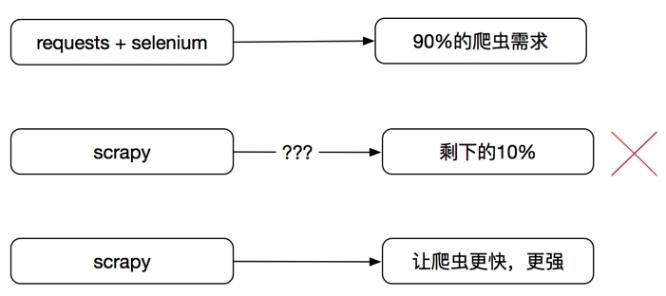
# Scrapy框架学习

课程概要

1. scrapy的基础概念
2. scrapy的工作流程
3. scrapy入门使用
4. scrapy的深入
5. crawlspider的使用

## 1. scrapy的基础概念

### 为什么要学习scrapy



### 什么是scrapy

Scrapy是一个为了爬取网站数据，提取结构性数据而编写的应用框架，我们只需要实现少量的代码，就能够快速的抓取

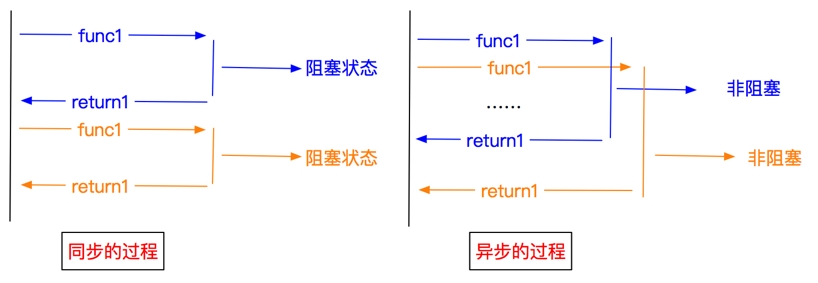
什么是框架, 与模块的区别. request只能发送请求, 获取响应, 只能实现单一的功能, 是一个模块. 而把多个模块有机地结合到一起, 实现一系列复杂的功能, 就是框架, 框架中实现一个功能的代码就是一个模块.

Scrapy 使用了 Twisted['twɪstɪd]异步网络框架，可以加快我们的下载速度。

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/intro/overview.html

在爬虫中, 速度最慢的就是发送网络请求获取响应的过程, 只要使用request.get发送了请求,一旦发送出去请求, 就处于等待状态了, 只有当服务器的响应返回给我们之后, 才能继续执行下一步的操作. 从发送请求到服务器的响应返回给我们这个过程都是我们不可控的, 网络延迟, 服务器的状态等都可能会加大我们的等待时间. scrapy可以使用多线程同时发送多个请求, 以减少总体等待的时间.

### 补充：异步和非阻塞的区别：



异步：**调用**在发出之后，这个调用就直接返回，不管有无结果.

非阻塞：关注的是程序在等待调用结果（消息，返回值）时的状态，指在不能立刻得到结果之前，该调用不会阻塞当前线程。

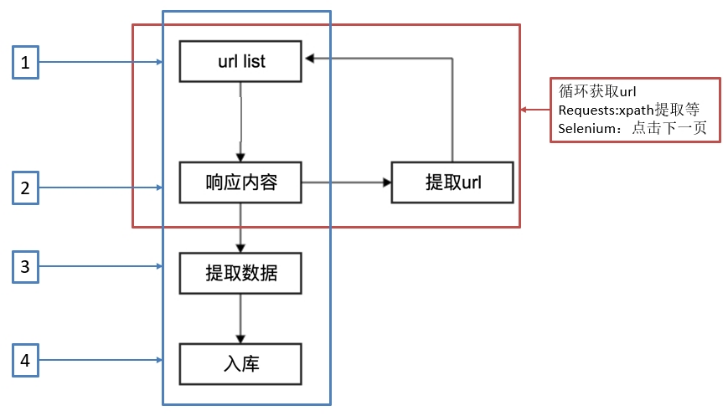
阻塞或非阻塞是拿到结果之前的状态, 如果拿到结果之前是在等待中, 就是阻塞的, 否则就是非阻塞的. 如果在一个过程中有大量的阻塞状态, 这个过程就是同步的. 如果拿到结果之前没有阻塞的状态, 过程就是异步的.

阻塞或非阻塞是拿到结果之前的状态, 而同步或异步是整个过程.

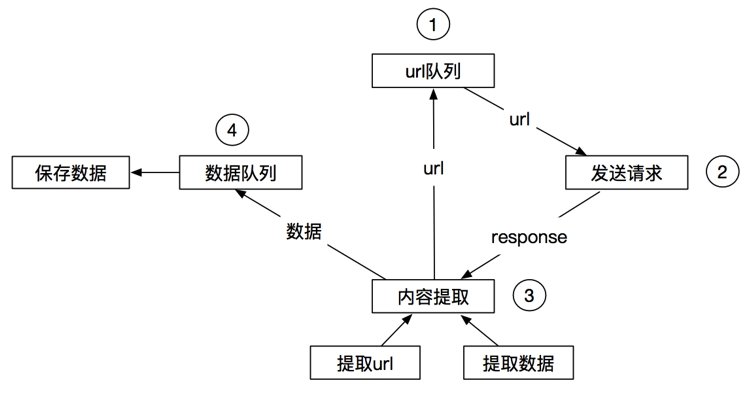
## 2. scrapy的工作流程

那么，scrapy是如何帮助我们抓取数据的呢？

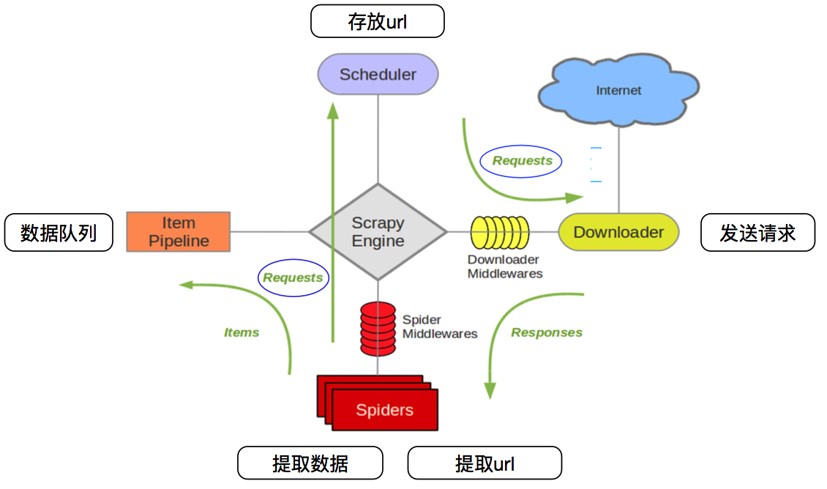
### 回顾前面的爬虫流程



### 另一种形式爬虫流程



### scrapy的爬虫流程



Python中Pipeline管道的作用, 把前一次的结果做为下一次的输入.

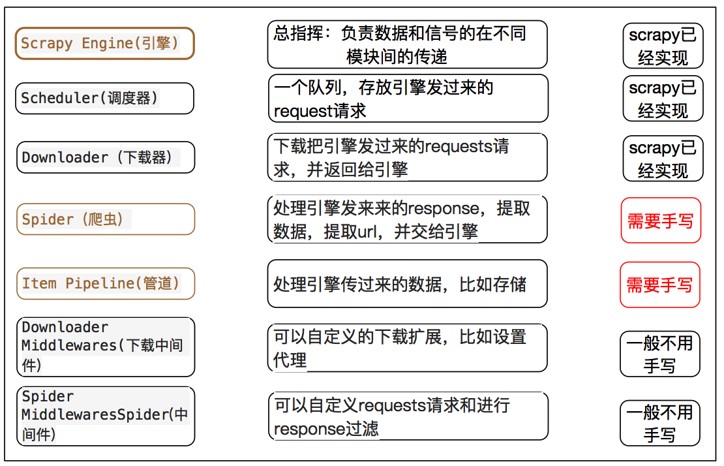
调度器中保存的不是url地址, 而是request对象. request对象中保存有url, headers, proxy等爬虫相关的信息.

不使用Spider Middlewares对提取到的数据进行处理, 而是使用item Pipeline对数据进行处理. 不是Spider Middlewares不能处理, 而是一般不这样来处理. 而是使用解耦的思想, 把不同的功能在不同的模块中完成, 不同的模块又在不同的线程中完成, 就提高了运行的效率.

scrapy各个模块之间功能相对独立, 使用Scrapy Engine连接起来. scrapy engine就相当于十字路口的交警.

### scrapy组件详解

We need know more



## 3. Scrapy入门

1. 创建一个scrapy项目
   * scrapy startproject mySpider
2. 生成一个爬虫
   * scrapy genspider itcast "itcast.cn"
3. 提取数据
   * 完善spider，使用xpath等方法
4. 保存数据
   * pipeline中保存数据

### 安装Scrapy

pip install scrapy

### 创建scrapy项目

命令：scrapy startproject +<项目名字>

scrapy startproject myspider

>cd PycharmProjects

>scrapy startproject mySpider

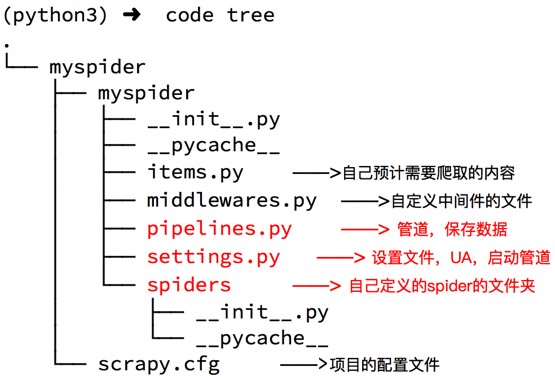
New Scrapy project 'mySpider', using template directory 'c:\\users\\david\\envs\\python3\_spider\\lib\\site-packages\\scrapy\\templates\\project', created in:

C:\Users\David\PycharmProjects\mySpider

You can start your first spider with:

cd mySpider

scrapy genspider example example.com



scrapy.cfg

# Automatically created by: scrapy startproject

#

# For more information about the [deploy] section see:

# https://scrapyd.readthedocs.io/en/latest/deploy.html

# 指定项目使用的设置文件

**[settings]**

**default** = **mySpider.settings**

# scrapy提供了一个scrapyd的功能, 可以把代码发布到本机或服务器中

**[deploy]**

#url = http://localhost:6800/

**project** = **mySpider**

### 创建scrapy爬虫

命令：scrapy genspider + <爬虫名字> + <允许爬取的域名>

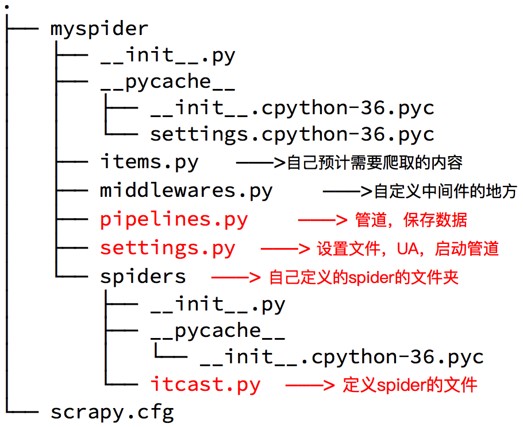
cd myspider

# 指定要爬取的域名, 以免爬到其它网站上的内容.

mySpider>scrapy genspider itcast "itcast.cn"

Created spider 'itcast' using template 'basic' in module:

mySpider.spiders.itcast



mySpider/spiders/itcast.py

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

**import** scrapy

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

name = **'itcast'**

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

start\_urls = [**'http://itcast.cn/'**]

# parse方法就是用来处理start\_url地址的响应, parse方法中的response就是对应的起始url的响应. 注意这里的parse方法名也不能够进行修改, 如果不使用parse这个固定的名称, 就会报错. Spider error processing <GET … , 错误的名称为 raise NotImplementedError, 这个错误指的是方法没有完成. 也就是说, 在父类方法中已经定义了一个不完整的parse方法, 我们必须要把它补全. 否则就会出错. 在我们自己写代码时, 如果代码会被别人调用, 在未完成的状态下也可以自定义一个类似的方法, raise一个错误告诉用户还没有完成.

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

**pass**

### 完善spider

想要爬取itcast师资库中所有老师的信息

<http://www.itcast.cn/channel/teacher.shtml#ajavaee>

首先要确认在响应的内容中有着我们要爬取的老师的信息.

查看网页源码, 在其中搜索老师的信息. div class="tea\_con", 确认能找到老师的信息之后, 就可以使用这个url来爬取信息了.

scrapy会向这个url地址发送请求, 服务器的响应会进入到Spider类的parse方法中.

*# 自定义spider类, 继承自scrapy.Spider*

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

*# 爬虫名称, 掉启动时使用scrapy crawl itcast*

name = **'itcast'**

*# 允许爬取的范围, 防止爬虫爬到了别的网站*

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

*# 开始爬取的地址*

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

*# parse为数据提取的方法, 接收下载中间件传过来的response*

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

*# 使用xpath返回包含选择器的列表*

name = response.xpath(**"//div[@class='tea\_con']//li/div/h3/text()"**)

print(names)

从选择器中提取字符串的方法:

1. extract() 返回一个包含有字符串数据的列表

2. extract\_first() 返回列表中的第一个字符串

注意:

1. spider中的parse方法名不能修改

2. 需要爬取的url地址必须要是属于allow\_domain的链接

3. response.xpath() 返回的是一个含有selector对象的列表

视频演示

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

**import** scrapy

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

name = **'itcast'** *#爬虫名*

allowed\_domains = [**'itcast.cn'**] *#允许爬取的范围*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml'**] *#最开始请求的url地址. 有无锚点发送的都是同一个响应.*

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

*#处理start\_url地址对应的响应, 提取数据., 打印的ret1是一个列表, 列表中的元素是Selector对象., xpath是我们定义的提取规则 , data中是我们想要提取的信息.*

ret1 = response.xpath(**"//div[@class='tea\_con']//h3/text()"**)

print(ret1)

启动爬虫进行测试.

需要进入到爬虫项目根目录中启动爬虫.

scrapy crawl itcast

在启动爬虫之后会出现很多的日志信息. 可以在settings.py中定义显示的日志信息的等级. 一共有4个等级的信息, DEBUG, INFO, WARNING, ERROR, 这样设置之后, 比等级WARNING低的信息都不会显示出来. 即只会显示WARNING和ERROR的信息.

LOG\_LEVEL = **"WARNING"**

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

**import** scrapy

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

name = **'itcast'** *#爬虫名*

allowed\_domains = [**'itcast.cn'**] *#允许爬取的范围*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml'**] *#最开始请求的url地址*

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

*#处理start\_url地址对应的响应, 提取数据. 使用extract()方法从响应中提取xpath匹配到的信息. 注意这里的列表并不是python中的列表, 而是scrapy经过处理后的列表了, 即是scrapy自定义的特殊列表类型了. 所以可以使用extract()方法从中提取信息.*

ret1 = response.xpath(**"//div[@class='tea\_con']//h3/text()"**).extract()

print(ret1)

先对信息进行分组, 然后再循环提取.

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

**import** scrapy

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

name = **'itcast'** *#爬虫名*

allowed\_domains = [**'itcast.cn'**] *#允许爬取的范围*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml'**] *#最开始请求的url地址*

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

*#处理start\_url地址对应的响应, 提取数据*

*# ret1 = response.xpath("//div[@class='tea\_con']//h3/text()").extract()*

*# print(ret1)*

*#分组之后再提取信息. 每个li标签是一组.*

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

**for** li **in** li\_list:

item = {}

*# extract()也是得到的列表对象, 可以使用[0]来提取其中的第1个元素*

item[**"name"**] = li.xpath(**".//h3/text()"**).extract()[0]

item[**"title"**] = li.xpath(**".//h4/text()"**).extract()[0]

print(item)

注意在对列表中的元素进行选取时, 必须要先保证元素存在, 才能提取, 否则会报错. 也就是必须要在后面进行判断. 但scrapy提供了一个extract\_first()方法来提取列表中的第一个元素, 如果有第1个元素, 就提取, 否则就返回None.

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

**import** scrapy

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

name = **'itcast'** *#爬虫名*

allowed\_domains = [**'itcast.cn'**] *#允许爬取的范围*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml'**] *#最开始请求的url地址*

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

*#处理start\_url地址对应的响应, 提取数据*

*# ret1 = response.xpath("//div[@class='tea\_con']//h3/text()").extract()*

*# print(ret1)*

*#分组之后再提取信息. 每个li标签是一组.*

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

**for** li **in** li\_list:

item = {}

item[**"name"**] = li.xpath(**".//h3/text()"**).extract\_first()

item[**"title"**] = li.xpath(**".//h4/text()"**).extract\_first()

print(item)

### 使用pipeline

#### spider的数据传到pipeline

pipeline是专门用来处理数据的组件, 可以把打印item的功能放在pipeline中去实现. 只需要在spider中使用yield来返回item对象就可以了, item对象会由scrapy engine会逐个的从yield生成器中出来, 发送到pipeline中进行处理. 这里为什么使用生成器而不是直接把所有结果返回呢? 使用生成器, 每次生成一个对象, scrapy engine把这个对象交给pipeline处理, 一个对象处理完后再生成第二个对象, 传递到pipeline中处理, 这样一个对象接一个对象的处理, 就可以减少内存的占用.

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

*# xpath分组提取*

teachers = response.xpath(**"//div[@class='tea\_con']//li"**)

**for** t **in** teachers:

name = t.xpath(**"./div/h3/text()"**).extract\_first()

position = t.xpath(**"./div/h4/text()"**).extract\_first()

profile = t.xpath(**"./div/p/text()"**).extract\_first()

item = dict(

name = name,

position = position,

profile = profile

)

*# yield就可以了*

**yield** item

为什么要使用yield

让整个函数变成一个生成器, 变成generator生成器有什么好处?

每次遍历的时候挨个读到内存的, 不会导致内存的占用量瞬间变高.

python3中的range和python2中的xrange同理

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

**import** scrapy

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

name = **'itcast'** *#爬虫名*

allowed\_domains = [**'itcast.cn'**] *#允许爬取的范围*

start\_urls = [**'http://www.itcast.cn/channel/teacher.shtml'**] *#最开始请求的url地址*

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

*#处理start\_url地址对应的响应, 提取数据*

*# ret1 = response.xpath("//div[@class='tea\_con']//h3/text()").extract()*

*# print(ret1)*

*#分组之后再提取信息. 每个li标签是一组.*

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

**for** li **in** li\_list:

item = {}

item[**"name"**]=li.xpath(**".//h3/text()"**).extract\_first()

item[**"title"**] = li.xpath(**".//h4/text()"**).extract\_first()

*# print(item)*

*#Request, BaseItem, dict or None*

*# 这里的item名称其实是可以随意定义的. 但注意item必须是Request, BaseItem, dict 或者None对象. 不能是列表等其它的对象. Spider must return Request, BaseItem, dict or None.*

**yield** item

修改piplines.py文件

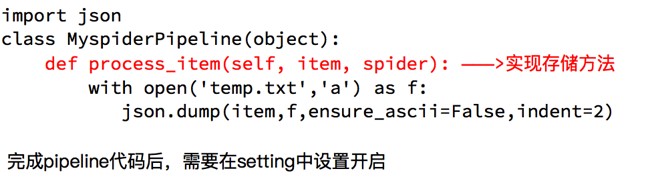
**class** MyspiderPipeline(object):

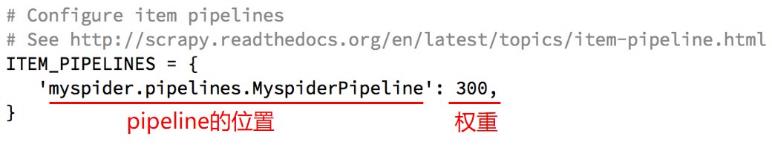
**def** process\_item(self, item, spider):

print(item)

**return** item

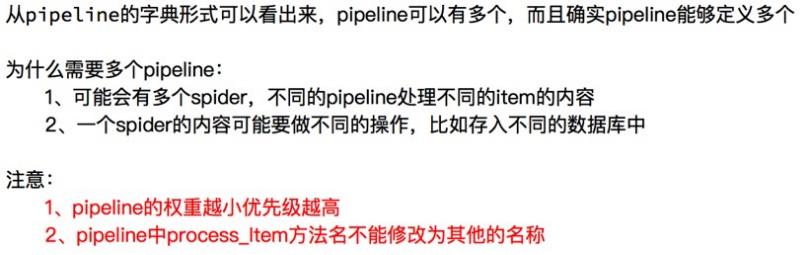
#### 使用pipeline





pipeline位于项目myspider根目录下的myspider/pipelines.py中的MyspiderPipeline类中. 这里定义的是一个相对目录, 因为项目执行的时候是在项目根目录下执行的, 在这里执行就可以通过myspider/pipelines.py找到对应的pipeline.

权重可以认为是距离scrapy engine的远近, 权重的值越小距离scrapy engine就越近, 数据就会先进入到这个pipeline中进行处理.



#### 使用多个pipeline管道

多个pipeline的作用, 可以定义多个pipeline来处理来自不同spider的数据, 还可以使用多个pipeline对所有spider的数据进行分步处理, 在各个pipeline中实现不同的功能, 如一个pipeline专门对数据进行处理, 而另一个pipeline专门用于对数据进行存储.

##### 对数据进行分步处理

修改pipelines.py文件, 添加多个管道.

**class** MyspiderPipeline(object):

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

*#* ***TODO***

item[**"hello"**] = **"world"**

**#** print(item)

**return** item

**class** MyspiderPipeline1(object):

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

print(item)

**return** item

在settings.py中添加这两个pipeline并修改权重.

ITEM\_PIPELINES = {

'myspider.pipelines.MyspiderPipeline': 300,

'myspider.pipelines.MyspiderPipeline1': 301,

}

再次执行爬虫, 就可以看到每条信息中都被添加上了hello world. 也就是数据先进入了MyspiderPipeline, 然后再进入MyspiderPipeline1进行处理. 因为在pipeline中只有process\_item这个方可以用来接收来自spider的item, 所以管道中的process\_item这个名称必须是固定的, 不能修改, 否则数据就不能进入到管道中处理. 并且为了使数据能够在各个pipeline中进行传递, 前一个pipeline中必须要使用return把处理后的数据返回, 数据才能进入到下一个处理的过程中, 否则就会出现None值.

##### 多个爬虫对应多个pipeline

可以在一个项目中创建多个爬虫.

新建项目 scrapy startproject myspider

新建爬虫 scrapy genspider itcast itcast.cn

scrapy genspider itcast1 itcast.n

scrapy genspider itcast2 itcast.cn

新建的爬虫都位于spiders文件夹下, itcast.py, itcast1.py, itcast2.py . 实际中的爬虫项目都会包含多个爬虫, 实现多个网站的同时爬取或多个任务的同时运行.

多个爬虫爬取到的item都会进入到pipelines.py中进行处理. 可以使用多个pipeline类分别处理多个各个spider的item, 也可以在一个pipeline类中处理多个spider的item, 只需要使用if判断即可, 为了区分不同的爬虫, 可以在每个spider的item中添加一个字段come\_from.

spiders/itcast.py

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

**import** scrapy

**import** logging

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

name = **'itcast'**

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

start\_urls = [**'http://www.itcast.cn/'**]

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

**for** i **in** range(10):

item = {}

item[**"come\_from"**] = **"itcast"**

**yield** item

修改pipelines.py文件, 使用一个pipeline来处理多个spider的item

**class** MyspiderPipeline(object):

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

*# if spider.name == "itcast":*

**if** item[**"come\_from"**]==**"itcast"**:

print(**"come from itcast"**)

**if** item[**"come\_from"**] == **"itcast1"**:

print(**"come from itcast1"**)

**if** item[**"come\_from"**] == **"itcast2"**:

print(**"come from itcast2"**)

**return** item

也可以在多个pipeline类中处理不同spider的item, 只需要把上面的if判断分拆放在多个pipeline类中即可.

**class** MyspiderPipeline(object):

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

*# if spider.name == "itcast":*

**if** item[**"come\_from"**]==**"itcast"**:

print(**"come from itcast"**)

**return** item

**class** MyspiderPipeline1(object):

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

**if** item[**"come\_from"**] == **"itcast1"**:

print(**"come from itcast1"**)

**return** item

**class** MyspiderPipeline2(object):

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

**if** item[**"come\_from"**] == **"itcast2"**:

print(**"come from itcast2"**)

**return** item

##### process\_item中spider参数的作用

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

定义的每一个爬虫都会把item传递到pipeline中进行处理, 这里的spider就是与item对应的爬虫. 即item传递到pipeline中时也会把spider本身传递过去, 更准确的说, 这里的spider就是在各个爬虫的py文件中定义的spider类. 由于在定义spider类的时候有一个name属性, 这个属性会随着spider类一起传递到pipeline中, 所以就可以在pipeline中根据spider的name属性来判断每个item是来自哪个spider的, 这样就不需要专门在爬虫的item中添加一个字段来识别item是来自哪个spider的.

一个pipeline处理多个spider的item

**class** MyspiderPipeline(object):

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

**if** spider.name == **"itcast"**:

print(**"come from itcast"**)

**if** spider.name == **"itcast1"**:

print(**"come from itcast1"**)

**if** spider.name == **"itcast2"**:

print(**"come from itcast2"**)

**return** item

多个pipeline处理多个spider的item

**class** MyspiderPipeline(object):

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

**if** spider.name == **"itcast"**:

print(**"come from itcast"**)

**return** item

**class** MyspiderPipeline1(object):

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

**if** spider.name == **"itcast1"**:

print(**"come from itcast1"**)

**return** item

**class** MyspiderPipeline2(object):

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

**if** spider.name == **"itcast2"**:

print(**"come from itcast2"**)

**return** item

### logging模块的使用

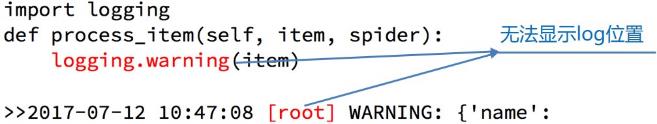
可以使用logging模块来实现日志信息的更进一步定制. logging模块是python中专门用来进行日志信息处理的模块. 如果程序比较大, 很多py文件中都使用print打印信息, 就无法区分打印的信息是来自哪个py文件的, 使用logging来代替print打印日志信息, 一方面可以区分信息来自哪个py文件, 另一方面可以把日志信息保存到本地文件中, 对程度的运行状态做进一步的分析.

为了让我们自己希望输出到终端的内容能容易看一些：

可以在setting中设置log级别. 在setting中添加一行（全部大写）：

LOG\_LEVEL = "WARNING"

scrapy运行的时候终端显示的是WARNING和WARNING级别以上的log信息



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

**import** scrapy

**import** logging

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

name = **'itcast'**

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

start\_urls = [**'http://www.itcast.cn/'**]

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

**for** i **in** range(10):

item = {}

item[**"come\_from"**] = **"itcast"**

*# 这里不再使用print, 而是使用logging.warning把item的信息在终端显示出来. logging.info, logging.debug, logging.error, logging.warning*

logging.warning(item)

**yield** item

在终端中执行scrapy crawl itcast, 就会显示出如下信息.

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

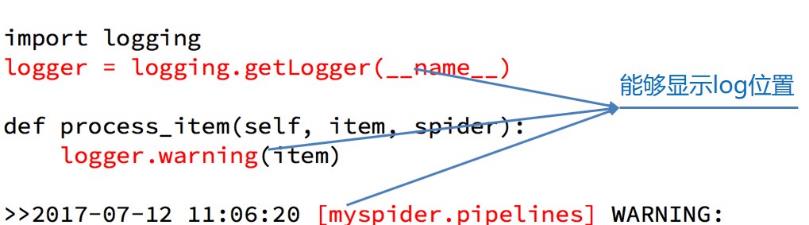
2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

2018-05-22 09:15:29 [root] WARNING: {'come\_from': 'itcast'}

scrapy默认的日志的格式为:

日期 时间 输出日志的文件位置 日志的等级 日志内容.

[root] 表示日志输出的位置, 即日志是从根目录中输出的. 问题就是我们是在myspider下执行的爬虫, 而日志却显示是在根目录中输出的, 并且也没有显示出日志是来自于哪个py文件的, 这不是我们想要的结果, 我们想把日志保存到执行scrapy命令的目录中即scrapy项目的根目录中. 所以需要对日志的输出格式进行定制. 使用logging模块中的getLogger方法来完成日志信息的定制.



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

**import** scrapy

**import** logging

logger = logging.getLogger(\_\_name\_\_)

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

name = **'itcast'**

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

start\_urls = [**'http://www.itcast.cn/'**]

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

**for** i **in** range(10):

item = {}

item[**"come\_from"**] = **"itcast"**

logger.warning(item)

*# yield item*

**pass**

执行scrapy crawl itcast, 就会打印出如下信息

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:30:30 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

此时在日志输出的位置处就显示了是哪个文件输出的日志信息了. 因为是从项目根目录myspider中执行的程度, 显示的路径就是相对于项目根目录的路径信息.

logger = logging.getLogger(\_\_name\_\_)

因为当前并没有指定文件为itcast.py, 所以此时的\_\_name\_\_的名字就是它的文件名"itcast", 如果指定了文件为itcast.py, 那么打印出来就是"\_\_main\_\_" ???

同样, 也可以在pipelines.py中文件中使用同样的方法来输出日志信息.

修改pipelines.py文件

**import** logging

logger = logging.getLogger(\_\_name\_\_)

**class** MyspiderPipeline(object):

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

**if** spider.name == **"itcast"**:

*# if item["come\_from"]=="itcast":*

logger.warning(**"-"**\*10)

**return** item

**class** MyspiderPipeline2(object):

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

**if** item[**"come\_from"**]==**"jd"**:

**pass**

**return** item

同时, 修改itcast.py文件, 使用yield向pipeline中传递item

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

**import** scrapy

**import** logging

logger = logging.getLogger(\_\_name\_\_)

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

name = **'itcast'**

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

start\_urls = [**'http://www.itcast.cn/'**]

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

**for** i **in** range(10):

item = {}

item[**"come\_from"**] = **"itcast"**

logger.warning(item)

**yield** item

确保settings.py中的pipeline已经开启.

ITEM\_PIPELINES = {

'myspider.pipelines.MyspiderPipeline': 300,

}

然后在项目根目录中执行

scrapy crawl itcast

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

2018-05-22 10:41:38 [myspider.spiders.itcast] WARNING: {'come\_from': 'itcast'}

2018-05-22 10:41:38 [myspider.pipelines] WARNING: ----------

这样, 就可以在不同的文件中同时输出日志文件, 在日志文件中就可以使用输出的路径来区分是哪个文件中输出的日志.

还可以把日志信息保存到文件中. 在settings.py中进行设置.

LOG\_LEVEL = **"WARNING"**

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

LOG\_FILE表示把日志信息保存到当前目录下的log.log文件中, 再次执行itcast爬虫, 就可以看到在项目的根目录下生成了log.log文件. 同时终端中不再显示出日志信息.

以上是在scarpy项目中使用的logging模块. 在普通的py文件中也可以使用logging模块来输出日志信息.

新建log\_a.py, log\_b.py, 修改log\_a.py文件

# coding=utf-8

**import** logging

# level为日志输出的等级, format为日志输出的格式. filename为把日志保存的位置

logging.basicConfig(level=logging.INFO,

format=**'levelname:%(levelname)s filename: %(filename)s '**

**'outputNumber: [%(lineno)d] thread: %(threadName)s output msg: %(message)s'**

**' - %(asctime)s'**,

datefmt=**'[%d/%b/%Y %H:%M:%S]'**,

*# filename='./loggmsg.log'*

)

# 实例化一个logger对象

logger = logging.getLogger(\_\_name\_\_)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

# 使用logger对象的.info方法来输出info等级的日志

logger.info(**"this is a info log"**)

logger.info(**"this is a info log 1"**)

执行python log\_a.py

levelname:INFO filename: log\_a.py outputNumber: [24] thread: MainThread output msg: this is a info log - [22/May/2018 11:01:30]

levelname:INFO filename: log\_a.py outputNumber: [25] thread: MainThread output msg: this is a info log 1 - [22/May/2018 11:01:30]

也可以对输出的格式进行修改, 修改成为我们想要的格式

*# coding=utf-8*

**import** logging

*#设置日志的输出样式, level为日志输出的等级, format为日志输出的格式. filename为把日志保存的位置*

logging.basicConfig(level=logging.INFO,

format=**'%(asctime)s'**

**'%(levelname)s [%(filename)s:%(lineno)d] '**

**': %(message)s'**,

datefmt=**'[%d/%b/%Y %H:%M:%S]'**,

)

*# 实例化一个logger对象*

logger = logging.getLogger(\_\_name\_\_)

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

*# 使用logger对象的.info方法来输出info等级的日志*

logger.info(**"this is a info log"**)

logger.info(**"this is a info log 1"**)

再次执行log\_a.py, 会输出如下日志信息

[22/May/2018 11:42:08]INFO [log\_a.py:25] : this is a info log

[22/May/2018 11:42:08]INFO [log\_a.py:26] : this is a info log 1

只要在一个文件中定义了logger, 在其它py文件中就不用重复定义了, 只要引入定义好的logger对象就可以了. 这个思想也适用于一个大的项目, 在一个项目中, 只需要在一个py文件中定义输出日志的样式, 再实例化一个logger, 其它py文件中只需要把定义好的logger引入就可以了.

修改log\_b.py, 从log\_a.py中引入logger.

*# coding=utf-8*

**from** log\_a **import** logger

**if** \_\_name\_\_ == **'\_\_main\_\_'**:

logger.warning(**"this is log b 1"**)

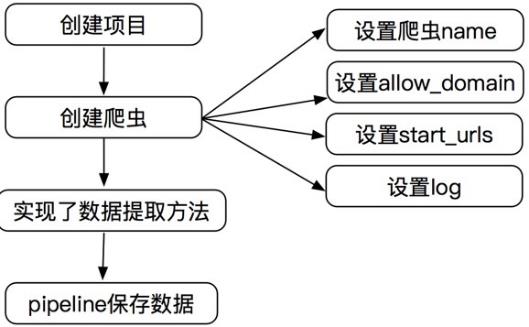
logger.warning(**"this is log b 2"**)

执行log\_b.py

[22/May/2018 11:47:36]WARNING [log\_b.py:5] : this is log b 1

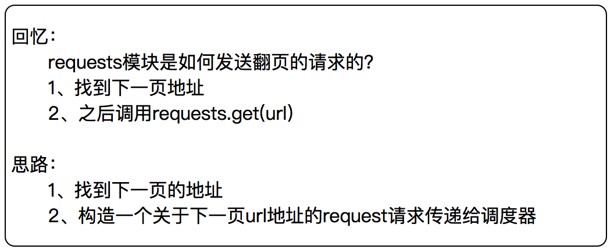
[22/May/2018 11:47:36]WARNING [log\_b.py:6] : this is log b 2

### scrapy爬虫回顾



### 实现翻页请求





#### 通过爬取腾讯招聘学习实现翻页请求

<http://hr.tencent.com/position.php>

在setting中设置User-Agent：

USER\_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36'

实现翻页请求



先找到下一页的url, 然后构建 request对象交给scrapy engine. 使用scrapy.Request(next\_page\_url, callback=self.parse)来构建request对象, 其中的callback为回调函数, 说明从下一页的url地址构建的request对象要交给谁去处理. 起始url地址是交给parse方法去处理的, 如果下一页的url地址也交给parse去处理, 这里就写为callback=self.parse.

http://hr.tencent.com/position.php

腾讯社会招聘的页面, 这个url能不能做为起始url是看这个url对应的响应中有没有我们要的内容, 如果有, 就可以作为起始的url地址. 先检查元素, 找到页面中想要的内容, 再查看url对应的response, 看其中有没有对应的元素内容.

注意xpath的规则是要按照response中的内容来写的, 不是按照检查元素中的网页结构来写的. 如在检查元素中有tbody的内容, 但是在url对应的response中却没有tbody. 所以一定要注意, xpath要以response中的内容为准. 往往在检查元素中有tbody, 但是在response中却没有tbody.

创建项目

scrapy startproject tencent

创建爬虫

scrapy genspider hr tencent.com

修改hr.py中的start\_urls, 并在parse方法中对页面的信息进行提取

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

**import** scrapy

**from** tencent.items **import** TencentItem

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

name = **'hr'**

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

start\_urls = [**'http://hr.tencent.com/position.php'**]

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

*# 第1个tr和最后一个tr中的内容不是招聘的信息, 所以使用列表切片把它去除掉.*

tr\_list = response.xpath(**"//table[@class='tablelist']/tr"**)[1:-1]

**for** tr **in** tr\_list:

*# 定义一个字典保存信息*

item = {}

item[**"title"**] = tr.xpath(**"./td[1]/a/text()"**).extract\_first()

item[**"position"**] = tr.xpath(**"./td[2]/text()"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[5]/text()"**).extract\_first()

**yield** item

信息会进入到pipeline中进行处理, 首先在settings.py中设置pipeline, 同时把日志的等级设置为WARNING, 这样就只显示重要的信息了.

LOG\_LEVEL = **"WARNING"**

ITEM\_PIPELINES = {

'tencent.pipelines.TencentPipeline': 300,

}

在pipelines.py文件中定义pipeline, 把数据保存到mongodb中.

**from** pymongo **import** MongoClient

*# 实例化mongoclient对象*

client = MongoClient()

*# 创建集合即数据表*

collection = client[**"tencent"**][**"hr"**]

**class** TencentPipeline(object):

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

print(item)

collection.insert(dict(item))

**return** item

运行scrapy crawl hr. 就能够爬取到对应的内容.

打开一个新的终端, 输入mongo, 进入到mongodb的交互环境中, 执行查询

use tencent

db.hr.find()

db.hr.count()

以上过程只是获取到了第一页中的信息, 需要继续提取下一页的信息.

修改hr.py文件

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

**import** scrapy

**from** tencent.items **import** TencentItem

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

name = **'hr'**

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

start\_urls = [**'http://hr.tencent.com/position.php'**]

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

*# 第1个tr和最后一个tr中的内容不是招聘的信息, 所以使用列表切片把它去除掉.*

tr\_list = response.xpath(**"//table[@class='tablelist']/tr"**)[1:-1]

**for** tr **in** tr\_list:

*# 定义一个字典保存信息*

item = {}

item[**"title"**] = tr.xpath(**"./td[1]/a/text()"**).extract\_first()

item[**"position"**] = tr.xpath(**"./td[2]/text()"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[5]/text()"**).extract\_first()

**yield** item

*#找到下一页的url地址*

next\_url = response.xpath(**"//a[@id='next']/@href"**).extract\_first()

*# 如果下一页的href地址为"javascript:;"就说明当前页是最后一页了. 如果当前页不是最后一页, 就提取下一页的链接并补全url地址*

**if** next\_url != **"javascript:;"**:

next\_url = **"http://hr.tencent.com/"** + next\_url

*# 可以使用yield来发送下一页的请求, callback函数中定义当前的url即next\_url交给哪个方法去处理. 因为下一页的处理方法和当前页的处理方法相同, 所以可以使用同一个函数进行处理. 如果下一页的处理方式与当前页的不同, 就需要另外再定义一个方法处理. 如parse\_next()方法, 并在这里的回调函数中指定此函数. callback=self.parse\_next(). 这里callback的调用方式很类似于递归, 直到next\_url="javascript:;"时停止调用.*

**yield** scrapy.Request(

next\_url,

callback=self.parse,

*# meta = {"item":item}*

)

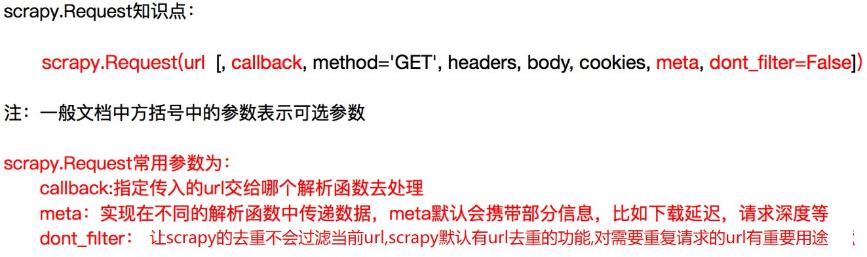
在settings.py文件中定义user-agent

USER\_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'

再次运行scrapy crawl hr

在mongo交互环境下输入db.hr.count()就能看到爬取到的数据的数量.

在实例化request对象时, 还可以传递其它的参数. 如method, headers等信息.



可以使用method="POST"来发送POST请求, 此时要加上请求体body.

需要注意的是, 这里的cookies不能放到headers中, 把cookies作为字符串放到headers中是没有任何效果的, 只能独立使用cookies参数进行传递.

meta: 实现在不同的解析函数中传递数据.

parse就是一个解析函数. 例如, 如果下一页响应的处理方法不同, 就需要重新定义一个解析函数, 如定义一个parse\_detail(), parse()函数用来处理列表页的信息, 而parse\_detail()用来处理详情页的信息. 此时就要使用meta来在两个解析函数之间传递数据. 如在列表页中提取的item信息不完整, 需要在详情页的解析函数中进一步处理, 就使用meta函数来把item从parse中传递到parse\_detail中.

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

**import** scrapy

**from** tencent.items **import** TencentItem

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

name = **'hr'**

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

start\_urls = [**'http://hr.tencent.com/position.php'**]

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

*# 第1个tr和最后一个tr中的内容不是招聘的信息, 所以使用列表切片把它去除掉.*

tr\_list = response.xpath(**"//table[@class='tablelist']/tr"**)[1:-1]

**for** tr **in** tr\_list:

*# 定义一个字典保存信息*

item = {}

item[**"title"**] = tr.xpath(**"./td[1]/a/text()"**).extract\_first()

item[**"position"**] = tr.xpath(**"./td[2]/text()"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[5]/text()"**).extract\_first()

**yield** item

*#找到下一页的url地址*

next\_url = response.xpath(**"//a[@id='next']/@href"**).extract\_first()

*# 如果下一页的href地址为"javascript:;"就说明当前页是最后一页了. 如果当前页不是最后一页, 就提取下一页的链接并补全url地址*

**if** next\_url != **"javascript:;"**:

next\_url = **"http://hr.tencent.com/"** + next\_url

*# 可以使用yield来发送下一页的请求, callback函数中定义当前的url即next\_url交给哪个方法去处理. 因为下一页的处理方法和当前页的处理方法相同, 所以可以使用同一个函数进行处理. 如果下一页的处理方式与当前页的不同, 就需要另外再定义一个方法处理. 如parse\_next()方法, 并在这里的回调函数中指定此函数. callback=self.parse\_next(). 这里callback的调用方式很类似于递归, 直到next\_url="javascript:;"时停止调用.*

**yield** scrapy.Request(

next\_url,

callback=self.parse,

*# meta是字典的格式, 可以传入多个内容, 把item对象放在meta中传递给其它的解析函数.*

meta = {**"item"**:item}

)

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

*# meta是字典格式, 使用response.meta["key\_name"]来取出meta中的内容.*

response.meta[**"item"**]

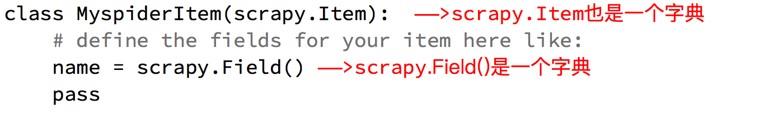
dont\_filter是不过滤, =False表示取反, 即过滤url地址. scrapy默认有去重的功能, 如果当前的url已经被请求过, 之后的请求中就会自动过滤此url, 如果希望本url在之后的请求中依然能够被请求到, 就要设置dont\_filter=True.

现在问题来了, 哪些情况下希望再次对请求过的url地址进行请求呢? 如百度贴吧这种更新非常快的网站, 不仅列表页的数据是实时变化的, 某个帖子的内容也是一直变化着的, response的响应是一直变化的, 这种情况下就需要把dont\_filter设置为True, 才能够反复去请求此url地址去获取最新的内容.

总结, 主要是使用scrapy.Request实例化一个request对象, 并在这个对象中指定callback回调函数, 用于发送下一页的request请求. 然后使用yield来把这个对象返回给scrapy engine, 由engine来进行调度. yield的内容可以是request对象, None, dict和BaseItem

## 4. scrapy深入

### Scrapy深入之自定义Item

  
所以总的来说, 可以把我们定义的MyspiderItem理解为一个字典.

那么 scrapy 吃饱了撑着为什么要定义一个字典呢?

个人理解:

在获取到数据的时候, 使用不同的Item来存放不同的数据

在把数据交给pipeline的时候, 可以通过 isinstance(item, MyspiderItem) 来判断数据是属于哪个item 的, 从而对不同的item数据进行不同的处理.

items.py文件, 可以在其中预先定义好我们要爬取的字段, 如要腾讯社招中要爬取title, position, publish\_date字段, 就可以在items.py中预先进行定义.

**import** scrapy

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

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

title = scrapy.Field()

position = scrapy.Field()

publish\_date = scrapy.Field()

在items.py中定义了TencentItem, 继承自scrapy.Item, 打开源码查看scrapy.Item的代码.

**class** Item(DictItem):

**pass**

Item继承自DictItem

**class** DictItem(MutableMapping, BaseItem):

fields = {}

DictItem又继承自BaseItem, 可以使用yield发送的数据类型中就有一种是BaseItem, 即在items.py中定义的Item类就是继承于BaseItem的类对象.

查看DictItem类的源码

**class** DictItem(MutableMapping, BaseItem):

fields = {}

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

self.\_values = {}

**if** args **or** kwargs: *# avoid creating dict for most common case*

**for** k, v **in** six.iteritems(dict(\*args, \*\*kwargs)):

self[k] = v

**def** \_\_getitem\_\_(self, key):

**return** self.\_values[key]

**def** \_\_setitem\_\_(self, key, value):

**if** key **in** self.fields:

self.\_values[key] = value

**else**:

**raise** KeyError(**"%s does not support field: %s"** %

(self.\_\_class\_\_.\_\_name\_\_, key))

**def** \_\_delitem\_\_(self, key):

**del** self.\_values[key]

可以看到, 在DictItem中定义了几个魔法方法, 一旦定义了魔法方法, 就可以使用类似于字典的[]的方法对它进行取值和赋值了, DictItem的目的就是 avoid creating dict for most common case, 避免在大多数的情况下创建字典. 为什么要尽量避免创建字典呢, 因为在对字典中的键进行赋值时, 在没有约束的情况下, 很可能会出现错误, 如把title拼写成tittle, 这样, 在数据最终保存到数据库中时, 也是使用的这种错误的键, 后继数据处理会带来很多的问题. 所以在Items.py中先定义一些会在spider中使用到的字段, 在spider中引入items.py中定义的类, 通过对类的实例化来创建item, 在对item进行赋值时, 一旦出现了没有在items.py中定义的字段, 就会报错. 这样相当于加了一层的限制, 保证了最终存储的数据的正确性.

在items.py中定义过要爬取的字段之后, spider中就不需要再定义item={}这个字典了, 就可以直接使用items.py中定义的这些字段了. 如, 修改hr.py文件, 使用items.py中定义的字段.

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

**import** scrapy

**from** tencent.items **import** TencentItem

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

name = **'hr'**

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

start\_urls = [**'http://hr.tencent.com/position.php'**]

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

*# 第1个tr和最后一个tr中的内容不是招聘的信息, 所以使用列表切片把它去除掉.*

tr\_list = response.xpath(**"//table[@class='tablelist']/tr"**)[1:-1]

**for** tr **in** tr\_list:

*# 不再自定义字典保存爬取到的信息, 而是实例化一个item对象来保存. 实例化的item对象与字典的操作方式很像, 可以使用字典的键值对赋值方式进行赋值. 注意这里的键只能使用items.py中已经定义过的字段, 否则就会报错.*

item = TencentItem()

item[**"title"**] = tr.xpath(**"./td[1]/a/text()"**).extract\_first()

item[**"position"**] = tr.xpath(**"./td[2]/text()"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[5]/text()"**).extract\_first()

**yield** item

*#找到下一页的url地址*

next\_url = response.xpath(**"//a[@id='next']/@href"**).extract\_first()

*# 如果下一页的href地址为"javascript:;"就说明当前页是最后一页了. 如果当前页不是最后一页, 就提取下一页的链接并补全url地址*

**if** next\_url != **"javascript:;"**:

next\_url = **"http://hr.tencent.com/"** + next\_url

**yield** scrapy.Request(

next\_url,

callback=self.parse,

*# meta是字典的格式, 可以传入多个内容, 把item对象放在meta中传递给其它的解析函数.*

*# meta = {"item":item}*

)

注意, 在从items.py中导入包时, 使用的是绝对路径, 这个路径是相对于执行scrapy爬虫时的路径. 因为是在项目的根目录中执行的爬虫, 所以在导入items.py文件中定义的类时, 要从项目的根目录中起进行导入. from tencent.items import TencentItem, 如果pycharm中的工作目录没有设置到项目的根目录中, pycharm会提示错误, 此时可以忽略错误提示. 在系统终端中可以正常执行.

还需要对pipelines.py中的内容进行修改. 因为item是一个类字典的对象, 并不是一个真实的字典, 使用collection.insert(item)向mongodb中插入数据时, 就会出错, 只需要使用dict()把item强制转换成字典格式即可.

另外, 也可以在pipelines.py中使用items.py中定义的Item类.

**from** pymongo **import** MongoClient

**from** tencent.items **import** TencentItem

*# 实例化mongoclient对象*

client = MongoClient()

*# 创建集合即数据表*

collection = client[**"tencent"**][**"hr"**]

**class** TencentPipeline(object):

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

*# 如果item对象是属于TencentItem这个类的一个对象, 就进行以下的处理, 否则就不进行任何操作.*

**if** isinstance(item,TencentItem):

print(item)

collection.insert(dict(item))

**return** item

如果有多个爬虫, 就可以在items.py中定义多个Item类, 每一个类对应一个爬虫, 如分别对应dangdang的爬虫, jd的爬虫, amazon的爬虫. 在不同的spider中实例化不同的Item类, 在Pipeline中也可以通过上面的方法来识别出对应的item是来自于哪个Item类的或来自于哪个爬虫的, 然后对不同的item进行不同的处理.

这样做还可以从items.py中定义的字段中直观的看到要爬取的字段,

#### Item的使用-阳光政务平台爬虫

通过爬取阳光热线问政平台来学习item的使用

目标：所有的投诉帖子的编号、帖子的url、帖子的标题和内容

url： http://wz.sun0769.com/index.php/question/questionType?type=4&page=0

查看在检查元素中的信息在url的响应中是否能够找到, 能够找到的时候才能把此url地址作为起始的url地址.

注意tbody在url地址的响应中往往是没有的, 在写xpath规则时一要特别需要注意.

首先要爬取起始url中的帖子列表, 还要爬取每个帖子的详情, 这时response的解析函数就与列表页的解析函数不同, 需要另外定义一个详情页的解析函数.

在爬取完一页的内容后, 还要爬取下一页, 下一页列表页的内容与parse函数中的处理方式相同, 下一页详情页与上一页详情页的处理方式相同. 可以分别使用与一页相同的解析函数.

新建项目: scrapy startproject yangguang

新建爬虫; scrapy genspider yg sun0769.com

修改items.py文件, 在其中定义要爬取的字段,

**import** scrapy

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

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

*# 帖子标题*

title = scrapy.Field()

*# 帖子详情页地址*

href = scrapy.Field()

*# 发帖时间*

publish\_date = scrapy.Field()

*# 详情页img地址*

content\_img = scrapy.Field()

*# 帖子详情*

content = scrapy.Field()

修改yg.py文件, 进行信息的提取.

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

**import** scrapy

**from** yangguang.items **import** YangguangItem

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

name = **'yg'**

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

start\_urls = [**'http://wz.sun0769.com/index.php/question/questionType?type=4&page=0'**]

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

*# 分组, 数据都在tr中, 每一个tr就是一条数据*

tr\_list = response.xpath(**"//div[@class='greyframe']/table[2]/tr/td/table/tr"**)

**for** tr **in** tr\_list:

item = YangguangItem()

item[**"title"**] = tr.xpath(**"./td[2]/a[@class='news14']/@title"**).extract\_first()

item[**"href"**] = tr.xpath(**"./td[2]/a[@class='news14']/@href"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[last()]/text()"**).extract\_first()

*# 构建爬取详情页的请求. 注意此时详情页不能交给parse处理, parse处理的是列表页的响应. 由于在列表页中已经获取了一部分数据, 需要把item通过meta传递到详情页的解析函数中进一步添加信息.*

**yield** scrapy.Request(

item[**"href"**],

callback=self.parse\_detail,

meta = {**"item"**:item}

)

*# 重新定义一个解析函数, 用来解析详情页的响应*

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

*# 取出列表页parse函数中传递过来的meta信息中的item, 进一步向其中添加信息*

item = response.meta[**"item"**]

*# 提取详情页中的信息并保存到item中.*

*# 在这里提取信息时要使用extract(), 因为xpath规则中有多个对应的内容. 注意在有图片和无图片时, 帖子的内容看似处在不同的结构中, 实际上还是可以从它们共同的父级对象中去提取.*

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

*# 图片也可能有多张, 所以要使用extract()进行提取*

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

*# 图片的地址不完整, 需要进行拼接.*

item[**"content\_img"**] = [**"http://wz.sun0769.com"**+i **for** i **in** item[**"content\_img"**]]

print(item)

在settings.py中定义User\_Agent, 并且把日志等级修改为WARNING

LOG\_LEVEL = **"WARNING"**

USER\_AGENT = **'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'**

运行爬虫进行测试.

以上只是爬取了一页的信息, 要进行翻页, 提取下一页的信息.

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

**import** scrapy

**from** yangguang.items **import** YangguangItem

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

name = **'yg'**

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

start\_urls = [**'http://wz.sun0769.com/index.php/question/questionType?type=4&page=0'**]

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

*# 分组, 数据都在tr中, 每一个tr就是一条数据*

tr\_list = response.xpath(**"//div[@class='greyframe']/table[2]/tr/td/table/tr"**)

**for** tr **in** tr\_list:

item = YangguangItem()

item[**"title"**] = tr.xpath(**"./td[2]/a[@class='news14']/@title"**).extract\_first()

item[**"href"**] = tr.xpath(**"./td[2]/a[@class='news14']/@href"**).extract\_first()

item[**"publish\_date"**] = tr.xpath(**"./td[last()]/text()"**).extract\_first()

*# 构建爬取详情页的请求. 注意此时详情页不能交给parse处理, parse处理的是列表页的响应. 由于在列表页中已经获取了一部分数据, 需要把item通过meta传递到详情页的解析函数中进一步添加信息.*

**yield** scrapy.Request(

item[**"href"**],

callback=self.parse\_detail,

meta = {**"item"**:item}

)

*# 翻页, 提取下一页的信息, 首先要找到下一页的url地址, 然后构建request对象, 指定使用parse方法来解析返回的response请求.*

*# 由于下一页的url地址所在的标签没有明显的属性, 所以可以使用文本进行定位, 查找文本等于>的标签, 提取它的href值. 查看最后一页, 其中没有>, 所以可以通过extract\_first()进行下一页的ulr地址的提取, 当得到的内容为None时, 就处于最后一页了. 由于这是全新的下一页爬取, 类似于start\_url, 所以不需要写meta信息.*

next\_url = response.xpath(**"//a[text()='>']/@href"**).extract\_first()

*# 如果当前页不是最后一页, 就再次构建下一页的请求.*

**if** next\_url **is not None**:

**yield** scrapy.Request(

next\_url,

callback=self.parse

)

*# 重新定义一个解析函数, 用来解析详情页的响应*

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

*# 取出列表页parse函数中传递过来的meta信息中的item, 进一步向其中添加信息*

item = response.meta[**"item"**]

*# 提取详情页中的信息并保存到item中.*

*# 在这里提取信息时要使用extract(), 因为xpath规则中有多个对应的内容. 注意在有图片和无图片时, 帖子的内容看似处在不同的结构中, 实际上还是可以从它们共同的父级对象中去提取.*

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

*# 图片也可能有多张, 所以要使用extract()进行提取*

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

*# 图片的地址不完整, 需要进行拼接.*

item[**"content\_img"**] = [**"http://wz.sun0769.com"**+i **for** i **in** item[**"content\_img"**]]

*# print(item)*

**yield** item

由于爬取到的信息中有很多不规范的内容, 需要在pipeline中进行进一步的处理. 产生要在settings.py中开启pipeline

ITEM\_PIPELINES = {

'yangguang.pipelines.YangguangPipeline': 300,

}

修改pipelines.py文件, 处理content的内容.

**import** re

**class** YangguangPipeline(object):

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

*# 调用下面定义的process\_content方法, 对爬取到的帖子的详情进行处理*

item[**"content"**] = self.process\_content(item[**"content"**])

print(item)

**return** item

*# 定义一个方法, 专门用来处理爬取到的帖子的详情.*

**def** process\_content(self,content):

*# 把\xa0和空格都替换为空字符串*

content = [re.sub(**r"\xa0|\s"**,**""**,i) **for** i **in** content]

*# 去除列表中的空字符串*

content = [i **for** i **in** content **if** len(i)>0]

**return** content

运行爬虫, 查看打印的结果.

修改pipeline文件, 把处理过的信息保存到mongodb数据库中.

**import** re

**from** pymongo **import** MongoClient

*# 实例化mongoclient对象*

client = MongoClient()

*# 创建集合即数据表*

collection = client[**"tencent"**][**"hr"**]

**class** YangguangPipeline(object):

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

*# 调用下面定义的process\_content方法, 对爬取到的帖子的详情进行处理*

item[**"content"**] = self.process\_content(item[**"content"**])

*# 把数据保存到mongodb数据库中*

collection.insert(dict(item))

*# print(item)*

**return** item

*# 定义一个方法, 专门用来处理爬取到的帖子的详情.*

**def** process\_content(self,content):

*# 把\xa0和空格都替换为空字符串*

content = [re.sub(**r"\xa0|\s"**,**""**,i) **for** i **in** content]

*# 去除列表中的空字符串*

content = [i **for** i **in** content **if** len(i)>0]

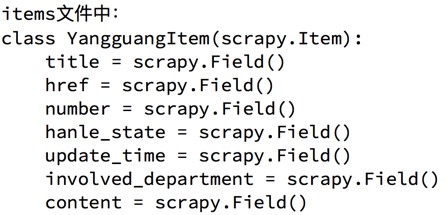
**return** content

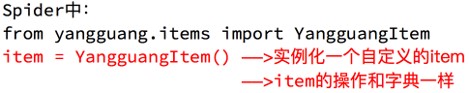
对此爬虫的总结,

1. 需要在详情页中获取下一列的url信息, 然后构建request对象, 并且指定下一页的url的响应交由parse函数来处理, 即递归调用parse函数.

2. 列表页和详情页的处理方法不同, 需要另外构建一个解析函数专门处理详情页的内容. 因为在列表页和详情页中都有部分帖子的信息, 所以需要把列表列解析函数parse中获取的关于帖子信息的item通过meta参数传递给详情页的解析函数parse\_detail中.

3. scrapy中包含着twisted异步网络框架, 所以parse\_detail和parse这两个解析函数的处理很可能是同时进行的.





### 在不同的解析函数中传递参数



### Scrapy深入之认识程序的debug信息

把爬虫yg.py中的allowed\_domins修改为['sun07691'], 同时有settings.py中注释掉LOG\_LEVEL = "WARNING", 再次运行爬虫, 就会在debug信息中看到提示, Filtered offsite request to 'wz.sun0769.com', 即由于请求的地址不符合定义的域名, 所以这个地址就被过滤掉了. 如果程序在运行的过程中出现并没有报错, 也没有输出任何信息, 这时就应该把debug打开, 看一下debug的信息, 可能就会找到问题所在.

爬虫开启

2018-05-22 19:08:04 [scrapy.utils.log] INFO: Scrapy 1.5.0 started (bot: yangguang)

使用的模块的信息

2018-05-22 19:08:04 [scrapy.utils.log] INFO: Versions: lxml 4.2.1.0, libxml2 2.9.5, cssselect 1.0.3, parsel 1.4.0, w3lib 1.19.0, Twisted 17.9.0, Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit (AMD64)], pyOpenSSL 17.5.0 (OpenSSL 1.1.0h 27 Mar 2018), cryptography 2.2.2, Platform Windows-10-10.0.16299-SP0

覆盖程序默认的设置, settings中自定义的设置都会出现在这里

2018-05-22 19:08:04 [scrapy.crawler] INFO: Overridden settings: {'BOT\_NAME': 'yangguang', 'NEWSPIDER\_MODULE': 'yangguang.spiders', 'ROBOTSTXT\_OBEY': True,

设置USER\_AGENT

'SPIDER\_MODULES': ['yangguang.spiders'], 'USER\_AGENT': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'}

Enabled extensions, Enabled downloader middlewares, Enabled spider middlewares都是爬虫默认开启的一些功能.

extensions即一些插件. 如开启了telnet功能, 在下面的item pipelines中可以看到2018-05-22 19:08:04 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023, 可以通过这个地址和端口来对程序进行高度, 如控制爬虫的运行, 使其暂停或继续.

2018-05-22 19:08:04 [scrapy.middleware] INFO: Enabled extensions:

['scrapy.extensions.corestats.CoreStats',

'scrapy.extensions.telnet.TelnetConsole',

'scrapy.extensions.logstats.LogStats']

2018-05-22 19:08:04 [scrapy.middleware] INFO: Enabled downloader middlewares:

['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',

'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',

超时处理

'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',

'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',

useragent

'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',

请求失败时重新尝试

'scrapy.downloadermiddlewares.retry.RetryMiddleware',

重定向功能

'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',

'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',

'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',

'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',

'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',

'scrapy.downloadermiddlewares.stats.DownloaderStats']

2018-05-22 19:08:04 [scrapy.middleware] INFO: Enabled spider middlewares:

['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',

'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',

'scrapy.spidermiddlewares.referer.RefererMiddleware',

'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',

'scrapy.spidermiddlewares.depth.DepthMiddleware']

2018-05-22 19:08:04 [scrapy.middleware] INFO: Enabled item pipelines:

因为只定义了一个pipeline, 所以这里就显示了1个. 如果定义了多个, 都会在这里显示出来. 在settings.py中进行查看, 还有SPIDER\_MIDDLEWARES, DOWNLOADER\_MIDDLEWARES, EXTENSIONS等选项, 添加了某个属性, 也会显示在对应模块的对应位置处. 除了自定义的之外, scrapy默认还启动了很多默认的来帮助我们实现相应的功能.

['yangguang.pipelines.YangguangPipeline']

爬虫开启

2018-05-22 19:08:04 [scrapy.core.engine] INFO: Spider opened

爬虫开启之后就开始进行统计, 统计抓取的页面数和item的数量

2018-05-22 19:08:04 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)

Telnet插件, 可以用来对程序进行调试

2018-05-22 19:08:04 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023

因为在settings.py中设置了遵守robots协议, 所以程序先去查找网站的rebots.txt文件, 因为没有查找到robots.txt, 所以会重定向到404.html

2018-05-22 19:08:04 [scrapy.downloadermiddlewares.redirect] DEBUG: Redirecting (302) to <GET http://www.sun0769.com/error/404.htm> from <GET <http://wz.sun0769.com/robots.txt>>

请求404.htm页面, 200表示请求成功了.

2018-05-22 19:08:04 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://www.sun0769.com/error/404.htm> (referer: None)

开始请求start\_urls中定义的url地址

2018-05-22 19:08:04 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://wz.sun0769.com/index.php/question/questionType?type=4&page=0> (referer: None)

在第1页中提取到详情页的url地址时, 会构建request对象, 发送请求, 此时会对url地址进行域名的过滤, 以防止提取到站外的内容. 因为与定义的允许的域名不符合, 就会被过滤掉. 出现一个debug信息. allowed\_domain中定义的域名是为了防止爬虫在运行的过程中提取到站外的url地址, 要想提取到url地址, 必须要首先至少请求一个页面, 所以scrapy中不会对start\_url进行域名的过滤检查. 所以入口url可以不满足定义的域名, 只要其中提取的url满足域名的要求即可.

2018-05-22 19:08:05 [scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'wz.sun0769.com': <GET <http://wz.sun0769.com/html/question/201805/371843.shtml>>

正在关闭爬虫

2018-05-22 19:08:05 [scrapy.core.engine] INFO: Closing spider (finished)

一些统计信息

2018-05-22 19:08:05 [scrapy.statscollectors] INFO: Dumping Scrapy stats:

{'downloader/request\_bytes': 969,

'downloader/request\_count': 3,

'downloader/request\_method\_count/GET': 3,

'downloader/response\_bytes': 33947,

'downloader/response\_count': 3,

'downloader/response\_status\_count/200': 2,

'downloader/response\_status\_count/302': 1,

'finish\_reason': 'finished',

'finish\_time': datetime.datetime(2018, 5, 22, 11, 8, 5, 81719),

'log\_count/DEBUG': 5,

'log\_count/INFO': 7,

'offsite/domains': 1,

'offsite/filtered': 31,

'request\_depth\_max': 1,

'response\_received\_count': 2,

出调度器的次数

'scheduler/dequeued': 1,

'scheduler/dequeued/memory': 1,

入调度器的次数

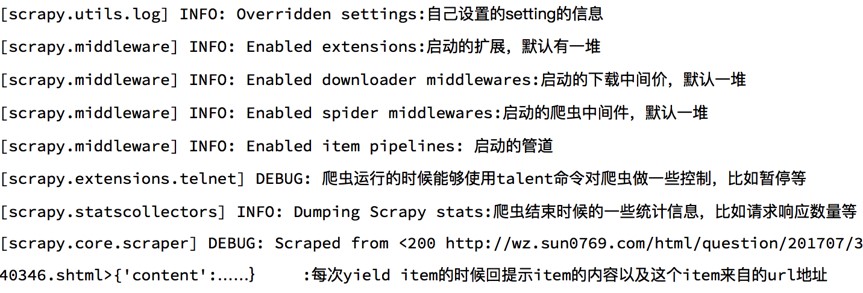
'scheduler/enqueued': 1,

'scheduler/enqueued/memory': 1,

'start\_time': datetime.datetime(2018, 5, 22, 11, 8, 4, 541837)}

2018-05-22 19:08:05 [scrapy.core.engine] INFO: Spider closed (finished)

主要关注其中的错误信息, 请求是否成功, 是否被重定向, 或一些debug错误信息.



### Scrapy深入之scrapy shell

Scrapy shell是一个交互终端，我们可以在未启动spider的情况下尝试及调试代码，也可以用来测试XPath表达式

使用方法：

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

[s] Available Scrapy objects:

[s] scrapy scrapy module (contains scrapy.Request, scrapy.Selector, etc)

[s] crawler <scrapy.crawler.Crawler object at 0x00000256B2120080>

[s] item {}

[s] request <GET http://www.itcast.cn/channel/teacher.shtml>

[s] response <200 http://www.itcast.cn/channel/teacher.shtml>

[s] settings <scrapy.settings.Settings object at 0x00000256B48138D0>

[s] spider <DefaultSpider 'default' at 0x256b4c74cf8>

[s] Useful shortcuts:

[s] fetch(url[, redirect=True]) Fetch URL and update local objects (by default, redirects are followed)

[s] fetch(req) Fetch a scrapy.Request and update local objects

[s] shelp() Shell help (print this help)

[s] view(response) View response in a browser

response.url：当前响应的url地址

response.request.url：当前响应对应的请求的url地址

response.headers：响应头

response.requests.headers：当前响应的请求头

response.text: 响应的str类型的html字符串

response.body：响应体，也就是html代码，默认是bytes类型. 网页的二进制响应

response.body.decode(): 变为str类型的html字符串

response.body\_as\_unicode(): 以unicode编码显示的响应内容

也可以使用xpath对结果进行调试

response.xpath("//h3/text()").extract()

spider的一些方法

spider.name

spider.log("\*"\*100)

2018-05-22 19:51:36 [default] DEBUG: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

response.request.headers

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

b'Accept-Language': b'en',

b'User-Agent': b'Scrapy/1.5.0 (+https://scrapy.org)',

b'Accept-Encoding': b'gzip,deflate'}

也可以在pycharm中进行查看. 点击

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

中的Spider, 就会进入到Spider的源码中, 然后点项目边框中的scroll from Source, 就会跳转到Spider类对应的文件位置处. 可以通过源码的方式来查看其中的方法和属性.

### Scrapy深入之认识setting文件

为什么需要配置文件

配置文件存放一些公式的变量, 比如数据库的地址, 账号密码等, 方便自己和别人修改. 达到的效果就是只需要在settings中定义或修改一次, 其它所有的地方都不需要再进行修改了, 或者都可以从settings中调用其中设置的变量了.

一般用全大写字母命名变量, 如SQL\_HOST = "192.168.0.1"

settings.py中选项的含义

# 项目名称

BOT\_NAME = **'yangguang'**

# 爬虫的位置

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

# 新建的爬虫保存的位置

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

# LOG\_LEVEL = "WARNING"

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

USER\_AGENT = **'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'**

# 是否遵守robots协议, 默认情况下遵守, 此时会先请求网站的robots.txt文件.

# Obey robots.txt rules

ROBOTSTXT\_OBEY = **True**

# 最大并发请求的数量, 越大时爬取的越快, 但也越容易被识别为爬虫.

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

#CONCURRENT\_REQUESTS = 32

# 下载延迟.

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

# See https://doc.scrapy.org/en/latest/topics/settings.html#download-delay

# See also autothrottle settings and docs

#DOWNLOAD\_DELAY = 3

# download delay只可以配合下面的两者之一进行使用

# The download delay setting will honor only one of:

# 每个域名的最大并发请求数量

#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16

# 每个IP的最大并发数量

# CONCURRENT\_REQUESTS\_PER\_IP = 16

# 是否开启cookies, 默认是开启的

# Disable cookies (enabled by default)

#COOKIES\_ENABLED = False

# 打开Telnet console这个插件, 可以使用它来对程序进行调试

# Disable Telnet Console (enabled by default)

#TELNETCONSOLE\_ENABLED = False

# 默认的请求头, 注意不能把USER\_AGENT放在请求头中, 这里的设置类似于在构建request对象时的cookies处理方式, 不能放在headers中.

# Override the default request headers:

#DEFAULT\_REQUEST\_HEADERS = {

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

# 'Accept-Language': 'en',

#}

# 爬虫中间件

# Enable or disable spider middlewares

# See https://doc.scrapy.org/en/latest/topics/spider-middleware.html

#SPIDER\_MIDDLEWARES = {

# 'yangguang.middlewares.YangguangSpiderMiddleware': 543,

#}

# 下载中间件

# Enable or disable downloader middlewares

# See https://doc.scrapy.org/en/latest/topics/downloader-middleware.html

#DOWNLOADER\_MIDDLEWARES = {

# 'yangguang.middlewares.YangguangDownloaderMiddleware': 543,

#}

# 爬虫中间件, 下载中间件, 插件. 这三者的使用与pipelines的使用类似. 都是指定位置, 指定权重. 开启

# Enable or disable extensions

# See https://doc.scrapy.org/en/latest/topics/extensions.html

#EXTENSIONS = {

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

#}

# Configure item pipelines

# See https://doc.scrapy.org/en/latest/topics/item-pipeline.html

ITEM\_PIPELINES = {

**'yangguang.pipelines.YangguangPipeline'**: 300,

}

# 自动限速插件, 默认是关闭的

# Enable and configure the AutoThrottle extension (disabled by default)

# See https://doc.scrapy.org/en/latest/topics/autothrottle.html

#AUTOTHROTTLE\_ENABLED = True

# The initial download delay

#AUTOTHROTTLE\_START\_DELAY = 5

# The maximum download delay to be set in case of high latencies

#AUTOTHROTTLE\_MAX\_DELAY = 60

# The average number of requests Scrapy should be sending in parallel to

# each remote server

#AUTOTHROTTLE\_TARGET\_CONCURRENCY = 1.0

# Enable showing throttling stats for every response received:

#AUTOTHROTTLE\_DEBUG = False

# HTTP缓存, 默认是关闭的

# Enable and configure HTTP caching (disabled by default)

# See https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#httpcache-middleware-settings

#HTTPCACHE\_ENABLED = True

#HTTPCACHE\_EXPIRATION\_SECS = 0

#HTTPCACHE\_DIR = 'httpcache'

#HTTPCACHE\_IGNORE\_HTTP\_CODES = []

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

**Writing your own item pipeline**

Each item pipeline component is a Python class that must implement the following method:

process\_item用来接收从spider中传递过来的数据, 进行数据的处理和保存.

**process\_item(self, item, spider)**

This method is called for every item pipeline component. [**process\_item()**](https://doc.scrapy.org/en/latest/topics/item-pipeline.html#process_item) must either: return a dict with data, return an [**Item**](https://doc.scrapy.org/en/latest/topics/items.html#scrapy.item.Item) (or any descendant class) object, return a [Twisted Deferred](https://twistedmatrix.com/documents/current/core/howto/defer.html) or raise [**DropItem**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.DropItem) exception. Dropped items are no longer processed by further pipeline components.

|  |  |
| --- | --- |
| Parameters: | **item** ([Item](https://doc.scrapy.org/en/latest/topics/items.html#scrapy.item.Item) object or a dict) – the item scraped  **spider** ([Spider](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider which scraped the item |

Additionally, they may also implement the following methods:

在spider开启的时候执行open\_spider方法. 在保存数据时要把数据保存到数据库中, 此时就要建立与数据库的连接, 数据库的连接不能建立在process\_item方法中, 否则每一个item过来都会重新建立一个到数据库的连接. 但可以把建立连接的过程放在open\_spider方法中, 当爬虫开启的时候就会建立数据库的连接, 并且只会执行一次. 同时在close\_spider中关闭连接.

**open\_spider(self, spider)**

This method is called when the spider is opened.

|  |  |
| --- | --- |
| **Parameters:** | **spider** ([**Spider**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider which was opened |

**在spider关闭的时候执行close\_spider方法**

**close\_spider(self, spider)**

This method is called when the spider is closed.

|  |  |
| --- | --- |
| **Parameters:** | **spider** ([**Spider**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider which was closed |

在settings.py中进行了设置, 如何在程序中使用呢.

如在settings.py中添加了保存到mongodb中的地址.

MONGO\_HOST = **"localhost"**

想要在spider中使用settings.py中设置的内容. 只需要导入就能使用.

**from** yangguang.settings **import** MONGO\_HOST

同样, 在piplines.py中也可以导入settings.py中的内容

**from** yangguang.settings **import** MONGO\_HOST

还有一种更简单的方法, class YgSpider(scrapy.Spider)中的Spider本身就有一个settings的属性, 可以直接使用如下的方法, 这样就不用再导入settings文件了.

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

# self.settings[**"MONGO\_HOST"**]

# 使用get时如果不存在这个键, 就会返回None, 可以设置为空字符串. 这里的self就是spider, 即当前爬虫的实例.

self.settings.get(**"MONGO\_HOST"**,**""**)

print(self.hello,**"\*"**\*100)

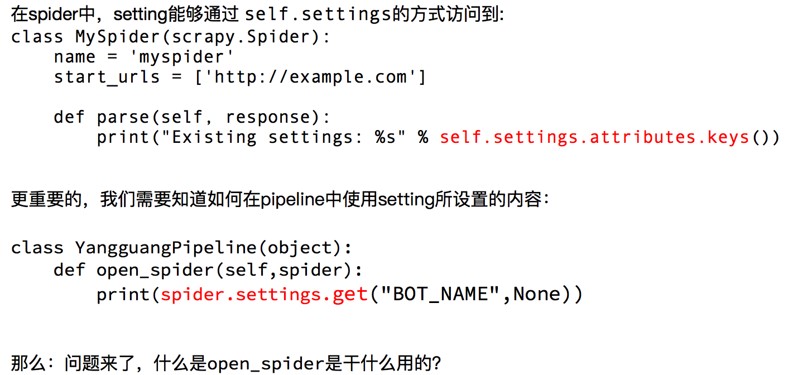
如果要在pipelines.py用类似的方法, 就要使用spider.settings

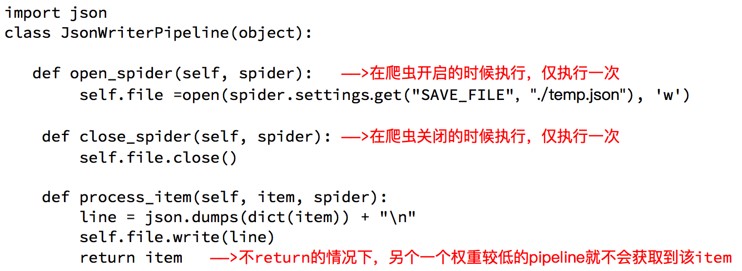
**def** process\_item(self, item, spider):

spider.settings.get(**"MONGO\_HOST"**)

item[**"content"**] = self.process\_content(item[**"content"**])

print(item)





open\_spider方法和close\_spider方法都只执行一次, 这样设置的好处是可以把连接服务器和关闭服务器连接分别放在open\_spider和close\_spider中. 还可以给程序设置一个初始化的值. 如

在pipelines.py中设置open\_spider方法, 给spider设置一个键.

**class** YangguangPipeline(object):

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

spider.hello = **"world"**

client = MongoClient()

self.collection = client[**"test"**][**"test"**]

这样, 在执行到spider的爬虫时, spider中就有一个hello的键, 就可以直接在spider中获取到这个键.

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

name = **'yg'**

allowed\_domains = [**'sun07691.com'**]

*# start\_urls = ['http://wz.sun0769.com/index.php/question/questionType?type=4&page=0']*

start\_urls = [**'http://www.baidu.com'**]

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

self.settings[**"MONGO\_HOST"**]

self.settings.get(**"MONGO\_HOST"**,**""**)

print(self.hello,**"\*"**\*100)

可以再次设置域名为["sun06791.com"], 这样程序就不会继续提取信息了. 运行spider, 就可以看到打印的world和 \* 了.

更多的是把连接服务器和关闭服务器的功能放在pipelines.py中的open\_spider方法和close\_spider方法中. 这样与之前把创建数据库连接的代码放在外面的作用是完全一样的, 但把创建数据库连接放在open\_spider方法中, 就可以直接在其它方法中使用self来调用, 更加方法, 并且代码也更加美观.

**import** re

*# from yangguang.settings import MONGO\_HOST*

**from** pymongo **import** MongoClient

**class** YangguangPipeline(object):

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

spider.hello = **"world"**

*# 实例化MongoClient对象*

client = MongoClient()

*# 创建数据库和collection*

self.collection = client[**"test"**][**"test"**]

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

spider.settings.get(**"MONGO\_HOST"**)

item[**"content"**] = self.process\_content(item[**"content"**])

print(item)

*# 把数据转化为dict之后保存到数据库中*

self.collection.insert(dict(item))

**return** item

**def** process\_content(self,content):

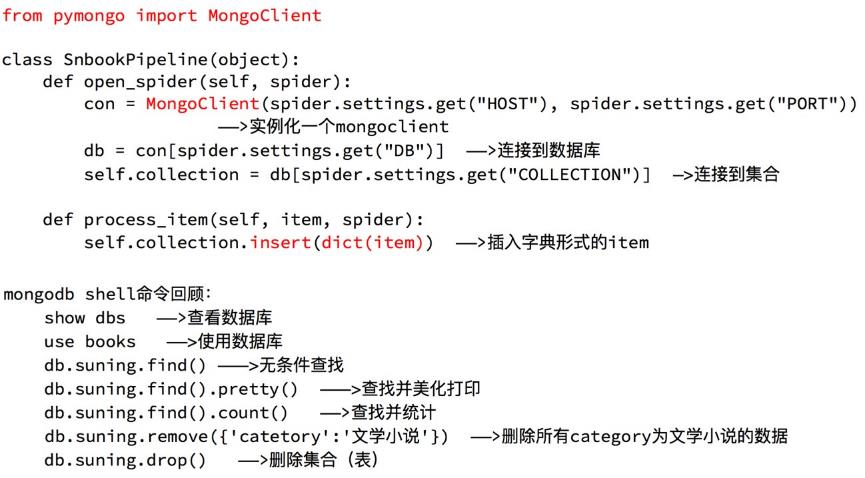
content = [re.sub(**r"\xa0|\s"**,**""**,i) **for** i **in** content]

content = [i **for** i **in** content **if** len(i)>0] *#去除列表中的空字符串*

**return** content

既然open\_spider和close\_spider中的代码都只执行一次, 那是否就可以把打开文件的代码写在open\_spider中, 把关闭文件的代码写到close\_spider中, 只间不停的向文件中写入内容呢. 这种方法是可以的, 但是会有一定的风险的, 如果程序中间出错, 以前要保存的数据是不会写入到文件中的, 因为对文件的写入只有当文件关闭的时候才真正写入到文件中. 但是数据库的操作却不存在这种风险, 因为mongodb使用insert可以直接把数据保存到数据库中, 而mysql有commit操作, 也可以手动把数据保存到数据库中.

### Mongodb回顾



### 苏宁图书爬虫

需求：爬取苏宁易购所有下所有图书和图书分类信息，以及子链接页面的价格内容。

url : http://snbook.suning.com/web/trd-fl/999999/0.htm

目标：熟悉前面的知识点

爬取图书对应的大分类, 小分类, 列表页图书的详情信息, 以及图书详情页的价格信息.

在爬取信息时要先进行分组, 可以先根据大分类进行分组, 按照如下的顺序进行爬取:

每个大分类下面的小分类, 每个小分类下面的列表页, 每个列表页下面的详情页,

从起始url中获取每一个大分类和大分类下每一个小分类, 然后请求每个小分类下面的列表页.

要从列表页中获取每一本图书的信息, 列表页中每一本图书也做为一个分类 > 爬取列表页图书的内容, 之后还要进入到图书的详情页中, 爬取价格, 这样一本书的信息才完全获取到.

一页的信息爬取完成后还要进行翻页, 接着爬取下一页的信息.

图书的信息中要包含自己所属大分类和小分类的信息.

程序的入口应该是各个大分类所在的页面, 由大分类获取各个小分类的url地址, 再请求各个小分类的url地址, 从小分类的响应中获取小分类对应的列表页中的图书信息, 还需要进入到每一本图书的详情页中获取价格信息.

测试start\_url, 看它是否包含了我们想要的数据. 测试得出响应和元素中网页的结构是相同的, 所以就可以使用元素中的网页结构写xpath提取规则了.

scrapy startproject book

cd book

scrapy genspider suning suning.com

修改suning.py, 修改start\_urls地址, 并提取响应的信息.

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

**import** scrapy

**import** re

**from** copy **import** deepcopy

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

name = **'suning'**

allowed\_domains = [**'suning.com'**]

start\_urls = [**'http://snbook.suning.com/web/trd-fl/999999/0.htm'**]

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

*#1.大分类分组*

li\_list = response.xpath(**"//ul[@class='ulwrap']/li"**)

*# 遍历获取每个大分类下的内容*

**for** li **in** li\_list:

item = {}

item[**"b\_cate"**] = li.xpath(**"./div[1]/a/text()"**).extract\_first()

*#2.小分类分组*

a\_list = li.xpath(**"./div[2]/a"**)

*# 遍历获取每个小分类下的内容*

**for** a **in** a\_list:

*# 小分类的url地址和名称*

item[**"s\_href"**] = a.xpath(**"./@href"**).extract\_first()

item[**"s\_cate"**] = a.xpath(**"./text()"**).extract\_first()

*# 使用extract\_first时, 如果没有获取到内容, 会返回None值, 如果不是None值, 就构建request对象, 使用yield传递给scrapy引擎, 进一步获取小分类的列表页.*

**if** item[**"s\_href"**] **is not None**:

item[**"s\_href"**]= **"http://snbook.suning.com/"** + item[**"s\_href"**]

**yield** scrapy.Request(

item[**"s\_href"**],

callback=self.parse\_book\_list,

*# 需要使用meta把item的信息传递过去*

meta = {**"item"**:deepcopy(item)}

)

*# 定义获取小分类列表页的解析函数*

**def** parse\_book\_list(self,response):

*# 获取item信息*

item = deepcopy(response.meta[**"item"**])

*# 图书列表页分组.*

*# 同样也要先确定列表页的响应中包含着图书的信息. 然后才能获取信息. 如果图书列表页的响应中没有包含图书的信息, 就要修改parse中的yield, 传递真正返回图书信息的url地址到parse\_book\_list函数中.*

*# 因为clearfix是用来清除一些默认样式的, 并不是用来区分每个标签的, 所以很可能不是唯一的, 就要取li[@class='clearfix']的上一级标签. 确定响应中包含图书的信息, 如果响应与元素中的内容相同, 可以使用检查元素中的网页结构来写xpath规则.*

li\_list = response.xpath(**"//div[@class='filtrate-books list-filtrate-books']/ul/li"**)

*# 遍历图书列表获取每一本书的信息*

**for** li **in** li\_list:

item[**"book\_name"**] = li.xpath(**".//div[@class='book-title']/a/@title"**).extract\_first()

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src"**).extract\_first()

*# 有的图书图片的信息是在src2中的, 所以要进行判断, 如果src中没有获取到信息, 就去获取src2中的信息.*

**if** item[**"book\_img"**] **is None**:

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src2"**).extract\_first()

item[**"book\_author"**] = li.xpath(**".//div[@class='book-author']/a/text()"**).extract\_first()

item[**"book\_press"**] = li.xpath(**".//div[@class='book-publish']/a/text()"**).extract\_first()

item[**"book\_desc"**] = li.xpath(**".//div[@class='book-descrip c6']/text()"**).extract\_first()

item[**"book\_href"**]= li.xpath(**".//div[@class='book-title']/a/@href"**).extract\_first()

*# 获取到图书的详情页的url时要构建request对象, 指定此url的响应由parse\_book\_detail解析函数进行解析. 同时要通过meta把item传递过去.*

**yield** scrapy.Request(

item[**"book\_href"**],

callback=self.parse\_book\_detail,

meta = {**"item"**:deepcopy(item)}

)

*# 定义图书详情页响应的解析函数*

**def** parse\_book\_detail(self,response):

item = response.meta[**"item"**]

*# 价格是在script标签听, 可以使用re来进行提取. 要从响应的字符串中进行提取, 所以要使用response.body.decode(), 在re的匹配规则中, 如果有双引号, 就需要进行转义.*

item[**"book\_price"**] = re.findall(**"\"bp\":'(.\*?)',"**,response.body.decode())

*# 判断获取到的book\_price, 如果列表的长度大于0, 就提取第0个元素, 否则就设置为None*

item[**"book\_price"**] = item[**"book\_price"**][0] **if** len(item[**"book\_price"**])>0 **else None**

print(item)

在settings.py中设置LOG\_LEVEL="WARNING", 并且添加USER\_AGENT

LOG\_LEVEL = "WARNING"

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

USER\_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'

启动爬虫进行测试, 发现很多book\_img都为None, 打开为None值的book\_url, 检查response, 发现有的图书的图片是放在src2中的, 所以要在spider中进行处理, 如果src中获取的图书的图片是None, 就在src2中进行获取.

**for** li **in** li\_list:

item[**"book\_name"**] = li.xpath(**".//div[@class='book-title']/a/@title"**).extract\_first()

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src"**).extract\_first()

*# 有的图书图片的信息是在src2中的, 所以要进行判断, 如果src中没有获取到信息, 就去获取src2中的信息.*

**if** item[**"book\_img"**] **is None**:

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src2"**).extract\_first()

item[**"book\_author"**] = li.xpath(**".//div[@class='book-author']/a/text()"**).extract\_first()

在打印出来的信息中, 还会发现一个问题, 就是很多图书的信息, 除了价格不同之外, 其它的内容都是重复的, 这是因为在parse()方法中, 对于a\_list会进行多次的循环, 每次循环都会构造一个request请求对象, 交给引擎进行调度, 同时把同一个item字典传递给parse\_book\_list解析函数, 由于scrapy使用的是异步的网络框架, 下一次的请求不会等待上一次请求的结束, 而是直接开始, 由于在循环中使用的是同一个item, 如果在上一个循环把item交给parse\_book\_list处理了, parse\_book\_list的处理尚未结束时下一个循环中响应的数据已经写入到item字典中, 就会把上一次的item中的内容覆盖掉, 同样, 如果parse\_book\_list中的item发生了变化, 也可能会把parse中的item修改掉, 这样就造成了结果中内容的重复, 同样, parse\_book\_list中的循环也存在着类似的问题. 而在request模块的爬虫中, 因为request对请求的处理不是异步的, 而是使用的单线程, 下一次的请求必须要等待上一次的请求结束才能进行, 所以不会出现类似scrapy中的下一次的数据把上一次的数据覆盖的问题.

要解决上面的问题, 就要使用深拷贝.

a = {"a":100,"b":20}

b=a

b

b["a"]=10

b

a

from copy import deepcopy

c=deepcopy(a)

c

c["a"]=50

c

a

b

在直接使用b=a对b赋值时, 只是把a的引用地址赋值给了a, b和a指向同一个引用地址. 而使用deepcopy时, 是在内存中另外开辟了一块空间, 把a中的所有内容复制到这块内存空间中, 然后把b指向新内存空间的引用. 所以deepcopy相当于是内容的拷贝, 这样就不会出现内容覆盖的问题了.

在程序中使用deepcopy, 再次运行, 就不会再出现前后内容重复的现象了.

问题是在之前的腾讯招聘和阳光政务平台的爬虫中为什么没有出来前后内容重复的现在呢. 在上两个爬虫中, 是对每一个帖子的内容进行分组的, 每一个帖子使用一个item, 每一个循环爬取一个帖子的内容, 上一个循环与下一个循环使用的是不同的item, 所以就不会出现下一个循环中的item把上一个循环中的覆盖掉的问题了. 而在suning图书爬虫中, 是对每一 个大分类进行分组的, 只在每一个大分类下新建了一个item, 所以同一个大分类下的所有小分类, 以及所有小分类下面的所有图书都是共用一个item的, 如果不使用deepcopy, 肯定会发生重复的现象, 一旦一个item发生变化, 同一大分类下的其它item都会变化为最后发生变化的那个值. 这就是上面的问题的来源.

解决了上面的问题, 还要给爬虫添加翻页的功能.

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

**import** scrapy

**import** re

**from** copy **import** deepcopy

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

name = **'suning'**

allowed\_domains = [**'suning.com'**]

start\_urls = [**'http://snbook.suning.com/web/trd-fl/999999/0.htm'**]

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

*#1.大分类分组*

li\_list = response.xpath(**"//ul[@class='ulwrap']/li"**)

*# 遍历获取每个大分类下的内容*

**for** li **in** li\_list:

item = {}

item[**"b\_cate"**] = li.xpath(**"./div[1]/a/text()"**).extract\_first()

*#2.小分类分组*

a\_list = li.xpath(**"./div[2]/a"**)

*# 遍历获取每个小分类下的内容*

**for** a **in** a\_list:

*# 小分类的url地址和名称*

item[**"s\_href"**] = a.xpath(**"./@href"**).extract\_first()

item[**"s\_cate"**] = a.xpath(**"./text()"**).extract\_first()

*# 使用extract\_first时, 如果没有获取到内容, 会返回None值, 如果不是None值, 就构建request对象, 使用yield传递给scrapy引擎, 进一步获取小分类的列表页.*

**if** item[**"s\_href"**] **is not None**:

item[**"s\_href"**]= **"http://snbook.suning.com/"** + item[**"s\_href"**]

**yield** scrapy.Request(

item[**"s\_href"**],

callback=self.parse\_book\_list,

*# 需要使用meta把item的信息传递过去*

meta = {**"item"**:deepcopy(item)}

)

*# 定义获取小分类列表页的解析函数*

**def** parse\_book\_list(self,response):

*# 获取item信息*

item = deepcopy(response.meta[**"item"**])

*# 图书列表页分组.*

*# 同样也要先确定列表页的响应中包含着图书的信息. 然后才能获取信息. 如果图书列表页的响应中没有包含图书的信息, 就要修改parse中的yield, 传递真正返回图书信息的url地址到parse\_book\_list函数中.*

*# 因为clearfix是用来清除一些默认样式的, 并不是用来区分每个标签的, 所以很可能不是唯一的, 就要取li[@class='clearfix']的上一级标签. 确定响应中包含图书的信息, 如果响应与元素中的内容相同, 可以使用检查元素中的网页结构来写xpath规则.*

li\_list = response.xpath(**"//div[@class='filtrate-books list-filtrate-books']/ul/li"**)

*# 遍历图书列表获取每一本书的信息*

**for** li **in** li\_list:

item[**"book\_name"**] = li.xpath(**".//div[@class='book-title']/a/@title"**).extract\_first()

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src"**).extract\_first()

*# 有的图书图片的信息是在src2中的, 所以要进行判断, 如果src中没有获取到信息, 就去获取src2中的信息.*

**if** item[**"book\_img"**] **is None**:

item[**"book\_img"**] = li.xpath(**".//div[@class='book-img']//img/@src2"**).extract\_first()

item[**"book\_author"**] = li.xpath(**".//div[@class='book-author']/a/text()"**).extract\_first()

item[**"book\_press"**] = li.xpath(**".//div[@class='book-publish']/a/text()"**).extract\_first()

item[**"book\_desc"**] = li.xpath(**".//div[@class='book-descrip c6']/text()"**).extract\_first()

item[**"book\_href"**]= li.xpath(**".//div[@class='book-title']/a/@href"**).extract\_first()

*# 获取到图书的详情页的url时要构建request对象, 指定此url的响应由parse\_book\_detail解析函数进行解析. 同时要通过meta把item传递过去.*

**yield** scrapy.Request(

item[**"book\_href"**],

callback=self.parse\_book\_detail,

meta = {**"item"**:deepcopy(item)}

)

*#翻页. 当点击下一页时, 整个页面没有刷新, 只是局部的刷新, 所以这是ajax的请求. 点击下一页之后, 在network中找到/0.htm?pageNumber=2&sort=0, 查看它的url地址. http://snbook.suning.com/web/trd-fl/999999/0.htm?pageNumber=2&sort=0, pageNumber就是每一页的页码, 这样就可以进行翻页了.*

*# 什么时候向下翻页, 什么时候已经是最后页, 不需要翻页了呢, 在response中查看script标签中有一个var pagecount=5, currentPage=2. 表示当前的分类有5页, 当前是第2页. 所以只需要提取出这两个值, 判断currentPage是否小于pagecount, 如果小于, 就让页码加1, 相当于请求下一页. 很多网站下一页的url地址都与此类似是通过js生成的, 在原页面的下一页中检查元素, div标签下是空值.*

*# 总页码数*

page\_count = int(re.findall(**"var pagecount=(.\*?);"**,response.body.decode())[0])

*# 当前所在页数*

current\_page = int(re.findall(**"var currentPage=(.\*?);"**,response.body.decode())[0])

**if** current\_page<page\_count:

*# 构造下一页的url地址, 只需要在小分类的地址后面加上页码即可*

next\_url = item[**"s\_href"**] +**"?pageNumber={}&sort=0"**.format(current\_page+1)

*# 构造下一页的request请求对象, 其中指定解析函数为parse\_book\_list, 即对图书列表页进行解析的函数, 即上一下与下一页的处理方式相同. 注意由于每一个大分类下的小分类都是使用的同一个item, 还需要把item传递给下一页的解析函数, 否则得到的数据就是不完整的. 从另一个角度来看, 构建的request请求对象得到的响应是由parse\_book\_list进行处理的, 这个函数的第一步就是从meta信息中取出来item, 如果下一页的请求中不传递item, 就会报错, 所以从这个角度来理解, 也需要把item通过meta构建在request请求对象中. 因为parse\_book\_list这个解析函数在解析图书列表中的数据, 如同在解析图书列表页第一页的数据时类似, 需要传递大小分类以及小分类的url地址信息, 由于item中包含有当前页面所在的大分类小分类以及小分类对应的url地址信息, 所以需要所item传递过去, 否则就无法正确进行下一步的操作.*

**yield** scrapy.Request(

next\_url,

callback=self.parse\_book\_list,

*# 在parse\_book\_list这个解析函数第一次被调用的时候, item传递过来的是包含大分类,小分类以及小分类的url地址信息这三个键, 经过parse\_book\_list中代码的处理, 给item添加了一些键或字段, 但是大小分类以及小分类的url信息还是存在的, 所以这里可以直接使用经过本解析函数处理后的item, 在下一次调用parse\_book\_list解析函数时, 上一次经过parse\_book\_list添加的信息会被覆盖掉. 因为在处理下一页的请求时, 只需要有大分类小分类以及小分类对应的url地址, 所以也可以使用程序最初的item. 如果这里要使用程序最初接收到的item, 可以从response.meta中去获取, 注意此时要把程序最初获取 item的地方使用deeepcopy来复制数据, 否则下一个循环中生成的item会把上一个循环中的覆盖掉. 一旦在parse\_book\_list解析函数最初使用了deeepcopy, item中就只包含了大小分类及小分类的url地址信息, 这样这里就可以直接使用item这个变量了.*

*# meta = {"item":item}*

meta = {**"item"**: response.meta[**"item"**]}

)

*# 定义图书详情页响应的解析函数*

**def** parse\_book\_detail(self,response):

item = response.meta[**"item"**]

*# 价格是在script标签听, 可以使用re来进行提取. 要从响应的字符串中进行提取, 所以要使用response.body.decode(), 在re的匹配规则中, 如果有双引号, 就需要进行转义.*

item[**"book\_price"**] = re.findall(**"\"bp\":'(.\*?)',"**,response.body.decode())

*# 判断获取到的book\_price, 如果列表的长度大于0, 就提取第0个元素, 否则就设置为None*

item[**"book\_price"**] = item[**"book\_price"**][0] **if** len(item[**"book\_price"**])>0 **else None**

print(item)

整个程序的思路是非常明确的, 先对大分类进行分组, 然后对大分类中的每一个元素进行循环, 提取每一个大分类下面对应的小分类的url地址. 对每一个小分类也要进行分组, 并使用循环提取每一个小分类的名称和url地址信息, 然后构造每个小分类url地址的request请求对象, 把小分类的url地址和item数据传递过去. 由于下一次的循环会影响上一次循环中的item信息, 所以需要使用deepcopy来完整拷贝item的数据并传递给列表页的解析函数parse\_book\_list中. 当小分类的url地址响应传递到列表页的解析函数中之后, 获取到图书的列表, 也需要对图书列表页进行分组, 然后使用循环提取出每一本图书的信息. 由于在列表页中只能提取到图书的一部分信息, 还需要进入到每一本图书的详情页中获取价格信息, 所以还需要构建图书详情页的request请求对象, 把图书详情页的信息和图书信息的item传递给图书详情页的解析函数parse\_book\_detail, 由parse\_book\_detail对图书详情页响应进行处理, 提取价格信息, 并保存到图书的item中. 当一页的数据处理完成后, 还需要实现列表页的翻页, 在翻页时就需要对翻页方式进行判断, 由于下一页的url地址在当前页的响应中没有找到, 就意味着是通过js的方式生成的下一页的信息, 需要找到对应的ajax请求的页面, 找到其中的下一页的url地址, 然后根据下一页url地址的特征组合成下一页的url地址, 再传递给parse\_book\_list进行下一页响应的解析.

## 5. Scrapy中CrawlSpider

回头看之前的代码，我们有很大一部分时间在寻找下一页的url地址或者是内容的url地址上面，这个过程能更简单一些么？

思路：

1、从response中提取所有的a标签对应的url地址

2、自动的构造自己requests请求，发送给引擎

上面的功能可以做的更好：

满足某个条件的url地址，我们才发送给引擎，同时能够指定callback函数

crawlSpider能够根据一定的规则自动提取其中url地址, 然后发送请求获取响应.

对于百度贴吧上的页面, 帖子详情页和翻页的url地址都是有规律的, 只需要使用//href提取出所有页面的url链接, 再使用正则匹配就能分别匹配到这两种地址, 把这两种地址分开, 交给不能的解析函数进行处理.

### 爬取csdn上面所有的博客专家及其文章

Url地址：http://blog.csdn.net/experts.html

目标：通过csdn爬虫了解crawlspider的使用

生成crawlspider的命令：

scrapy genspider –t crawl csdn "csdn.cn"





所以在写rules中的提取规则时, 尽量写的更明确些, 尽量匹配单一类型的url地址. 不要让一个url能同时匹配到多个规则.

### circ帖子爬虫

<http://circ.gov.cn/web/site0/tab5240/>

抓取详情页的帖子标题和发布日期

生成crawlspider的命令：

scrapy genspider –t crawl csdn "csdn.cn"

创建项目

scrapy startproject circ

cd circ

创建爬虫

scrapy genspider -t crawl cf circ.gov.cn

查看cf.py文件

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

**import** scrapy

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

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

**class** CfSpider(CrawlSpider):

name = **'cf'**

allowed\_domains = [**'circ.gov.cn'**]

start\_urls = [**'http://circ.gov.cn/'**]

#

rules = (

Rule(LinkExtractor(allow=**r'Items/'**), callback=**'parse\_item'**, follow=**True**),

)

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

i = {}

*#i['domain\_id'] = response.xpath('//input[@id="sid"]/@value').extract()*

*#i['name'] = response.xpath('//div[@id="name"]').extract()*

*#i['description'] = response.xpath('//div[@id="description"]').extract()*

**return** i

修改爬虫文件, 从中提取url和帖子的信息

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

**import** scrapy

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

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

**import** re

**class** CfSpider(CrawlSpider):

name = **'cf'**

allowed\_domains = [**'circ.gov.cn'**]

*# 确实start\_urls中是否有我们想要的内容*

start\_urls = [**'http://www.circ.gov.cn/web/site0/tab5240/module14430/page1.htm'**]

*#定义提取url地址规则. rules是一个元组, 元组中元素是有顺序的. rules中可以定义多个规则, 每个规则提取一种类型的url地址*

rules = (

*# Rule(), Rule也是一个类, Rule()就相当于实例化一个对象. 相当于定义了一个规则对象.*

*# LinkExtractor 链接提取器，提取url地址*

*# allow=r''中是正则表达式, 用来匹配某一类型的url地址. 提取到的url地址会通过parse函数发送请求. 所以下面的解析函数不能定义为parse, 否则会覆盖系数的parse函数, 就不能够去发送请求了.*

*# callback 提取出来的url地址的response会交给callback处理. 对于有些规则提取器, 不需要有callback函数. 也就是不需要把提取到的链接地址的响应进行处理. 如下一列的url地址, 只需要提取出地址, 默认交给parse函数进行处理即可, parse函数会再次调用Rule从中提取出帖子详情页的url地址, 就不需要指定callback函数了.*

*# follow 当前url地址的响应是否重新使用rules来提取url地址，如果设置为True, 就类似于一个循环的功能, 当前Rule规则提取出来的url地址会继续进入到rules规则中再次依次匹配各个规则, 再次提取url地址. 如在circ爬虫项目中, 对于提取到的下一页的url地址, 它的响应就需要再次进入到rules中匹配并提取url地址, 所以要设置为True. 而提取到的帖子的详情url地址, 就不需要从中提取url地址, 只需要默认使用parse函数处理url的顺应即可. 所以callback和follow要根据实际情况选择是否填写.*

Rule(LinkExtractor(allow=**r'/web/site0/tab5240/info\d+\.htm'**), callback=**'parse\_item'**), *# 提取详情页的url地址, 需要有callback, 不需要再次提取url响应中的url地址*

*#parse函数在crawl spider中有特殊功能，不能使用parse作为函数名*

*# 提取详情页中帖子的内容*

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

item = {}

item[**"title"**] = re.findall(**"<!--TitleStart-->(.\*?)<!--TitleEnd-->"**,response.body.decode())[0]

item[**"publish\_date"**] = re.findall(**"发布时间：(20\d{2}-\d{2}-\d{2})"**,response.body.decode())[0]

print(item)

修改settings.py中的LOG\_LEVEL和USER\_AGENT.

启动爬虫进行测试. scrapy crawl cf

以上只是提取了第一页中的响应, 还需要设置提取下一页的rules规则.

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

**import** scrapy

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

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

**import** re

**class** CfSpider(CrawlSpider):

name = **'cf'**

allowed\_domains = [**'circ.gov.cn'**]

*# 确实start\_urls中是否有我们想要的内容*

start\_urls = [**'http://www.circ.gov.cn/web/site0/tab5240/module14430/page1.htm'**]

*#定义提取url地址规则. rules是一个元组, 元组中元素是有顺序的. rules中可以定义多个规则, 每个规则提取一种类型的url地址*

rules = (

*# Rule(), Rule也是一个类, Rule()就相当于实例化一个对象. 相当于定义了一个规则对象.*

*# LinkExtractor 链接提取器，提取url地址*

*# allow=r''中是正则表达式, 用来匹配某一类型的url地址. 提取到的url地址会通过parse函数发送请求. 所以下面的解析函数不能定义为parse, 否则会覆盖系数的parse函数, 就不能够去发送请求了.*

*# callback 提取出来的url地址的response会交给callback处理. 对于有些规则提取器, 不需要有callback函数. 也就是不需要把提取到的链接地址的响应进行处理. 如下一列的url地址, 只需要提取出地址, 默认交给parse函数进行处理即可, parse函数会再次调用Rule从中提取出帖子详情页的url地址, 就不需要指定callback函数了.*

*# follow 当前url地址的响应是否重新使用rules来提取url地址，如果设置为True, 就类似于一个循环的功能, 当前Rule规则提取出来的url地址会继续进入到rules规则中再次依次匹配各个规则, 再次提取url地址. 如在circ爬虫项目中, 对于提取到的下一页的url地址, 它的响应就需要再次进入到rules中匹配并提取url地址, 所以要设置为True. 而提取到的帖子的详情url地址, 就不需要从中提取url地址, 只需要默认使用parse函数处理url的顺应即可. 所以callback和follow要根据实际情况选择是否填写.*

Rule(LinkExtractor(allow=**r'/web/site0/tab5240/info\d+\.htm'**), callback=**'parse\_item'**), *# 提取详情页的url地址, 需要有callback, 不需要再次提取url响应中的url地址*

Rule(LinkExtractor(allow=**r'/web/site0/tab5240/module14430/page\d+\.htm'**),follow=**True**), *# 提取下一页的url地址, 不需要有callback, 需要再次提取url响应中的地址*

)

*#parse函数在crawl spider中有特殊功能，不能使用parse作为函数名*

*# 提取详情页中帖子的内容*

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

item = {}

item[**"title"**] = re.findall(**"<!--TitleStart-->(.\*?)<!--TitleEnd-->"**,response.body.decode())[0]

item[**"publish\_date"**] = re.findall(**"发布时间：(20\d{2}-\d{2}-\d{2})"**,response.body.decode())[0]

print(item)

通过rules来抓取页面中的url地址, 就不需要我们太去关心如何去提取页面中的url地址, 如何去发送请求的问题了.

在以上的项目中, 我们要提取的信息全部来自帖子的详情页, 所以只需要在parse\_item中定义提取数据的规则, 进行数据的提取即可. 如果信息的一部分来自详情页, 一部分来自列表页, 就可以在parse\_item中再手动构建一个request请求对象, 即crawl spider也可以在parse函数中手动构建 request请求对象, 指定处理函数. 并且也可以在crawl spider中同时使用两种url的提取规则, 即可以使用rules提取url地址, 也可以手动提取url地址, 手动构建request请求对象.

也就是说, 在crawl spider中也可以使用之前scrapy.Spider中的方法. 如下:

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

**import** scrapy

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

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

**import** re

**class** CfSpider(CrawlSpider):

name = **'cf'**

allowed\_domains = [**'circ.gov.cn'**]

*# 确实start\_urls中是否有我们想要的内容*

start\_urls = [**'http://www.circ.gov.cn/web/site0/tab5240/module14430/page1.htm'**]

*#定义提取url地址规则. rules是一个元组, 元组中元素是有顺序的. rules中可以定义多个规则, 每个规则提取一种类型的url地址*

rules = (

*# Rule(), Rule也是一个类, Rule()就相当于实例化一个对象. 相当于定义了一个规则对象.*

*# LinkExtractor 链接提取器，提取url地址*

*# allow=r''中是正则表达式, 用来匹配某一类型的url地址. 提取到的url地址会通过parse函数发送请求. 所以下面的解析函数不能定义为parse, 否则会覆盖系数的parse函数, 就不能够去发送请求了.*

*# callback 提取出来的url地址的response会交给callback处理. 对于有些规则提取器, 不需要有callback函数. 也就是不需要把提取到的链接地址的响应进行处理. 如下一列的url地址, 只需要提取出地址, 默认交给parse函数进行处理即可, parse函数会再次调用Rule从中提取出帖子详情页的url地址, 就不需要指定callback函数了.*

*# follow 当前url地址的响应是否重新使用rules来提取url地址，如果设置为True, 就类似于一个循环的功能, 当前Rule规则提取出来的url地址会继续进入到rules规则中再次依次匹配各个规则, 再次提取url地址. 如在circ爬虫项目中, 对于提取到的下一页的url地址, 它的响应就需要再次进入到rules中匹配并提取url地址, 所以要设置为True. 而提取到的帖子的详情url地址, 就不需要从中提取url地址, 只需要默认使用parse函数处理url的顺应即可. 所以callback和follow要根据实际情况选择是否填写.*

Rule(LinkExtractor(allow=**r'/web/site0/tab5240/info\d+\.htm'**), callback=**'parse\_item'**), *# 提取详情页的url地址, 需要有callback, 不需要再次提取url响应中的url地址*

Rule(LinkExtractor(allow=**r'/web/site0/tab5240/module14430/page\d+\.htm'**),follow=**True**), *# 提取下一页的url地址, 不需要有callback, 需要再次提取url响应中的地址*

)

*#parse函数在crawl spider中有特殊功能，不能使用parse作为函数名*

*# 提取详情页中帖子的内容*

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

item = {}

item[**"title"**] = re.findall(**"<!--TitleStart-->(.\*?)<!--TitleEnd-->"**,response.body.decode())[0]

item[**"publish\_date"**] = re.findall(**"发布时间：(20\d{2}-\d{2}-\d{2})"**,response.body.decode())[0]

print(item)

**yield** scrapy.Request(

url = **''**,

callback=self.parse\_detail,

meta = {**"item"**:item}

)

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

item = response.meta[**"item"**]

item[**"price"**] = **"///"**

**yield** item

生成爬虫的命令不同, 在爬虫中需要定义提取url地址的匹配规则, 注意什么时候使用follow, 什么时候使用callback. 如果数据全部位于详情页中, 只需要定义一个详情页的解析函数, 如果数据只有一部分在详情页中, 一部分在列表页中, 就需要像在scrapy.Spider中的那样, 手动去提取url地址, 手动构建 request请求对象, 并指定用来提取对应url响应的解析函数.

### CrawlSpider补充(了解)

<https://www.tqcto.com/article/code/664108.html>

LinkExtractor 更多常见参数

allow: 满足括号中"正则表达式" 的URL会被提取, 如果为空, 则全部匹配.

deny: 满足括号中 "正则表达式" 的URL一定不提取, 优先级高于allow

allow\_domains：会被提取的链接的domains。

deny\_domains：一定不会被提取链接的domains。

restrict\_xpaths：使用xpath表达式，和allow共同作用过滤链接。xpath满足范围内的url地址会被提取.

spider.Rule 常见参数

link\_extractor: 是一个Link Extractor 对象, 用于定义需要提取的链接.

callback: 从link\_extractor中每获取到去了如甲方, 参数所指定的值作为回调函数.

follow: 是一个布尔 (boolean) 值, 指定了根据该规则从 response 提取的链接是否需要进行跟进. 如果callback 为None, follow 默认设置为True, 否则默认为 False.

process\_links: 指定该spider 中哪个函数将会被调用, 从link\_extractor中获取到的链接列表时将会调用该函数, 该方法主要用来过滤 url.

process\_requests: 指定该spider中哪个函数将会被调用, 该规则提取到每个request时都会调用该函数. 该方法用来过滤request.

process\_links, 例如在rules中定义了链接的提取规则, 提取到的url地址会自动构建request请求对象, 获取的request响应是使用parse函数进行处理. 但是如果需要在构建request对象和发送请求之前对提取出来的链接进行进一步的处理, 就可以使用process\_links指定一个处理函数来处理.

Rule(LinkExtractor(allow=r'/web/site0/tab5240/info\d+\.htm'), callback='parse\_item'),

但是在上面的circ爬虫中, 响应中的链接是不完整的, 但是我们并没有手动去对链接进行补全, 却没有报错, 这是因为crawl spider会自动对链接进行补全操作.

process\_request, 如果要对某一个提取规则提取出来的url构建的request请求对象进行进一步的处理, 就可以使用process\_request来指定处理函数. 这样, 此提取规则提取出来的url构建的request请求对象在发送请求前会先交给此函数处理, 之后再交给引擎发送请求.

### ### crawlspider的使用

- 创建爬虫 scrapy genspider -t crawl 爬虫名 allow\_domain

- 指定start\_url，此url对应的响应会经过rules提取url地址

- 完善rules，添加Rule

`Rule(LinkExtractor(allow=r'/web/site0/tab5240/info\d+\.htm'), callback='parse\_item'),`

- 注意点:

- url地址不完整，crawlspider会自动补充完整之后再请求

- 不能使用parse为解析函数的函数名，parse解析函数有特殊的功能需要实现

- callback：连接提取器提取出来的url地址对应的响应交给他处理

- follow：连接提取器提取出来的url地址对应的响应是否继续被rules来过滤并提取其中的url链接地址.

### 下载中间件

中间件有两种, 一种是Spider Middlewares即爬虫中间件, 另一种是Downloader Middlewares即下载中间件, 经常需要自定义的是下载中间件.

下载中间件位于scrapy engine和下载器之间, request请求对象会由engine经过下载中间件发送给下载器, 下载器获取到的响应也会经过下载中间件发送给engine. 所以就可以在下载中间件中对请求对象和响应进行进一步的处理.

使用方法：

使用方法和pipeline的使用方法类似.

编写一个Downloader Middlewares, 和我们编写一个pipeline一样，定义一个类，然后在setting中开启

Downloader Middlewares默认的方法：

process\_request(self, request, spider)：

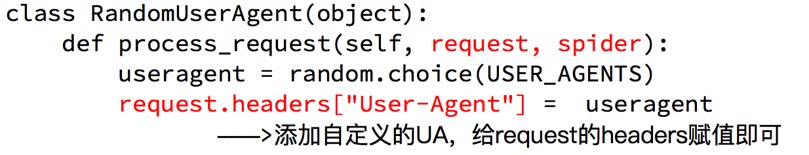
当每个request通过下载中间件时，该方法被调用。其中的spider用来指明这个request是来自哪个spider. pipeline中的process\_item(self, item, spider)方法中也有一个spider, 指明对应的item是来自哪个spider.

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

当下载器完成http请求，传递响应给引擎的时候调用. 在这个方法中除了要传递response参数外, 还需要传递request参数, 指明这个response是哪个request请求的响应.

可以在下载中间件中定义一些反反爬的规则, 如设置User-Agent, 设置proxy代理.

添加请求头的方法. 因为request中有headers请求头信息, 就可以使用request.headers["User-Agent"]来修改请求头.



修改settings.py文件, 把User-Agent的列表放在settings.py文件中.

USER\_AGENTS\_LIST = [ "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64; Trident/5.0; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 2.0.50727; Media Center PC 6.0)", "Mozilla/5.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; .NET CLR 1.0.3705; .NET CLR 1.1.4322)", "Mozilla/4.0 (compatible; MSIE 7.0b; Windows NT 5.2; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.2; .NET CLR 3.0.04506.30)", "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN) AppleWebKit/523.15 (KHTML, like Gecko, Safari/419.3) Arora/0.3 (Change: 287 c9dfb30)", "Mozilla/5.0 (X11; U; Linux; en-US) AppleWebKit/527+ (KHTML, like Gecko, Safari/419.3) Arora/0.6", "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.2pre) Gecko/20070215 K-Ninja/2.1.1", "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9) Gecko/20080705 Firefox/3.0 Kapiko/3.0", "Mozilla/5.0 (X11; Linux i686; U;) Gecko/20070322 Kazehakase/0.4.5" ]

然后修改middlewares.py文件, 定义一个middleware

**import** random

**class** RandomUserAgentMiddleware:

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

*# 给request添加一个随机的请求头, 需要从settings.py中取到定义的USER\_AGENTS\_LIST列表. 使用spider.settings.get("USER\_AGENTS\_LIST")来获取*

ua = random.choice(spider.settings.get(**"USER\_AGENTS\_LIST"**))

*# 在每次发送请求的时候都会经过这个下载中间件, 都调用这个方法, 随机选择一个User-Agent.*

request.headers[**"User-Agent"**] = ua

**class** CheckUserAgent:

*# 每次请求的响应都会经过下载中间件, 并且响应都会经过process\_response这个函数来处理, 在这里就来判断 一下上面设置的User-Agent是否生效了.*

**def** process\_response(self,request,response,spider):

*# dir(response.request)显示response.request的所有方法, 注意response中有response.request这个方法, 但是却没有response.request.headers这个方法, 所以这里只能使用request.headers方法来验证设置的User-Agent是否生效. 注意response.headers是响应头, 其中没有User-Agent的选项*

*# print(dir(response.request))*

print(request.headers[**"User-Agent"**])

# 注意在process\_response这个方法中必须要使用return返回一个对象, 或者返回request, 或者返回response, 或者返回一个Nonetype对象. 如果返回request, request就会交给下载器进行下载, 如果返回response, 就会交给引擎, 如果没有写return, 返回的就是Nonetype对象. 由于这里处理的是响应, 处理过后要交给引擎, 所以要return response. 而在上面的process\_request方法中因为已经指明是处理请求的了, 处理过的请求都会发送给下载器, 所以就不需要使用return返回对象.

**return** response

在settings.py中添加并打开定义的这两个中间件. 注意, 因为一个下载中间件是处理请求的, 一个是处理响应的, 所以这里写的权重对二者的执行顺序并没有影响.

DOWNLOADER\_MIDDLEWARES = {

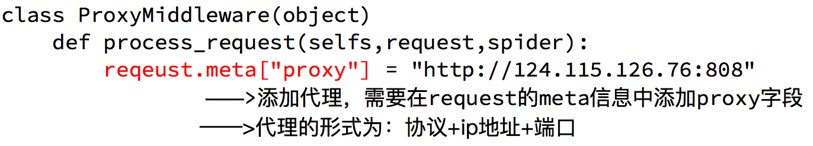
'circ.middlewares.RandomUserAgentMiddleware': 543,

'circ.middlewares.CheckUserAgent': 544,

}

运行爬虫, 查看输出的内容. 检验User-Agent是否每次都是随机选择的.

使用下载中间件添加代理地址



在使用代理的时候也会经常使用到下载中间件, 只需要给request.meta添加一个proxy的键值对就行了, proxy键的值为 "协议+ip地址+端口" 的形式. 需要注意的是, 如果请求的url地址是http的地址时, 就要使用http协议的代理, 如果请求的是https的url地址, 就要使用https的代理. 和上面随机选择User-Agent的方法一样, 也需要设置一个代理地址的列表, 从中随机选择代理才能达到反爬的目的. 注意的是在选择之间需要保证代理列表中的代理地址对于该网站都是可用的. 所以就需要在使用代理之前对代理列表中的代理的可用性进行检查. 可以使用request模块用代理地址发送请求的方式来对代理地址进行检查. 实际上, 可以把所有的代理地址组成代理池放在redis中, 使用一个程序不停的检查代理池中的代理是否可用, 如果不可用, 就把它删除, 后续再使用scrapy从代理池中提取代理的时候, 得到的代理就都是可用的了.

在购买第三方代理地址的时候, 可能会提供用户名和密码, 在使用代理的时候, 还需要把用户名和密码带上, 可能就要把用户名和密码使用base64方法加密, 再和协议, 地址和端口组合起来使用.

查看scrapy中下载中间件的使用

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

先要在settings.py中开启下载中间件

然后定义一个类, 类中有一些方法. process\_request, process\_response, process\_exception. process\_exception是专门用来处理异常的方法. 如在使用代理IP的时候可能会出现一些异常, 就可以使用process\_exception来接收并处理这些异常. 同样要返回None, Response对象, Request对象.在scrapy后台默认开启了很多的下载中间件, 在返回None时, scrapy会继续使用其它的下载中间件来处理这些异常. 并不是定义了一个方法之后就只会在自定义的方法中处理一次, 也是会根据返回的对象的不同进行下一步的处理.

**Writing your own downloader middleware**

Each middleware component is a Python class that defines one or more of the following methods:

**class scrapy.downloadermiddlewares.DownloaderMiddleware**

**Note**

Any of the downloader middleware methods may also return a deferred.

**process\_request(request, spider)**

This method is called for each request that goes through the download middleware.

[**process\_request()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_request) should either: return **None**, return a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object, return a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request)object, or raise [**IgnoreRequest**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.IgnoreRequest).

If it returns **None**, Scrapy will continue processing this request, executing all other middlewares until, finally, the appropriate downloader handler is called the request performed (and its response downloaded).

If it returns a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object, Scrapy won’t bother calling any other [**process\_request()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_request) or [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) methods, or the appropriate download function; it’ll return that response. The [**process\_response()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_response) methods of installed middleware is always called on every response.

If it returns a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object, Scrapy will stop calling process\_request methods and reschedule the returned request. Once the newly returned request is performed, the appropriate middleware chain will be called on the downloaded response.

If it raises an [**IgnoreRequest**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.IgnoreRequest) exception, the [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) methods of installed downloader middleware will be called. If none of them handle the exception, the errback function of the request (**Request.errback**) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

|  |  |
| --- | --- |
| **Parameters:** | * **request** ([**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object) – the request being processed * **spider** ([**Spider**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider for which this request is intended |

**process\_response(request, response, spider)**

[**process\_response()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_response) should either: return a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object, return a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object or raise a [**IgnoreRequest**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.IgnoreRequest) exception.

If it returns a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) (it could be the same given response, or a brand-new one), that response will continue to be processed with the [**process\_response()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_response) of the next middleware in the chain.

If it returns a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object, the middleware chain is halted and the returned request is rescheduled to be downloaded in the future. This is the same behavior as if a request is returned from [**process\_request()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_request).

If it raises an [**IgnoreRequest**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.IgnoreRequest) exception, the errback function of the request (**Request.errback**) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

|  |  |
| --- | --- |
| **Parameters:** | * **request** (is a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object) – the request that originated the response * **response** ([**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object) – the response being processed * **spider** ([**Spider**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider for which this response is intended |

**process\_exception(request, exception, spider)**

Scrapy calls [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) when a download handler or a [**process\_request()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_request) (from a downloader middleware) raises an exception (including an [**IgnoreRequest**](https://doc.scrapy.org/en/latest/topics/exceptions.html#scrapy.exceptions.IgnoreRequest) exception)

[**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) should return: either **None**, a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object, or a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object.

If it returns **None**, Scrapy will continue processing this exception, executing any other [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) methods of installed middleware, until no middleware is left and the default exception handling kicks in.

If it returns a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response) object, the [**process\_response()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_response) method chain of installed middleware is started, and Scrapy won’t bother calling any other [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) methods of middleware.

If it returns a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object, the returned request is rescheduled to be downloaded in the future. This stops the execution of [**process\_exception()**](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#scrapy.downloadermiddlewares.DownloaderMiddleware.process_exception) methods of the middleware the same as returning a response would.

|  |  |
| --- | --- |
| **Parameters:** | * **request** (is a [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object) – the request that generated the exception * **exception** (an **Exception** object) – the raised exception * **spider** ([**Spider**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider) object) – the spider for which this request is intended |

## 6. scrapy模拟登陆

为什么需要模拟登陆？

获取cookie，能够爬取登陆后的页面

回顾：

requests是如何模拟登陆的？

1、直接携带cookies请求页面. 把已经登陆后的cookies放在请求头中, 再次请求页面时就是以已经登陆的状态进行请求的.

2、找接口发送post请求存储cookie, 使用session发送post请求, 服务器会在session中设置cookies, 在下一次请求是就会带上上一次服务器设置的cookies.

3、可以把cookies放在request中作为字典形式使用cookies参数来接收.

selenium是如何模拟登陆的？

找到对应的input标签，输入用户名和密码, 点击登录

### scrapy模拟登陆的两个方法

1、直接携带cookie

2、只需要在setttings.py中设置打开cookies\_anabled = True, 默认是开启的状态. 然后找到发送post请求的url地址，带上信息，发送请求. scrapy就会自动保存服务器发送的cookies信息, 在下一次再次请求时, 会带上上一次服务器保存到本地的cookies, 就相当于是以登陆状态进行的请求.

### scrapy模拟登陆之携带cookie

应用场景：

1、cookie过期时间很长，常见于一些不规范的网站

2、能在cookie过期之前把所有的数据拿到

3、配合其他程序使用，比如其使用selenium把登陆之后的cookie获取到保存到本地，scrapy发送请求之前先读取本地cookie

### 携带cookie登录之前

我们在spider下面定义了start\_urls,那么这个start\_urls是交给谁去处理的？

start\_urls地址的响应是由parse(self, response) 这个方法进行处理的, 其中的response就是start\_urls地址对应的响应. 那start\_urls中的url地址是由哪个方法进行处理并发送请求的呢.

为什么要提出这个问题呢, 如果start\_urls中的地址是一个需要登陆后才能访问的页面, 如12306的个人主页, 直接访问这个地址是不能获取到想要的响应的. 所以就需要对发送请求之前的url地址进行一些处理. 如加上cookies. 所以要弄明白start\_urls中的地址是哪个方法或函数进行处理并发送请求的.

打开一个scray Spider, 点击class SuningSpider(scrapy.Spider):其中的Spider, 跳转到Spider的源码.

**class** Spider(object\_ref):

...

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

cls = self.\_\_class\_\_

**if** method\_is\_overridden(cls, Spider, **'make\_requests\_from\_url'**):

warnings.warn(

**"Spider.make\_requests\_from\_url method is deprecated; it "**

**"won't be called in future Scrapy releases. Please "**

**"override Spider.start\_requests method instead (see %s.%s)."** % (

cls.\_\_module\_\_, cls.\_\_name\_\_

),

)

# 对于每一个起始url, 调用self.mak\_requests\_from\_url来构建一个request请求对象.

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

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

**else**:

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

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

# 构建request请求对象. 就是在Scrapy中手动构建的request请求对象. 在scrapy中, 如果构建的request请求对象中定义callback函数, 它就会默认交给parse()方法进行处理.

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

*""" This method is deprecated. """*

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

只要在spider中自定义一个start\_requests方法, 就可以自己对请求进行额外的处理, 如添加headers, 添加cookies.

创建项目 scrapy startproject login

cd login

创建爬虫 scrapy genspider renren renren.com

修改renren.py文件

**import** scrapy

**import** re

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

name = **'renren'**

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

start\_urls = [**'http://www.renren.com/327550029/profile'**]

*# 因为start\_url是需要登陆之后才能访问的页面, 所以此时必须要自定义start\_requests. 这样就会覆盖掉父类中的同名方法, scrapy发送过来的start\_url的请求就会由这个自定义的start\_requests方法来处理.*

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

*# 在scrapy中添加的cookies是一个字典的形式, 所以要对原始的cookies进行处理, 把字符串格式的cookies转换为字典格式的cookies*

cookies = **"anonymid=jcokuqturos8ql; depovince=GW; jebecookies=f90c9e96-78d7-4f74-b1c8-b6448492995b|||||; \_r01\_=1; JSESSIONID=abcx4tkKLbB1-hVwvcyew; ick\_login=ff436c18-ec61-4d65-8c56-a7962af397f4; \_de=BF09EE3A28DED52E6B65F6A4705D973F1383380866D39FF5; p=90dea4bfc79ef80402417810c0de60989; first\_login\_flag=1; ln\_uact=mr\_mao\_hacker@163.com; ln\_hurl=http://hdn.xnimg.cn/photos/hdn421/20171230/1635/main\_JQzq\_ae7b0000a8791986.jpg; t=24ee96e2e2301bf2c350d7102956540a9; societyguester=24ee96e2e2301bf2c350d7102956540a9; id=327550029; xnsid=e7f66e0b; loginfrom=syshome; ch\_id=10016"**

*# 先使用分号来进行切割, 得到以等号连接的每个键值对, 再使用等号进行切割, 等号前的部分作为字典的键, 等号后的部分作为对应的值*

cookies = {i.split(**"="**)[0]:i.split(**"="**)[1] **for** i **in** cookies.split(**"; "**)}

*# headers = {"Cookie":cookies}*

*# 取出来起始页的url地址, 构建request请求对象. 使用callback指定start\_url的响应由parse函数来进行处理.*

**yield** scrapy.Request(

self.start\_urls[0],

callback=self.parse,

cookies=cookies

*# headers = headers*

)

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

*# 先看使用cookies进行的请求是否成功. 使用正则匹配查看是否有用户的名字*

print(re.findall(**"毛兆军"**,response.body.decode()))

在settings.py中设置不遵守robots协议, 这样在爬虫中就不会再先去请求一次robots.txt了.

ROBOTSTXT\_OBEY = False

启动爬虫进行测试, 看是否会打印出来用户的名字.

假如还想访问其它在登陆之后才能访问的页面, 就需要再在parse方法中构建另一个request对象, 在其中指定url地址和解析函数.

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

**import** scrapy

**import** res

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

name = **'renren'**

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

start\_urls = [**'http://www.renren.com/327550029/profile'**]

*# 因为start\_url是需要登陆之后才能访问的页面, 所以此时必须要自定义start\_requests. 这样就会覆盖掉父类中的同名方法, scrapy发送过来的start\_url的请求就会由这个自定义的start\_requests方法来处理.*

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

*# 在scrapy中添加的cookies是一个字典的形式, 所以要对原始的cookies进行处理, 把字符串格式的cookies转换为字典格式的cookies*

cookies = **"anonymid=jcokuqturos8ql; depovince=GW; jebecookies=f90c9e96-78d7-4f74-b1c8-b6448492995b|||||; \_r01\_=1; JSESSIONID=abcx4tkKLbB1-hVwvcyew; ick\_login=ff436c18-ec61-4d65-8c56-a7962af397f4; \_de=BF09EE3A28DED52E6B65F6A4705D973F1383380866D39FF5; p=90dea4bfc79ef80402417810c0de60989; first\_login\_flag=1; ln\_uact=mr\_mao\_hacker@163.com; ln\_hurl=http://hdn.xnimg.cn/photos/hdn421/20171230/1635/main\_JQzq\_ae7b0000a8791986.jpg; t=24ee96e2e2301bf2c350d7102956540a9; societyguester=24ee96e2e2301bf2c350d7102956540a9; id=327550029; xnsid=e7f66e0b; loginfrom=syshome; ch\_id=10016"**

*# 先使用分号来进行切割, 得到以等号连接的每个键值对, 再使用等号进行切割, 等号前的部分作为字典的键, 等号后的部分作为对应的值*

cookies = {i.split(**"="**)[0]:i.split(**"="**)[1] **for** i **in** cookies.split(**"; "**)}

# 注意, 此时如果把cookies放在headers中并传递hdaders的话, 是不起作用的, 只能把cookies作为单独的参数传递, 这里和之前的request中使用cookies登陆是的方法是不同的.

*# headers = {"Cookie":cookies}*

*# 取出来起始页的url地址, 构建request请求对象. 使用callback指定start\_url的响应由parse函数来进行处理.*

**yield** scrapy.Request(

self.start\_urls[0],

callback=self.parse,

cookies=cookies

*# headers = headers*

)

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

*# 先看使用cookies进行的请求是否成功. 使用正则匹配查看是否有用户的名字*

print(re.findall(**"毛兆军"**,response.body.decode()))

**yield** scrapy.Request(

**"http://www.renren.com/327550029/profile?v=info\_timeline"**,

callback=self.parse\_detial

)

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

print(re.findall(**"毛兆军"**,response.body.decode()))

再次运行爬虫, 测试在访问新的url地址时是否能够打印出用户名.

为了确定在发送各个请求是是否携带了cookies参数, 可以在setttings.py设置中打开cookies debug的功能, 这样就可以看到cookies在不同请求之间传递的过程.

COOKIES\_DEBUG = **True**

会在[scrapy.downloadermiddlewares.cookies] DEBUG: 中看到把cookies传递给哪个请求了.

我们定义在spider下的start\_urls=[] 都是默认交给start\_requests处理的, 所以如果有必要，我们可以重写start\_requests方法

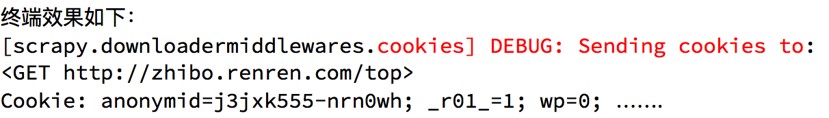


### scrapy模拟登陆之携带cookie



### 那么问题来了：如何知道我的cookie确实是在不同的解析函数中传递呢？





### scrapy模拟登陆之发送post请求



注意：github的部分页面只允许一处登录，比如https://github.com/settings/security

scrapy使用scrapy.FormRequest方法来发送POST请求,

打开github.com/login, 对登陆进行测试, 选中Preserve log. 输入错误的密码, 查看请求中发送的数据以及请求的url地址, 刷新页面, 再次输入错误的密码, 查看请求的数据中哪些是变化的, 哪些是不变的.

post请求的发送地址是github.com/session.

在发送的数据中, 每次刷新页面都会生产新的authenticity\_token, 这个值是变化的, 其它的值都没有变化. 发送post请求中的参数可能是在当前的响应中, 也可能是在之前的响应中, 也可能是通过js生成的, 通过search all files找到authenticity\_token, 看它是有哪个请求中的. 确认这些值都是在github.com/login网页中生成的, 只需要在网页中提取出这些值, 然后在post请求中把这些值发送出去就可以了.

a=10

b=20

c=dict(a=a,b=b)

c

{'a': 10, 'b': 20}

新建一个github的爬虫.

scrapy genspider github github.com

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

**import** scrapy

**import** re

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

name = **'github'**

allowed\_domains = [**'github.com'**]

*# 起始url地址是login的页面, 其中包含着发送post请求的数据.*

start\_urls = [**'https://github.com/login'**]

*# 定义parse函数, 从start\_url的响应中提取出来post请求发送的数据*

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

authenticity\_token = response.xpath(**"//input[@name='authenticity\_token']/@value"**).extract\_first()

utf8 = response.xpath(**"//input[@name='utf8']/@value"**).extract\_first()

commit = response.xpath(**"//input[@name='commit']/@value"**).extract\_first()

*# 构建字典格式的post\_data的数据. post的数据中除了用户名和密码之外 , 还有其它的参数.*

post\_data = dict(

login=**"noobpythoner"**,

*# 可以先输入错误的密码进行测试, 看能否匹配到用户名*

*# password="zhouda23wei123",*

password=**"zhoudawei123"**,

authenticity\_token=authenticity\_token,

utf8=utf8,

commit=commit

)

*# 使用scrapy.FormRequest来发送POST请求, 需要传入POST的地址, POST的数据, 和callback解析函数, 即不管前面的POST请求是否成功, POST请求对应的响应都要使用这个callback函数进行处理*

**yield** scrapy.FormRequest(

**"https://github.com/session"**,

formdata=post\_data,

callback=self.after\_login

)

**def** after\_login(self,response):

*# 当输入错误的验证码后, 还是会匹配到一个noobpythoner的用户名, 可以把响应的内容保存到本地中. 查看响应的内容. 在输入密码错误时, 会在返回的响应中的input标签中把输入用户名填写进去, 这就是输入错误的密码时也会匹配到用户名的原因.*

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

*# f.write(response.body.decode())*

print(re.findall(**"noobpythoner|NoobPythoner"**,response.body.decode()))

分别输入错误的密码和正确的密码, 运行爬虫, 查看返回的数据.

在输入正确的密码时, 可以在[scrapy.downloadermiddlewares.cookies] DEBUG: 中看到Set-Cookie的字段, 即在访问<https://github.com/login时>, 如果输入的密码正确, 服务器会向客户端中设置cookies. 然后把上一步中接收到的cookies发送到<https://github.com/session这个页面中>. 这个页面接收到cookies之后, 还会从<https://github.com/session>这个页面再次向本地设置cookies. 之后会进行一个302的重定向, 从https//gitbhub.com/session跳转到github.com的主页. 并把上一步中写入到本地的cookies全部发送到github.com的主页中. 这个主页会再次向本地写入cookies. 这样就完成了整个登陆的流程.

其实, 如果登陆时发送POST请求的表单中有action的地址, 就会简化登陆的流程. github.com/login页面表单的action地址为/session, 只要向这个地址发送用户名和密码就可以了, 这样就不用构建POST请求的formdata数据了. 其实sccrapy会自动识别登陆的表单中的action地址, 并把我们输入的数据提交到这个地址中.

scrapy.FormRequest提供了一个from\_response方法, 它能够自动从response的响应中找到form表单的action地址. 我们只用构建用户名和密码的字典就可以了. 在这个字典中, 用户名标签的name值作为用户名字段的键, 真正的用户名作为值, 密码标签的name值作为密码字段的键. 真正的密码作为值. 这样scrapy就会自动把构建的字典传递到action中的地址.

新建爬虫 scrapy genspider github2 github.com

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

**import** scrapy

**import** re

**class** Github2Spider(scrapy.Spider):

name = **'github2'**

allowed\_domains = [**'github.com'**]

*# 起始地址为登陆页*

start\_urls = [**'https://github.com/login'**]

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

*# scrapy.FormRequest.from\_response方法会自动从起始url地址的响应中查找登陆表单的action地址. 我们只要构建字典格式的包含用户名和密码的formdata参数即可.*

**yield** scrapy.FormRequest.from\_response(

response, *#自动的从response中寻找from表单*

formdata={**"login"**:**"****noobpythoner"**,**"password"**:**"zhoudawei123"**},

*# 设置callback函数, 指定把formdata传递到action中的地址后的响应的解析函数.*

callback = self.after\_login

)

**def** after\_login(self,response):

print(re.findall(**"noobpythoner|NoobPythoner"**,response.body.decode()))

运行爬虫, 看是否从页面中匹配到了用户名.

问题来了, 如果在登陆页面中存在着多个form表单, scrapy到底会提取哪个表单中的action地址并向其中发送formdata的数据呢.

点击**yield** scrapy.FormRequest.from\_response中的from\_response, 查看源码, 可以看到, from\_response中还可以传入很多其它的参数. 如formname, formid, formnumber等. 通过这些参数就可以对要提取action的表单进行定位.

**def** from\_response(cls, response, formname=**None**, formid=**None**, formnumber=0, formdata=**None**,

clickdata=**None**, dont\_click=**False**, formxpath=**None**, formcss=**None**, \*\*kwargs):

scrapy模拟登陆的3种方式.

携带cookies登陆

使用scrapy.FormRequest发送POST请求, 还可以使用scrapy.Request发送请求, 指定method="POST"即可.

如果表单中对应有action的地址, 还可以使用scrapy.FormRequest.from\_response方法自动从表单中提取action地址, 构建用户名和密码的字典formdata, 自动发送向action中的地址发送POST请求并发送formdata.

### scrapy模拟登陆之自动登录



### 12306网站登陆介绍与总结

找到12306登陆的url地址, 打开一个无痕浏览器, 输入登陆页面的url地址, 模拟爬虫的行为, 或模拟第1次登陆12306网站的行为. 在network中init的响应中可以看到, request headers中并没有携带cookies信息. 而服务器则通过response向本地设置cookies了. 使用request模块的session或者scrapy就可以自动接收这些cookies.

在uamtk中, 会设置发送POST请求, 发送appid otn这个数据. 它的response是用户是否登陆的信息. 它也会向本地设置cookies. 所以这个地址也是在模块登陆时需要请求的地址.

在captcha中就开始请求验证码了. 可以使用打码平台对验证码进行识别.

logdevice中设置了一些验证的数据. 会向本地设置cookieCode和dfp等cookies信息. 但是在response中却没有设置向本地设置cookies的选项, 所以需要手动提到到其中的cookies信息, 然后手动设置cookies

输入正确的验证码和错误的密码, 点击提交, 在captcha-check中会对验证码进行识别, 在response中就会显示出验证码是否成功的信息.

验证码验证成功之后就会在login中进行登陆, 向login中发送用户名和密码等信息.

输入正确的密码, 正确的验证码进行登陆, 也会在captcha-check中对验证码进行识别, 然后在login中进行登陆, 登陆成功后会跳转到userLogin页面, 然后再次重定向到passport…这个地址中, 这个地址会向本地设置cookies.

然后再次请求uamtk这个地址, 提示是否登陆成功.

会请求uamauthcient, 向本地设置cookies

再次请求userLogin地址, 然后会重定向到initMy12306这个最终呈现给用户的地址. 到这一步, 就算是登陆成功了.

### 贴吧爬虫

urllib中的urljoin的使用

import urllib

a="http://tieba.baidu.com/mo/q----,sz@320\_240-1-3---2/m?kw=%E6%9D%8E%E6%AF%85&lp=9001"

b="m?kz=5715089480&amp;is\_bakan=0&amp;lp=5010&amp;pinf=1\_2\_0"

urllib.parse.urljoin(a,b)

'http://tieba.baidu.com/mo/q----,sz@320\_240-1-3---2/m?kz=5715089480&amp;is\_bakan=0&amp;lp=5010&amp;pinf=1\_2\_0'

为什么crawlspider能自动补全地址呢, 就因为它使用了urllib.parse.urljoin()的功能.

请求极速版贴吧

创建项目 scrapy startproject tieba

cd tieba

创建爬虫 scrapy genspider tb tieba.baidu.com

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

**import** scrapy

**import** urllib

**import** requests

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

name = **'tb'**

allowed\_domains = [**'tieba.baidu.com'**]

*# 请求贴吧的极速手机版, 结构单一, 反爬措施低*

start\_urls = [**'http://tieba.baidu.com/mo/q----,sz@320\_240-1-3---2/m?kw=%E6%9D%8E%E6%AF%85&lp=9001'**]

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

*#根据帖子进行分组*

div\_list = response.xpath(**"//div[contains(@class,'i')]"**)

**for** div **in** div\_list:

item = {}

item[**"href"**] = div.xpath(**"./a/@href"**).extract\_first()

item[**"title"**] = div.xpath(**"./a/text()"**).extract\_first()

item[**"img\_list"**] = []

*# 提取到的href地址不完整, 如果使用的是crawlspider版本的爬虫, scrapy会自动补全网址. 但scrapy.Spider版本的爬虫只能手动补全地址. crawlspider使用的是urllib.parse.urljoin()函数来进行地址补全的.*

**if** item[**"href"**] **is not None**:

*# 使用urllib.parse.urljoin()来对地址进行补全. 根据response.url的地址把item['href]中的地址补充完整*

item[**"href"**] = urllib.parse.urljoin(response.url,item[**"href"**])

*# 构建request请求对象, 对于帖子的详情页的响应交给parse\_detail这个函数进行处理.*

**yield** scrapy.Request(

item[**"href"**],

callback=self.parse\_detail,

*# 这里的item也不需要进行deepcopy了. 为什么之前suning图书爬虫中需要使用deepcopy呢, 因为item是大分类下面创建的, 大分类下的每个小分类, 上分类下的每个图书列表都会共用这个item, 这样不同小分类或小分类下的不同列表页都会把上一次的item覆盖掉, 所以需要使用deepcopy. 而在这里, 是针对每一个帖子创建的item, 每次循环都会针对每个帖子创建一个新的单独的item, 不同的item之间不会相互影响, 所以不用使用deepcopy.*

meta = {**"item"**:item}

)

*# 列表页的翻页, 提取列表页下一页的url地址*

next\_url = response.xpath(**"//a[text()='下一页']/@href"**).extract\_first()

**if** next\_url **is not None**:

*# 进行地址的拼接*

next\_url = urllib.parse.urljoin(response.url,next\_url)

*# 构建请求对象并指定解析函数为parse, 继续提取当页中的url地址. 注意因为item提取的是帖子详情页的信息, 所以这里不需要传递meta信息.*

**yield** scrapy.Request(

next\_url,

callback=self.parse

)

*# 解析帖子详情页的响应*

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

item = response.meta[**"item"**]

*# 方法1: 注意如果前面不定义item["img\_list"] = [], 这样真的写是有问题的, 在解析详情页的下一页时, 会把上一页的img\_list给覆盖掉, 得到的只是最后的img\_list*

*# item["img\_list"] = response.xpath("//img[@class='BDE\_Image']/@src").extract()*

*# 方法2: 对img\_list这个键进行判断, 如果这个键不存在, 就添加, 同时从网页中提取出图片的链地址, 如果img\_list这个键已经存在, 就把解析出来的url地址添加到img\_list这个键中. 使用这种方法时在parse函数中不需要事先定义item["img\_list"] = [].*

*# if "img\_list" not in item:*

*# item["img\_list"] = response.xpath("//img[@class='BDE\_Image']/@src").extract()*

*# else:*

*# item["img\_list"].extend(response.xpath("//img[@class='BDE\_Image']/@src").extract())*

*# 方法3: 在parse函数中定义item["img\_list"] = []这个键, 在解析详情页的图片时只需要extend添加到img\_list中即可.*

item[**"img\_list"**].extend(response.xpath(**"//img[@class='BDE\_Image']/@src"**).extract())

*# 注意提取到的图片的url地址的形式为*

*# "http://c.hiphotos.baidu.com/forum/w%3D96%3Bq%3D45%3Bg%3D0/sign=55684c9524381f309e1981af9270713c/a2d70a55b319ebc47c0208348e26cffc1f17160f.jpg?&amp;src=http%3A%2F%2Fimgsrc.baidu.com%2Fforum%2Fpic%2Fitem%2Fa2d70a55b319ebc47c0208348e26cffc1f17160f.jpg"*

*# 只需要取后半部分就好了. 注意后一部分的url地址是编码之后 的格式, 需要先使用requests.utils.unquote对提取到的地址进行解码, 然后使用src=进行切割, 再取最后一个地址即可.*

item[**"img\_list"**] = [requests.utils.unquote(i).split(**"src="**)[-1] **for** i **in** item[**"img\_list"**]]

*# 提取帖子详情页中下一页的标签*

next\_url = response.xpath(**"//a[text()='下一页']/@href"**).extract\_first()

*# 如果有下一页, 就补全地址, 然后构建请求对象, 再次使用parse\_detail解析页面内容*

**if** next\_url **is not None**:

next\_url = urllib.parse.urljoin(response.url,next\_url)

**yield** scrapy.Request(

next\_url,

callback=self.parse\_detail,

meta={**"item"**:item}

)

**else**:

*# 如果没有下一页, 表示 一个帖子的详情已经爬取完毕, 就需要把提取到的item返回.*

print(item)

**yield** item

修改设置文件, 设置loglevel和user-agent, 并且不遵守robotx协议

LOG\_LEVEL = "WARNING"

USER\_AGENT = 'Mozilla/5.0 (iPhone; CPU iPhone OS 10\_3 like Mac OS X) AppleWebKit/602.1.50 (KHTML, like Gecko) CriOS/56.0.2924.75 Mobile/14E5239e Safari/602.1'

ROBOTSTXT\_OBEY = **False**

运行爬虫. scrapy crawl tb

### 腾讯招聘爬虫crawlspider版

scrapy genspider -t crawl tt hr.tencent.com

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

**import** scrapy

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

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

**class** TtSpider(CrawlSpider):

name = **'tt'**

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

*# 修改start\_urls地址*

start\_urls = [**'https://hr.tencent.com/position.php'**]

rules = (

*# 提取详情页的url地址, 这里不需要follow, 指定详情页的响应交给parse\_item解析函数进行处理. 注意url地址中一般都有?和句号, 所以都要进行转义处理*

Rule(LinkExtractor(allow=**r'position\_detail\.php\?id=\d+&keywords=&tid=0&lid=0'**), callback=**'parse\_item'**),

*# 提取下一页的url地址. 列表页不需要另外从中提取数据, 只需要使用默认的parse即可. 列表页要继续使用rules规则进行匹配, 从中提取链接.*

Rule(LinkExtractor(allow=**r'position\.php\?&start=\d+#a'**),follow=**True**),

)

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

*# 提取详情页中的信息*

item = {}

item[**"title"**] = response.xpath(**"//td[@id='sharetitle']/text()"**).extract\_first()

item[**"aquire"**] = response.xpath(**"//div[text()='工作要求：']/../ul/li/text()"**).extract()

print(item)

*# 因为每个详情页中只有一个item, 所以不需要使用yeild返回信息, 只需要使用return即可*

*# return item*

修改settings.py中的loglevel和user-agent, 并且设置不遵守robots协议

运行爬虫.

什么时候使用crawlspider比较方便? 当不从列表页提取数据, 所要提取的数据全部都可以在详情页进行提取时, 使用crawlspider就比较方便, 否则使用scrapy.spider和crawlspider的代码量都差不多.

如果列表页和详情页中都要提取数据, 只使用crawlspider来帮助我们提取分页的url地址, 而详情页的url地址就不需要使用rules规则进行提取, 而是手动提取详情页的url地址, 再手动构建requests对象, 指定解析函数进行处理.

腾讯招聘爬虫列表详情页都提取数据

创建爬虫 scrapy genspider -t crawl tt1

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

**import** scrapy

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

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

**class** Tt1Spider(CrawlSpider):

name = **'tt1'**

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

*# 修改url地址*

start\_urls = [**'https://hr.tencent.com/position.php/'**]

rules = (

*# 只使用rules提取列表页的URL地址.*

Rule(LinkExtractor(allow=**r'position\.php\?&start=\d+#a'**), callback=**'parse\_item'**, follow=**True**),

)

*# 手动提取详情页的url地址, 并构建request请求对象并指定解析函数*

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

*# 对每一个帖子进行分组, 注意没有tbody, 并且去除第1个和最后1个元素*

tr\_list = response.xpath(**"//table[@class='tablelist']/tr"**)[1:-1]

**for** tr **in** tr\_list:

item = {}

item[**"title"**] = tr.xpath(**"./td[1]/a/text()"**).extract\_first()

*# 补全url地址, 可以使用urllib.request.urljoin方法来自动补全, 也可以手动补全*

item[**"href"**] = **"https://hr.tencent.com/"**+tr.xpath(**"./td[1]/a/@href"**).extract\_first()

**yield** scrapy.Request(

item[**"href"**],

callback=self.parse\_detail,

*# 由于列表页和详情页都要提取数据, 所以需要传递item*

meta = {**"item"**:item}

)

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

item = response.meta[**"item"**]

item[**"aquire"**] = response.xpath(**"//div[text()='工作要求：']/../ul/li/text()"**).extract()

print(item)

**yield** item

### Fiddler手机抓包

## 7. SCRAPY\_REDIS学习

### 第六部分课程概要

1. Scrapy\_redis的基础概念
2. Scrapy\_redis的流程
3. Scrapy\_redis的使用

### 什么是scrapy\_redis

基于redis组件的scrapy

Scrapy\_redis：Redis-based components for Scrapy.

Github地址：https://github.com/rmax/scrapy-redis

### 为什么要学习scrapy\_redis

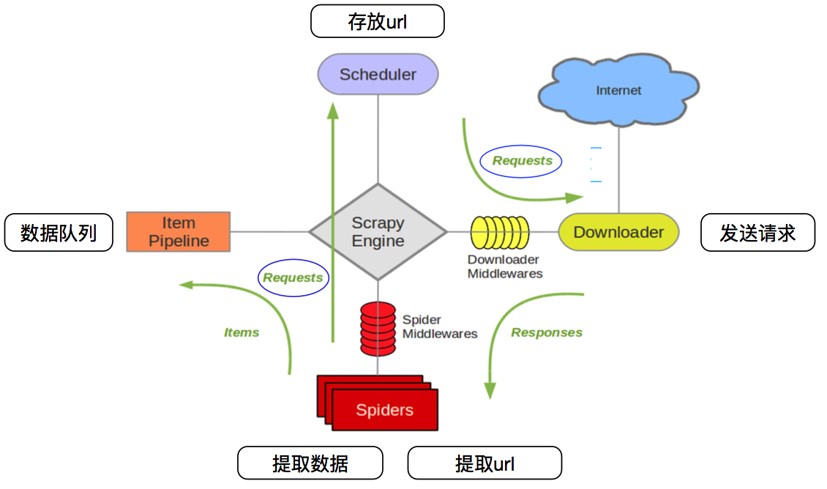
Scrapy\_redis在scrapy的基础上实现了更多，更强大的功能，具体体现在：reqeust去重，爬虫持久化，和轻松实现分布式

在scrapy中, 每一次打开爬虫都会把以前所有的url再请求一次, 包括之前请求过的url和未请求过的url地址. 但是在实际中, 一般不会对之前请求过的url地址再次请求. scrapy\_redis就实现了这个功能, 只要是之前爬取过的链接, 不仅在此次爬虫运行中不会再次请求, 只要redis的相关文件存在, 在以后每一次打开爬虫时都不会再次请求. 这就相当于是基于url地址的增量式爬虫.

### Scrapy\_redis的工作流程

那么，scrapy\_redis是如何帮助我们抓取数据的呢？

#### 回顾scrapy的爬虫流程



#### scrapy\_redis的爬虫流程

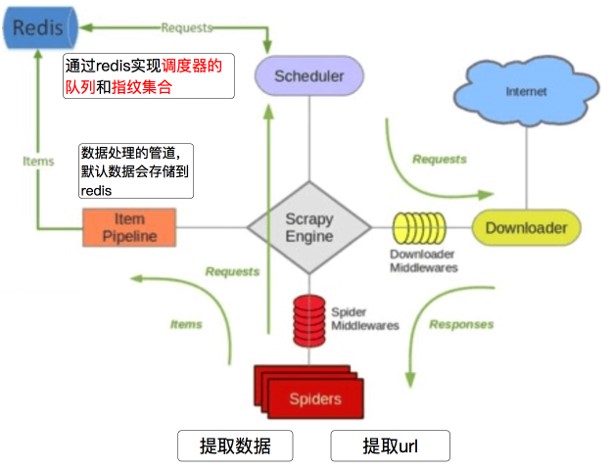
如何在scrapy的基础上实现分布式, 不同电脑上的url地址一定要保存到一个地方, 如存到一个数据库中, 使用redis对url地址进行去重, 然后统一的调度和分发, 保证每个url地址只会被一个电脑爬取到, 爬取过的url地址不会再被其它的电脑爬取.

调度器和Redis相连接, 它们之间会传递Request请求对象, 调度器会从redis中读取request请求对象, 也会把request请求对象保存到redis数据库中.

同样, 也可以把item pipeline与redis数据库相连接, 把爬取到的数据也保存到redis数据库中. 这一步不是必要的, 也可以把数据保存到其它任何地方.

多台电脑都连接同一个redis数据库, 都从redis数据库中读取请求对象, 并把抓取到的url构建的请求对象都保存到redis中, 只要一台电脑爬取过一个url地址对应的请求对象, 就把这个对象从数据库中删除, 这样就可以保证不会爬取到同样的数据了. 同样, 不同的电脑在把抓取到的url构建的请求对象保存到redis之前, 也要进行去重的操作. 实现的原理就是对每一个request对象都生成一个指纹,

这样就能实现不同的电脑通过redis对同一个项目进行爬取的目的了.



### 复习redis的使用

redis有向本地保存数据的机制, 但并不是实时保存的, 而是隔一段时间保存一次, 所以可能会存在着数据的丢失.

<https://github.com/SpringMagnolia/jianlispider> 使用celery实现的一个招聘网站的分布式爬虫.

redis中间件. 可以用于生产者与消费者模型. 对于scrapy\_redis爬虫来说, 有一台电脑是生产者, 所有的数据都保存在这台电脑中. 爬取数据的其它所有电脑都可以看成是消费者. 生产者把生产的产品放在redis数据库中, 其它电脑都从这种电脑中读取生产者生产的产品进行消费, 此时redis就是一个中间件, 起着临时保存数据的作用.



哈希, 散列或字典, 三者在redis中的意义相同.

服务器有两种启动方式: apt-get install 方式安装的包, 使用下面命令来启动.

/etc/init.d/redis-server start

如果是使用源码方式安装的, 使用redis-server + 配置文件的方式启动. 配置文件位于/etc/redis/config中.

这两个启动方式与mongodb的启动方式相似.

在客户端是连接服务器并进行操作.

如果是本机的操作, 就不用写hostname和port

redis-cli

ping

PONG

# 选择数据库

SELECT 0

OK

SELECT 1

OK

# 查看所有的数据表

keys \*

1) "jd:requests"

2) "jd:dupefilter"

# 删除数据表

del id:requests

del id:dupefilter

# redis中数据类型不同, 查看数据的方式也不同, 必须要先知道数据的数据类

# 查看数据类型

type jd:requests

zset # 有序集合

type jd:dupefilter

set # 无序集合

# 清空当前的数据库

flushdb

# 清空所有的数据库

flushall

中文文档 http://www.redis.cn/commands.html

redis中的几种数据类型

redis中都是以键值对存储的, 键是名字, 值是数据的类型

列表, 命令都是以L开头的

LPUSH mylist "world" # 向mylist中从左边添加一个值

LRANGE mylist 0 -1 # 返回mylist中所有的值. 0表示第一个, -1表示最后一个

LLEN mylist # 返回mylist的长度

操作实例

lpush mylist a b c d

(integer) 4

lpush mylist e f g h

(integer) 8

lrange mylist 0 -1

# 因为是lpush, 从左边插入数据, 所以最后插入的数据会在最左边

1) "h"

2) "g"

3) "f"

4) "e"

5) "d"

6) "c"

7) "b"

8) "a"

llen mylist

(integer) 8

集合set.

redis> SADD myset "hello" # 往set中添加数据, 向myset的键中插入一个"hello"的值

redis> SMEMBERS myset # 获取myset中所有的元素

redis> SCARD myset # scrad 获取数量

sadd myset a b c d

(integer) 4

sadd myset a b c d

(integer) 0

# 查看其中的数据, 因为是集合, 所以只有4个元素.

SMEMBERS myset

1) "a"

2) "b"

3) "c"

4) "d"

scard myset

(integer) 4

有序集合 zset

给每一个值一个分数的字段, 分数的值可以是小数或整数, 分数可以相等, 后继就可以以分数对字段进行操作. 有序集合中的有序就是指的分数有序而不是值有序.

zadd 向一个zset中添加一个值和分数, 如果存在值, 就更新分数, 分数可以相同.

zrange 遍历myzset

zcard 返回zset中元素的数量

redis> ZADD myzset 1 "one"

(integer) 1

redis> ZADD myzset 2 "two" 3 "three"

(integer) 2

redis> ZRANGE myzset 0 -1 WITHSCORES

1) "one"

2) "1"

3) "two"

4) "2"

5) "three"

6) "3"

redis> ZCARD myzset

视频演示

zadd myzset 10 a 11 b 12 c

(integer) 3

zadd myzset 10 a 15 b 20 d

(integer) 1 # 为什么只插入了1个, 因为a,b存在, 就更新它们的分数, d不存在, 插入d

zrange myzset 0 -1

1) "a"

2) "c"

3) "b"

4) "d"

zrange myzset 0 -1 withscores

1) "a"

2) "10"

3) "c"

4) "12"

5) "b"

6) "15"

7) "d"

8) "20"

zcard myzset

(integer) 4

下载源码

git clone [git@github.com:rmax/scrapy-redis.git](mailto:git@github.com:rmax/scrapy-redis.git)

# 把example-project复制到当前文件夹下

cp -rf scrapy-redis/example-project ./

### Scrapy\_redis使用

1、clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

2、研究项目自带的三个demo

mv scrapy-redis/example-project ~/scrapyredis-project

example-project为项目文件夹,

example-project/scrapy.cfg为配置文件

example-project/requirements.txt 为需要安装的内容

example-project/readme.txt 为介绍

example-project/process\_item 如果爬取速度非常快, 使用mysql或mongodb数据库写入数据时可能会存在着速度的限制, 出现堵塞的现象. 因为redis的读写速度比较快, 所以默认情况下, scrapy\_reids会把提取到的数据保存到redis数据库中, 然后再从redis中读取数据保存到其它数据库中. process\_item实现的就是从redis中取出来数据写入到文件或写入到其它数据库的功能. 但一般情况下, 并不会直接把数据保存到redis中, 而是直接保存到mysql或mongodb中. 并且数据爬取的速度并不是很快的情况下也不会发生阻塞.

example-project/Dockerfile, example-project/docker-compose.yml, Docker是一种新兴的技术, 能够帮助我们创建一个容器, 能够在这个容器中运行一些程序. 在这里使用Docker创建了一个容器, 在这个容器中运行scrapy\_redis爬虫项目.

example-project/example中的内容就是之前scrapy爬虫项目中的内容

#### Scrapy\_redis之domz



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

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

**class** DmozSpider(CrawlSpider):

*"""Follow categories and extract links."""*

name = **'dmoz'**

allowed\_domains = [**'dmoztools.net'**]

start\_urls = [**'http://dmoztools.net/'**]

rules = [

*# 使用css选择器的方式去定位url地址, 并不是使用的正则匹配. restrict\_css是提取这个类对应的标签下的url地址, 并没有说明要提取哪个具体的url. 也就是说, 只需要使用restrict\_css或restrict\_xpath方法定位到包含要提取的url地址的div标签或a标签, 程序会自动提取其中的url地址. '.top-cat'这个类下面所有的url地址都会提取出来, 它的响应会交给parse\_directory这个解析函数进行处理.*

Rule(LinkExtractor(

restrict\_css=(**'.top-cat'**, **'.sub-cat'**, **'.cat-item'**)

), callback=**'parse\_directory'**, follow=**True**),

]

*# 定义解析函数*

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

*# 进行分组*

**for** div **in** response.css(**'.title-and-desc'**):

*# 从div中提取出来数据, 使用yield返回给pipeline*

**yield** {

**'name'**: div.css(**'.site-title::text'**).extract\_first(),

**'description'**: div.css(**'.site-descr::text'**).extract\_first().strip(),

**'link'**: div.css(**'a::attr(href)'**).extract\_first(),

}

pipelines.py

pipeline与scrapy的也没有什么区别, 只是给item添加了两个键而已.

**from** datetime **import** datetime

**class** ExamplePipeline(object):

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

*# dmoz.py爬虫传递过来的数据会添加2个键, 分别是crawled和spider. 即抓取的时间和爬虫的名字*

item[**"crawled"**] = datetime.utcnow()

item[**"spider"**] = spider.name

**return** item

而在items.py中定义的内容并没有引入到domz.py爬虫文件中或pipeline文件中.

settings.py

*# 爬虫的位置*

SPIDER\_MODULES = ['example.spiders']

NEWSPIDER\_MODULE = 'example.spiders'

USER\_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

*# 指定了一个去重的类, 使用scrapy\_redis中的去重方法*

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

*# 指定调度器的类, 使用scrapy\_redis中的调度器*

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

*# 调度器的内容是否要持久化. 调度器中保存的是待爬取的对象和已爬取过的对象的指纹, 如果这里设置为False, 关闭爬虫之后调度器中的内容就不会被保存. 如果设置为True, 调度器中的内容不会被清除, 下一次启动爬虫时可以继续使用其中的数据.*

SCHEDULER\_PERSIST = True

*# 定义调度器的优先级, 分别为优先队列, 队列和栈.*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderPriorityQueue"*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"*

ITEM\_PIPELINES = {

*# ExamplePipeline是自定义的Pipeline, 这个pipeline的权重比较小,数据会先经过这个pipeline处理.*

'example.pipelines.ExamplePipeline': 300,

*# RedisPipeline是scrapy\_redis自带的Pipeline. scrapy\_redis会默认把爬取到的数据保存到redis数据库中. 就是使用这个pipeline来实现的.*

'scrapy\_redis.pipelines.RedisPipeline': 400,

}

LOG\_LEVEL = 'DEBUG'

*# Introduce an artifical delay to make use of parallelism. to speed up the*

*# crawl.*

DOWNLOAD\_DELAY = 1

*# 如果要把数据保存到redis中, 还需要添加redis的地址和端口号*

REDIS\_URL = "redis://127.0.0.1:6379"

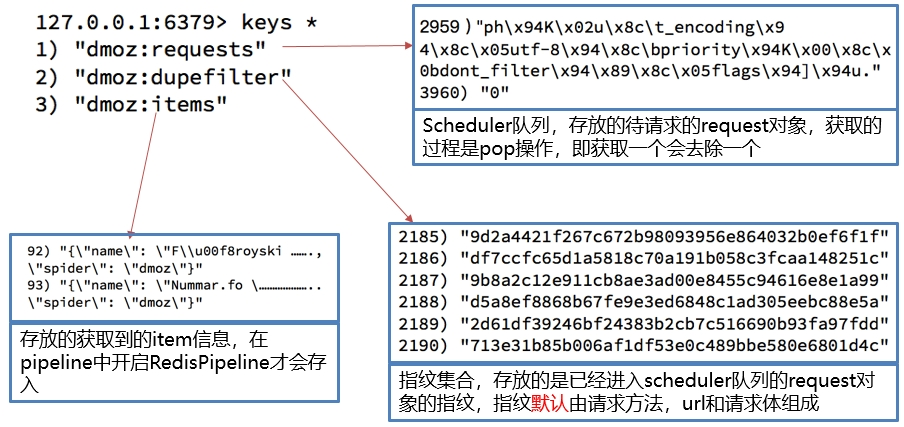


执行scrapy\_redis爬虫

cd example-project

scrapy crawl dmoz

我们执行domz的爬虫，会发现redis中多了一下三个键：



在redis-cli中执行

keys \*

1) "dmoz:requests"

2) "dmoz:dupefilter"

3) "dmoz:items"

# 查看这三个键的类型

type dmoz:requests

zset # 保存的是已经构造好的待爬取的requests请求对象

zrange dmoz:requests 0 -1 # 查看其中的数据. dmoz:requests中保存的是待爬取的requests对象. redis中本不能保存对象, 这里是把对象进行了一次序列化, 序列化是类似于json的dumps的操作, 能够把一个类转换为一个字符串. 在爬虫运行时, 从redis中读取出字符串, 再进行反序列化把字符串转换为一个对应的request对象, 然后使用对象去发送请求下载数据.

ZCARD dmoz:request

(integer) 1528

type dmoz:items

list #

zrange dmoz:items 0 -1 # 其中保存着提取到的类似于json字符串格式的item的内容. 每个item中有5个字段, 但是在我们自定义的ExamplePipeline中并没有把数据保存到redis中, 所以这些数据是使用默认的 'scrapy\_redis.pipelines.RedisPipeline' 保存到redis中的.

llen dmoz:items

(integer) 79

type dmoz:dupefilter

SMEMBERS dmoz:dupefilter # 实现对request对象的过滤和去重. 其中保存的是所有提取到的request对象的指纹, 包括已经交给下载器下载的request对象, 和没有交给下载器下载过的request对象. dmoz:requests中保存的是待抓取/待下载的. 所以二者相减dmoz:dupefilter - dmoz:requests就得到了已经下载过的request对象的指纹.

scard dmoz:dupefilter

(integer) 1557

我们可以尝试在setting中关闭redispipeline，观察redis中三个键的存储数据量的变化.

执行爬虫, 会看到信息中出现了很多302的DEBUG信息.

DEBUG: Redirecting (302) to <GET XXX> from <GET XXX>, 在爬取后面from中的地址时, 发现它已经爬取过了, 就把它重定向到前面to中的url地址, 爬取to中的地址.

ZCARD dmoz:request

(integer) 1705

llen dmoz:items

(integer) 79

scard dmoz:dupefilter

(integer) 1785

变化结果：

dmoz:requests 有变化(变多或者变少或者不变). 待抓取的requests对象的数量不一定会永远增加. 在刚开始爬取时, 提取到的url地址非常多, 而交给下载器下载的url地址有限, 这个值就会一直增大. 到网站的爬取接近尾声时, 大部分url都已经交给下载器提取过了, 其中的内容就会一直减小, 直到为0时, 整个网站的数据就全部抓取到了.

dmoz:dupefilter 变多, 已经提取到的url地址, 只要爬虫一直在运行, 它的数量一直会增加, 直到整个网站的url地址都提取到时, 它的数量才达到最大值, 就不会再增加了.

dmoz:items 不变. 因为把RedisPipeline关闭了, 数据不会再写入到redis数据库了.

变化结果分析:

redispipeline中仅仅实现了item数据存储到redis的过程，我们可以新建一个pipeline（或者修改默认的ExamplePipeline），让数据存储到任意地方

通过以上知识点的学习，我们会发现：

domz相比于之前的spider多了持久化和request去重的功能

在之后的爬虫中，我们可以模仿domz的用法，使用scrapy\_redis实现相同的功能

注意：setting中的配置都是可以自己设定的，意味着我们的可以重写去重和调度器的方法，包括是否要把数据存储到redis(pipeline)

### scrapy\_redis源码分析

那么问题来了：以上的这些功能scrapy\_redis都是如何实现的呢？

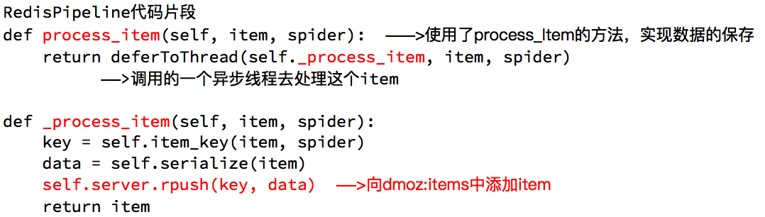
1. 如何通过redispipeline把数据保存到redis中的

2. 如何通过DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter" 这个去重类达到对request对象的去重的

3. 如何通过SCHEDULER = "scrapy\_redis.scheduler.Scheduler" 实现对数据的调度的.

spider和pipelines都与之前的crawl scrapy项目没有任何区别, 只是在settings.py中多出来了几行就能实现基于redis的持久化去重功能, 或者称为增量式爬虫功能. 为什么能够持久化去重, 因为程序可以把之前爬取过的request对象的指纹保存到redis中. 下次运行时, 会先去判断要爬取的url对应的request对象在之前是否爬取过. 之前爬取过的url地址在以后运行的时候不会再去爬取, 所以是个基于url地址的增量式爬虫. 查看scrapy\_redis的源码.

#### Scrapy\_redis之RedisPipeline



C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\_redis

pipelines.py文件, 看一个pipeline就先要看它的process\_item方法

**from** scrapy.utils.misc **import** load\_object

**from** scrapy.utils.serialize **import** ScrapyJSONEncoder

**from** twisted.internet.threads **import** deferToThread

**from** . **import** connection, defaults

default\_serialize = ScrapyJSONEncoder().encode

**class** RedisPipeline(object):

*"""Pushes serialized item into a redis list/queue*

*Settings*

*--------*

*REDIS\_ITEMS\_KEY : str*

*Redis key where to store items.*

*REDIS\_ITEMS\_SERIALIZER : str*

*Object path to serializer function.*

*"""*

**def** \_\_init\_\_(self, server,

key=defaults.PIPELINE\_KEY,

serialize\_func=default\_serialize):

*"""Initialize pipeline.*

*Parameters*

*----------*

*server : StrictRedis*

*Redis client instance.*

*key : str*

*Redis key where to store items.*

*serialize\_func : callable*

*Items serializer function.*

*"""*

self.server = server

self.key = key

self.serialize = serialize\_func

@classmethod

**def** from\_settings(cls, settings):

params = {

**'server'**: connection.from\_settings(settings),

}

**if** settings.get(**'REDIS\_ITEMS\_KEY'**):

params[**'key'**] = settings[**'REDIS\_ITEMS\_KEY'**]

**if** settings.get(**'REDIS\_ITEMS\_SERIALIZER'**):

params[**'serialize\_func'**] = load\_object(

settings[**'REDIS\_ITEMS\_SERIALIZER'**]

)

**return** cls(\*\*params)

@classmethod

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

**return** cls.from\_settings(crawler.settings)

*# process\_item可以接收从其它pipeline或引擎传递过来的item, 顺便也会把spider传递过来*

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

*# 调用了一个线程去把item和spider交给\_process\_item来进行处理*

**return** deferToThread(self.\_process\_item, item, spider)

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

*# 通过item\_key方法把传入的item和spider生成key, 得到类似于dmoz:requests中的dmoz键*

key = self.item\_key(item, spider)

*# 把item序列化之后生成data变量. item是一个类字典格式的对象, 保存到redis中之后就成为字符串格式了. 中间应该是先把item转换为字典, 再进行了类似于json.dumps的操作把它转换为字符串.*

data = self.serialize(item)

*# 通过rpush向对应的键key中放入数据data, self.server应该就是一个与redis的连接. rpush从右向左放入. self.server这个与redis的连接通过rpush向key这个键中放入了data这个内容. 以上就是RedisPipeline所做的事情, 即把数据保存到redis中.*

self.server.rpush(key, data)

**return** item

**def** item\_key(self, item, spider):

*"""Returns redis key based on given spider.*

*Override this function to use a different key depending on the item*

*and/or spider.*

*"""*

*# 因为有一个%, 所以key为通过spider.name生成的一个字符串, 得到的效果就是在redis中使用keys \* 显示出来的数据表的名称. dmoz:requests, dmoz就是爬虫名*

**return** self.key % {**'spider'**: spider.name}

#### Scrapy\_redis之RFPDupeFilter



C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\_redis\dupefilter.py

scrapy\_redis.dupefilter.RFPDupeFilter

从request\_seen方法开始看起

**def** request\_seen(self, request):

"""Returns True if request was already seen.

当request已经被看到过时就返回True. 什么是已经被看到过了呢, 就是已经在页面上提取过url并构建成了request对象. 这些request对象有两种情况, 一种情况是已经把它请求过了. 第二种情况是没有请求过, 但是已经放在了待爬取的队列中了. 一个是准备爬, 一个是已经爬过了

Parameters

----------

request : scrapy.http.Request

Returns

-------

bool

"""

# 是如何返回True的, 首先调用request\_fingerprint方法, 把request对象传过去, 得到对应的request对象的指纹

fp = self.request\_fingerprint(request)

# This returns the number of values added, zero if already exists.

# sadd是向一个集合中插入数据, 当数据已存在时返回0, 数据不存在时返回插入的数据的条数. 把指纹保存到self.key中, 其中的key就是pipelines.item\_key中生成的return self.key % {'spider': spider.name}, 即包含着爬虫名的键.

added = self.server.sadd(self.key, fp)

# added=0的时候返回True. 即上一步中server.sdded的操作返回0的时候, 即数据已经存在的时候返回0. 即request对象的指纹已经存在的时候就返回True.

**return** added == 0

因为scrapy\_redis是调用的scrapy中的方法来生成指纹的, C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\utils\request.py中的request\_fingerprint方法来生成指纹, scrapy也是使用同样的方法生成指纹并对request对象去重的.

**def** request\_fingerprint(request, include\_headers=**None**):

"""

Return the request fingerprint.

返回request对象的指纹

The request fingerprint is a hash that uniquely identifies the resource the

request points to. For example, take the following two urls:

request指纹是一个唯一标识出request对象指向的资源的hash值. 如下面两个url地址, 虽然字符串是不一样的, 但都指向同一个资源. 所以应该是以唯一标记出某个资源的url地址生成hash值, 而不是以唯一的url地址生成hash值. 这里的处理可能会对参数进行排序, 然后再生成hash值.

http://www.example.com/query?id=111&cat=222

http://www.example.com/query?cat=222&id=111

Even though those are two different URLs both point to the same resource

and are equivalent (ie. they should return the same response).

cookies是用来存储session id的, 假如下面的地址只能被已经认证过的用户访问.

Another example are cookies used to store session ids. Suppose the

following page is only accesible to authenticated users:

http://www.example.com/members/offers.html

许多网站都使用cookie来存储session id, 这就给HTTP的请求头添加了一个随机的组成部分, 因此在计算指纹时应该被忽略掉.

Lot of sites use a cookie to store the session id, which adds a random

component to the HTTP Request and thus should be ignored when calculating

the fingerprint.

原因, 不同的用户名对应的session id不同, 所以生成的cookies也不同, 如果计算指纹时带上了cookies, 那么不同用户对同一个url生成的reqeust对象的指纹就不同了, 在这种情况下, 即使同一个url地址在前面已经被请求过, 如果使用不同的账号登陆, 因为生成的指纹不同, 这个url还会再次被请求. 但这与scrapy的去重理念是不同的, 因为请求的是一个url地址, 同一个url地址对应的指纹应该是相同的, 所以在计算指纹时不能带上cookies

因为这个原因, 在计算指纹时是默认忽视掉request的请求头的. 如果想要把特殊的请求头包括进指纹中, 就使用include\_headers参数, 这个参数是一个包含着请求头的列表.

For this reason, request headers are ignored by default when calculating

the fingeprint. If you want to include specific headers use the

include\_headers argument, which is a list of Request headers to include.

"""

**if** include\_headers:

include\_headers = tuple(to\_bytes(h.lower())

**for** h **in** sorted(include\_headers))

cache = \_fingerprint\_cache.setdefault(request, {})

**if** include\_headers **not in** cache:

# 实例化一个sha1加密方法

fp = hashlib.sha1()

# 把request的请求方法转化为bytes类似, 然后添加到fp中. 因为sha1的加密方法不能接受字符串类型的数据, 只能接受bytes类型的

fp.update(to\_bytes(request.method))

# 对请求的url地址进行排序, 然后转化为bytes类型的, 添加到fp中.

fp.update(to\_bytes(canonicalize\_url(request.url)))

# 把请求体添加到fp中, 因为只能post请求才有请求体, 所以对于get请求, 就使用b''bytes类型的空白字符串

fp.update(request.body **or b''**)

**if** include\_headers:

**for** hdr **in** include\_headers:

**if** hdr **in** request.headers:

fp.update(hdr)

**for** v **in** request.headers.getlist(hdr):

fp.update(v)

# fp.hexdigest()转换为16进制的字符串, 保存到cahce的include\_headers键中.

cache[include\_headers] = fp.hexdigest()

**return** cache[include\_headers]

在ipython中进行操作

In [1]: import hashlib

In [2]: f = hashlib.sha1()

In [3]: f.update("传智播客")

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-3-6d44982f20ce> in <module>()

----> 1 f.update("传智播客")

TypeError: Unicode-objects must be encoded before hashing

In [4]: f.update("传智播客".encode())

In [5]: f.hexdigest()

Out[5]: '02e3e0629b82aaf8a5cd3e7d02b7e2dddb30e743'

这里生成的16进制的字符串与scrapy\_redis中生成的request对象的指纹格式相似.

所以scrapy\_redis中就是使用sha1生成的request对象的指纹

scrapy\_redis, scrapy, 自己的程序中如何对url地址进行去重, 自己的程序中如何对数据进行去重, hashlib.sha1

注意上面在生成指纹的时候是把request.method请求方法, request.url请求的url地址, request.body请求体放进sha1对象中生成指纹的, 所以在使用request模块编写爬虫时, 也可以使用类似的方法, 在下次请求时, 通过要请求的url地址与已请求的url地址的指纹进行对比进行去重即可.

对数据也可以进行类似的操作, 如以字典格式保存的键值对, 如果某些键值对比较重要, 可以对这些重要的键值对生成hashlib.sha1的指纹, 把它们保存到redis中去. 下一次的数据也生成对应的指纹, 然后判断是否已经存在, 如果存在, 就不用再保存了.

不仅可以使用sha1, 还可以使用其它的方法.

import hashlib

import pdir

pdir(hashlib)

property:

\_\_all\_\_, \_\_builtin\_constructor\_cache, \_\_builtins\_\_, \_hashlib, algorithms\_available, algorithms\_guaranteed

class:

blake2b: Return a new BLAKE2b hash object.

blake2s: Return a new BLAKE2s hash object.

sha3\_224: Return a new SHA3 hash object with a hashbit length of 28 bytes.

sha3\_256: sha3\_256([string]) -> SHA3 object

sha3\_384: sha3\_384([string]) -> SHA3 object

sha3\_512: sha3\_512([string]) -> SHA3 object

shake\_128: shake\_128([string]) -> SHAKE object

shake\_256: shake\_256([string]) -> SHAKE object

function:

\_\_get\_builtin\_constructor:

md5: Returns a md5 hash object; optionally initialized with a string

new: new(name, data=b'') - Return a new hashing object using the named algorithm;

pbkdf2\_hmac: pbkdf2\_hmac(hash\_name, password, salt, iterations, dklen=None) -> key

sha1: Returns a sha1 hash object; optionally initialized with a string

sha224: Returns a sha224 hash object; optionally initialized with a string

sha256: Returns a sha256 hash object; optionally initialized with a string

sha384: Returns a sha384 hash object; optionally initialized with a string

sha512: Returns a sha512 hash object; optionally initialized with a string

#### start\_urls的去重

在C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\spider.py中

from scrapy.spiders import \*

查看spider的源码

C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\spiders\\_\_init\_\_.py

scrapy.spiders.Spider#start\_requests

**class** Spider(object\_ref):

…

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

cls = self.\_\_class\_\_

**if** method\_is\_overridden(cls, Spider, **'make\_requests\_from\_url'**):

warnings.warn(

**"Spider.make\_requests\_from\_url method is deprecated; it "**

**"won't be called in future Scrapy releases. Please "**

**"override Spider.start\_requests method instead (see %s.%s)."** % (

cls.\_\_module\_\_, cls.\_\_name\_\_

),

)

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

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

**else**:

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

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

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

*""" This method is deprecated. """*

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

start\_request中发送的是start\_urls中的请求, 其中的dont\_filter=True, 即如果start\_urls中放的是之前已经抓取过的url地址, 它依然会被再次抓取. 即如果一个地址在start\_urls地址列表中, 即使之前被抓取过了, 它就会反复的被抓取. start\_urls中的地址相当于是程序的入口, 如果start\_urls中的地址不会被请求, 程序根本就不会去启动, 所以必须要设置start\_urls中的地址为不过滤. 这样还能兼顾到start\_urls地址中的内容被更新的情况下.

查看redis的dupefilters.py文件

C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\dupefilters.py

scrapy.dupefilters.RFPDupeFilter

**from** \_\_future\_\_ **import** print\_function

**import** os

**import** logging

**from** scrapy.utils.job **import** job\_dir

**from** scrapy.utils.request **import** request\_fingerprint

**class** BaseDupeFilter(object):

@classmethod

**def** from\_settings(cls, settings):

**return** cls()

**def** request\_seen(self, request):

**return False**

**def** open(self): *# can return deferred*

**pass**

**def** close(self, reason): *# can return a deferred*

**pass**

**def** log(self, request, spider): *# log that a request has been filtered*

**pass**

**class** RFPDupeFilter(BaseDupeFilter):

*"""Request Fingerprint duplicates filter"""*

**def** \_\_init\_\_(self, path=**None**, debug=**False**):

self.file = **None**

*# self.fingerprints为存在于内存中的一个集合*

self.fingerprints = set()

self.logdupes = **True**

self.debug = debug

self.logger = logging.getLogger(\_\_name\_\_)

*# self.file默认为None, 不会进行下面打开file的操作, 也不会在request\_seen中向file中写入内容, 即默认情况下指纹都会保存在fingerprints的集合中. 当path参数不为None时, 会把指纹保存到本地文件中. 因为是指纹在默认情况下都是保存在内存中的set集合中的, 爬虫关闭时都会被清空.这就是为什么scarpy的爬虫只能在一次运行的过程中实现url地址的去重, 但却不能实现url地址的持久化去重的原因. 而scrapy\_redis因为把指纹保存到了redis数据库中, 每次都从redis中读取待爬取的request对象, 通过redis来判断这个对象之前有没有被抓 取过, 所以可以实现指纹的持久化去重.*

**if** path:

*# 使用os.path.join把path和request.see字符串拼接到一起, 即相当于操作path路径下的requests.seen文件. 以追加的可读可写方式打开文件*

self.file = open(os.path.join(path, **'requests.seen'**), **'a+'**)

*# 把文件操作符定位到第0个处*

self.file.seek(0)

*# for x in self.file读取file中的每一行. rstrip切除右边的空白字符, 就得到了request的指纹, 再把指纹更新到fingerprints的集合中. 这里的操作就相当于读取path路径下的requests.seen文件, 把其中的指纹全部放到fingerprints的集合中去.*

self.fingerprints.update(x.rstrip() **for** x **in** self.file)

@classmethod

**def** from\_settings(cls, settings):

debug = settings.getbool(**'DUPEFILTER\_DEBUG'**)

**return** cls(job\_dir(settings), debug)

**def** request\_seen(self, request):

*# 调用request\_fingerprint生成指纹. scrpy\_redis也是调用的scrapy中相同的方法*

fp = self.request\_fingerprint(request)

*# 判断新生成的request对象是否在self.fingerprints这个集合中. 如果已经存在, 就返回True. 表示这个request对象在之前已经被提取过. 即与scapy\_redis中的相同, 如果已经被抓取过, 就返回True.*

**if** fp **in** self.fingerprints:

**return True**

*# 如果指纹不存在, 就把它添加到指纹的set中去*

self.fingerprints.add(fp)

*# 如果file文件存在, 就向其中写入fp + os.linesep. fp为刚刚生成的request对象的指纹. os.linesep为系统的换行符. linux的换行符为\n, windows的换行符为\r\n. 使用os.linesep就能保证在不同的平台下面都能成功的写入换行符. 上面是从file中读取内容, 这里是向file中写入内容.*

**if** self.file:

self.file.write(fp + os.linesep)

**def** request\_fingerprint(self, request):

**return** request\_fingerprint(request)

*# 爬虫关闭的时候会把文件保存, 即所有的指纹都会存在到path路径下的requests.seen文件中.*

**def** close(self, reason):

**if** self.file:

self.file.close()

**def** log(self, request, spider):

**if** self.debug:

msg = **"Filtered duplicate request: %(request)s"**

self.logger.debug(msg, {**'request'**: request}, extra={**'spider'**: spider})

**elif** self.logdupes:

msg = (**"Filtered duplicate request: %(request)s"**

**" - no more duplicates will be shown"**

**" (see DUPEFILTER\_DEBUG to show all duplicates)"**)

self.logger.debug(msg, {**'request'**: request}, extra={**'spider'**: spider})

self.logdupes = **False**

spider.crawler.stats.inc\_value(**'dupefilter/filtered'**, spider=spider)

#### Scrapy\_redis之Scheduler



C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\_redis\scheduler.py

**class** Scheduler(object):

# 在scheduler调度器关闭的时候会调用close方法, 程序会读取settings.py中的persist的设置, 如果不是持久化爬取的, 就执行flush()方法, 把request的指纹和调度器中保存的待爬取的request对象的队列都给清除掉.

**def** close(self, reason):

**if not** self.persist:

self.flush()

**def** flush(self):

# df为dupefilter, 其中保存的是request的指纹

self.df.clear()

# queuq队列为待爬取的request对象

self.queue.clear()

# 定义request请求对象进入队列的方法

**def** enqueue\_request(self, request):

# dont\_filter转为为False, 即进行过滤. 加上not时and前面的条件就为True, 如果request中设置了为进行过滤. request\_seen在request对象之前已经被抓取到并把它的指纹保存到redis数据库的情况下为True. 两个条件加在一起表示当设置为对request进行过滤, 并且在指纹已经保存到数据库的情况下就返回Flase, request请求对象就不入队列.

# dont\_filter=False, not dont\_filter=True, request指纹已经存在, True 不会入队列

# dont\_filter=False, not dont\_filter=True, request指纹不存在, 全新的url, False, 会入队列

# dont\_filter=True, not dont\_filter=False, 会入队列, 会入队列就意味着不管以前有没有被seen过, 被提取过, 都会再次被请求.

**if not** request.dont\_filter **and** self.df.request\_seen(request):

self.df.log(request, self.spider)

**return False**

**if** self.stats:

self.stats.inc\_value(**'scheduler/enqueued/redis'**, spider=self.spider)

# 当设置为进行过滤, 并且指纹没有保存到redis数据库的情况下, 也就是遇到的是一个全新的url时, 它对应的request对象就会入队列.

self.queue.push(request)

**return True**

**def** next\_request(self):

block\_pop\_timeout = self.idle\_before\_close

request = self.queue.pop(block\_pop\_timeout)

**if** request **and** self.stats:

self.stats.inc\_value(**'scheduler/dequeued/redis'**, spider=self.spider)

**return** request

**def** has\_pending\_requests(self):

**return** len(self) > 0

### 京东图书爬虫

需求：抓取京东图书的信息

目标：抓取京东图书包含图书的名字、封面图片地址、图书url地址、作者、出版社、出版时间、价格、图书所属大分类、图书所属小的分类、分类的url地址

url：https://book.jd.com/booksort.html

大分类, 小分类, 小分类的url地址, 列表页, 详情页, 详情页中的价格等信息.

scrapy startproject jdbook

cd jdbook

scracy genspider jd book.jd.com

完善爬虫

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

**import** scrapy

**from** copy **import** deepcopy

**import** json

**import** urllib

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

name = **'jd'**

*# 因为要向p.3.cn发送请求获取图书的价格信息, 所以也要把它加入到允许的域名列表中.*

allowed\_domains = [**'jd.com'**,**'p.3.cn'**]

*# 确定想要的内容是否在start\_url的响应中. Elements中的元素与response中的元素相同.*

start\_urls = [**'https://book.jd.com/booksort.html'**]

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

*# 根据大分类进行分组*

dt\_list = response.xpath(**"//div[@class='mc']/dl/dt"**) *# 大分类列表*

**for** dt **in** dt\_list:

*# 在大分类下面建立的item, 后面使用meta传递时要使用deepcopy*

item = {}

*# 向item中添加大分类的名字字段*

item[**"b\_cate"**] = dt.xpath(**"./a/text()"**).extract\_first()

*# 根据小分类进行分组*

*# xpath获取下一个兄弟节点*

em\_list = dt.xpath(**"./following-sibling::dd[1]/em"**) *#小分类列表*

**for** em **in** em\_list:

*# 添加小分类的url地址.*

item[**"s\_href"**] = em.xpath(**"./a/@href"**).extract\_first()

item[**"s\_cate"**] = em.xpath(**"./a/text()"**).extract\_first()

*# 判断小分类的地址是否存在, 如果存在, 要补齐地址*

**if** item[**"s\_href"**] **is not None**:

item[**"s\_href"**] = **"https:"** + item[**"s\_href"**]

*# 构造小分类的request请求对象, 并把对应的响应交给parse\_book\_list进行解析*

**yield** scrapy.Request(

item[**"s\_href"**],

callback=self.parse\_book\_list,

meta = {**"item"**:deepcopy(item)}

)

*# 小分类对应图书列表页解析函数*

**def** parse\_book\_list(self,response):

item = response.meta[**"item"**]

*# 在netwrok中查看图书列表页的响应, 发现服务器会对我们的请求进行重定向. scrapy可以帮我们解决重定向的问题.*

*# 对每一本进行分组, 获取图书列表*

li\_list = response.xpath(**"//div[@id='plist']/ul/li"**)

**for** li **in** li\_list:

*# 获取图书列表页的图书图片地址, 要把它拼接完整*

item[**"book\_img"**] = li.xpath(**".//div[@class='p-img']//img/@src"**).extract\_first()

**if** item[**"book\_img"**] **is None**:

# 有的页面中图书的图片的url链接地址不一样, 所以如果上面的取不到内容, 就要使用下面的方法来取

item[**"book\_img"**] = li.xpath(**".//div[@class='p-img']//img/@data-lazy-img"**).extract\_first()

item[**"book\_img"**]=**"https:"**+item[**"book\_img"**] **if** item[**"book\_img"**] **is not None else None**

item[**"book\_name"**] = li.xpath(**".//div[@class='p-name']/a/em/text()"**).extract\_first().strip()

*# 图书可能有多个作者, 所以要使用extract*

item[**"book\_author"**] = li.xpath(**".//span[@class='author\_type\_1']/a/text()"**).extract()

item[**"book\_press"**]= li.xpath(**".//span[@class='p-bi-store']/a/@title"**).extract\_first()

item[**"book\_publish\_date"**] = li.xpath(**".//span[@class='p-bi-date']/text()"**).extract\_first().strip()

*# 提取每一本图书的j\_id值*

item[**"book\_sku"**] = li.xpath(**"./div/@data-sku"**).extract\_first()

*# 价格信息在列表页的响应中不存在, 在其它响应中, 所以还要重新发送请求获取价格信息.*

**yield** scrapy.Request(

*# 拼接一本书的json数据格式的地址.*

**"https://p.3.cn/prices/mgets?skuIds=J\_{}"**.format(item[**"book\_sku"**]),

callback=self.parse\_book\_price,

meta = {**"item"**:deepcopy(item)}

)

*#列表页翻页. 确定url地址的响应中是否有下一页的信息. 很多下一页的信息都是用js生成的.*

next\_url = response.xpath(**"//a[@class='pn-next']/@href"**).extract\_first()

**if** next\_url **is not None**:

*# 使用urllib.parse.urljoin自动拼接url地址*

next\_url = urllib.parse.urljoin(response.url,next\_url)

**yield** scrapy.Request(

next\_url,

callback=self.parse\_book\_list,

meta = {**"item"**:item}

)

*# 解析图书的价格信息*

**def** parse\_book\_price(self,response):

item = response.meta[**"item"**]

*# 根据图书的价格进行全局搜索, 会搜索到很多相同的价格, 根据页面上图书价格的顺序判断是哪个响应. 找到价格对应的响应之后, 看到对应的url非常复杂, 有很多%, 所以要进行解码.*

*# 从json中获取图书的价格*

item[**"book\_price"**] = json.loads(response.body.decode())[0][**"op"**]

print(item)

**yield** item

**"""**

**url = "https://p.3.cn/prices/mgets?callback=jQuery4805996&ext=11000000&pin=&type=1&area=1\_72\_2799\_0&skuIds=J\_11757834%2CJ\_12090377%2CJ\_12018031%2CJ\_10937943%2CJ\_12114139%2CJ\_10162899%2CJ\_12182317%2CJ\_11970294%2CJ\_11783136%2CJ\_12161302%2CJ\_12026957%2CJ\_10367073%2CJ\_11711801%2CJ\_12052514%2CJ\_11837713%2CJ\_12098764%2CJ\_10199768%2CJ\_11711801%2CJ\_11775777%2CJ\_10960247%2CJ\_12098746%2CJ\_12325789%2CJ\_11138599%2CJ\_11506268%2CJ\_11660913%2CJ\_12192773%2CJ\_11673849%2CJ\_11467452%2CJ\_12174895%2CJ\_11660916&pdbp=0&pdtk=&pdpin=&pduid=152279554607457015143&source=list\_pc\_front&\_=1527343335953"**

**https://p.3.cn/prices/mgets?callback=jQuery4805996&ext=11000000&pin=&type=1&area=1\_72\_2799\_0&skuIds=J\_11757834,J\_12090377,J\_12018031,J\_10937943,J\_12114139,J\_10162899,J\_12182317,J\_11970294,J\_11783136,J\_12161302,J\_12026957,J\_10367073,J\_11711801,J\_12052514,J\_11837713,J\_12098764,J\_10199768,J\_11711801,J\_11775777,J\_10960247,J\_12098746,J\_12325789,J\_11138599,J\_11506268,J\_11660913,J\_12192773,J\_11673849,J\_11467452,J\_12174895,J\_11660916&pdbp=0&pdtk=&pdpin=&pduid=152279554607457015143&source=list\_pc\_front&\_=1527343335953**

**这个地址返回的是json格式的图书信息数据.**

**callback可以删除**

**在页面中搜索 skuIds 中的数字 11757834, 12090377, 它们位于每本图书的div标签中, 类似于商品的编号,**

**前面的参数 "ext=11000000&pin=&type=1&area=1\_72\_2799\_0&" 和**

**后面的 "&pdbp=0&pdtk=&pdpin=&pduid=152279554607457015143&source=list\_pc\_front&\_=1527343335953" 一般都可以删除. 在浏览器中尝试, 看删除之后返回的数据是否有变化.**

**通过下面的地址也能返回正确的json数据**

**https://p.3.cn/prices/mgets?skuIds=J\_11757834,J\_12090377,J\_12018031,J\_10937943,J\_12114139,J\_10162899,J\_12182317,J\_11970294,J\_11783136,J\_12161302,J\_12026957,J\_10367073,J\_11711801,J\_12052514,J\_11837713,J\_12098764,J\_10199768,J\_11711801,J\_11775777,J\_10960247,J\_12098746,J\_12325789,J\_11138599,J\_11506268,J\_11660913,J\_12192773,J\_11673849,J\_11467452,J\_12174895,J\_11660916**

**上面的地址返回了很多书的信息, 使用下面的地址来获取一本书的数据**

**https://p.3.cn/prices/mgets?skuIds=J\_11757834**

**"""**

因为要实现持久化增量的爬虫效果, 需要修改settings.py文件

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

SCHEDULER\_PERSIST = **True**

REDIS\_URL = **"redis://127.0.0.1:6379"**

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

USER\_AGENT = **'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'**

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = **False**

只需要加入scrapy\_redis中的4行信息, 就可以把scrapy的spider修改为基于scrapy\_redis的持久化的爬虫. 修改useragent, 并且不遵守robots协议.

先在redis中执行flushdb, 删除掉redis中的所有数据.

执行爬虫, 等打印出提取的数据时停止爬虫.

使用keys \* 来查看redis中的所有内容

再次执行爬虫, 此时很快就会打印出提取到的数据. 第一次执行爬虫时要等待较长时间才能提取到数据, 而第二次执行时很快就得到了数据, 说明第二次爬取是从redis中已保存的待爬取队列中读取的数据. 即是接着上次运行的任务继续运行的.

所以在scrapy的settings.py中添加上4行scrapy\_redis的内容, 就可以实现增量式持久性的爬虫了, 之前爬取过的就不会再爬取了, 并且可以实现断点续爬的效果.

如果还想要爬取图书的评价信息. 就可以使用与价格类似的方法. 使用全局搜索找到评价的响应. 然后在Network中搜索响应的地址, 查看响应的预览.

https://sclub.jd.com/comment/productPageComments.action?callback=fetchJSON\_comment98vv48814&productId=11757834&score=0&sortType=5&page=0&pageSize=10&isShadowSku=0&fold=1

callback可以去掉. pageSize=10表示一页中显示出10条评价信息. 由于评价的内容都非常多, 如果按默认的一页显示10条评论, 要请求非常多的次数才能把所有的评论都爬下来, 可以尝试着修改10为100等, 再从中提取出评价的数据, 这样就可以大大的减少爬取的次数.

### Scrapy\_redis之RedisSpider



分布式爬虫的特点, 第1, 起始url地址并没有直接放在代码中, 而是放在了redis的myspider:start\_urls这个键对应的值中. 把编写好的爬虫代码复制到所有的电脑中, 在爬虫开始运行时, 不同电脑上的爬虫都会处于就绪的状态, 都在等待着读取redis中myspider:start\_urls中的起始url地址列表. 一旦向myspider:start\_urls中push放入了start\_urls的列表, 所有电脑上的爬虫都会尝试读取其中的地址, 这里的读取是pop方式的读取, 一旦某个爬虫最先读取到这个地址之后, 就会把它删除, 从而保证只有一台电脑上的爬虫能读取到1个start\_url地址, 从而避免了所有电脑都读取同一个start\_urls的问题. 当一台电脑读取到start\_urls的时候,读取到start\_urls的电脑就开始发送请求, 提取url地址构建request请求对象, 并把构建的request对象保存到redis中的 myspider:requests键中. 与此同时, 其它未读取到start\_urls地址的电脑都处于等待中, 直到redis的requests待爬取队列中出现request对象的时候. 这些处于等待的电脑才开始进行工作, 从request待爬取队列中读取request, 发送请求获取响应.

**from** scrapy\_redis.spiders **import** RedisSpider

# 第1点不同, 继承自redisSpider父类, 以前的爬虫都是继承自crawlspider或scrapy.spider.

**class** MySpider(RedisSpider):

"""Spider that reads urls from redis queue (myspider:start\_urls)."""

# 爬虫名

name = **'myspider\_redis'**

# 第2点不同. redis\_key对应于redis中的一个列表, spider启动之后, 会从redis中读取myspider:start\_urls对应的值. 如果按之前的方式启动这个爬虫, 由于缺少start\_urls, 爬虫启动之后是无法立即抓取数据的, 它会一直等待着myspider:start\_urls这个键中出出url地址, 一旦这个键中出现值, 爬虫就会从中取出url地址, 发送请求开始爬取数据. 这个基于RedisSpider的爬虫就是分布式的爬虫, 它能实现分布式爬虫的功能, 即多台电脑或服务器同时运行一个爬虫项目.

# start\_urls = [""]

redis\_key = **'myspider:start\_urls'**

# 不使用\_\_init\_\_方法中定义的自动获取domains的方法, 还使用手动确定allowed\_domain的方法.

# allowed\_domain = ["jd.com", "p.3.cn"]

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

# \_\_init\_\_方法动态的定义allowed domians列表, 相当于之前的allowed\_domains列表.

# allowed\_domain = ["jd.com", "p.3.cn"]

# Dynamically define the allowed domains list.

# kwargs是一个字典, 从中pop一个domain键对应的值, 保存到domain变量中. 如果其中没有domain这个键, 就把空字符串赋值给domain.

domain = kwargs.pop(**'domain'**, **''**)

# 实际上这里提取到的domain有时候是空值, 即使用这个方法的效果有时候并不好. 所以可以把这里的代码注释掉, 还使用以前的alowed\_domain方法,

# print(domain)

# 对domain进行分隔, 得到一个列表, 列表中的每个元素都是一个start\_url地址.

self.allowed\_domains = filter(**None**, domain.split(**','**))

# 继承父类的\_\_init\_\_方法.

super(MySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

# parse方法会最先处理\_urls中地址的响应, 返回name和url这两个键到pipeline中.

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

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

pipelines.py 与之前的pipeline相同, 给item添加了两个键.

**from** datetime **import** datetime

**class** ExamplePipeline(object):

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

item[**"crawled"**] = datetime.utcnow()

item[**"spider"**] = spider.name

**return** item

所以这个爬虫除了spider中的allowed\_domain和redis\_key不同之外, 其它的内容都是相同的. 这样就能够实现分布式爬虫的效果.

### dangdang爬虫

需求：抓取当当图书的信息

目标：抓取当当图书又有图书的名字、封面图片地址、图书url地址、作者、出版社、出版时间、价格、图书所属大分类、图书所属小的分类、分类的url地址

url：http://book.dangdang.com/

在之前的book项目中创建danagdang图书的爬虫

cd book

# 创建的方法与之前一样, 只需要把爬虫中继承的父类和redis\_key进行修改即可.

scrapy genspider dangdang dangdang.com

修改dangdang.py

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

**import** scrapy

**from** scrapy\_redis.spiders **import** RedisSpider

**from** copy **import** deepcopy

**import** urllib

*# 修改继承的父类为RedisSpider*

**class** DangdangSpider(RedisSpider):

name = **'dangdang'**

allowed\_domains = [**'dangdang.com'**]

*# 确定start\_urls, 还是首先要确定响应中有没有想要的内容. 使用检查元素在Elements中查找想要的内容, 然后在响应是看是否存在.*

*# start\_urls = ['http://book.dangdang.com/']*

*# 使用redis\_key来代替start\_urls, 其中的键可以任意命名, 这里取名为dangdang, 就可以在redis中向dangdang这个键写入start\_urls列表*

redis\_key = **"dangdang"**

*# 处理start\_urls的响应*

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

*#大分类分组*

div\_list = response.xpath(**"//div[@class='con flq\_body']/div"**)

**for** div **in** div\_list:

item = {}

*# 获取大分类的名称, 有的大分类名称在elements中dt下面的span标签中, 但源码中没有span标签, 所以不能写span标签. 而有的是在dt下面的a标签中, 所以这里要使用//text()来提取内容. 这里就不能使用extract\_first()来提取内容了, 只能使用extract(). 一些大分类有多个, 所以这里要使用extract, 并且返回一个列表.*

item[**"b\_cate"**] = div.xpath(**"./dl/dt//text()"**).extract()

*# 因为取的是dt//text(), dt和a标签之间可能会有换行符, 所以要进行处理. 如果去掉换行符等之后不是空的内容, 就把它保存下来. 可以放在pipeline中进行处理, 也可以在这里处理.*

item[**"b\_cate"**] = [i.strip() **for** i **in** item[**"b\_cate"**] **if** len(i.strip())>0]

*#中间分类分组.*

dl\_list = div.xpath(**"./div//dl[@class='inner\_dl']"**)

**for** dl **in** dl\_list:

*# 中间分类的名称*

*# 与上面中间分类相似, 有的中间分类是放在a标签下的, 有的没有放在a标签下, 所以如果使用dt/a/text()可能会取不到内容. 所以这里还使用dt//text()来提取. 并要对提取的内容进行处理. 每个小分类只有1个, 所以可以使用[0]取出来它的文本. 如果不放心, 也可以使用类似大分类的处理方式, 不提取其中的元素, 只保存为列表即可.*

item[**"m\_cate"**] = dl.xpath(**"./dt//text()"**).extract()

item[**"m\_cate"**] = [i.strip() **for** i **in** item[**"m\_cate"**] **if** len(i.strip())>0][0]

*#小分类分组*

a\_list = dl.xpath(**"./dd/a"**)

**for** a **in** a\_list:

item[**"s\_href"**] = a.xpath(**"./@href"**).extract\_first()

item[**"s\_cate"**] = a.xpath(**"./text()"**).extract\_first()

*# 构造每个小分类下获取图书的列表的请求.*

**if** item[**"s\_href"**] **is not None**:

**yield** scrapy.Request(

item[**"s\_href"**],

callback=self.parse\_book\_list,

meta = {**"item"**:deepcopy(item)}

)

*# 解析图书列表页的response响应*

**def** parse\_book\_list(self,response):

item = response.meta[**"item"**]

*# 对图书进行分组. 首先需要确定要提取的数据在响应中, 如果在响应中就可以提取*

li\_list = response.xpath(**"//ul[@class='bigimg']/li"**)

**for** li **in** li\_list:

*# 图书的内容在响应中都存在, 提取图书的信息*

item[**"book\_img"**] = li.xpath(**"./a[@class='pic']**[**/img/@src"**).extract\_first](mailto:/img/@src%22).extract_first)()

# 有些图书用上面的规则提取到的内容是**url\_none.png时**, 要进行判断后重新提取.

**if** item[**"book\_img"**] == **"images/model/guan/****url\_none.png"**:

item[**"book\_img"**] = li.xpath(**"./a[@class='pic']/img/@data-original"**).extract\_first()

item[**"book\_name"**] = li.xpath(**"./p[@class='name']/a/@title"**).extract\_first()

item[**"book\_desc"**] = li.xpath(**"./p[@class='detail']/text()"**).extract\_first()

item[**"book\_price"**] = li.xpath(**".//span[@class='search\_now\_price']/text()"**).extract\_first()

*# 可能会存在多个作者, 所以要使用extract()*

item[**"book\_author"**] = li.xpath(**"./p[@class='search\_book\_author']/span[1]/a/text()"**).extract()

item[**"book\_publish\_date"**] = li.xpath(**"./p[@class='search\_book\_author']/span[2]/text()"**).extract\_first()

item[**"book\_press"**] = li.xpath(**"./p[@class='search\_book\_author']/span[3]/a/text()"**).extract\_first()

print(item)

先清空redis中的内容. flushdb

keys \*

启动spider进行测试. scrapy crawl dangdang

会出现 telnet console listening on 127.0.0.1:6073

等待着从redis的dangdang这个键中读取start url.

向redis的dangdang键中插入url地址

lpush dangdang http;//book.dangdang.com/ # dangdang此时就是一个列表, 只有列表才有lpush和rpush的操作

只要执行了向redis的danagdang键中写入数据, 爬虫就会开始运行.

补全下一页的提取代码

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

**import** scrapy

**from** scrapy\_redis.spiders **import** RedisSpider

**from** copy **import** deepcopy

**import** urllib

*# 修改继承的父类为RedisSpider*

**class** DangdangSpider(RedisSpider):

name = **'dangdang'**

allowed\_domains = [**'dangdang.com'**]

*# 确定start\_urls, 还是首先要确定响应中有没有想要的内容. 使用检查元素在Elements中查找想要的内容, 然后在响应是看是否存在.*

*# start\_urls = ['http://book.dangdang.com/']*

*# 使用redis\_key来代替start\_urls, 其中的键可以任意命名, 这里取名为dangdang, 就可以在redis中向dangdang这个键写入start\_urls列表*

redis\_key = **"dangdang"**

*# 处理start\_urls的响应*

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

*#大分类分组*

div\_list = response.xpath(**"//div[@class='con flq\_body']/div"**)

**for** div **in** div\_list:

item = {}

*# 获取大分类的名称, 有的大分类名称在elements中dt下面的span标签中, 但源码中没有span标签, 所以不能写span标签. 而有的是在dt下面的a标签中, 所以这里要使用//text()来提取内容. 这里就不能使用extract\_first()来提取内容了, 只能使用extract()*

item[**"b\_cate"**] = div.xpath(**"./dl/dt//text()"**).extract()

*# 因为取的是dt//text(), dt和a标签之间可能会有换行符, 所以要进行处理. 如果去掉换行符等之后不是空的内容, 就把它保存下来. 可以放在pipeline中进行处理, 也可以在这里处理.*

item[**"b\_cate"**] = [i.strip() **for** i **in** item[**"b\_cate"**] **if** len(i.strip())>0]

*#中间分类分组.*

dl\_list = div.xpath(**"./div//dl[@class='inner\_dl']"**)

**for** dl **in** dl\_list:

*# 中间分类的名称*

*# 与上面中间分类相似, 有的中间分类是放在a标签下的, 有的没有放在a标签下, 所以如果使用dt/a/text()可能会取不到内容. 所以这里还使用dt//text()来提取. 并要对提取的内容进行处理. 当中间分类没有放在a标签中时, 中间分类只有1个, 此时是可以使用extract\_first()来提取的, 只需要把提取的内容进行strip()处理即可,*

*# item["m\_cate"] = dl.xpath("./dt//text()").extract\_first().strip()*

item[**"m\_cate"**] = dl.xpath(**"./dt//text()"**).extract()

item[**"m\_cate"**] = [i.strip() **for** i **in** item[**"m\_cate"**] **if** len(i.strip())>0][0]

*#小分类分组*

a\_list = dl.xpath(**"./dd/a"**)

**for** a **in** a\_list:

item[**"s\_href"**] = a.xpath(**"./@href"**).extract\_first()

item[**"s\_cate"**] = a.xpath(**"./text()"**).extract\_first()

*# 构造每个小分类下获取图书的列表的请求.*

**if** item[**"s\_href"**] **is not None**:

**yield** scrapy.Request(

item[**"s\_href"**],

callback=self.parse\_book\_list,

meta = {**"item"**:deepcopy(item)}

)

*# 解析图书列表页的response响应*

**def** parse\_book\_list(self,response):

item = response.meta[**"item"**]

*# 对图书进行分组. 首先需要确定要提取的数据在响应中, 如果在响应中就可以提取*

li\_list = response.xpath(**"//ul[@class='bigimg']/li"**)

**for** li **in** li\_list:

*# 图书的内容在响应中都存在, 提取图书的信息*

item[**"book\_img"**] = li.xpath(**"./a[@class='pic']/img/@src"**).extract\_first()

**if** item[**"book\_img"**] == **"images/model/guan/url\_none.png"**:

item[**"book\_img"**] = li.xpath(**"./a[@class='pic']/img/@data-original"**).extract\_first()

item[**"book\_name"**] = li.xpath(**"./p[@class='name']/a/@title"**).extract\_first()

item[**"book\_desc"**] = li.xpath(**"./p[@class='detail']/text()"**).extract\_first()

item[**"book\_price"**] = li.xpath(**".//span[@class='search\_now\_price']/text()"**).extract\_first()

*# 可能会存在多个作者, 所以要使用extract()*

item[**"book\_author"**] = li.xpath(**"./p[@class='search\_book\_author']/span[1]/a/text()"**).extract()

item[**"book\_publish\_date"**] = li.xpath(**"./p[@class='search\_book\_author']/span[2]/text()"**).extract\_first()

item[**"book\_press"**] = li.xpath(**"./p[@class='search\_book\_author']/span[3]/a/text()"**).extract\_first()

print(item)

yield item

*# 提取下一页的url地址*

next\_url = response.xpath(**"//li[@class='next']/a/@href"**).extract\_first()

**if** next\_url **is not None**:

*# 对下一页的url地址进行补全.*

next\_url = urllib.parse.urljoin(response.url,next\_url)

**yield** scrapy.Request(

next\_url,

callback=self.parse\_book\_list,

*# 因为处理的是列表页, 不同的列表页中主分类和子分类的地址是不变的, 所以即使下一页的数据把上一页的数据覆盖掉, 也不会出现数据的重复. 所以这里不用使用deepcopy*

meta = {**"item"**:item}

)

使用ifconifg或ipconfig查看本机的ip地址

修改settings.py中的redis\_url为本机的ip地址.

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

SCHEDULER\_PERSIST = **True**

REDIS\_URL = **"redis://192.168.82.64:6379"**

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

USER\_AGENT = **'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_13\_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36'**

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = **False**

在本机redis中执行flushdb清空所有的数据库.

压缩book项目文件夹. 分发到所有分布的电脑上, 然后在各个电脑上运行爬虫, 此时所有的爬虫都处于等待状态.

在本机上向redis中的dangdang键中添加值.

lpush dangdang <http://book.dangdang.com/>

客户端的某一台电脑读取到这个start\_url地址, 就开始抓取数据, 随后所有的电脑都开始抓取数据. 当一台电脑从本机的redis中读取到一个request对象后, 就把它删除掉, 这样就保证了各个电脑中爬取到的数据是不重复的.

回顾当当爬虫, 分类为大分类, 中间分类和小分类.

注意构建图书列表页的request对象时传递的meta信息要使用deepcopy.

要抓取的图书信息都处于列表页, 不需要进入图书详情页抓取数据.

想要加快爬取的速度, 最简单的方法就是使用分布式. 只需要把scrapy爬虫继承的类修改为RedisSpider, 修改allowed\_domains和redis\_key, 并在settings.py中进行一系列的设置, 就能把scrapy爬虫改造为分布式的爬虫.

使用redis\_key来代替start\_url, 向各个电脑公用的redis数据库的dangdang键中插入一条地址. 只有1台电脑会抓取到这个地址.

### Scrapy\_redis之RedisCrawlSpider



**from** scrapy.spiders **import** Rule

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

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

*# 注意这里是继承自RedisCrawlSpider, 即为crawlspider的分布式爬虫类, 可以实现自动提取url地址和分布式的功能.*

**class** MyCrawler(RedisCrawlSpider):

*"""Spider that reads urls from redis queue (myspider:start\_urls)."""*

name = **'mycrawler\_redis'**

*# redis\_key = 'mycrawler:start\_urls'*

*# 注意这里可以使用任意值作为start\_urls的键名.*

redis\_key = **'spider'**

*# 可以不使用\_\_init\_\_方法中的自动获取allowed\_domain的功能, 而使用自定义的功能.*

allowed\_domain = []

rules = (

*# follow all links*

*# 把提取到的链接地址的响应交给parse\_page来提取. 注意在crawl spider中的parse函数是用来自动对提取到的链接的响应进行处理的. 如果不指定callback函数, 提取到的链接的响应是自动交给parse函数处理的.*

Rule(LinkExtractor(), callback=**'parse\_page'**, follow=**True**),

)

*# def \_\_init\_\_(self, \*args, \*\*kwargs):*

*# # Dynamically define the allowed domains list.*

*# domain = kwargs.pop('domain', '')*

*# self.allowed\_domains = filter(None, domain.split(','))*

*# super(MyCrawler, self).\_\_init\_\_(\*args, \*\*kwargs)*

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

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

### 亚马逊图书爬虫

需求：抓取亚马逊图书的信息

目标：抓取亚马逊图书又有图书的名字、封面图片地址、图书url地址、作者、出版社、出版时间、价格、图书所属大分类、图书所属小的分类、分类的url地址

url：<https://www.amazon.cn/%E5%9B%BE%E4%B9%A6/b/ref=sd_allcat_books_l1?ie=UTF8&node=658390051>

https://www.amazon.cn/图书/b/ref=sd\_allcat\_books\_l1?ie=UTF8&node=658390051

<https://www.amazon.cn/gp/book/all_category>

新建爬虫

scrapy genspider -t crawl amazon amazon.cn

修改amazon.py

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

**import** scrapy

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

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

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

**import** re

*# 修改继承的父类*

**class** AmazonSpider(RedisCrawlSpider):

name = **'amazon'**

allowed\_domains = [**'amazon.cn'**]

*# 修改start\_urls为redis\_key*

*# start\_urls = ['https://www.amazon.cn/%E5%9B%BE%E4%B9%A6/b/ref=sd\_allcat\_books\_l1?ie=UTF8&node=658390051']*

redis\_key = **"amazon"**

rules = (

*# 匹配大分类的url地址和小分类的url. 这里不再使用re正则匹配来获取url地址, 而是使用xpath来匹配并提取url地址. 只需要写到url所在的a标签的父级标签即可, 程序会自动获取li下面所有a标签的url地址. 提取到的是大分类的url地址, 所以不需要指定解析函数, 而是使用默认的parse函数来解析. 并且设置follow=True, 继续对子分类中的url地址进行提取. 子分类和大分类的xpath的提取规则相同, 所以使用这一个xpath提取规则可以提取出大分类和小分类的url地址.*

Rule(LinkExtractor(restrict\_xpaths=(**"//div[@class='categoryRefinementsSection']/ul/li"**,)), follow=**True**),

*#匹配图书的url地址. 可以先定位到图书的标题标签h2, 再定位到它的父级, 由程序自动获取a标签中的href链接. 因为是详情页, 需要指定其响应的解析函数. 因为是详情页, 就不需要对链接进行跟进了.*

Rule(LinkExtractor(restrict\_xpaths=(**"//div[@id='mainResults']/ul/li//h2/.."**,)),callback=**"parse\_book\_detail"**),

*#列表页翻页*

Rule(LinkExtractor(restrict\_xpaths=(**"//div[@id='pagn']"**,)),follow=**True**),

)

*# 解析详情页的响应, 提取图书的信息*

**def** parse\_book\_detail(self,response):

*# 匹配不到图书的简介, 可以把response的内容写入到本地文件中, 进行测试*

*# with open(response.url.split("/")[-1]+".html","w",encoding="utf-8") as f:*

*# f.write(response.body.decode())*

item = {}

item[**"book\_title"**] = response.xpath(**"//span[@id='productTitle']/text()"**).extract\_first()

item[**"book\_publish\_date"**] = response.xpath(**"//h1[@id='title']/span[last()]/text()"**).extract\_first()

*# 因为图书作者可能有多个, 所以使用extract()提取*

item[**"book\_author"**] = response.xpath(**"//div[@id='byline']/span/a/text()"**).extract()

*# 提取到的图片是经过base64转码后的. 如果想要把图片也保存在数据库中, 可以保存图片经过base64转码后的字符串. 因为图片的字符串非常长, 在调试时可以先注释掉, 在写入数据库时再取消注释*

*# item["book\_img"] = response.xpath("//div[@id='img-canvas']/img/@src").extract\_first()*

item[**"book\_price"**] = response.xpath(**"//div[@id='soldByThirdParty']/span[2]/text()"**).extract\_first()

*# 提取图书详情页中面包屑中的分类信息, 因为是多级的分类, 所以使用extract()*

item[**"book\_cate"**] = response.xpath(**"//div[@id='wayfinding-breadcrumbs\_feature\_div']/ul/li[not(@class)]/span/a/text()"**).extract()

*# item["book\_cate"] = response.xpath("//div[@id='wayfinding-breadcrumbs\_feature\_div']/ul/li/span/a/text()").extract()*

*# 去掉分类中空白字符*

item[**"book\_cate"**] = [i.strip() **for** i **in** item[**"book\_cate"**]]

item[**"book\_url"**] = response.url

*# 出版社的位置不确定, 所以使用文本定位*

item[**"book\_press"**] = response.xpath(**"//b[text()='出版社:']/../text()"**).extract\_first()

*# 获取图书的介绍. 图书的介绍是放在noscript标签中的, 然后使用js渲染到页面中的指定位置处. 因为noscript标签没有可以定位的class或id等结构, 所以这里要使用re进行匹配. re.S匹配到换行符.*

*# item["book\_desc"] = re.findall(r'<noscript>.\*?<div>(.\*?)</div>.\*?</noscript>',response.body.decode(),re.S)*

*# item["book\_desc"] = item["book\_desc"][0].split("<br>",1)[0] if len(item["book\_desc"])>0 else None*

*# 由于使用re无法正确匹配图书简介, 在这里使用xpath提取图片的简介信息*

item[**"book\_desc"**] = response.xpath(**"//noscript/div/text()"**).extract()

*# 对简介的内容进行处理, 提取内容不是空的且内容不是海报的*

item[**"book\_desc"**] = [i.strip() **for** i **in** item[**"book\_desc"**] **if** len(i.strip())>0 **and** i!=**'海报：'**]

*# 有些图书的简介中有<br>的海报图片, 所以要以<br>对book\_desc进行切割, 设置为1表示只切割为2部分. 只取第1部分就得到图书的简介了. 如果图书的简介中并没有br标签, 以br进行切割后得到简介自身的列表, 取第0个元素就得到图书的简介本身. 可以把图书简介的文本复制到ipython中进行测试. 或者使用requests模块进行测试.*

print(item)

# **yield** item

清空redis中的数据 flushdb

启动爬虫 scrapy crawl amazon

向redis中写入start\_url, 为了能更快的抓取到内容, 可以把图书的详情页作为起始地址.

lpush amazon https://www.amazon.cn/dp/B07BF25DNQ/ref=lp\_658508051\_1\_1?s=books&ie=UTF8&qid=1527397943&sr=1-1

可以使用htmpparser对编码后的html的内容进行解码

回顾amazon爬虫, 它继承的类不一样, 现在的爬虫继承于RedisCrawlSpider, 就能成为一个分布式的crawlspider爬虫.

自动动手修改dangdang的爬虫为redis crawl spider爬虫. 当使用crawl spider的爬虫时, 因为程序能够自动提取出链接地址, 所以能够少写很多代码.

### 使用Pycharm来发布代码

Tools > Deployment > Configuration > Add Server > Name: my\_spider, Type: SFTP > 打开ubuntu虚拟机, 查看它的ip地址, 为10.211.55.11, 修改my\_spider > 在connection选项卡中, SFTP host : 10.211.55.11, Test SFTP connection, 输入ubuntu的用户名和密码进行连接测试. > Root path: /home/zdw > 在Mapping选项卡中, 修改Deployment path on server 'my\_spider': 这里选择的是Root path中定义的目录下面的目录. 可以在zdw下面新建文件夹, 也可以使用zdw作为目录. 确定

在项目根目录下右键 > Upload to my\_spider >

在ubuntu的虚拟机中打开/home/zdw, 就会看到book项目目录.

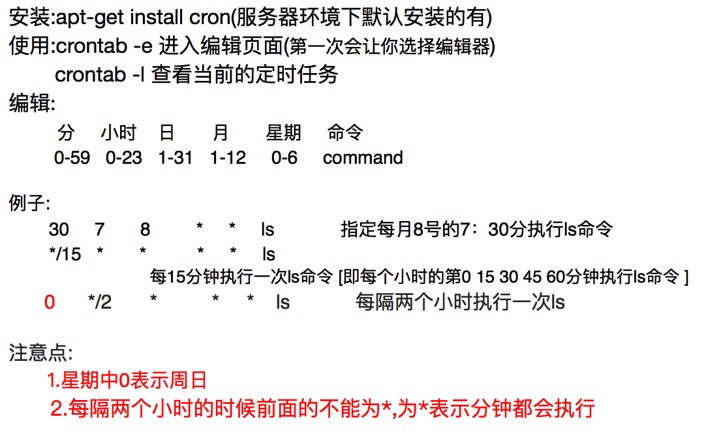
如果在ubuntu这个服务器中的代码被修改了, 还可以在本机上的book项目根目录中右键选择"Synchronize 'book' " 即同步服务器和本地的代码. 也可以选择compare with 进行比较.

### Crontab爬虫定时执行

For example, you can run a backup of all your user accounts at 5 a.m every week with:

0 5 \* \* 1 tar -zcf /var/backups/home.tgz /home/

\*表示每个. 这里表示在每周执行. \* 在哪个时间段下就表示每个这个时间都会执行.



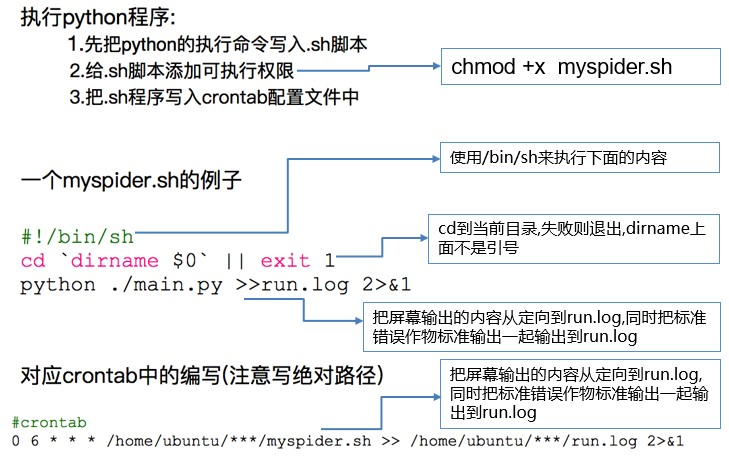
0 \*/2 \* \* \* ls 每隔两个小时执行一次ls

注意分钟这里必须要写为0, 如果写为\*就表示每分钟执行一次了.

可以类似上面的, 使用crontab -e进入编辑页面, 然后创建一个定时任务.

0 0 \* \* \* scrapy spider dangdang

就可以在每天的0点0分执行程序了. 但一般不这么使用. 而是按下面的步骤创建定时任务.



第1步: 把python的执行命令写入到.sh脚本中.

创建 myspider.sh

# !/bin/sh

# 使用/bin/sh来执行下面的内容

# cd到当前的目录中, 两个||表示或, 如果前面的代码执行失败就执行||后面的代码, 即如果没有成功进入当前目录就退出, dirname要放在反引号``中.

cd `dirname $0` || exit 1

# 如果成功进入到dirname的目录中, 就执行下面的代码.

# 使用python执行当前目录下的main.py程序, 把执行main.py过程中屏幕输出的内容重定向到run.log中, 同时把标准错误作为标准输出一起输出到run.log中. 前面的内容只能实现把标准输出重定向到run.log中, 0表示标准输入, 1表示标准输出, 2表示标准错误, 默认情况下只会把标准输出重定向到run.log中, 标准错误是不会重定向到run.log中去的, 2>&1, 就表示把标准错误也作为标准输出, 这时标准错误也会和标准输出一起重定向到run.log中.

python ./main.py >>run.log 2>&1

第2步: 给脚本添加可执行权限.

chmod +x myspider.sh

第3步: 把脚本sh程序写入到crontab配置文件中.

对应于crontab中的编写(注意使用的是绝对路径)

# crontab

# 把屏幕输出的内容重定向到run.log, 同时把标准错误作为标准输出 一起输出到run.log. 这里写的2>&1 表示把脚本执行过程中的错误也重定向到run.log中去.

0 6 \* \* \* /home/ubuntu/\*\*\*/myspider.sh >> /home/ubuntu/\*\*/run.log 2>&1

在ubuntu中进行操作

1. 编写自己的程序

emacs hello.py

print("hello world")

print("\*"\*100)

print("hello world")

2. 编写sh文件

emacs test.sh

# !/bin/sh

cd `dirname $0` || exit 1

python ./main.py >>run.log 2>&1

3. 给sh文件添加可执行权限

chmod + x test.sh

4. 执行sh文件

./test.sh

5. 就会生成run.log文件

cat run.log

python: can't open file 'main.py': [Errno 2] No such file or directory

出现这个错误的原因是在test.sh文件中使用的不是hello.py, 而是main.py. 修改test.sh文件

# !/bin/sh

cd `dirname $0` || exit 1

python ./hello.py >>run.log 2>&1

rm run.log

rm test.sh~

再次执行程序.

./test.sh

cat run.log # 就看到hello.py中执行时打印的内容了.

把sh文件放入到crontab中.

crontab -e

# 每分钟都会执行

\* \* \* \* \* /home/zdw/test.sh >> /home/zdw/run1.lgo 2>&1

crontab -l # 查看添加的定时任务

等待1分钟, ls查看, 就会在当前目录下出现run1.log文件.

cat un.log

/home/zdw/test.sh: 3: /home/zdw/test.sh: python: not found

whereis python # 查看python的可执行文件的位置

# 补全test.sh中的python命令

emacs test.sh

# !/bin/sh

cd `dirname $0` || exit 1

/usr/bin/python3.5 ./hello.py >>run.log 2>&1

rm run.log

rm run1.log

rm test.sh~

cat run.log

cat run1.log

# 会停留在运行run.log的窗口中, 只要向run.log中写入了新的内容, 都会显示出来. tail即显示文件中最后写入的几行内容

tail -f run.log