentMiddleware(object):  # 随机更换 User-Agent  def \_\_init\_\_(self,crawler):  super(RandomUserAgentMiddleware,self).\_\_init\_\_()  self.user\_agent\_list = crawler.settings.get("user\_agent\_list")  @classmethod  def from\_crawler(cls, crawler):  return cls(crawler)  def process\_request(self, request, spider):  import random  random\_index = random.randint(0, len(self.user\_agent\_list) - 1)  random\_agent = self.user\_agent\_list[random\_index]  request.headers.setdefault('User-Agent',random\_agent) |

然后像**方式一**一样需要在 settings.py 中配置 user\_agent\_list 代码，代码与**方式一**中的配置一样。这样就达到了代码的松耦合且不会有代码的累赘。但是这种做法还是不太方便，因为 user\_agent\_list 是定义在 settings.py 文件当中的，这样每次修改 user\_agent\_list 的值，就需要重新加载 settings.py 文件，也就是说需要重启蜘蛛程序，这样很麻烦。

方式四：

在这里我们将使用 scrapy 的一个库 fake-useragent 来实现随机更换 User-Agent 同时又不需要依赖 settings.py 文件。

fake-useragent 项目地址 https://github.com/hellysmile/fake-useragent

fake-useragent 的 User-Agent 维护地址: https://fake-useragent.herokuapp.com/browsers/0.1.5 通过 pip list 查看 fake-useragent 的版本号，然后把上面的地址修改为对应的版本号即可。

我们可以通过 pip install fake-useragent 进行安装，或者用 pip install -i https://pypi.doubanio.com/simple/ fake-useragent 豆瓣源来安装更快。安装过程如下：

|  |
| --- |
| C:\Users\Administrator>pip install -i https://pypi.doubanio.com/simple/ fake-useragent  Collecting fake-useragent  Downloading https://pypi.doubanio.com/packages/76/0a/54b883e9ef1193aab88ca39e672d3d9758ce108be1f89bf110dff4803aac/fake-useragent-0.1.7.tar.gz  Building wheels for collected packages: fake-useragent  Running setup.py bdist\_wheel for fake-useragent ... done  Stored in directory: C:\Users\Administrator\AppData\Local\pip\Cache\wheels\bf\16\39\f55216c95061a86449637bf03cde28589fc88297874527410a  Successfully built fake-useragent  Installing collected packages: fake-useragent  Successfully installed fake-useragent-0.1.7 |

使用 fake-useragent 来随机更换 User-Agent 的代码如下：

|  |
| --- |
| from fake\_useragent import UserAgent  class RandomUserAgentMiddleware(object):  # 随机更换 User-Agent  def \_\_init\_\_(self,crawler):  super(RandomUserAgentMiddleware,self).\_\_init\_\_()  self.ua = UserAgent()  @classmethod  def from\_crawler(cls, crawler):  return cls(crawler)  def process\_request(self, request, spider):  request.headers.setdefault('User-Agent',self.ua.random) |

这个代码中的 self.ua.random 就是随机调用 User-Agent 了。**我们以后使用主要用这个方式**。

**额外说明**：

通过阅读 https://github.com/hellysmile/fake-useragent 这个文档，我们知道通过 ua.ie、ua.opera、ua.chrome、ua.google、ua.firefox、ua.safari、ua.random 可以实现各种浏览器的不同 User-Agent 的随机调用，那么如何对 ua 的 ie、opera、chrome、google、firefox、safari、random 这些属性达到更灵活的配置呢？

很简单，在 settings.py 中加入如下配置当中的一个：

|  |
| --- |
| RANDOM\_UA\_TYPE = "random"  # 或者  RANDOM\_UA\_TYPE = "ie"  # 或者  RANDOM\_UA\_TYPE = "opera"  # 或者  RANDOM\_UA\_TYPE = "chrome"  # 或者  RANDOM\_UA\_TYPE = "firefox" |

然后把 RandomUserAgentMiddleware 代码修改成如下：

|  |
| --- |
| from fake\_useragent import UserAgent  class RandomUserAgentMiddleware(object):  # 随机更换 User-Agent  def \_\_init\_\_(self,crawler):  super(RandomUserAgentMiddleware,self).\_\_init\_\_()  self.ua = UserAgent()  # RANDOM\_UA\_TYPE 使用 settings.py 中配置的 RANDOM\_UA\_TYPE 变量的值  # "random" 是设置的默认值  self.ua\_type = crawler.settings.get("RANDOM\_UA\_TYPE","random")  @classmethod  def from\_crawler(cls, crawler):  return cls(crawler)  def process\_request(self, request, spider):  def get\_ua():  return getattr(self.ua,self.ua\_type)  request.headers.setdefault('User-Agent',get\_ua()) |

只是，这种做法相对上面的方式四的做法来说，依赖了 settings.py 配置文件，所以我在这里就不使用这种方式了。

**中间件的一点说明**：

中间件的处理是双向的，重载了 process\_request 就会处理 request，重载了 process\_response 就会处理 response。

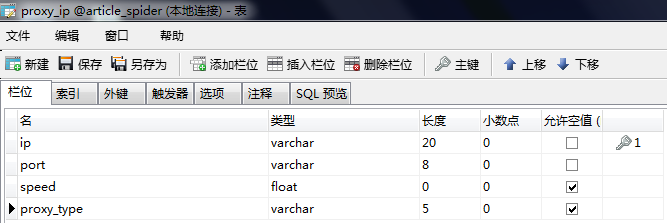
中间件文档:

<https://doc.scrapy.org/en/latest/topics/downloader-middleware.html>

## IP 切换与代理

在开发 IP代理中间件之前，我们要先从第三方网站采集代理IP，并保存到我们的数据库，然后中间件再通过代码的调用获取到随机的代理IP。

这里，我们将从 <http://www.xicidaili.com/> 这个网站去采集代理IP，首先设计数据库，如下图所示：



对应的SQL语句代码：

|  |
| --- |
| DROP TABLE IF EXISTS `proxy\_ip`;  CREATE TABLE `proxy\_ip` (  `ip` varchar(20) NOT NULL,  `port` varchar(8) NOT NULL,  `speed` float DEFAULT NULL,  `proxy\_type` varchar(5) DEFAULT NULL,  PRIMARY KEY (`ip`)  ) ENGINE=MyISAM DEFAULT CHARSET=utf8; |

数据库设计完成之后，我们就编写代码来采集代理IP的数据，我们在 ArticleSpider 创建目录 tools 在该目录下创建 crawl\_xici\_ip.py，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  # 在这里我们将爬取 http://www.xicidaili.com/nn/ 里面的 高匿代理IP 并保存到数据库中  # 这里直接采用 requests 模块来编写代码  import requests  from scrapy.selector import Selector  import MySQLdb  conn = MySQLdb.connect(host="127.0.0.1", user="root", passwd="root", db="article\_spider", charset="utf8")  cursor = conn.cursor()  def crawl\_ips():  #爬取西刺的免费高匿ip代理  headers = {"User-Agent":"Mozilla/5.0 (Windows NT 6.1; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0"}  for i in range(100):  re = requests.get("http://www.xicidaili.com/nn/{0}".format(i), headers = headers)  selector = Selector(text = re.text)  all\_trs = selector.css("#ip\_list tr")  ip\_list = []  for tr in all\_trs[1:]:  speed\_str = tr.css(".bar::attr(title)").extract()[0]  if speed\_str:  speed = float(speed\_str.split("秒")[0]) # 速度  all\_texts = tr.css("td::text").extract()  ip = all\_texts[0] # ip  port = all\_texts[1] # 端口  proxy\_type = all\_texts[5] # 类型  ip\_list.append((ip, port, proxy\_type, speed))  for ip\_info in ip\_list:  cursor.execute(  "insert proxy\_ip(ip, port, speed, proxy\_type) VALUES('{0}', '{1}', {2}, 'http')".format(  ip\_info[0], ip\_info[1], ip\_info[3]  )  )  conn.commit()  # 数据库操作类  class GetIP(object):  def delete\_ip(self, ip):  #从数据库中删除无效的ip  delete\_sql = "delete from proxy\_ip where ip='{0}'".format(ip)  cursor.execute(delete\_sql)  conn.commit()  return True  def judge\_ip(self, ip, port):  #判断ip是否可用  http\_url = "https://www.baidu.com/"  proxy\_url = "http://{0}:{1}".format(ip, port)  try:  proxy\_dict = {  "http":proxy\_url,  }  # 设置代理，并使用代理来访问百度  response = requests.get(http\_url, proxies = proxy\_dict)  except Exception as e:  print ("无效的 ip 和 port")  self.delete\_ip(ip)  return False  else:  code = response.status\_code # 状态码  if code >= 200 and code < 300:  print ("有效的 ip 和 port")  return True  else:  print ("无效的 ip 和 port")  result = self.delete\_ip(ip)  if result:  print("无效的 IP 已经删除")  else:  print("无效的 IP 删除失败")  return False  def get\_random\_ip(self):  #从数据库中随机获取一个可用的ip  random\_sql = "SELECT ip, port FROM proxy\_ip ORDER BY RAND() LIMIT 1"  #result = cursor.execute(random\_sql)  cursor.execute(random\_sql)  for ip\_info in cursor.fetchall():  ip = ip\_info[0]  port = ip\_info[1]  judge\_re = self.judge\_ip(ip, port) # 判断 IP 是否可用  if judge\_re:  return "http://{0}:{1}".format(ip, port)  else:  return self.get\_random\_ip()  if \_\_name\_\_ == "\_\_main\_\_":  crawl\_ips()  get\_ip = GetIP()  get\_ip.get\_random\_ip() |

然后我们在 middlewares.py 添加 RandomProxyMiddleware 这个类，代码如下：

|  |
| --- |
| from ArticleSpider.tools.crawl\_xici\_ip import GetIP  # 添加代理 IP 格式为 http(s)://IP:端口  # 所以我们需要一个代理IP池，这里我们将采集 http://www.xicidaili.com/nn/ 里面的代理IP，  # 包括 IP地址，端口，类型  class RandomProxyMiddleware(object):  #动态设置ip代理  def process\_request(self, request, spider):  get\_ip = GetIP()  request.meta["proxy"] = get\_ip.get\_random\_ip() |

这样就可以了。记得把我们的 RandomProxyMiddleware 配置到 settings.py 文件的 DOWNLOADER\_MIDDLEWARES 中去。

建议使用自己的IP地址，这样采集也比较快，然后要做好采集速度的控制，防止本机IP被封。

下面我们介绍一个处理代理IP 的库文件，地址 https://github.com/aivarsk/scrapy-proxies，这个不好的地方就是它是从文件里面去读取代理IP的，如果文件要更新和无效IP的删除就比较麻烦，不过可以查看它的源代码结合我们自己的代码改造成我们的更好的代码出来。

另外一个不错的处理代理IP的库，地址 https://github.com/scrapy-plugins/scrapy-crawlera 很多东西都帮我们处理好了，使用起来也非常方便，需要申请 apikey，且是收费的。

tor 洋葱浏览器，可以做多次转发，隐藏我们的IP，不过使用 洋葱浏览器 需要 VPN。

到这里，动态的设置 代理IP 就学习完毕了。

## 验证码的识别

验证码的识别有三种方式：

编码实现（tesseract-ocr：谷歌提供的工具，识别率非常低，不适用）

在线打码

人工打码

这里我们演示 在线代码 的代码实现，在 ArticleSpider/tools 目录下新建 yundama.requests.py 文件，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  import json  import requests  '''  使用在线打码的方式实现验证码的识别功能，在这里使用 http://www.yundama.com/  它的开发文档地址为 http://www.yundama.com/apidoc/  这里由于我们是基于 python 开发，所以使用 PythonHTTP示例下载 来查看开发代码，下面的代码是在示例代码基础上修改而来  在测试的时候，需要两个账号，开发者账号和普通用户账号，  代码中的 appid 和 appkey 是属于开发者账号的  代码中的 用户名 和 密码 是属于普通用户账号的  '''  class YDMHttp(object):  apiurl = 'http://api.yundama.com/api.php'  username = ''  password = ''  appid = ''  appkey = ''  def \_\_init\_\_(self, username, password, appid, appkey):  self.username = username  self.password = password  self.appid = str(appid)  self.appkey = appkey  def balance(self):  data = {'method': 'balance', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey}  response\_data = requests.post(self.apiurl, data=data)  ret\_data = json.loads(response\_data.text)  if ret\_data["ret"] == 0:  print ("获取剩余积分", ret\_data["balance"])  return ret\_data["balance"]  else:  return None  def login(self):  data = {'method': 'login', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey}  response\_data = requests.post(self.apiurl, data=data)  ret\_data = json.loads(response\_data.text)  if ret\_data["ret"] == 0:  print ("登录成功", ret\_data["uid"])  return ret\_data["uid"]  else:  return None  def decode(self, filename, codetype, timeout):  data = {'method': 'upload', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey, 'codetype': str(codetype), 'timeout': str(timeout)}  files = {'file': open(filename, 'rb')}  response\_data = requests.post(self.apiurl, files=files, data=data)  ret\_data = json.loads(response\_data.text)  if ret\_data["ret"] == 0:  print ("识别成功", ret\_data["text"])  return ret\_data["text"]  else:  return None  def ydm(file\_path):  username = 'da\_ge\_da1' # 普通用户的账号  password = 'da\_ge\_da' # 普通用户的密码  # 软件ＩＤ，开发者分成必要参数。登录开发者后台【我的软件】获得！  appid = 3129  # 软件密钥，开发者分成必要参数。登录开发者后台【我的软件】获得！  appkey = '40d5ad41c047179fc797631e3b9c3025'  # 图片文件  filename = 'image/captcha.jpg'  # 验证码类型，# 例：1004表示4位字母数字，不同类型收费不同。请准确填写，否则影响识别率。  # 在此查询所有类型 http://www.yundama.com/price.html  codetype = 5000  # 超时时间，秒  timeout = 60  # 检查  yundama = YDMHttp(username, password, appid, appkey)  if (username == 'username'):  print('请设置好相关参数再测试')  else:  # 开始识别，图片路径，验证码类型ID，超时时间（秒），识别结果  return yundama.decode(file\_path, codetype, timeout);  if \_\_name\_\_ == "\_\_main\_\_":  # 用户名  username = 'da\_ge\_da1'  # 密码  password = 'da\_ge\_da'  # 软件ＩＤ，开发者分成必要参数。登录开发者后台【我的软件】获得！  appid = 3129  # 软件密钥，开发者分成必要参数。登录开发者后台【我的软件】获得！  appkey = '40d5ad41c047179fc797631e3b9c3025'  # 图片文件  filename = 'image/captcha.jpg'  # 验证码类型，# 例：1004表示4位字母数字，不同类型收费不同。请准确填写，否则影响识别率。在此查询所有类型 http://www.yundama.com/price.html  codetype = 5000  # 超时时间，秒  timeout = 60  # 检查  if (username == 'username'):  print ('请设置好相关参数再测试')  else:  # 初始化  yundama = YDMHttp(username, password, appid, appkey)  # 登陆云打码  uid = yundama.login();  print('uid: %s' % uid)  # 登陆云打码  uid = yundama.login();  print ('uid: %s' % uid)  # 查询余额  balance = yundama.balance();  print ('balance: %s' % balance)  # 开始识别，图片路径，验证码类型ID，超时时间（秒），识别结果  text = yundama.decode(filename, codetype, timeout); |

## 其他反爬虫配置

### cookies 的禁用

在 settings.py 文件中可以配置 cookies 的禁用，代码如下：

|  |
| --- |
| COOKIES\_ENABLED = False |

False 表示禁用 cookies 。因为有些网站会根据 cookies 的值来判断是否为爬虫，所以我们在这里禁用 cookies 网站就无法通过 cookies 来做判断了，特别是那些不需要登陆就可以爬取内容的网站。禁用 cookies 之后，request 就不会把我们的 cookies 带进去了。

### 采集的限速问题

Scrapy 提供的限速文档地址：

<http://scrapy-chs.readthedocs.io/zh_CN/latest/topics/autothrottle.html>

下面是控制AutoThrottle扩展的设置：

|  |
| --- |
| AUTOTHROTTLE\_ENABLED  AUTOTHROTTLE\_START\_DELAY  AUTOTHROTTLE\_MAX\_DELAY  AUTOTHROTTLE\_DEBUG  CONCURRENT\_REQUESTS\_PER\_DOMAIN  CONCURRENT\_REQUESTS\_PER\_IP  DOWNLOAD\_DELAY |

这些扩展设置都可以在 settings.py 文件中设置。

### 个性化setting配置

需要登陆才能采集的网站是不能禁用 cookies 的，所以对于需要登陆的 spider 来说，就需要进行个性化的 setting 的配置，比如我们之前的 知乎 采集，就需要配置自己的个性化的 setting 配置。

比如知乎的采集，就需要把 COOKIES\_ENABLED 设置为 True。通过 custom\_settings 这个变量可以实现个性化的 setting 配置，注意 custom\_settings 是类属性，不是对象属性，所以它和 allowed\_domains 一样，放在类里面，而不是方法里面。配置代码如下：

|  |
| --- |
| custom\_settings = {  "COOKIES\_ENABLED":True  } |

通过这种个性化的配置，它会把 settings.py 中**相同配置项**的值进行覆盖，而其他没有个性化配置的配置项仍然是使用 settings.py 中的配置。

## 模拟浏览器

### Selenium框架

#### Selenium 抓取动态网站

我们先查看淘宝商品，比如说： <https://detail.tmall.com/item.htm?id=540616520279&sku_properties=4065700:30197460> 这个页面，我们打开这个页面的时候可以看到它是有商品价格的 “¥ 1.00-265.00” 但是我们查看该页面的源码的时候，发现里面只有价格，没有具体的价格区间的数字，因此，我们用以前的方式来抓取的话是永远也抓不到具体的价格的。

这个时候我们的 Selenium 这个框架就派上用场了，它类似于浏览器一样，可以获取整个页面的执行结果（包括 js、css 的执行结果）然后再进一步执行内容提取。我们可以查看一下 Selenium 的百度百科词条：<http://baike.baidu.com/item/Selenium/18266> 来进一步了解它的含义。

我们在 ArticleSpider 的 tools 目录新建 selenium.spider.py 文件。然后通过 pip install selenium 来安装 selenium 库文件。我们这里使用豆瓣源来安装速度更快，代码如下：

|  |
| --- |
| pip install -i https://pypi.doubanio.com/simple/ selenium |

安装完成之后，我们在 selenium.spider.py 文件进行导入，selenium 如何使用呢？我们在百度搜索 selenium python api 可以找到 selenium 的文档，具体地址为： <http://selenium-python.readthedocs.io/api.html> 。然后导入，代码如下：

|  |
| --- |
| from selenium import webdriver |

selenium 导入之后还不够，我们还需要安装我们要使用的浏览器的 driver，这个 driver 可以在上面的文档中找到安装地址，我们这里安装的是 chrome 的 driver。

我们打开 chrome driver 的下载地址 <https://sites.google.com/a/chromium.org/chromedriver/downloads> 找到 Latest Release: ChromeDriver 2.30 安装最新版本，我们把下载下来的压缩包进行解压，把解压缩后的路径记录下来。比如我这里是 D:\workspace\chromedriver\_win32\chromedriver.exe 。

到这里 selenium 的准备工作就算完成了，接下来我们来编写 selenium.spider.py 文件代码，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  from scrapy.selector import Selector  browser = webdriver.Chrome(executable\_path = "D:/workspace/chromedriver\_win32/chromedriver.exe")  # 打开浏览器并获取页面  browser.get("https://detail.tmall.com/item.htm?id=540616520279&sku\_properties=4065700:30197460")  # 这个获取的是浏览器的执行结果后的代码，在这里我们就可以看到价格的数值了  page\_source = browser.page\_source  #print(page\_source)  # selenium 也提供了获取页面元素操作的函数，但是它使用 纯python 编写，速度比较慢，  # 所以我们提取页面仍然使用 scrapy 的 selector ，  # 但是页面的模拟点击操作仍然是需要使用 selenium 的函数的。  t\_selector = Selector(text = page\_source)  price = t\_selector.css("dd span.tm-price::text").extract()  print(price)  browser.quit() # 关闭浏览器 |

#### Selenium 模拟登陆 和 鼠标滚动

下面我们使用 selenium 来实现知乎的模拟登陆，我们在上节的 selenium.spider.py 文件上进行修改，模拟登陆知乎的代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  browser = webdriver.Chrome(executable\_path = "D:/workspace/chromedriver\_win32/chromedriver.exe")  browser.get("https://www.zhihu.com/#signin")  # 获取用户名输入框并输入值  browser.find\_element\_by\_css\_selector(".view-signin input[name='account']").send\_keys("13459282910")  # 获取密码输入框并输入值  browser.find\_element\_by\_css\_selector(".view-signin input[name='password']").send\_keys("a5s7sh4u")  # 获取登陆按钮并单击登陆  browser.find\_element\_by\_css\_selector(".view-signin button.sign-button.submit").click()  #browser.quit() # 关闭浏览器 |

模拟登陆微博的代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  import time  browser = webdriver.Chrome(executable\_path = "D:/workspace/chromedriver\_win32/chromedriver.exe")  browser.get("http://weibo.com/login.php")  # 这里加入暂停是因为页面加载比较慢的时候  # 页面还没加载完下面代码就开始去获取元素,就会失败  time.sleep(15) # 暂停 15 秒  # 获取用户名输入框并输入值  browser.find\_element\_by\_css\_selector("input#loginname").send\_keys("13459282910")  # 获取密码输入框并输入值  browser.find\_element\_by\_css\_selector("div.info\_list.password input[node-type='password']").send\_keys("a5s7sh4u$")  time.sleep(25) # 暂停 20 秒  # 获取登陆按钮并单击登陆  browser.find\_element\_by\_css\_selector("div.info\_list.login\_btn a.W\_btn\_a.btn\_32px").click()  #browser.quit() # 关闭浏览器 |

到这里模拟登陆代码就完成了。

接下来，我们来完成鼠标下滑的操作，这里以**开源中国**的博客作为例子，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  import time  browser = webdriver.Chrome(executable\_path = "D:/workspace/chromedriver\_win32/chromedriver.exe")  browser.get("https://www.oschina.net/blog")  time.sleep(10) # 暂停 10 秒  for i in range(3):  time.sleep(3)  # 鼠标滚动操作  browser.execute\_script("""  window.scrollTo(0,document.body.scrollHeight);  var lenOfPage=document.body.scrollHeight;  return lenOfPage;  """)  time.sleep(3)  time.sleep(10)  browser.quit() # 关闭浏览器 |

#### Selenium 设置不加载图片

selenium 在加载网页的时候，默认是加载图片的，加载图片需要消耗带宽，同时导致页面打开速度比较慢，所以我们这里学习一下，如何设置不加载图片。

设置不加载图片是通过对浏览器的driver设置来完成的，我们这里使用的是 chrome 浏览器，所以要对 chrome 的 driver 进行设置。

我们对上节的 selenium.spider.py 文件的代码进行修改，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  import time  # 设置 chrome 的 driver 不加载图片  chrome\_opt = webdriver.ChromeOptions()  prefs = {  "profile.managed\_default\_content\_settings.images":2 # 设置为 2 表示不加载图片  }  chrome\_opt.add\_experimental\_option("prefs",prefs)  browser = webdriver.Chrome(  executable\_path = "D:/workspace/chromedriver\_win32/chromedriver.exe",  chrome\_options = chrome\_opt,  )  browser.get("https://www.taobao.com/")  time.sleep(10)  browser.quit() |

这样就不会加载图片了。

#### Selenium 集成到 scrapy

我们知道 selenium 是模拟浏览器的，所以，我们把 selenium 集成到 scrapy 的时候，我们应该先考虑是不是所有的 spider 都要用 selenium，但是事实上，并不是所有的 spider 都需要使用 selenium 。

所以，我们接着要考虑的应该是如何把 selenium 集成到 具体的 spider 中去。

我们采用中间件的写法，在 middlewares.py 添加如下代码：

|  |
| --- |
| from selenium import webdriver  from scrapy.http import HtmlResponse  class JSPageMiddleware(object):  #通过chrome请求动态网页  def process\_request(self, request, spider):  if spider.name == "jobbole":  # browser = webdriver.Chrome(executable\_path="D:/Temp/chromedriver.exe")  spider.browser.get(request.url)  import time  time.sleep(3)  print ("访问:{0}".format(request.url))  return HtmlResponse(url=spider.browser.current\_url, body=spider.browser.page\_source, encoding="utf-8", request=request) |

并把中间件写到 settings.py 的 DOWNLOADER\_MIDDLEWARES 配置中去。

然后在 jobbole.py 中 类 增加 \_\_init\_\_ 方法，代码如下：

|  |
| --- |
| from selenium import webdriver  from scrapy.xlib.pydispatch import dispatcher  from scrapy import signals  def \_\_init\_\_(self):  self.browser = webdriver.Chrome(executable\_path="D:/Temp/chromedriver.exe")  super(JobboleSpider, self).\_\_init\_\_()  dispatcher.connect(self.spider\_closed, signals.spider\_closed)  def spider\_closed(self, spider):  #当爬虫退出的时候关闭chrome  print ("spider closed")  self.browser.quit() |

在这里我们加入了 spider 的 signal 信号 控制问题，这里用到的是，当 spider 关闭时，也把 chrome 浏览器也关闭。

但是注意，scrapy 是基于 twisted 异步请求的，在这里加入模拟浏览器，它就变成了同步了，降低了 scrapy 的性能，这个如果要变成异步话操作就需要修改 downloader 的代码，如何修改呢？参考 <https://github.com/flisky/scrapy-phantomjs-downloader> 这个源码。

### PhantomJS 使用

在 selenium 中，除了可以使用 chrome 、firefox 等浏览器之外，我们也可以使用无界面浏览器，比如 phantomjs。phantomjs 是一种无界面的浏览器，它的官网为 <http://phantomjs.org/> 。它比 firefox 和 chrome 这些浏览器更加轻量。但是该浏览器的一个很大的问题是：在多进程情况下，phantomjs 的性能下降会很严重。

所以我们这里只是演示如何使用 phantomjs 不过多学习它，因为后面我们也不使用 phantomjs 来模拟登陆。

在使用 phantomjs 之前，我们也是需要下载它的 driver 文件，打开官网下载页面 http://phantomjs.org/download.html 下载它的压缩包，并解压缩，这里我们解压缩到 D:\workspace\ 目录下，并把它的 exe 文件地址的路径记录下来，这里是：D:\workspace\phantomjs-2.1.1-windows\bin\phantomjs.exe。

现在我们开始修改之前的 selenium.spider.py 文件，使用 phantomjs 来操作浏览器，代码如下：

|  |
| --- |
| #!/usr/bin/env python  # \_\*\_ coding:utf-8 \_\*\_  from selenium import webdriver  import time  browser = webdriver.PhantomJS(executable\_path = "D:/workspace/phantomjs-2.1.1-windows/bin/phantomjs.exe")  browser.get("https://detail.tmall.com/item.htm?id=540616520279&sku\_properties=4065700:30197460")  print(browser.page\_source)  time.sleep(10)  # 因为是无界面的，我们无法手动关闭它，所以这里要加上 quit 函数  browser.quit() |

其实，phantomjs 也可以设置不加载图片的，另外，chrome 的 driver 也可以设置为无界面方式。以后有学到再做记录。

### 获取动态网页的其他方法

1、 chrome 无界面运行

我们之前使用 chrome 的时候模拟浏览器默认是打开浏览器界面的，那么如何实现 chrome 无界面运行呢？

安装插件 pip install pyvirtualdisplay 。

这里只写代码演示，因为它在 windows 中好像不行，只能运行在 linux 系统中，演示代码如下：

|  |
| --- |
| from pyvirtualdisplay import Display  # visible 设置为 0 表示 不显示界面  display = Display(visible = 0, size = (800, 600))  display.start()  browser = webdriver.Chrome(executable\_path = "D:/Temp/chromedriver.exe")  browser.get(request.url) |

这里只是使用的大概步骤。在 linux 中使用的时候，可能会碰到 出现 xvfb 错误的问题，需要安装 xvfb。

|  |
| --- |
| sudo apt-get install xvfb |

然后再执行下面代码：

|  |
| --- |
| pip install -i https://pypi.doubanio.com/simple/ xvfbwrapper |

就可以了。

2、 scrapy-splash

3、 selenium-grid

4、 splinter

后面这三种有空可以去研究。

# Scrapy框架介绍

## Scrapy框架图

Scrapy 文档地址：https://doc.scrapy.org/en/latest/

Scrapy 的架构图如下：

**老版本的架构图**：



**新版本的架构图**：



我们在架构图中所看到的所有的组件都在 **/site-packages/scrapy/core** 文件夹中。

### scrapy engine 对应 engine.py 文件

在这个文件中比较重要的两个函数是：

**def schedule(self, request, spider)**

在这个函数里面调用 scheduler 的 def enqueue\_request(self, request) 对请求进行调度处理。

**def \_next\_request\_from\_scheduler(self, spider)**

在这个函数里面调用 scheduler 的 def next\_request(self) 获取下一条请求。

所以，scrapy engine 并不是把 spider 发过来的请求直接交给 downloader 进行处理，而是发给 schedule 进行处理，通过 schedule 处理后返回的 request 再交给 download 进行处理。

### scheduler 对应 scheduler.py 文件

downloader （downloader 属于 middleware） 对应 downloader 文件夹：

在 downloader 里面有一个 handles 文件夹，说明 downloader 可以进行很多方式的下载处理，比如 file、ftp、http（包括 https） 、s3等等。

middleware 中间件：

middleware 包括 spider middleware 和 downloader middleware 这两种中间件。

综上可知，request 是由 spider 产生的，而 response 由 downloader 产生的。

所以我们在编写 spider 文件的时候，在函数的参数里面直接写 response 就可以接收到 downloader 返回来的 response 了。

## Request和Response

本节学习 Scrapy 重要的两个类 Request 和 Response。

**Request** 位于 **/site-packages/scrapy/http/request/\_\_init\_\_.py** 文件

**Response** 位于 **/site-packages/scrapy/http/response/\_\_init\_\_.py** 文件

### Request分析

初始化方法定义如下：

|  |
| --- |
| 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) |
| 参数分析 |
| priority 是一个 int 值，它相当于设置 request 处理的优先级，值越大，优先级越高，越早处理。 |
| dont\_filter 表示是否要被 scheduler 过滤，设置为 True 表示不过滤，设置为 False 表示过滤。 |
| errback 设置 request 出错时的回调函数。 |

**为什么 scrapy 的 request 会自动把 cookies 带上**？

首先我们打开cookies.py文件，该文件位于**/site-packages/scrapy/ downloadermiddlewares/** 文件夹内。

在cookies.py文件中有一个函数**def process\_request(self, request, spider)**在该函数里面把cookies放进了request当中。

### Response分析

初始化方法定义如下：

|  |
| --- |
| def \_\_init\_\_(self, url, status=200, headers=None, body=b'', flags=None, request=None) |
| 参数分析 |
| body 它是返回的 response 的整个内容，它必须是一个字符串，不能是一个 unicode。 |
| flags 包含一些初始化的变量和值。 |

Response子类：

|  |  |
| --- | --- |
| TextResponse | 直接继承 Response |
| HtmlResponse | 继承自TextResponse |
| XmlResponse | 继承自TextResponse |

所以这里只对 TextResponse 做个分析，在 TextResponse 中有两个方法：

|  |
| --- |
| xpath(self, query, \*\*kwargs) |
| css(self, query) |

## Scrapy 暂停 和 重启

这里通过 cmd 命令行操作，首先进入 项目 目录，比如我这里是 D:\workspace\ArticleSpider 。

然后通过命令：

|  |
| --- |
| scrapy crawl lagou -s JOBDIR=job\_info/001 |

说明：

|  |
| --- |
| lagou 表示 spider 的名称      -s 表示 set 的意思  JOBDIR 表示 spider 的工作目录,用来保存 spider 的工作状态  job\_info 在 ArticleSpider 项目下建立 job\_info 目录  001 ， 每个 spider 以及 spider 每次运行的时候 都不能共用同一个目录，否则会覆盖，所以需要在 job\_info 下建立子目录。 |

在运行上面的代码后，按一次 Ctrl + C 则会开始保存工作状态，按两次则会强制退出。

查看工作目录中的文件：

|  |
| --- |
| requets.seen 文件，保存访问过的 url 地址  request.queue 目录，保存未访问过的 url 地址  p0 文件，表示需要继续爬取的 request，当这里的 url 跑完了， p0 会自动清除  spider.state 文件，保存 spider 的状态信息 |

重启跟上面暂停的命令一样。如果要重新爬，则把上面的 001 修改为 002 ，也就是使用新的目录来保存状态就是了。

## Scrapy 去重原理分析

Scrapy本身自带了去重的中间件的。

去重的中间件位于 site-packages/scrapy/dupefilter.py 和 dupefilters.py 文件中。

dupefilter.py 声明抛弃过时的用法，dupefilter**s**.py 是真正处理去重的文件。

dupefilter**s**.py分析

class RFDupeFilter 去重器，该类里面有一个函数：

def request\_seen 该函数在 scrapy/core/scheduler.py 中的 def enqueue\_request 函数中调用。如果要定义自己的去重器，就要重写 def request\_seen 函数，在 def request\_seen 中调用函数 request\_fingerprint 指纹函数，该函数其实调用的是 scrapy/utils/request.py 文件中的 request\_fingerprint 函数。 在 request\_fingerprint 中，主要用到的是使用 sha1 的 hash 算法，把 url 生成固定长度的 哈希值。在 request\_fingerprint 中的 fp.hexdigest 则是生成摘要。

在 class RFDupeFilter 中，其实 url 的哈希值是保存带 set 中的。可以通过代码调试运行看看过程。

在 dupefilters.py 的 request\_seen 的 fp = self.request\_fingerprint(request) 设置断点进行测试。

## Scrapy 的 telnet 服务

在 windows 中，使用 telnet 需要 开启 Telnet服务器 和 Telnet客户端，这些需要通过 控制面板 去操作。

scrapy 默认是开启 telnet 服务的。在 cmd 中运行 scrapy crawl lagou ，可以观察到 telnet 的 ip 和 端口。

然后打开另外一个 cmd 来链接 ，telnet localhost 6023 就可以打开 telnet 链接了。

telnet 链接之后，可以使用 est() 查看 spider 状态。

位于 scrapy/extensions/telnet.py 文件中。

## Middleware 中间件

spider 的 middleware 中间件是介于 spider 和 engine 之间的。

无论是 spider 的中间件，还是 download 的中间件，都会被 middleware 的 manager 调用的。

我们以 ArticleSpider 项目中的 middlewares.py 中的默认中间件 class ArticlespiderSpiderMiddleware 来说明。

**def from\_crawler** 会被 middleware 的 manager 调用。无论是 spider 的中间件，还是 download 的中间件都可以重载 def from\_crawler 函数。

**def process\_spider\_input** 当 engine 把 response 交给 spider 的时候，那么在这里可以加上处理逻辑。

**def process\_spider\_output** 在 spider 解析完 request 交给 engine 的时候，可以在这里加上处理逻辑。

scrapy/spidermiddlewares 目录下有很多 中间件 的源码可以分析和学习，这里以 depth.py 和 httperror.py为例：

class DepthMiddleware 用来监控 spider 可以爬取的URL的深度。主要查看这两个文件的 上面 三个函数的处理代码逻辑。

## Scrapy 数据收集

数据收集(Stats Collection)文档：

<http://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/stats.html>

在 scrapy 里面的数据收集，其实也就是状态收集。

如何使用数据收集器呢？在 spider 文件中，添加如下代码，来收集 404 页面的 url 地址 和 404页面的条数，代码如下：

|  |
| --- |
| #收集伯乐在线所有404的url以及404页面数  handle\_httpstatus\_list = [404] # handle\_httpstatus\_list 参考 httperror.py 文件  def \_\_init\_\_(self, \*\*kwargs):  self.fail\_urls = []  def parse(self, response):  if response.status == 404:  self.fail\_urls.append(response.url)  self.crawler.stats.inc\_value("failed\_url") |

## Scrapy 信号详解

<http://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/signals.html>

在上节的代码上完善，代码如下：

|  |
| --- |
| #收集伯乐在线所有404的url以及404页面数  handle\_httpstatus\_list = [404] # handle\_httpstatus\_list 参考 httperror.py 文件  def \_\_init\_\_(self, \*\*kwargs):  self.fail\_urls = []  dispatcher.connect(self.handle\_spider\_closed, signals.spider\_closed)  def handle\_spider\_closed(self, spider, reason):  self.crawler.stats.set\_value("failed\_urls", ",".join(self.fail\_urls))  def parse(self, response):  if response.status == 404:  self.fail\_urls.append(response.url)  self.crawler.stats.inc\_value("failed\_url") |

在 PyCharm 中运行 spider 的时候，会在 Console 输出上面的变量的内容。

## Scrapy 扩展开发

<http://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/extensions.html>

我们可以分析 scrapy/extensions/corestats.py 和 memusage.py （定时监控内存）这个两个文件的代码。

# Scrapy-Redis分布式爬虫

## 分布式面临的问题

在使用分布式的过程中，将面临以下两个需要解决的问题：

* request队列的集中管理
* url去重的集中管理

为什么要解决上面的问题呢？我们通过scrapy的架构图可以知道，scrapy 的 request 任务调度和 url 去重是在内存的 set 集合中去处理的，而一台机器是不可能去调用另一台机器的内存的。

所以要使 scrapy 变成分布式，那么就必须解决上面这两个问题，同时由第三方对队列和去重进行管理。

而这些问题就是 redis 本身要解决的问题。

## Redis 内容相关

### Redis介绍

Redis\_百度百科： <http://baike.baidu.com/item/Redis>

Redis官网： <https://redis.io/>

Redis中文官网： <http://www.redis.cn/>

中文教程：

<http://www.runoob.com/redis/redis-tutorial.html>

<http://www.yiibai.com/redis/>

Redis 命令参考： <http://redisdoc.com/>

### Redis 安装

Linux中的安装命令：

|  |
| --- |
| sudo apt-get install redis-server |

使用Redis之前，我们还需要安装 redis 的客户端

windows:

<https://github.com/ServiceStack/redis-windows/tree/master/downloads>

从这里下载压缩版本，解压缩即可使用。这里的压缩包包含了服务端和客户端，而官网下载的 mis （https://github.com/MSOpenTech/redis/releases ） 文件则只包含服务端且安装完成之后会自动加入系统服务，并随系统启动而启动。

这里以 windows 环境为主学习。

### 数据类型

Redis中的数据类型主要包括如下几项：

* 字符串
* 散列/哈希 → dict字典
* 列表 → list
* 无序集合 → set
* 有序集合： 可对集合进行排序

字符串操作：

|  |  |
| --- | --- |
| set course\_name "scrapy-redis" | 设置字符串 course\_name 的值 |
| get course\_name | 获取字符串 course\_name 的值 |
| getrange course\_name start end | getrange course\_name 2 5 获取子串 |
| strlen course\_name | 获取字符串长度 |
| incr/decr course\_name | incr/decr 是 加一 和 减一 操作，变量必须是 int 或者 可以转换为 int 的 字符串 |
| append course\_name ".cn" | 字符串拼接 |

哈希操作：

|  |  |
| --- | --- |
| hset 哈希变量名 键 值 | hset myhash name "zls.com" |
| hget 哈希变量名 键 | hget myhash name |
| hgetall 哈希变量名 | hgetall myhash 返回键值 |
| hexists 哈希变量名 键 | 判断是否存在某个键: 1 表示存在; 0 表示不存在 |
| hdel 哈希变量名 键 | 删除键 |
| hkeys 哈希变量名 | 列出所有键 |
| hvals 哈希变量名 | 列出所有值 |

列表操作：

<http://www.yiibai.com/redis/redis_lists.html>

无序集合操作：

<http://www.yiibai.com/redis/redis_sets.html>

有序集合操作：

<http://www.runoob.com/redis/redis-sorted-sets.html>

## 编写分布式爬虫

scrapy-redis下载地址：

<https://github.com/rolando/scrapy-redis>

安装redis：

|  |
| --- |
| pip install -i https://pypi.doubanio.com/simple/ redis |

然后我们在 workspace 新建一个 scrapy 项目：

|  |
| --- |
| scrapy startproject ScrapyRedisTest |

创建完成之后，然后从 https://github.com/rolando/scrapy-redis 下载 scrapy-redis 的 zip 压缩包，下载下来之后进行解压缩，然后把 src/scrapy\_redis 文件夹复制粘贴到 D:\workspace\ScrapyRedisTest 项目目录下。

创建 spider 文件，与之前不同的是，我们不需要通过之前的方式创建，因为我们不使用 scrapy 提供的模板，在这里直接在 spiders 目录下新建 jobbole.py 文件，然后编写如下代码：

|  |
| --- |
| #!/usr/bin/python  # \_\*\_ coding:utf-8 \_\*\_  from scrapy.http import Request  from urllib import parse  from scrapy\_redis.spiders import RedisSpider  # 继承的是 RedisSpider 而不是 scrapy 提供的 Spider  # 而 RedisSpider 其实继承了两个类 其中一个是 scrapy 提供的 Spider  class JobboleSpider(RedisSpider):  name = 'jobbole'  allowed\_domains = ["blog.jobbole.com"]  redis\_key = 'jobbole:start\_urls'  handle\_httpstatus\_list = [404]  def parse(self, response):  if response.status == 404:  self.fail\_urls.append(response.url)  self.crawler.stats.inc\_value("failed\_url")  post\_nodes = response.css("#archive .floated-thumb .post-thumb a")  for post\_node in post\_nodes:  image\_url = post\_node.css("img::attr(src)").extract\_first("")  post\_url = post\_node.css("::attr(href)").extract\_first("")  yield Request(url=parse.urljoin(response.url, post\_url), meta={"front\_image\_url": image\_url},  callback=self.parse\_detail)  next\_url = response.css(".next.page-numbers::attr(href)").extract\_first("")  if next\_url:  yield Request(url=parse.urljoin(response.url, post\_url), callback=self.parse)  def parse\_detail(self, response):  pass |

然后在 scrapy 的 settings.py 编写如下配置：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  BOT\_NAME = 'ScrapyRedisTest'  SPIDER\_MODULES = ['ScrapyRedisTest.spiders']  NEWSPIDER\_MODULE = 'ScrapyRedisTest.spiders'  ROBOTSTXT\_OBEY = False  SCHEDULER = "scrapy\_redis.scheduler.Scheduler"  DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"  ITEM\_PIPELINES = {  'scrapy\_redis.pipelines.RedisPipeline': 300  } |

然后把之前的 main.py 文件复制过来。

然后设置一些断点。

开始运行 main.py 文件，这个时候 PyCharm 的 Console 开始 监听 scrapy-redis 。

我们在命令行输入：

|  |
| --- |
| redis-cli lpush jobbole:start\_urls http://blog.jobbole.com/all-posts/ |

这个时候 Console 又有新的输出了，在命令行输入 keys \* 可以查看所有变量，type jobbole:requests 查看数据类型。

使用说明：

1、 在 spider 文件中继承的类为 RedisSpider ，然后定义 redis\_key

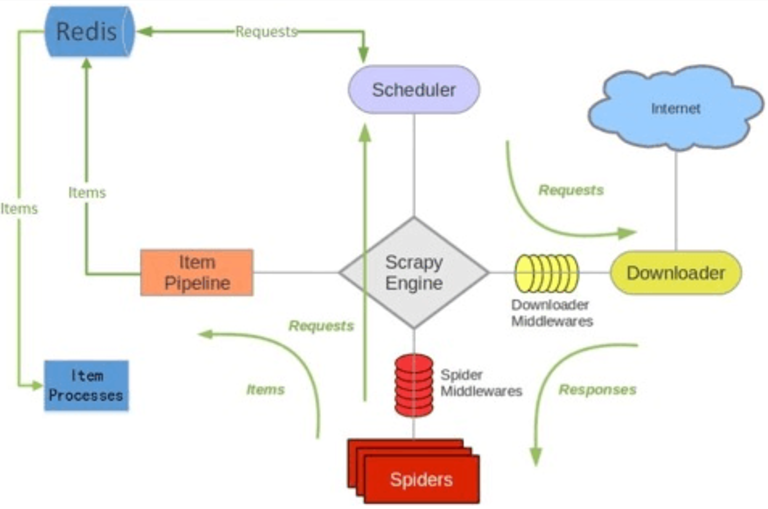
2、 在 settings.py 文件增加如下三行代码：

|  |
| --- |
| SCHEDULER = "scrapy\_redis.scheduler.Scheduler"  DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"  ITEM\_PIPELINES = {  'scrapy\_redis.pipelines.RedisPipeline': 300  } |

其他就没啥不同了。

## Scrapy-Redis 源码分析

Scrapy + Redis 的架构图：



**connection.py 分析：**

该文件是用来连接 redis 的，也是最重要的文件，本文件是连接 redis 最基本的文件，主要实现连接 redis 的配置参数设置。

**defaults.py 分析：**

该文件用来编写各个配置参数，代码如下：

|  |
| --- |
| import redis  # 保存访问过的 request 指纹  DUPEFILTER\_KEY = 'dupefilter:%(timestamp)s'  PIPELINE\_KEY = '%(spider)s:items'  # https://github.com/andymccurdy/redis-py redis 模块  REDIS\_CLS = redis.StrictRedis  # 链接 redis 指名编码  REDIS\_ENCODING = 'utf-8'  # 链接 redis 的参数,供 redis.py 调用  REDIS\_PARAMS = {  'socket\_timeout': 30,  'socket\_connect\_timeout': 30,  'retry\_on\_timeout': True,  'encoding': REDIS\_ENCODING,  }  # request 队列的 key  SCHEDULER\_QUEUE\_KEY = '%(spider)s:requests'  # 指名 request 队列 的类型，也就是先进先出，后进先出，优先级队列 这三种中的一种  SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.PriorityQueue'  # dupefilter 去重的 key  SCHEDULER\_DUPEFILTER\_KEY = '%(spider)s:dupefilter'  # dupefilter 去重器 的类型  SCHEDULER\_DUPEFILTER\_CLASS = 'scrapy\_redis.dupefilter.RFPDupeFilter'  START\_URLS\_KEY = '%(name)s:start\_urls'  START\_URLS\_AS\_SET = False |

**dupefilter.py分析：**

用来过滤重复的URL，替换了scrapy原来的去重器。该文件与 scrapy 的 dupefilters.py 的代码差不多。

|  |
| --- |
| @classmethod  def from\_settings(cls, settings):  # 这个 server 其实就是链接 redis 的  server = get\_redis\_from\_settings(settings) |
| # 这个用来把 request 生成指纹并加入序列,成功,说明没重复返回 True  # 失败,说明重复了返回 False  def request\_seen(self, request):  fp = self.request\_fingerprint(request)  added = self.server.sadd(self.key, fp)  return added == 0 |

**picklecompat.py介绍：**完成序列化操作。

**pipelines.py介绍：**

将 item 保存到 redis 中，实现了 item 的分布式保存。

|  |
| --- |
| def process\_item(self, item, spider):  # 异步操作,每次开启一个 新线程  return deferToThread(self.\_process\_item, item, spider) |

**queue.py介绍：**

处理 request 的队列，里面有三种模式的队列，先进先出，后进先出，优先级排序队列。scarpy 其实也是三种队列。主要是三种队列的不同操作函数。

|  |
| --- |
| # Fifo → first in first out 先进先出,有序队列  SpiderQueue = FifoQueue  # Lifo → last in first out 后进先出  SpiderStack = LifoQueue  # 优先级队列,有序集合,通过 score 来排序  SpiderPriorityQueue = PriorityQueue |

**scheduler.py介绍：**

url 调度器。可对比 scarpy 的 scheduler.py 进行源码分析。

主要函数为 enqueue\_request 和 next\_request。

**spiders.py介绍：**

通过 redis 读取 start\_urls，重点分析 next\_requests 函数。

**utils.py介绍：**

该文件只是实现了 py3 的兼容。

## 集成 bloomfilter 到 scrapy-redis 中

百度 “BloomFilter——大规模数据处理利器” 可找到很多BloomFilter 参考文章，比如：<https://www.cnblogs.com/heaad/archive/2011/01/02/1924195.html>

imooc 的一个实现：<https://github.com/liyaopinner/BloomFilter_imooc>，这里附上代码：

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import mmh3  import redis  import math  import time  class PyBloomFilter():  #内置100个随机种子  SEEDS = [543, 460, 171, 876, 796, 607, 650, 81, 837, 545, 591, 946, 846, 521, 913, 636, 878, 735, 414, 372,  344, 324, 223, 180, 327, 891, 798, 933, 493, 293, 836, 10, 6, 544, 924, 849, 438, 41, 862, 648, 338,  465, 562, 693, 979, 52, 763, 103, 387, 374, 349, 94, 384, 680, 574, 480, 307, 580, 71, 535, 300, 53,  481, 519, 644, 219, 686, 236, 424, 326, 244, 212, 909, 202, 951, 56, 812, 901, 926, 250, 507, 739, 371,  63, 584, 154, 7, 284, 617, 332, 472, 140, 605, 262, 355, 526, 647, 923, 199, 518]  #capacity是预先估计要去重的数量  #error\_rate表示错误率  #conn表示redis的连接客户端  #key表示在redis中的键的名字前缀  def \_\_init\_\_(self, capacity=1000000000, error\_rate=0.00000001, conn=None, key='BloomFilter'):  self.m = math.ceil(capacity\*math.log2(math.e)\*math.log2(1/error\_rate)) #需要的总bit位数  self.k = math.ceil(math.log1p(2)\*self.m/capacity) #需要最少的hash次数  self.mem = math.ceil(self.m/8/1024/1024) #需要的多少M内存  self.blocknum = math.ceil(self.mem/512) #需要多少个512M的内存块,value的第一个字符必须是ascii码，所有最多有256个内存块  self.seeds = self.SEEDS[0:self.k]  self.key = key  self.N = 2\*\*31-1  self.redis = conn  # print(self.mem)  # print(self.k)  def add(self, value):  name = self.key + "\_" + str(ord(value[0])%self.blocknum)  hashs = self.get\_hashs(value)  for hash in hashs:  self.redis.setbit(name, hash, 1)  def is\_exist(self, value):  name = self.key + "\_" + str(ord(value[0])%self.blocknum)  hashs = self.get\_hashs(value)  exist = True  for hash in hashs:  exist = exist & self.redis.getbit(name, hash)  return exist  def get\_hashs(self, value):  hashs = list()  for seed in self.seeds:  hash = mmh3.hash(value, seed)  if hash >= 0:  hashs.append(hash)  else:  hashs.append(self.N - hash)  return hashs  pool = redis.ConnectionPool(host='127.0.0.1', port=6379, db=0)  conn = redis.StrictRedis(connection\_pool=pool)  if \_\_name\_\_ == "\_\_main\_\_":  bf = PyBloomFilter(conn=conn)  bf.add('www.jobbole.com')  bf.add('www.zhihu.com')  print(bf.is\_exist('www.zhihu.com'))  print(bf.is\_exist('www.lagou.com')) |

在项目下建立 utils 包，然后新建一个 bloomfilter.py 文件，然后把上面的 imooc 的实现的代码复制进去，由于该代码依赖 mmh3 库文件，所以需要 pip install mmh3 进行安装。

如何使用呢？这里我们就需要对 scrapy-redis 的 dupefilter.py 这个文件进行改造了。主要改造两个函数：

def \_\_init\_\_(self, server, key, debug=False) 增加一句代码：

|  |
| --- |
| self.bf = PyBloomFilter(conn=conn, key=key) |

def request\_seen(self, request) 代码修改如下：

|  |
| --- |
| if self.bf.is\_exist(fp):  return True  else:  self.bf.add(fp)  return False |

整理后的代码如下：

|  |
| --- |
| from ScrapyRedisTest.utils.bloomfilter import conn, PyBloomFilter  def \_\_init\_\_(self, server, key, debug=False):  self.server = server  self.key = key  self.debug = debug  self.logdupes = True  self.bf = PyBloomFilter(conn=conn, key=key)    def request\_seen(self, request):  fp = self.request\_fingerprint(request)  if self.bf.is\_exist(fp):  return True  else:  self.bf.add(fp)  return False |

到这里 BloomFilter 集成到 scrapy-redis 就完成了。

## redis-cli 常用操作

请求所有数据：

|  |
| --- |
| redis-cli flushall |

执行批量URL地址：

|  |
| --- |
| redis-cli lpush mtsohucom:start\_urls http://mt.sohu.com/news http://mt.sohu.com/sports http://mt.sohu.com/business http://mt.sohu.com/game http://mt.sohu.com/fashion http://mt.sohu.com/it http://mt.sohu.com/auto http://mt.sohu.com/learning http://mt.sohu.com/health http://mt.sohu.com/travel http://mt.sohu.com/joke http://mt.sohu.com/yule http://mt.sohu.com/astro http://mt.sohu.com/acg http://mt.sohu.com/cul http://mt.sohu.com/history http://mt.sohu.com/pets http://mt.sohu.com/view http://mt.sohu.com/food |

## PyChram 安装 Iedis 插件

File → Settings → Plugins，搜索 Iedis，点击 Search in repositories 进行安装。安装完成之后，重启 PyCharm 就可以了。

# ElasticSearch搜索引擎的使用

# Django搭建搜索网站

# Scrapy项目部署

使用的是scrapyd来进行项目部署。

<http://scrapyd.readthedocs.io/en/stable/>

## 部署步骤

1、 安装 scrapyd ，安装完成之后，可以使用 scrapyd 命令。该命令在 Scripts 目录下。同时会在对应的 Python 环境的 Scripts 目录下安装一个 scrapyd-deploy 这样一个没有后缀的文件。这个文件是无法直接运行的，该命令在 Linux 中可以直接运行，在 windows 中无法直接运行，在该目录下我们新建一个 scrapyd-deploy.bat 文件，代码如下：

|  |
| --- |
| @echo off  "C:\Anaconda3\python.exe" "C:\Anaconda3\Scripts\scrapyd-deploy" %1 %2 %3 %4 %5 %6 %7 %8 %9 |

然后再次运行 scrapyd-deploy 就可以了，该命令是用来部署项目的。scrapyd-deploy 参数：-l 列出已部署的项目列表。

2、 进入要部署爬虫的一个空目录， 运行 scrapyd 命令，该目录会作为 scrapyd 服务器的工作目录。工作端口为 6800 。

3、 到这里 scrapyd 就运行起来了。

4、 安装 scrapyd 客户端 scrapyd-client 该软件是负责打包项目。

5、 cd 进入我们 scrapy 项目所在目录。

6、 编辑 scrapy.cfg 文件，把 url 的注释去掉，然后把 [deploy] 修改为 [deploy:myredis]

7、 scrapy list 查看 spider 的名称和数量。

8、 在项目目录下 运行 scrapyd-deploy zlsscrapy -p scrapypro 部署，比如：

|  |
| --- |
| [deploy:myredis]  url = http://localhost:6800/  project = RedisNews |

9、 使用 scrapyd 的项目命令来运行 scrapy 项目：

|  |
| --- |
| curl http://localhost:6800/listspiders.json?project=RedisNews 列出项目里面有多少爬虫  curl http://localhost:6800/listprojects.json 列出有多少项目  curl http://localhost:6800/daemonstatus.json 查看状态  curl http://localhost:6800/listjobs.json?project=RedisNews 列出所有 job  curl http://localhost:6800/schedule.json 部署  curl http://localhost:6800/schedule.json -d project=RedisNews -d spider=mtsohucom  curl http://localhost:6800/delproject.json 删除项目  curl http://localhost:6800/delproject.json -d project=RedisNews  curl http://localhost:6800/cancel.json 取消某个 spider job 任务  curl http://localhost:6800/cancel.json -d project=RedisNews -d job=58ea4d46715111e7840d685d43942450 |

注意在 spider 的 \_\_init\_\_ 中需要加 \*\*kwargs 不然部署会出问题。 def \_\_init\_\_(self , \*\*kwargs):

# 相关错误处理

## MySQLdb-cursors.py-313- Warning- (1300- -Invalid u

出现如下错误提示信息：

|  |
| --- |
| MySQLdb\cursors.py:313: Warning: (1300, "Invalid utf8 character string: 'F09F87'")  MySQLdb\cursors.py:313: Warning: (1366, "Incorrect string value: '\\xF0\\x9F\\x87\\xAB\\xF0\\x9F...' for column 'body' at row 1") |

这个问题是由编码问题造成的，当编码为 utf-8 时，只能保存 3 个长度的编码，当长度是 4 个的时候，需要使用 utf8mb4 才可以解决该问题。

下面是对应的修改相关编码的语句：

|  |
| --- |
| ALTER DATABASE mypro CHARACTER SET utf8mb4;  USE mypro;  SHOW VARIABLES LIKE 'CHARACTER\_SET\_DATABASE';  ALTER TABLE zlszzc CONVERT TO CHARACTER SET utf8mb4; |

PHP 或者 Python 中的数据库链接的编码也需要修改为 utf8mb4。

参考文件:

<http://blog.csdn.net/hhtnan/article/details/76769264>

<http://blog.csdn.net/hzw19920329/article/details/55670782>

<https://www.cnblogs.com/shihaiming/p/5855616.html>

## 设置 insert 的优先级高于 select 和 update

my.ini 加入如下语句

|  |
| --- |
| # 不管表有没有空洞,都允许在数据文件结尾并发插入  concurrent\_insert=2 |

然后在 insert 语句加 HIGH\_PRIORITY 优先设置，比如：