分布式爬虫打造亿级搜索引擎

**第2章 开发环境搭建**

1. IDE-------pycharm

2. 数据库------mysql、redis、elasticsearch

3. 开发环境----virtualenv

**2-1 pycharm的安装和简单使用**

pycharm在windows和linux下的安装

1. pycharm的安装（linux和windows）

2. pycharm查找license server以及导入工程

Linux中启动pycharm

使用 ./pycharm.sh来启动pycharm

vim ~/.bashrc

添加一行

alias pycharm="bash /home/david/Downloads/pycharm-2017.3.2/bin/pycharm.sh"

source ~/.bashrc

使用上面的命令生效.

在命令行中输入pycharm就能打开pycharm

settings中的搜索栏中输入keymap, 就可以设置常用的快捷方式样式.

interpreter中可以修改Project的Interpreter, 即此项目使用的解释器

**2-2 mysql和navicat的安装和使用**

# 安装mysql服务器和客户端

sudo apt install mysql-server mysql-client

# 查看mysql服务是否启动

ps aux|grep mysqld

# 连接mysql

mysql -uroot -p

输入密码

# 查看所有的数据库

show databases;

exit; 退出

如何来配置mysql, 使windows系统能连接到虚拟机中的mysql数据库

sudo vim /etc/mysql/mysql.conf.d/mysqld.cnf

vim /etc/mysql/mysql.conf.d/mysqld.cnf

其中有一个bind-address的选项, 默认情况下是只在本地启动的.

把它的值改为0.0.0.0就可以在外部访问了.

# bind-address 127.0.0.1

bind-address 0.0.0.0

修改之后要重启mysql服务器、

sudo service mysql restart

ifconif查看ubuntu虚拟机的ip地址, 然后在windows的Navicat中新建一个mysql的连接, 但是发现不能连接.

还必须要为root用户进行授权, 使它能够在非127.0.0.1的ip地址上也能访问.

在mysql命令行中运行.

GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'root' WITH GRANT OPTION;

# grant all privileges获取所有权限, 也可以设置只允许查找的权限. \*.\*是一个正则表达式, 表示对所有的数据表修改权限. TO 'root'表示对root这个用户赋权限. @'%'表示对所有外部的IP, 也可以指定某个特定的IP. IDENTIFIED BY 'root'表示密码.

# 刷新权限

flush privileges;

在navicat中连接的名称上右键新建数据库---->数据库名为scrapyspider---->字符集要选择utf8--UTF-8 Unicode---->排序规则要选择utf8\_general\_ci---->打开数据库, 在表中右键新建数据表.

添加栏位

栏位中设置

名: add\_time, 类型: datetime, 长度0, 小数点0.

名: name, 类型: varchar, 长度100

添加一个id的主键

名: id, 类型: int, 长度10, 勾选不是null, 点击工具栏上的主键按钮就可以把它设置为主键.

保存, 输入表名article

添加一个数据, add\_time中选择一个日期, name: bobby.

在查询中新建查询, 查询编辑器中就可以执行sql语句了.

select \* from article WHERE id=1

在某数据库上点击数据传输, 就可以把本地的数据库传输到目标数据库了.

在数据库上右击选择转储SQL文件, dump SQL File > structure and data, 就可以导出数据库了. 在数据库上右键选择 Execute SQL File, 就能运行并导入之前保存的SQL文件和数据了.

**2-3 windows和linux下安装python2和python3**

**windows安装python**

如果同时安装python2和python3, 在windows的环境变量下, 查看是哪个版本的python, 把python3的安装目标添加到path环境变量中. 同时删除python2的path路径.

安装到C盘根目录下.

**Linux安装python**

apt install python3.5

直接输入python默认是使用的python2

输入python3, 就是使用的python3 的版本

**使用pipy国内镜像**

https://pypi.douban.com/simple 豆瓣

https://mirrors.ustc.edu.cn/pypi/web/simple/ 中国科学技术大学

https://mirrors.aliyun.com/pypi/simple/ 阿里云

<https://pypi.tuna.tsinghua.edu.cn/simple/> 清华大学

使用方法

pypi清华大学源

https://mirrors.tuna.tsinghua.edu.cn/help/pypi/

1 临时使用

可以在使用pip的时候加参数-i https://pypi.tuna.tsinghua.edu.cn/simple

pip install -i https://pypi.tuna.tsinghua.edu.cn/simple some-package

注意, simple 不能少, 是 https 而不是 http

例如:

pip install -i https://pypi.tuna.tsinghua.edu.cn/simple/ requests

这样就会从清华的镜像去安装requests库.

2. 配置为默认pip更新源

Linux下, 修改 ~/.pip/pip.conf (没有就创建一个)

#在当前用户主目录创建.pip文件夹

cd ~

mkdir .pip

#在.pip目录创建并编辑pip.conf

#pip安装需要使用的https加密, 所以在此需要添加trusted-host

[install]

trusted-host = mirrors.ustc.edu.cn

[global]

index-url = https://mirrors.ustc.edu.cn/pypi/web/simple

3.对于window系统修改配置文件在 %HOMEPATH%\pip\pip.ini , 基本配置与linux版的配置相同

windows下, 直接在user目录中创建一个pip目录, 如: C:\Users\xxx\pip, 新建文件pip.ini.

PyPI 镜像使用帮助

地址

https://mirrors.shu.edu.cn/pypi/

使用说明

如何使用上海大学开源镜像站 加速 pip？

编辑 ~/.pip/pip.conf 文件（如果没有则创建之），将 index-url开头的一行修改为下面一行：(Windows配置文件位置为~/.pip/pip.ini,Mac配置文件位置为~/.pip.pip.conf)

[global]

index-url = https://mirrors.shu.edu.cn/pypi/web/simple

format = columns

**2-4 虚拟环境的安装和配置**

3. vitualenv和virtualenvwrapper安装和配置

虚拟环境可以将开发环境进行隔离, 相互之间不会影响.

如有的是用python2开发的, 有的是用python3开发的, 使用虚拟环境就能解决这个问题.

https://pip.pypa.io//en/latest/user\_guide/#

**Windows下安装虚拟环境**

python -m pip install --upgrade pip

pip install virtualenv

进入到想要保存虚拟环境的目录中

如D:\David\Documents\PythonVirutalEnv中, 执行

# 创建虚拟环境

virtualenv scrapytest

# 进行虚拟环境Scripts目录

cd ./scrapytest/Scripts

# 运行scrapytest的虚拟环境了

activate.bat

# 退出当前的虚拟环境

deactivate.bat

# 在虚拟环境中输入python就能看到python的版本了.

python --version

python -V

在创建虚拟环境时可以指定python的版本.

cd D:\David\Documents\PythonVirutalEnv

virtualenv -p C:\Python35\python.exe scrapytestpy3

./scrapytestpy3/Scripts/activate.bat启动虚拟环境

再次输入python查看版本

**在Linux下安装python虚拟环境**

# 安装虚拟环境包

sudo apt install python-virtualenv

# 新建虚拟环境

viurtalenv scrapytestpy2

# 进入虚拟环境bin目录

cd scrapytestpy2/bin

# 执行activate 运行虚拟环境

source activate

# 新建python3的虚拟环境

virtualenv -p /usr/bin/python3 scrpytestpy3

**虚拟环境管理的包virtualenvwapper**

但以上必须要进入到虚拟环境的目录中执行代码, 很不方便. 可以使用virtualenvwapper来管理虚拟环境.

**windows下安装**

pip install virtualenvwrapper-win

使用workon xxx就可以进入到对应的虚拟环境中.

如果只输入workon就可以列出所有虚拟环境了.

新建虚拟环境

mkvirtualenv scrapytest

使用virtualenvwrapper新建的虚拟环境默认放在C:\Users\David\Envs\中.

可以修改此默认保存的位置

新建系统环境变量, 变量名: WORKON\_HOME, 变量值E:\Envs\

重启命令行窗口. 再执行workon, 因为工作目录已经修改为E:\Envs\, 这个文件夹是空的, 就找不到虚拟环境了. 可以把以前的虚拟环境都拷备过去即可.

mkvirtualenv scrapytest, 就在WORKON\_HOME路径下创建了一个虚拟环境并且自动进入这个虚拟环境了.

使用deactivate退出虚拟环境.

可以在虚拟环境下使用pip进行软件包的安装.

workon scrapytest

pip install scrapy

会出错, 找不到python3.5对应的twisted版本

在http://www.lfd.uci.edu/~gohlke/pythonlibs/上查找python3.5的twisted. 下载, 然后进入到下载目录中进行安装.

pip install Twisted-17.1.0-cp35-cp35m-win\_amd64.whl

pip install scrapy就能成功安装了.

使用mkvirtualenv --python=

来指定某版本的python

scrapy安装说明

https://doc.scrapy.org/en/master/intro/install.html

**linux下安装**

pip install virtualenvwrapper

但linux下不能像windows下一样, 直接使用mkvirtualenv, 首先要找到virutalenvwrapper.sh

sudo find / -name virtualenvwrapper.sh

/home/david/.local/bin/virtualenvwrapper.sh

vim ~/.bashrc

# WORKON\_HOME设置虚拟环境保存的目录, $HOME为用户的主目录, 即设置虚拟环境保存在用户主目录下的.virtualenvs目录下.

export WORKON\_HOME=$HOME/.virtualenvs

#RTUALENVWRAPPER\_PYTHON='usr/bin/python3'

source /home/david./local/bin/virtualenvwrapper.sh

保存退出

source ~/.bashrc

使配置生效.

就可以使用mkvirtualenv创建虚拟环境了.

配置的WORKON\_HOME的路径为

cd ~/.virtualenvs

# 创建python3的虚拟环境

mkvirtualenv --python=/usr/bin/python3 python3\_scrapy

# 删除虚拟环境

rm -rf python3\_scrapy

# 第3章 爬虫基础知识回顾

## 3-1 技术选型——爬虫能做什么

scrapy vs requests+beautifulsoup

为什么使用scrapy而不使用request+beautifulsoup来实现爬虫.

1. requests和beautifulsoup都是库, scrapy是框架

requests和beautiful类似于jquery, 都是库.

2. scrapy框架中可以加入requests和beautifulsoup

3. scrapy基于twisted, 性能是最大的优势

是一个异步IO的框架

4. scrapy方便扩展, 提供了很多内置的功能

5. scrapy内置的css和xpath selector非常方便, beautifulsoup最大的缺点就是慢

网页分类

常见类型的服务

1. 静态网页

http://projectsedu.com/

2. 动态网页

3. webservice(restapi), ajax

爬虫能做什么

爬虫作用

1. 搜索引擎---百度、google、垂直领域搜索引擎

垂直领域搜索引擎, 只针对某一专业方面的搜索引擎, 如租房信息.

2. 推荐引擎---今日头条

根据我们的浏览习惯来推测我们可能感兴趣的内容, 把数据推送给我们.

3. 机器学习的数据样本

4. 数据分析(如金融数据分析)、舆情分析等

## 3-2 正则表达式

正则表达式介绍, 为什么必须会正则表达式?

能判断某一个字符串能否符合某一种模式, 能提取字符串中的某些信息

### 正则表达式基础

特殊字符

1) ^ $ \* ? + {2} {2, } {2,5} |

2) [] [^] [a-z] .

3) \s \S \w \W

4) [\u4E00-\u9FA5] () \d

使用python shell, ipython, cmd或Windows PowerShell来运行

import re

line = 'bobby123'

#### ^的作法

regex\_str='^b.\*'

# ^b表示以字母b为开头, .英文句号代表的是任意字符, 包括中英文和特殊字符. \*表示它前面的字符可以重复任意多遍, 0次, 1次或多次.

# 此匹配规则的总体意思就是匹配以b开关的字符串, b后面的字符可以出现任意次.

if re.match(regex\_str, line):

print('yes')

else:

print('no')

#### $的用法

$代表结尾字符.

regex\_str = '.\*3$'

表示匹配的是以3为结尾的字符串.

regex\_str = '^b.\*3$' 表示匹配以b开头, 以3结尾的字符串

但

regex\_str = '^b.3$'

不能匹配成功.

#### ?的用法

?在正则表达式中表示非贪婪匹配的模式.

正则表达式默认是开户的贪婪模式, 会根据匹配规则尽可能匹配最大的长度.

line = 'aaaboooooobby123'

regex\_str = '.\*(b.\*b).\*'

# 表示匹配有两个b的字符串, 而不管中间和开头有多少字符. 给(b.\*b)加上括号表示提取出以b开头以b结尾的字符串.

match\_obj = re.match(regex\_str, line)

if match\_obj:

print(match\_obj.group(1))

# 使用re.match(regex\_str, line).group(1)就把前面正则表达式中提取的以b开头和以b结尾的字符串给取出来了.

输出的结果为bb, 而不是boooooob. 为什么呢?

这就涉及到正则表达式的贪婪匹配模式, ~~为了尽可能的去匹配到更多的字符, 它实际上是反向匹配的, 匹配到的是最后面的两个bb, 而不是boooooob中的两个b.~~

要使用?把它变成非贪婪匹配的模式, 让它从左边开始匹配.

regex\_str = '.\*?(b.\*b).\*'

输出的结果为booooobb. 为什么会把最后面的一个b也匹配上了呢. 前面的?把.\*b变成了非贪婪匹配模式, 但后面的.\*b还是使用的贪婪匹配模式, 尽可能的去匹配到更多的字符, 是从右边开始匹配的, 匹配到的是最后一个b. 要改为

regex\_str = '.\*?(b.\*?b).\*'

这样就把前后两个对b的匹配都改为了非贪婪模式.

#### +号的用法

+号表示它前面的字符至少要出现1次才能匹配成功.

regex\_str = '.\*(b.+b).\*'

现在表示b和b之间要有一个字符, 并且这个字符要至少出现1次, 才能匹配成功, 所以现在就不能匹配到字符串最后面的bb了. 而之前的'.\*(b.\*b).\*'这种匹配模式下, b前面的字符出现0或多次, 所以能匹配到最后面的bb.

\*号和+号都是限定它前面的字符出现的次数的限定词. \*号为0次或多次包含0次, +号表示1次或多次不包含0次.

#### {}的用法

大括号中的数字, 有三种情况{2}, {2,}, {2,5}, 也是限定词, 限定它前面字符出现的次数. {2}表示出现的次数为2次, {2,}表示出现的次数为2次以上. {2,5}表示出现的次数最少2次, 最多5次.

line = 'boobaaaooobbbaaaby123'

regex\_str = '.\*(b.{1}b).\* '

match\_obj = re.match(regex\_str, line)

match\_obj.group(1)

#### | 的用法

|表示或的关系.

line = 'bobby123'

regex\_str = '(bobby|bobby123)'

结果是bobby

line = 'bobby123'

regex\_str = '(bobby123|bobby)'

结果则是bobby123

line = 'bobby'

regex\_str = '(bobby123|bobby)'

结果则是bobby

line = 'bobby'

regex\_str = '(bobby|boobby)123'

结果则是None

line = 'boobby123'

regex\_str = '(bobby|boobby)123'

结果则是boobby123

line = 'boobby123'

regex\_str = '((bobby|boobby)123)'

结果则是boobby123

把(bobby|boobby)123)的值作为group(1), 把bobby|boobby的值作为group(2)

#### []的用法

第一种用法:

表示可以匹配[]中的任意一个字符

如

line = 'bobby123'

regex\_str = '([abcd]obby123)'

第二种用法:

使用区间

在提取电话号码时会经常遇到这种模式.

line = '18302349424'

regex\_str = '(1[35789][0-9]{9})'

[]

第三种用法就是, 取反的意思, 如[^1]表示不等于1

如

line = '18302349124'

regex\_str = '(1[35789][^1]{9})'

这样就不能匹配了.

注意只要写到 [] 中的字符, 都不再表示正则匹配, 而是表示它本身. 如[.\*]就不再表示对任意一个字符匹配0或多次的意思了.

.表示任意的一个字符, 不管是字母, 数字或特殊的字符.

#### \s \S的用法

line = '你 好'

regex\_str = '(你\s好)'

\s表示匹配空白字符

\S与\s刚好相反, 表示匹配不是空格的任意一个字符, 如果line='你ha好', 这样\s就无法匹配了.

可以使用'(你\S+好)'来匹配多个非空格的任意字符.

line = '你w好'

regex\_str = '(你\S好)'

#### \w \W的用法

\w

表示匹配任意字母数字和下划线中的一个字符. 等价于[A-Za-z0-9\_]

\W

与\w刚好相反. 表示除字母数字和下划线之外的任意一个字符. 可以用来修改文件名

#### \d的用法

表示匹配数字.

如

line = 'XXX出生于2001年'

regex\_str = '.\*(\d+)年'

结果是只提取出了一个数字1, 这也是贪婪模式引起的问题.

有两种方法可以解决这个问题.

一是

regex\_str = '.\*(\d{4})年'

一是

regex\_str = '.\*?(\d+)年'

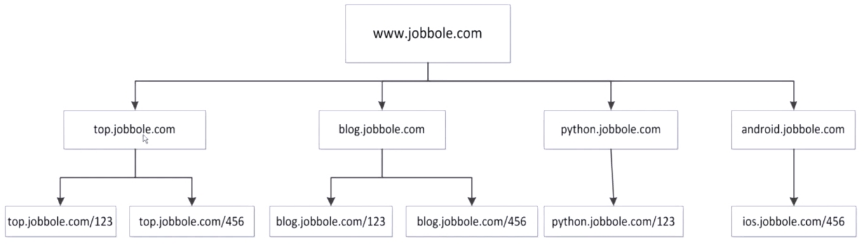
## 3-5 深度优先和广度优先原理

1. 网站的树结构

2. 深度优先算法和实现

3. 广度优先算法和实现

### 网站url的树形结构图,



url的分层设计.

一级域名www.jobbole.com,

二级域名python.jobbole.com,

二级域名下的分类python.jobbole.com/django,

文章python.jobbole.com/django/123.

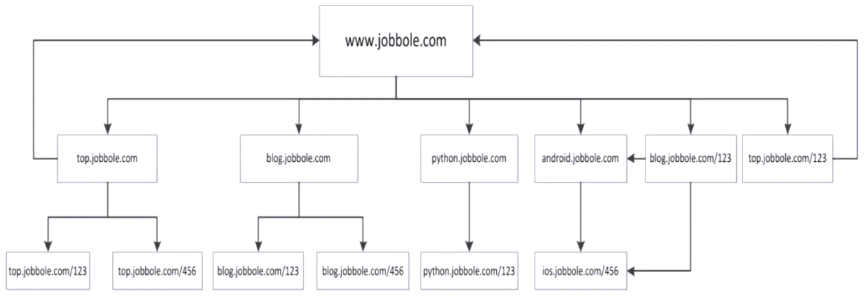
### 网站url链接的结构图

二叉树的遍历

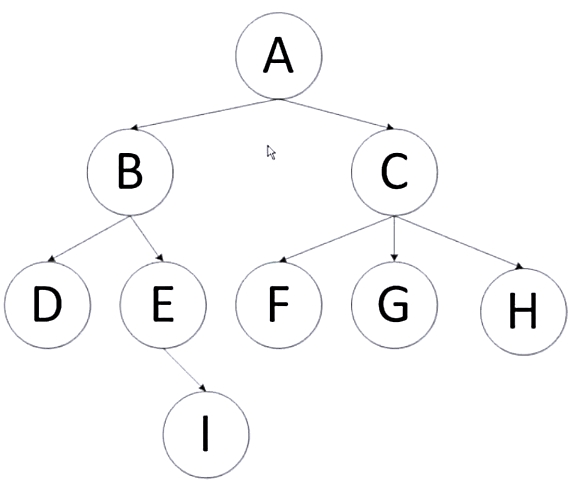
深度优先和广度优先

深度优先: 子孙节点的访问优先. scrapy默认是使用深度优先的算法.

广度优先: 兄弟节点的访问优先. 分层的访问.



二叉树



1. 深度优先输出 A、B、D、E、I、C、F、G、H（递归实现）

scrapy默认也是使用深度优先算法实现的

2. 广度优先输出 A、B、C、D、E、F、G、H、I（队列实现）

深度优先过程

def depth\_tree(tree\_node):

if tree\_node is not None:

print(tree\_node.\_data)

if tree\_node.\_left is not None:

return depth\_tree(tree\_node.\_left)

if tree\_node.\_right is not None:

return depth\_tree(tree\_node.\_right)

广度优先过程

#传递过来一个二叉树最顶层的一个节点. 二叉树和树的实现原理类似, 二叉树只有左节点和右节点.

def level\_queue(root):

#利用队列实现树的广度优先遍历

if root is None:

return

my\_queue = []

node = root

my\_queue.append(node)

while my\_queue:

node = my\_queue.pop(0)

print(node.elem)

if node.lchild is not None:

my\_queue.append(node.lchild)

if node.rchild is not None:

my\_queue.append(node.rchild)

数据爬取策略

1. url拼接分析(适用于所有数据的列表页和搜索关键词的获取)

2. 通用爬虫(广度优先和深度优先实现)

3. 广度优先算法和实现

爬虫更新策略

1. 根据网站权重分配爬虫资源

2. 根据last-modify-date、content-length字段判断

3. 对网站内容做一个指纹

## 3-6 url去重方法

1. 将访问过的url保存到数据库中

2. 将访问过的url保存到set中, 只需要o(1)的代价就可以查询url, 但内存占用会越来越大.

1亿个占50个字符的url, 占用的空间为, 字符串在python中为unicode, 每个字符占用2byte

100000000\*2byte\*50个字符/1024/1024/1024 = 9G

3. url经过md5等方法哈希后保存到set中, md5一般是128个bit, 大概等于16个byte字节, 而第2种方法是使用的是100个byte字节. scrapy使用的类似md5方法.

4. 用bitmap方法, 将访问过的url通过hash函数映射到某一位

申请8个bit位. 但是会出现冲突/重复.



5. bloomfilter方法对bitmap进行改进, 多重hash函数降低冲突

## 3-7 彻底搞清楚unicode和utf-8编码

字符串编码

1. 计算机只能处理数字, 文本转换为数字才能处理. 计算机中8个bit作为一个字节, 所以一个字节能表示最大的数字就是255.

1个bit是一个0或1的位

2. 计算机是美国人发明的, 所以一个字节可以表示所有字符了, 所以ASCII(一个字节)编码就成为美国人的标准编码

3. 但是ASCII处理中文明显是不够的, 中文不止255个汉字, 所以中国制定了GB2312编码, 用两个字节表示一个汉字. GB2312还把ASCII包含进去了, 同理, 日文, 韩文等等上百个国家为了解决这个问题就都发展了一套字节的编码, 标准就越来越多, 如果出现多种语言混合显示就一定会出现乱码

4. 于是unicode出现了, 通过16个个bit或32个bit将所有语言统一到一套编码里.

5. 看一下ASCII和unicode编码:

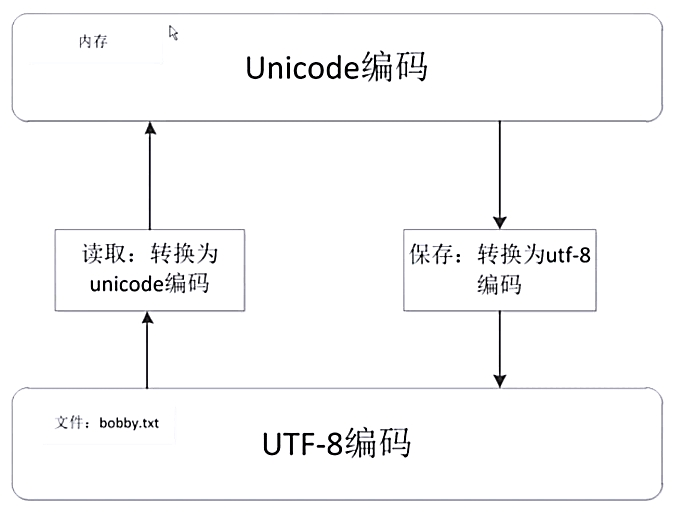
1）字母A 用ASCII编码十进制是65, 二进制 0100 0001

2）汉字"中"已近超出了ASCII编码的范围, 用unicode编码是20013, 二进制是 01001110 00101101

3）A用unicode编码只需要前面补0二进制是 00000000 0100 0001

6. 乱码问题解决了, 但是如果内容全是英文, unicode编码比ASCII需要多一倍的存储空间, 同时如果传输需要多一倍的传输.

7. 所以出现了可变长的编码"utf-8", 把英文变长一个字节, 汉字3个字节. 特别生僻的变成4-6字节, 如果传输大量的英文, utf-8作用就很明显了. utf-8在文件保存和传输的时候很方便.



但如果在编程的过程中把变量以utf-8来进行编码, 如果代码中既有中文又有英文, 在内存中, 有的字符串占用1个字节, 有的占用2个字节, 程序就很难正确进行处理. 因为unicode编码的字符串长度是一样的, 就会比utf-8处理起来更简单一些.

解决的方法就是在做文件保存或网络传输的时候, 把unicode编码转换为utf-8编码, 以减小文件大小和网络传输的数据量, 而在从文件中把数据读取到内存中的时候, 再把utf-8的编码转换为unicode的编码进行统一的处理.

读文件, 系统会自动转换为unicode, 但必须要指定文件的原始编码, python才能够正确的把这种编码转换为unicode的编码. 在保存时, 需要指定以utf-8的编码进行保存, 或者把unicode的编码转换为utf-8的编码进行保存.

在python2中编码和解码经常会出错, python2中有两种编码的格式, 一种是string的格式.

s = 'abc'

su = u'abc'

s是string格式的编码, su是unicode格式的编码. 两种编码方法都可以直接用encode方法转换编码格式.

s.encode('utf8')

su.encode('utf8')

但一旦字符串中出现了中文, 使用string格式的编码的字符串在用encode进行转码时就会出错.

如s = '我用python'

su = u'我用python'

python中, 字符串在内存中是使用unicode进行编码的, s = '我用python'在linux中是utf-8的编码, 在windows中是gb2312格式的编码, 在调用encode这个方法的时候, 必须要先将这个变量变为unicode格式的编码, 在上例中, 如果想将s转换为utf-8的编码, 就先要把它decode成unicode格式的编码. decode的作用是将其它编码的字符串转换为unicode编码, 但必须要指定字符串当前的编码格式. 如果是在linux, 则是utf8, 在windows下则是gb2312.

windows中

s.decode('gb2312').encode('utf-8')

这样就不会出错了.

但如果s.decode('gb2312').encode('ascii'), 因为ascii编码不支持中文, 也会出错.

对于su, 它已经是unicode格式的编码, 当然可以直接使用encode把它编码成utf-8的格式.

su.encode('utf-8')

linux中

s = '我用python'

su = u'我用python'

s.encode('utf-8')也会出错. linux中utf8和utf-8的意义相同.

这是因为, 在使用encode方法进行字符串编码格式转换的时候必须要保证前面的变量是unicode的字符串, 不然就会出错. linux下s是utf-8格式的编码, 直接使用s.encode('utf-8')就会出错.

在s.encode('utf-8')的时候, 程序实际上会先调用decode的默认解码方式来解码. decode默认是ascii编码.

import sys

sys.getdefaultencoding()

输出为'ascii'

对包含中文的字符串使用默认解码方式为ascii编码格式的decode, 肯定会报错.

s.decode('gb2312').encode('utf-8')也会出错. 因为在linux下s实际上就是utf8的编码.

s.decode('utf-8').encode('utf-8')

在linux中 su = u'我用python' 已经是unicode字符串了, 就可以直接使用encode来编码成utf-8编码

su.encode("utf-8")

python2与python3一个最大的不同是,

在python2中, 在存在中文的py文件中, 必须在文件头加上#coding=utf-8, 这样python才能够正确识别其中的中文.

在python3中就不需要进行声明了. python3实际上已经把python2中s = '我用python'的这种写法取消了. 直接把所有字符串的格式都默认为unicode.

在python3中.

s = '我用python'

s.encode('utf-8')

这样也不会出错.

在python3中, 直接把所有的字符都默认为unicode格式, 就解决了在python2中经常会遇到的编码问题.

# 第4章 scrapy爬取知名技术文章网站

爬取 "最新文章" 中的所有文章, 就达到了我们的目的. 在一个网站提供了所有文章, 所有内容的入口时, 就可以优先考虑通过这个入口来进行爬取. 在网站没有提供所有内容的入口, 再考虑使用深度优先或广度优先的算法来进行爬取.

## 4-1 scrapy安装以及目录结构介绍

Scrapy大量借鉴了Django的开发理念.

mkvirtualenv --python=C:\Python35\python.exe jobboleArticle

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

会出错, 需要手动安装twisted包

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

下载对应的安装包Twisted‑17.9.0‑cp36‑cp36m‑win\_amd64.whl

pip install wheel

进入虚拟环境, 进入whl文件所在目录, 安装whl文件

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

手动安装lxml包.

https://pypi.python.org/pypi/lxml/

在pycharm中没有新建scrapy的项目, 要通过手动来创建.

系统内容用驼峰法命名, 文件和文件夹用下划线命名.

进入到保存工程文件的目录中. 如E:\scrapy\_projects

workon jobboleArticle

scrapy startproject ArticleSpider

# scrapy在创建项目的时候可以自定义模板.

You can start your first spider with:

cd ArticleSpider

scrapy genspider example example.com

cd ArticleSpider

scrapy genspider jobbole blog.jobbole.com

创建一个名为jobbole的爬虫. 会自动生成一个jobbole.py的爬虫文件

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

import scrapy

class JobboleSpider(scrapy.Spider):

name = 'jobbole'

allowed\_domains = ['blog.jobbole.com']

start\_urls = ['http://blog.jobbole.com/']

def parse(self, response):

pass

## 4-2 pycharm调试scrapy执行流程

可以使用pycarm打开ArticleSpider

在pycharm settings中的project interpreter中导入之前创建的虚拟环境.

使用Pychram来导入本地的虚拟环境, Windows下要找到虚拟环境中\Scripts\python.exe. Linux下要找到bin下的python运行文件

pycharm中没有django的调试模式, 需要自己写一个文件.

在与scrapy.cfg同目录下新建一个main.py

#coding = utf-8

\_\_author\_\_ = 'bobby'

from scrapy.cmdline import execute

import sys, os

#os.path.abspath(\_\_file\_\_)得到的是当前的main文件所在的绝对目录, 而os.path.dirname得到的是某个文件的父级目录.

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

# print(os.path.dirname(os.path.abspath(\_\_file\_\_)))

execute(['scrapy', 'crawl', 'jobbole'])

**记录下: python 的 sys.path.append()**

当我们导入一个模块时: import  xxx, 默认情况下python解析器会搜索当前目录、已安装的内置模块和第三方模块, 搜索路径存放在**sys模块的path**中:

>>> **import** sys

>>> sys.path

['', 'C:\\Python352\\Lib\\idlelib', 'C:\\Python352\\python35.zip', 'C:\\Python352\\DLLs', 'C:\\Python352\\lib', 'C:\\Python352', 'C:\\Python352\\lib\\site-packages', 'C:\\Python352\\lib\\site-packages\\setuptools-28.6.1-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\pip-8.1.2-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\requests-2.11.1-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\xlutils-2.0.0-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\xlwt-1.1.2-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\pymongo-3.3.1-py3.5-win-amd64.egg', 'C:\\Python352\\lib\\site-packages\\pytz-2016.7-py3.5.egg', 'C:\\Python352\\lib\\site-packages\\zope.interface-4.3.3-py3.5-win-amd64.egg']

sys.path 返回的是一个列表！

该路径已经添加到系统的环境变量了, 当我们要添加自己的搜索目录时, 可以通过列表的append()方法；

对于模块和自己写的脚本不在同一个目录下, 在脚本开头加sys.path.append('xxx'):

**import** sys

sys.path.append(’引用模块的地址')

这种方法是运行时修改, 脚本运行后就会失效的.

另外一种方法是:

把路径添加到系统的环境变量, 或把该路径的文件夹放进已经添加到系统环境变量的路径内. 环境变量的内容会自动添加到模块搜索路径中.

运行main.py, 会出现一个no module named 'win32api'的错误.

pip install pypiwin32 -i <https://pypi.douban.com/simple>

使用scrapy crawl jobbole这个命令来启动爬虫

在settings中设置

ROBOTSTXT\_OBEY = False

在pycharm右上角的run中点下拉列表, 选择run configurations出现Run/Debug Configurations.

添加Python,

Name任意,

Script为虚拟环境中\Lib\site-ackages\scrapy\cmdline.py,

Script parameters为crawl spider\_name,

Python interpreter为虚拟环境的python.exe.

Working directory为当前的项目目录.

设置好之后在jobbole.py中的pass一行中点左边的空白位置设置一个断点. 然后在main.py或者jobbole.py中运行debug, 运行结束就可以看到debug的信息了. 查看jobbole.py中def parse中的response的内容.

## 4-3 xpath的用法

xpath

目录

1. xpath简介

1. xpath使用路径表达式在xml和html中进行导航

2. xpath包含标准函数库

3. xpath是一个w3c的标准

2. xpath术语

3. xpath语法

xpath节点关系

1. 父节点

2. 子节点

3. 同胞节点, 兄弟节点

4. 先辈节点, 包含父节点和所有爷节点的节点

5. 后代节点

xpath语法

|  |  |
| --- | --- |
| 表达式 | 说明 |
| article | 选取所有article元素的所有子节点 |
| /article | 选取根元素article |
| article/a | 选取所有属于article的子元素的a元素 |
| //div | 选取所有div子元素(不论出现在文档任何地方) |
| article//div | 选取所有属于article元素的后代的div元素, 不管它出现在article之下的任何位置 |
| //@class | 选取所有名为class的属性 |

xpath语法 谓语

|  |  |
| --- | --- |
| 表达式 | 说明 |
| /article/div[1] | 选取属于article子元素的第一个div元素 |
| /article/div[last()] | 选取属于article子元素的最后一个div元素 |
| /article/div[last()-1] | 选取属于article子元素的倒数第二个div元素 |
| //div[@lang] | 选取所有拥有lang属性的div元素 |
| //div[@lang='eng'] | 选取所有lang属性为eng的div元素 |

xpath语法 特殊字符

|  |  |
| --- | --- |
| 表达式 | 说明 |
| /div/\* | 选取属于div元素的所有子节点 |
| //\* | 选取所有元素 |
| //div[@\*] | 选取所有带属性的title元素 |
| /div/a | //div/p | 选取所有div元素的a和p元素 |
| //span | //ul | 选取文档中的span和ul元素 |
| article/div/p | //span | 选取所有属于article元素的div元素的p元素 以及文档中所有的span元素 |

**使用xpath进行信息的提取.**

把start\_urls设置为['http://blog.jobbole.com/110287/']

修改jobbole.py

import scrapy

class JobboleSpider(scrapy.Spider):

name = 'jobbole'

allowed\_domains = ['blog.jobbole.com']

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

start\_urls = ['http://blog.jobbole.com/111231/']

def parse(self, response):

xpth\_selector0 = response.xpath('/html/body/div[3]/div[3]/div[1]/div[1]/h1')

xpth\_selector1 = response.xpath('//\*[@id="post-110287"]/div[1]/h1')

xpth\_selector2 = response.xpath("//div[@class='entry-header']/h1/text()")

pass

在pass处添加断点, 再次进行debug, 可以看到xpth\_selector0没有取到内容, 而xpth\_selector1, 2提取到了内容. 为什么呢?

这是因为, 在网页中通过检查元素看到的网页的源码是所有代码包括js代码运行结束之后产生的效果, 和查看源码中看到的网页源码可能是有所不同的. 如源码中有一个js, 通过这个js在body中插入了一个div标签, 在浏览器中看到的就是插入这个div标签之后的效果, 检查元素时看到的代码也是插入这个div标签之后的代码. 而我们通过scrapy获取到的是response的请求, 服务器返回来的response请求即是网页的源码, 而我们使用xpath选择器是在浏览器中检查元素中的源码中进行选取的. 两个源码的不同就造成了一个xpath语法能够取到元素, 而另一个取不到的结果.

可以把源码复制到pycharm中的一个html中进行分析, 发现div class='container'这个标签是body中的第1个div标签, 而在网页中通过检查元素看到的此标签是body的第三个标签, 前面还有2个通过js动态生成的div标签.

把xpth\_selector0改为

xpth\_selector0 = response.xpath('/html/body/div[1]/div[3]/div[1]/div[1]/h1')

再次运行debug信息, 就看到xpth\_selector0也取到了标题的内容.

对于xpth\_selector1这种取法, 因为其中post中的id会随着文章的不同而不同, 故也不建议使用这种选择方法.

可以使用scrapy shell http://blog.jobbole.com/111231/这种方式来先对代码进行测试, 然后再把代码写入到文件中.

title = response.xpath("//div[@class='entry-header']/h1/text()")

title

[<Selector xpath="//div[@class='entry-header']/h1/text()" data='顺丰菜鸟大战背后真实原因 , 也许没有那么复杂'>]

可以看到title是一个selector, 这样就可以直接在title的基础上进行二次的匹配.

使用title.extract()来获取选择器中的文本信息, 返回一个列表.

title = response.xpath("//div[@class='entry-header']/h1/text()").extract()

title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()

['顺丰菜鸟大战背后真实原因, 也许没有那么复杂']

取列表中的第1个值, 就得到了文本的信息

title = response.xpath("//div[@class='entry-header']/h1/text()").extract()[0]

获取发表时间

create\_date=response.xpath('//p[@class="entry-meta-hide-on-mobile"]')

create\_date.extract()

create\_date=response.xpath('//p[@class="entry-meta-hide-on-mobile"]/text()')

p标签的text只会获取p标签中的内容, 其子孙标签中的所有内容都会被忽略.

create\_date=response.xpath('//p[@class="entry-meta-hide-on-mobile"]/text()').extract()

这样就获取得发表时间. 对文本信息进行处理, 删除其中的空格.

create\_date=response.xpath('//p[@class="entry-meta-hide-on-mobile"]/text()').extract()[0].strip()

'2017/06/04 ·'

再使用replace

create\_date=response.xpath('//p[@class="entry-meta-hide-on-mobile"]/text()').extract()[0].strip().replace("·","").strip()

要取得点赞数量

praise\_num = response.xpath("//span[contains(@class, 'vote-post-up')]/h10/text()").extract()[0]

获得收藏数量

fav\_num = response.xpath("//span[contains(@class, 'bookmark-btn')]/text()").extract()[0]

import re

re.match(r".\*(\d+).\*", fav\_num).group(1)

评论数

comment\_num = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]

re.match(r".\*(\d+).\*", comment\_num).group(1)

#文章分类

tag\_list = response.xpath('//p[@class="entry-meta-hide-on-mobile"]/a/text()').extract()

#去除掉评论数, 不能直接删除列表中的第2个元素, 因为没有评论的时候, 是不会产生1评论这个标签的.

tag\_list = [element for element in tag\_list if not element.strip().endswith('评论')]

# 把列表拼接成字符串

tags = ','.join(tag\_list)

## 4-6 css选择器实现字段解析

css选择器, 和xpath的功能类似, 都是用来定位html中的元素.

|  |  |
| --- | --- |
| 表达式 | 说明 |
| \* | 选择所有节点 |
| #container | 选择id为container的节点 |
| .container | 选取所有class包含container的节点 |
| li a | 选取所有li下的所有a节点 |
| ul + p | 选择ul后面的第一个p元素, 不是选择li中的节点, ul和p相当于兄弟节点 |
| div#container > ul | 选取id为container的div的第一个ul子元素 |

|  |  |
| --- | --- |
| 表达式 | 说明 |
| ul ~ p | 选取与ul相邻的所有p元素 |
| a[title] | 选取所有有title属性的a元素 |
| a[href="http://jobbole.com"] | 选取所有href属性为jobbole.com值的a元素 |
| a[href\*="jobole"] | 选取所有href属性包含jobbole的a元素 |
| a[href^="http"] | 选取所有href属性值以http开头的a元素 |
| a[href$=".jpg"] | 选取所有href属性值以.jpg结尾的a元素 |
| input[type=radio]:checked | 选择选中的radio的元素 |

|  |  |
| --- | --- |
| 表达式 | 说明 |
| div:not(#container) | 选取所有id不是container的div属性 |
| li:nth-child(3) | 选取第三个li元素 |
| tr:nth-child(2n) | 第偶数个tr |

**使用css选择器提取网页中的元素.**

先进入虚拟环境, 在虚拟环境中安装ipython, 使用虚拟环境进入scrapy shell

pip install ipython

安装ipython之后, 再使用scrapy shell, 就会自动进入到ipython中.

scrapy shell http://blog.jobbole.com/111231/

#提取标题, 得到一个selector对象

response.css(".entry-header h1")

[<Selector xpath="descendant-or-self::\*[@class and contains(concat(' ', normalize-space(@class), ' '), ' entry-header ')]/descendant-or-self::\*/h1" data='<h1>顺丰菜鸟大战背后真实原因, 也许没有那么复杂</h1>'>]

#提取h1标签

response.css(".entry-header h1").extract()

['<h1>顺丰菜鸟大战背后真实原因, 也许没有那么复杂</h1>']

#使用::text来提取h1标签的值

response.css("p.entry-header h1::text").extract()

['顺丰菜鸟大战背后真实原因, 也许没有那么复杂']

title = response.css("p.entry-header h1::text").extract()

**#发表时间**

response.css("p.entry-meta-hide-on-mobile::text").extract()

response.css("p.entry-meta-hide-on-mobile::text").extract()[0].strip()

'2017/06/04 ·'

response.css("p.entry-meta-hide-on-mobile::text").extract()[0].strip().replace("·","").strip()

'2017/06/04'

create\_date = response.css("p.entry-meta-hide-on-mobile::text").extract()[0].strip().replace("·", "").strip()

**#点赞数量**

praise\_num = response.css("span.vote-post-up h10::text").extract()[0]

**#收藏数量**

fav\_num = response.css("span.bookmark-btn::text").extract()[0]

match\_re = re.match(".\*(\d+).\*", fav\_num)

if match\_re:

fav\_num = match\_re.group(1)

**#评论数量**

comment\_num = response.css("a[href='#article-comment'] span::text").extract()[0]

**#正文内容, 不进行处理**

content = response.css("div.entry").extract()[0]

**#标签, 取[0]得到职场, 不取的话得到所有的标签**

tag\_list = response.css("p.entry-meta-hide-on-mobile a::text").extract()

# 去掉评论的内容

tag\_list = [element for element in tag\_list if not element.strip().endswith("评论")]

tags = ",".join(tag\_list)

所有代码都完成之后再进行最后的调试.

在xpath的fav\_num和match\_re, 以及pass处添加断点进行调试. F6. F6进入下一个调试的快捷键是在keymap中设置模板为Eclipse得到的.

提取出本页面的内容http://python.jobbole.com/87508/

发现原来的评论数是14, 但通过正则表达式来匹配之后只得到了4个评论数, 这是因为正则匹配模式默认的是贪婪模式, 是从右向左匹配的, 匹配到4之后就停止了. 对于评论数大于10的, 都会出现这个问题. 所以要把所有的正则匹配模式都改为非贪婪模式. 如

match\_re = re.match(".\*?(\d+).\*", fav\_num)

第二个问题是使用extract()生成列表之后, 再取其中的元素, 但这样做会存在一定的风险, 如果列表为空, 就会抛出异常. 可以使用extract\_fisrt("")来直接取出其中的第1个元素. 查看extract\_first()的源码:

def extract\_first(self, default=None):

"""

Return the result of ``.extract()`` for the first element in this list.

If the list is empty, return the default value.

"""

for x in self:

return x.extract()

else:

return default

在extract\_first()中有一个default的值. 如果提取不到, 就返回default中的值.

使用extract\_first("")就可以避免extract()中可以出现的问题了.

## 4-8 编写spider爬取jobbole的所有文章

parse函数中需要实现的两个功能

1. 获取文章列表页中所有文章的url并交给scrapy下载后进行解析. 下载所有文章中的链接<http://blog.jobbole.com/all-posts/>, 就不需要使用深度或广度优先的算法了, 只要把这个页面中所有的文章提取出来即可.

2. 获取下一页的url并交给scrapy进行下载, 下载完成后交给parse. 也可以使用自己拼凑url的方法, 但这种方法在以后网站内容增多的时候会有一些页面不能提取到.

import scrapy, re

from scrapy.http import Request

class JobboleSpider(scrapy.Spider):

name = 'jobbole'

allowed\_domains = ['blog.jobbole.com']

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

start\_urls = ['http://blog.jobbole.com/all-posts/']

def parse(self, response):

#1. 获取文章列表页中的文章url并交给scrapy下载后进行解析. 注意要添加#archive这个id, 不然的话就会取到很多不是文章列表中的url地址. id是唯一的, 就不需要再添加标签名了.

post\_urls = response.css("#archive div.floated-thumb .post-thumb a::attr(href)").extract()

for post\_url in post\_urls:

#print(post\_url)

#把提取到的文章详情页的url交给parse\_detail这个解析函数进行处理.

Request(url=post\_url, callback=self.parse\_detail)

可以在scrapy shell中进行调试.

但是在很多网页中, 获取到的href并不是完整的地址信息, 不包含网站的域名, 就要需要把地址信息补全, 把当前网站的域名和获取到的herf拼接起来. response.url + post\_url. 但这种方法不具有通用性, 只能适用于提取到的href不包含域名的情况.

在python3的urllib库提供了一个更简单的方法来实现url的拼接功能.

from urllib import parse

url=parse.urljoin(response.url, post\_url)

查看urljoin的源码:

def urljoin(base, url, allow\_fragments=True):

"""Join a base URL and a possibly relative URL to form an absolute interpretation of the latter."""

这个函数会把两个域名进行拼接. urljoin会把传进去的第1个base参数的域名提取出来. 如果第2个url参数已经是带有域名的完整的信息, 就不会进行处理.

把上面的for函数改写为:

for post\_url in post\_urls:

url=parse.urljoin(response.url, post\_url)

#使用yield把提取到的url交给详情解析函数parse\_detail进行详情的提取.

yield Request(url, callback=self.parse\_detail)

**提取下一页的url并交给scrapy进行post\_urls地址的提取**

#提取出了包含下一页的a标签. .next和.page-numbers是属于同一个div的class. 使用extract\_first就可以避免在最后一页时出错.

next\_urls = response.css(".next.page-numbers").extract\_first("")

#提取出下一页的链接.

next\_url = response.css(".next.page-numbers::attr(href)").extract\_first("")

#如果存在下一页的这个值. 在最后一页时这个值不存在.

if next\_url:

#使用回调函数把下一页的url传回到parse函数中重复进行提取. 不用写成self.parse(). scrapy是基于异步处理twisted完成的, 会自动根据函数名来调用函数.

yield Request(url=parse.urljoin(response.url, next\_url), callback=self.parse)

此时的起始url为http://blog.jobbole.com/all-posts/

class JobboleSpider(scrapy.Spider):

name = "jobbole"

allowed\_domains = ["python.jobbole.com"]

start\_urls = ['http://python.jobbole.com/all-posts/']

def parse\_detail(self, response):

#提取每篇文章的具体字段.

# xpth\_selector0 = response.xpath('/html/body/div[1]/div[3]/div[1]/div[1]/h1/text()')

# xpth\_selector1 = response.xpath('//div[@id="post-111231"]/div[1]/h1')

# xpth\_selector2 = response.xpath("//div[@class='entry-header']/h1/text()")

#文章标题

title = response.xpath("//div[@class='entry-header']/h1/text()").extract\_first()

#发表日期

create\_date = response.xpath('//p[@class="entry-meta-hide-on-mobile"]/text()').extract()[0].strip().replace("·","").strip()

#点赞数量

praise\_num = response.xpath("//span[contains(@class, 'vote-post-up')]/h10/text()").extract()[0]

#收藏数量

fav\_num = response.xpath("//span[contains(@class, 'bookmark-btn')]/text()").extract()[0]

match\_re = re.match(r".\*?(\d+).\*", fav\_num)

if match\_re:

fav\_num = int(match\_re.group(1))

else:

#如果没有取到数据, 就把它设为0

fav\_num = 0

#评论数

comment\_num = response.xpath("//a[@href='#article-comment']/span/text()").extract()[0]

match\_re = re.match(r".\*?(\d+).\*", comment\_num)

if match\_re:

comment\_num = int(match\_re.group(1))

else:

#如果没有取到数据, 就把它设为0

comment\_num = 0

#取正文内容, 但是不作处理

content = response.xpath("//div[@class='entry']").extract()[0]

# 文章分类

tag\_list = response.xpath('//p[@class="entry-meta-hide-on-mobile"]/a/text()').extract()

# 去除掉评论数

tag\_list = [element for element in tag\_list if not element.strip().endswith('评论')]

tags = ','.join(tag\_list)

pass

**列表页图片url的获取和图片的下载**

获取列表页的图片的url, 在用户没有自定义列表页图片时, 会自动获取正文中的第一张图片来作为列表页的缩略图.

可以直接在列表页获取图片的url.

对于列表页的每一篇文章, 不再直接获取其详情的url地址了, 而是获取详情url和封面图url的父节点, 再父节点的基础上再进行选取, 得到详情url和封面图的url

def parse(self, response):

# 获取文章列表中的每一篇文章的url交给Scrapy下载并解析

article\_nodes = response.css('div#archive .floated-thumb .post-thumb a')

for article\_node in article\_nodes:

# 解析每个文章的封面图

front\_image\_url = article\_node.css('img::attr(src)').extract\_first("")

# 解析每个文章的url

article\_url = article\_node.css('::attr(href)').extract\_first("")

# 智能对url进行拼接, 如果url中不带有域名则会自动添加域名, 如果传递过来的url中带有域名信息, 就不再进行拼接处理.

# meta是字典dict类型的, 通过在Request中设置meta信息来在不同的请求之间进行数据的传递.

yield Request(url=parse.urljoin(response.url, article\_url), meta={'front\_image\_url': parse.urljoin(response.url, front\_image\_url)}, callback=self.parse\_detail)

# 获取文章的下一页url地址, 并交给parse函数自身进行解析

next\_url = response.css('a.next.page-numbers::attr(href)').extract\_first('')

if next\_url:

yield Request(url=parse.urljoin(response.url, next\_url), callback=self.parse)

在parse\_detail中的title处放置断点来查看response中传递过来的数据中是否包含meta的信息. 可以看到meta是字典类型的, 其中有一个front\_image\_url的键值对. 通过response.meta.get('front\_image\_url', '')就能够把它的值取出来了. 使用get方法来取如果不存在时不会抛出异常, 使用meta['front\_image\_url']来取时如果不存在, 会抛出异常.

def parse\_detail(self, response):

article\_item = JobBoleArticleItem()

# 从response中获取数据

# 文章封面图

front\_image\_url = response.meta.get('front\_image\_url', '')

# CSS方式进行解析

# 文章标题

title = response.css('div.entry-header h1::text').extract\_first()

## 4-10 items设计

使用scrapy中的items把提取的内容保存到数据库中

数据爬取的目的就是从非结构化的数据中提取到结构化的数据. 最简单的是把所有字段放到字典中进行保存, 但字典中缺少结构化的数据, 必须要手动定义, 这样会比较容易出错. scrapy中定义了item类, 可以使用item类来指定字段.

在对items类进行实例化之后, 在spider中对它进行yield之后, scrapy会把这个items传递到pipeline中, 就可以在pipeline中集中进行数据的去重, 保存等操作.

定义items.py

class JobboleArticleItem(scrapy.Item):

title = scrapy.Field()

create\_time = scrapy.Field()

# 文章详情的url

url = scrapy.Field()

#url是变长的, 通过md5把它变成固定长度的.

url\_object\_id = scrapy.Field()

#封面图的url地址

front\_image\_url = scrapy.Field()

#封面图在本地保存的路径.

front\_image\_path = scrapy.Field()

praise\_num = scrapy.Field()

fav\_num = scrapy.Field()

comment\_num = scrapy.Field()

tags = scrapy.Field()

content = scrapy.Field()

修改jobbole.py这个spider文件, 实例化item对象, 并把提取的详情信息传递到item中.

from ArticleSpider.items import JobboleArticleItem

def parse\_detail(self, response):

#实例化article对象

article\_item = JobBoleArticleItem()

# 从response中获取数据

# 文章封面图

front\_image\_url = response.meta.get('front\_image\_url', '')

# CSS方式进行解析

# 文章标题

title = response.css('div.entry-header h1::text').extract\_first()

# 发布时间

create\_time = response.css('p.entry-meta-hide-on-mobile::text').extract\_first().replace('·','').strip()

# 点赞数

praise\_num = response.css('span.vote-post-up h10::text').extract\_first()

# 收藏数

fav\_num = response.css('span.bookmark-btn::text').extract\_first()

match\_re = re.match('.\*?(\d+).\*',fav\_num)

if match\_re:

fav\_num = match\_re.group(1)

else:

fav\_num = 0

# 评论数

comment\_num = response.css('a[href="#article-comment"] span::text').extract\_first()

match\_re = re.match('.\*?(\d+).\*', comment\_num)

if match\_re:

comment\_num = match\_re.group(1)

else:

comment\_num = 0

# 文章正文

content = response.css('div.entry').extract\_first()

# 获取标签

tags\_list = response.css('p.entry-meta-hide-on-mobile a::text').extract()

tags\_list = [element for element in tags\_list if not element.strip().endswith('评论')]

tags = ",".join(tags\_list)

article\_item["title"] = title

article\_item["create\_time"] = create\_time

article\_item["url"] = response.url

article\_item["front\_image\_url"] = [front\_image\_url]

article\_item["praise\_num"] = praise\_num

article\_item["fav\_num"] = fav\_num

article\_item["comment\_num"] = comment\_num

article\_item["content"] = content

article\_item["tags"] = tags

#调用yield, 把article\_item传递到pipeline中,

yield article\_item

在settings中修改ITEM\_PIPELINES

ITEM\_PIPELINES = {

'ArticleSpider.pipelines.JobboleArticlePipeline': 300,

}

在pipeline中添加def process\_item和reture item的断点, 运行debug, F6查看信息, 查看是否能传递过来item信息. 在item中有一个\_values的信息, 其中就包含了之前在jobbole爬虫中提取的信息.

可以在pipeline中进行信息的处理和保存.

想要把文章的封面图片下载下来, 并保存到本地, scrapy提供了一种自动下载图片的方法, 只需要在settings中的ITEM\_PIPELINES中添加一个images的item\_pipline即可完成图片的下载.

C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\pipelines.images.py

#从items.py中传递过来的数据都要经过ITEM\_PIPELINES这个管道的处理, 数字设置的越小越早经过这个pipeline.

在settings.py中修改或添加以下信息

ITEM\_PIPELINES = {

'bole.pipelines.BolePipeline': 300,

'scrapy.pipelines.images.ImagesPipeline' : 200

}

# 设置图片url的字段, scrapy将从item中找出此字段进行图片下载

IMAGES\_URLS\_FIELD = "front\_image\_url"

# 设置图片下载保存的目录, 这里不直接使用本机的绝对路径, 而是通过程序获取路径. 这样在程序迁移后也能够正常运行.

import os

# os.path.dirname(\_\_file\_\_)获取当前文件所在的文件夹名称.

# os.path.abspath(os.path.dirname(\_\_file\_\_))获得当前文件所在的绝对路径.

project\_path = os.path.abspath(os.path.dirname(\_\_file\_\_))

# 想要把下载的图片保存在与settings同目录的images文件夹中, 要先在项目中新建此文件夹.

# IMAGES\_STORE中定义保存图片的路径

IMAGES\_STORE = os.path.join(project\_path, "images")

# 表示只下载大于100x100的图片, 查看images.py的源码, 程序会自动的从settings.py中读取设置的IMAGES\_MIN\_HEIGHT和IMAGES\_MIN\_WIDTH值

# IMAGES\_MIN\_HEIGHT = 100

# IMAGES\_MIN\_WIDTH = 100

之后运行项目可能包PIL未找到, 因此需要pip install pillow

此外scrapy的图片下载默认是接受一个数组/列表, 因此jobbole.py要修改为:

article\_item["front\_image\_url"] = [front\_image\_url]

**自定义管道获取下载的图片保存的路径.**

运行spider, 就能够下载到图片了. 现在图片已经保存在本地了, 要提取到图片保存的路径, 与对应的文章绑定在一起, 把它放在front\_image\_path这个字段中. 只需要自定义一个pipeline即可, 继承系统的ImagePipeline.

获取到front\_image\_path图片保存的路径之后, 就可以把content中对应的图片链接全部转换为本地的保存路径, 这样就可以把别人的网页按按照原样照搬过来.

打开scrapy.pipelines.images.ImagesPipeline进行查看.

# 在scrapy.pipelines.images.ImagesPipeline中有一个get\_media\_requests函数, 这个函数是用for循环的方式处理的, 所以之前在对图片封面进行赋值的时候要使用一个数组, 从数组中取得地址, 使用Request发送这个地址的请求, 交给scrapy下载器进行下载.

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

return [Request(x) for x in item.get(self.images\_urls\_field, [])]

# 使用item\_completed这个函数能够获取到图片在本地的保存路径, 图片的路径是保存在result这个参数中的, 需要对这个函数进行重写.

def item\_completed(self, results, item, info):

if isinstance(item, dict) or self.images\_result\_field in item.fields:

item[self.images\_result\_field] = [x for ok, x in results if ok]

return item

**修改pipelines.py, 自定义图片下载管道**

from scrapy.pipelines.images import ImagesPipeline

# 自定义图片下载处理的中间件

class ArticleImagePipeline(ImagesPipeline):

# 重写函数, 改写item处理完成的函数

def item\_completed(self, results, item, info):

pass

result的结构不清楚, 可以使用断点的方式进行调试. 把images下载的所有图片都删除. 修改settings中的ITEM\_PIPELINES, 使用我们自定义的管道.

ITEM\_PIPELINES = {

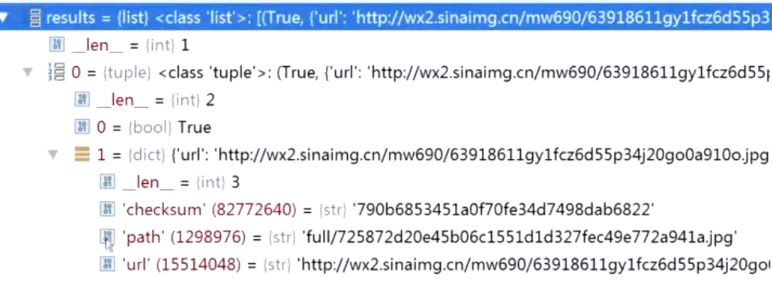
'ArticleSpider.pipelines.JobboleArticlePipeline': 300,

#'scrapy.pipeline.images.ImagesPipeline' : 200

'ArticleSpider.pipelines.ArticleImagePipeline': 200,

}

在pipelines.py中的pass添加断点, 开始进行调试, 查看result的结构, 可以看到它是一个list. list中的每一个元素都是一个tuple, tuple中第1个值是一个bool, 表示它有没有成功, 第2个值是一个字典, 字典中有一个path属性, 保存着下载的image文件的路径. 因为使用的是yield方法, 所以这里的result中只有一个值.



pipelines.py

class JobboleArticlePipeline(object):

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

return item

class ArticleImagePipeline(ImagesPipeline):

# 重写函数, 改写item处理完成的函数

def item\_completed(self, results, item, info):

# result是一个list的结构. 可以获取多个图片保存的信息. 但由于使用yield, 一次只传递过一个item, 所以这里的result中只有一个元素.

for key, value in results:

# value是一个字典的结构

front\_image\_path = value["path"]

item["front\_image\_path"] = front\_image\_path

# 在完成处理后一定要返回item, 这样数据才能被下一个pipeline接收并处理.

return item #在此处添加断点再次进行调试, 看item中是否保存了图片下载的路径.

因为ITEM\_PIPELINES中设置的ArticleImagePipeline的优先级高于JobboleArticlePipeline, 在ArticleImagePipeline处理完成后才开始处理JobboleArticlePipeline, 在处理JobboleArticlePipeline时已经保存有front\_image\_path的值了.

可以在JobboleArticlePipeline中的return item处设置断点, 进行调试, 查看item > values中有一个front\_image\_path的键值对, 就是图片在本地的保存地址.

**把url进行md5的处理. 就可以把url变成一个唯一且长度固定的值, 作为数据库的主键.**

与spider同级目录下新建一个名为utils的python package来存放常用的函数.

新建utils/common.py,

import hashlib

def get\_md5(url):

m = hashlib.md5()

m.update(url)

return m.hexdigest()

if \_\_name\_\_ = "\_\_main\_\_"

# 进行hash的字符串必须要先进行utf-8的编码, 才能对它进行hash操作

print(get\_md5("http://jobbole.com".encode("utf-8")))

可以看到把这个url地址转换为md5字符串了.

由于unicode格式的字符不能进行md5的转换, 传递过来的url不确定是unicode格式的还是经过utf-8编码的, 所以要进行一个判断. python3中所有的字符串都是使用的unicode编码, 已经没有类似于python2中的unicode这个关键词了.

import hashlib

def get\_md5(url):

#如果url是以unicode进行编码的, 就把它进行utf-8的编码.

if isinstance(url, str):

url = url.encode("utf-8")

m = hashlib.md5()

m.update(url)

return m.hexdigest()

在jobbole.py这个爬虫中引入这个函数

from ArticleSpider.utils.common import get\_md5

在article\_item中添加处理后的url\_object\_id字段.

article\_item['url\_object\_id'] = get\_md5(response.url)

至此, 数据都已经提取并处理完成, 只需要进行数据的保存即可.

## 4-13 数据表设计和保存item到json文件

item都会经过item pipeline的处理, 可以在item pipeline中把数据保存到mysql或json中.

**将数据保存到json文件中.**

import codecs, json

# 自定义管道将Item导出为Json文件

class JsonWithEncodingPipeline(object):

class ArticleWithJsonPipeline(object):

# 初始化时调用

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

# 打开json文件

# 使用codecs完成文件的打开和写入能够解决编码方面的问题

self.file = codecs.open('article.json','w',encoding="utf-8")

# 重写Item处理

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

# 先把item转换为dict格式, 再使用json.dump把它转换为字符串.

# 需要关闭ensure\_ascii, 否则会以unicode方式写入中文字符, 中文字符就无法正确显示.

lines = json.dump(dict(item), ensure\_ascii=False)+'\n'

# 将一行数据写入

self.file.write(lines)

# 重写process\_item方法时必须要使用return把它返回去, 以供其它的pipeline使用.

return item

# 爬虫结束关闭spider时调用spider\_closed方法

def spider\_closed(self, spider):

# 关闭文件句柄

self.file.close()

修改settings中的item pipeline设置

ITEM\_PIPELINES = {

'ArticleSpider.pipelines.ArticleWithJsonPipeline': 300,

#'scrapy.pipeline.images.ImagesPipeline' : 200

'ArticleSpider.pipelines.ArticleImagePipeline': 200,

}

执行爬虫, 就会生成一个article.json文件.

scrapy自身也提供了把数据导出为不同文件格式的方法, 以导出为json格式进行说明.

修改pipelines.py文件.

#打开JsonItemExporter文件查看, 可以看到scrapy提供了把数据导出为多种格式的方法.

from scrapy.exporters import JsonItemExporter

#调用scrapy提供的json export导出json文件.

class JsonExporterPipeline(objcet):

# 调用scrapy提供的json exporter导出json文件

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

# 以二进制方法打开json文件

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

# 实例化一个JsonItemExporter对象exporter, 在实例化时需要传递几个参数.

self.exporter = JsonItemExporter(self.file, encoding="utf-8", ensure\_ascii = False)

#使用start\_exporting方法开始导出

self.exporter.start\_exporting()

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

self.exporter.export\_item(item)

return item

def close\_spider(self, spider):

#使用finish\_exporting方法结束导出

self.exporter.finish\_exporting()

self.file.close()

修改settings中的item pipeline设置

ITEM\_PIPELINES = {

'ArticleSpider.pipelines.JsonExporterPipeline': 300,

#'scrapy.pipeline.images.ImagesPipeline' : 200

'ArticleSpider.pipelines.ArticleImagePipeline': 200,

}

运行spider, 就可以看到生成的json文件了, 但是整体是放在一个列表中的. 查看exporters.py的源码, 在start\_exporting(self)和finish\_exporting(self)中, 分别写入了 b"[\n" 和 b"\n]" 的符号. 这样就保证了导出的数据是标准的json格式.

## 4-14 通过pipeline保存数据到mysql

pip install mysqlclient 安装Mysql客户端的python驱动程序

在linux中安装可能会失败, 对于ubuntu, 使用sudo apt install libmysqlclient-dev

对于centos, 使用sudo yum install python-devel mysql-devel来安装支持包, 再执行pip install mysqlclient的操作, 就会成功.

需要先设计mysql数据表的结构, scrapy中items中字段与mysql数据库表中字段的关系类似于django中models与mysql的关系. 可以通过django来对数据表进行管理. 也可以使用navicat手动建立数据表.

这里使用navicat来手动建立数据表

新建一个article\_spider的数据库, 新建一个article的数据表.

在栏位中添加

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名 | 类型 | 长度 | 小数点 | 不是null | default |
| title | varchar | 200 | 0 | TRUE |  |
| create\_date | date | 0 | 0 | FALSE |  |
| url | varchar | 300 | 0 | TRUE |  |
| url\_object\_id | varchar | 50 | 0 | TRUE | Empty string |
| front\_image\_url | varchar | 300 | 0 | FALSE |  |
| front\_image\_path | varchar | 200 | 0 | FALSE |  |
| comment\_num | int | 11 | 0 | TRUE | 0 |
| fav\_num | int | 11 | 0 | TRUE | 0 |
| praise\_num | int | 11 | 0 | TRUE | 0 |
| tags | varchar | 200 | 0 | FALSE |  |
| content | longtext | 0 | 0 | TRUE |  |

comment\_num, fav\_num和praise\_num设置为不能为null, 这时可以设置它们的默认值为0.

再把url\_object\_id设置为主键, 在后面就可以根据主键来进行更新.

因为在提取数据时是把日期当成字符串处理的, 要修改jobbole.py文件, 转化为日期.

imort datetime

# 时间转换处理

try:

create\_date = datetime.datetime.strptime(value, "%Y/%m/%d").date()

except Exception as e:

#如果出现异常, 就将它的时间设置为当前的时间

create\_date = datetime.datetime.now().date()

article\_item['create\_date'] = create\_date

在此处添加断点, 进行调试, 可以正常转换为日期.

**将数据保存到mysql中**

同步化将Item保存入数据库

import MySQLdb

# 同步机制写入数据库

class ArticleWithMysqlPipeline(object):

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

#创建一个连接MySQLdb.connect('host', 'user', 'password', 'dbname', charset, use\_unicode), 可以打开connect函数查看其源码.

self.conn = MySQLdb.connect('127.0.0.1', 'root', 'mysql', ' article\_spider', charset="utf8", use\_unicode=True)

#执行数据库的具体操作是由cursor来完成的.

self.cursor = self.conn.cursor()

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

#这里insert插入的操作要与之前数据库中设置的字段的名称和顺序相同.

insert\_sql = '''

insert into ArticleSpider(title, create\_date, url, url\_object\_id, front\_image\_url, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

self.cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

# 注意使用的是conn.commit, 不是cursor.commit

self.conn.commit()

def spider\_closed(self, spider):

self.conn.close()

在settings中配置ITEM\_PIPELINE

ITEM\_PIPELINES = {

# 'ArticleSpider.pipelines.JsonExporterPipeline': 300,

# 'scrapy.pipeline.images.ImagesPipeline' : 200

# 'ArticleSpider.pipelines.ArticleImagePipeline': 200,

'ArticleSpider.pipelines. ArticleWithMysqlPipeline': 300,

}

在self.cursor.execute打上断点, 在jobbole.py中article\_item['create\_date']上打上断点, 进行debug. F6继续执行.

回到数据库中进行查看, 已经插入了一条数据. 按F8取消断点, 就可以看到数据会源源不断的写进数据库.

**异步化将Item保存入数据库**

因为Scrapy的解析速度非常快, 加上文章的内容较大, 上面mysql数据库的写入是同步的操作, 可能会出现数据库的写入速度赶不上解析速度, 就会产生阻塞, 因此采用异步化的方式来进行数据的插入, twisted框架提供了一种连接池, 可以把mysql数据库的插入变成异步的操作.

在上面的操作中, self.cursor.execute和self.conn.commit()是同步的操作, 必须在完成上面的操作之后才能进行下一步的操作.

在settings中添加mysql的设置

MYSQL\_HOST = '127.0.0.1'

MYSQL\_DBNAME = 'article\_spider' #这里设置的是数据库的名字, 不是数据表的名字

MYSQL\_USER = 'root'

MYSQL\_PASSWORD = 'mysql'

MYSQL\_PORT = "3306"

修改pipelines.py文件

import MySQLdb

import MySQLdb.cursors

from twisted.enterprise import adbapi

# 异步操作写入数据库

class MysqlTwistedPipline(object):

# from\_settings和\_\_init\_\_这两个方法就能实现在启动spider时, 就把dbpool传递进来

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

self.dbpool = dbpool

# 在初始化时scrapy会调用from\_settings方法, 将setting文件中的配置读入, 成为一个settings对象, 这种写法是固定的, 其中的参数不能修改.

@classmethod

def from\_settings(cls, settings):

dbparas = dict(

host = settings["MYSQL\_HOST"], #可以在settings中设置此pipeline, 在此处放置断点, 进行debug, 查看能否导入. 在attribute中可以看到settings中定义的所有的值. F6执行, 就能看到取到了settings中设置的host的值了.

db = settings["MYSQL\_DBNAME"],

user = settings["MYSQL\_USERNAME"],

passwd = settings["MYSQL\_PASSWORD"],

charset = "utf8",

# pymysql模块中也有类似的模块pymysql.cursors.DictCursor

cursorclass = MySQLdb.cursors.DictCursor,

use\_unicode = True

)

# 创建twisted的mysql连接池, 使用twisted的连接池, 就能把mysql的操作转换为异步操作.

# twisted只是提供了一个异步的容器, 并没有提供连接mysql的方法, 所以还需要MySQLdb的连接方法.

$ adbapi可以将mysql的操作变成异步化的操作. 查看ConectionPool, def \_\_init\_\_(self, dbapiName, \*connargs, \*\*connkw).

# 需要指定使用的连接模块的模块名, 第一个参数是dbapiName, 即mysql的模块名MySQLdb. 第二个参数是连接mysql的参数, 写为可变化的参数形式. 查看MySQLdb的源码, 在from MySQLdb.connections import Connection中查看Connection的源码, 在class Connection中就能看到MySQLdb模块在连接mysql数据库时需要传递的参数. param这个dict中参数的名称要其中的参数名称保持一致. 即与connections.py中 class Connection中的def \_\_init\_\_中定义的参数保持一致.

dbpool = adbapi.ConnectionPool("MySQLdb", \*\*dbparas)

# 如果对上面的写法不太理解, 可以写成下面的形式

# dbpool = adbapi.ConnectionPool("MySQLdb", host = settings["MYSQL\_HOST"], db = settings["MYSQL\_DBNAME"], user = settings["MYSQL\_USERNAME"], passwd = settings["MYSQL\_PASSWORD"], charset = "utf8", cursorclass = MySQLdb.cursors.DictCursor, use\_unicode = True)

# 因为使用@classmethod把这个方法转换为类方法了, 所以cls就是指的MysqlTwistedPipline这个类, 所以cls(dbpool) 就相当于使用dbpool这个参数实例化MysqlTwistedPipline类的一个对象, 再把这个对象返回. 然后在init方法中接收这里创建的异步连接对象.

return cls(dbpool)

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

# 使用twisted将mysql数据库的插入变成异步插入, 第一个参数是自定义的函数, runInteraction可以把这个函数的操作变成异步的操作. 第二个参数是要插入的数据, 这里是item.

query = self.dbpool.runInteraction(self.do\_insert, item)

#处理异常, 这里也可以不传递item和spider

query.addErrback(self.handle\_error, item, spider)

# 自定义错误处理, 处理异步插入的异常, 这里也可以不传递item和spider, 只传递failure即可.

def handle\_error(self, failure, item, spider):

print(failure)

print(item)

# 执行具体的插入, 此时的cursor就是self.dbpool.runInteraction中传递过来的cursor, 使用这个cursor, 就可以把mysql的操作变成异步的操作. 并且此时也不用再手动执行commit的操作了.

def do\_insert(self, cursor, item):

insert\_sql = '''

INSERT INTO

ArticleSpider(title, create\_time, url, url\_object\_id, front\_image\_url, front\_image\_path, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["front\_image\_path"], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

在settings中把ITEM\_PIPELINE修改为上面定义的异步操作

ITEM\_PIPELINES = {

# 'ArticleSpider.pipelines.JsonExporterPipeline': 300,

# 'scrapy.pipeline.images.ImagesPipeline' : 200

# 'ArticleSpider.pipelines.ArticleImagePipeline': 200,

'ArticleSpider.pipelines.MysqlTwistedPipline': 100,

}

在执行操作前清空原有的所有数据库. twisted只支持关系型数据库的异步操作.

### 把django的ORM集成到scrapy中

https://github.com/scrapy-plugins/scrapy-djangoitem

实际上是scrapy的一个插件, 通过django model的形式将scrapy的item写入到数据库中.

在定义item的时候就定义为django item, 在保存item的时候就可以像django中的样, 直接使用save()来保存了. 会比上面使用sql语句写入数据库简单的多.

## 4-16 scrapy itemloader机制

随着爬取的网站的增多和提取的信息的增多, 要写大量的css或xpath selector语法, 不同的爬虫之间的代码不能重用, 后期维护的成本也比较高. scrapy提供了一种itemloader的机制, 可以使后期的维护变得很简单.

itemloader提供了一个容器. 这个容器中可以配置item中的某个字段需要用到哪种css或xpath规则来解析, 配置完成后调用itemloader方法, 就可以直接生成itemloader.

修改jobbole.py

form scrapy.loader import ItemLoader

#通过itemloader加载item, 先要实例化ItemLoader, 查看ItemLoader源码, 它有两个重要的参数, 一个是item, 一个是response. item即之前在parse\_detail第一行中定义的JobBoleArticleItem(), 它实例上是一个实例化的对象.

item\_loader = ItemLoader(item=JobBoleArticleItem(), response=response)

# ItemLoader一共有3个重要的方法.

# item\_loader.add\_css()

# item\_loader.add\_xpath()

# item\_loader.add\_value()

item\_loader = JobBoleArticleItemLoader(item=JobBoleArticleItem(),response=response)

# add\_css()中的第一个参数就是在items.py中定义的JoBoleArticleItem中的参数. 第二个参数就是它对应的css样式.

如

item\_loader.add\_css("title",".entry-header h1::text")

在item\_loader中实例化的JoBoleArticleItem()就会调用后面的css选择器, 从response中选择出内容放在title字段中.

form scrapy.loader **import** **ItemLoader**

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

name = "jobbole"

allowed\_domains = ["blog.jobbole.com"]

start\_urls = ['http://blog.jobbole.com/all-posts/']

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

*# 通过item loader加载item*

*# 文章封面图*

front\_image\_url = response.meta.get("front\_image\_url", "")

item\_loader = ItemLoader(item=JobBoleArticleItem(), response=response)

*# 通过item\_loader加载item*

item\_loader.add\_css('title', 'div.entry-header h1::text')

item\_loader.add\_css('create\_time', 'p.entry-meta-hide-on-mobile::text')

*# url的值并不是调用css选择器提取出来的, 而是通过response.url直接传递过来的, 要使用add\_value直接添加值的方法. 而add\_css则是调用css选择器把值取出来后再传递给前面的字段中.*

item\_loader.add\_value('url', response.url)

item\_loader.add\_value('url\_object\_id', get\_md5(response.url))

item\_loader.add\_value('front\_image\_url', [front\_image\_url])

item\_loader.add\_css('comment\_num', 'a[href="#article-comment"] span::text')

item\_loader.add\_css('content', 'div.entry')

item\_loader.add\_css('tags', 'p.entry-meta-hide-on-mobile a::text')

item\_loader.add\_css('praise\_num', '.vote-post-up h10::text')

item\_loader.add\_css('fav\_num', '.bookmark-btn::text')

*#写完上面的规则之后要调用item\_loader的load\_item()方法对以上的规则进行解析. 解析之后就生成了item对象.*

article\_item = item\_loader.load\_item()

**yield** article\_item

运用item\_loader的方法之后, 代码就变得非常简洁, 并且更具有可移植性和可配置性. 可以把css或xpath的匹配规则写入到数据库或文件中, 在解析时动态来获取这些匹配规则就可以了.

但还存在两个问题. 在yield article\_item处添加断点, debug. 看是否生成了item对象. 生成了一个JoboleArticleItem()的对象. 其中的values值就是在item\_loader中添加的所有的字段和add\_css或add\_value得到的值. 注意得到的所有的值都是列表的形式, 即使是使用add\_value方法传递了一个值, 也得到一个列表. 这也是符合逻辑的, 因为使用css选择器可能会选取到多个的值, 对于多值的css规则, 提取出来的值就会保存到列表中, 这里为了保持一致性, 就把提取出来的所有的元素都放在列表中了. 可以用提取列表中的第1个元素的方法解决. 第二个问题是如果没有人评论, 只会提取到评论两个字, 不会取到评论数为0, 可以给comment\_num的提取规则加一个处理函数, 像前面一样使用正则表达式进行判断, 如果没有提取到评论数, 就把它设置为0.

为了解决以上两个问题, 需要修改items.py文件.

定义的scrpy.Field()实际上有两个参数, 一个是input\_processor, 当这个字段的值传递过来时, 可以通过input\_processor进行一些预处理.

from scrapy.loader.processors import MapCompose

# MapCompose这个方法可以传递任意多的函数, 能传递过来的值进行预处理. 如对于title字段, 想要加一个后缀-jobbole, 在这个类之前先定义一个函数add\_jobbole()

def add\_jobbole(value):

return value+'-jobbole'

title = scrapy.Field(

input\_processor = MapCompose(add\_jobbole)

)

在MapCompose调用add\_jobbole这个函数的时候, 会把title中获取的值传递到add\_jobbole这个函数中, add\_jobbole这个函数以value的形参接收这个title的值, 进行处理后返回到title中. 这样就完成了传递过来的值的预处理.

甚至可以在MapCompose中传递一个匿名函数

title = scrapy.Field(

input\_processor = MapCompose(lambda x:x+'-jobbole')

)

屏蔽上面的add\_jobbole函数, 在jobbole.py中yield article\_item一行添加断点进行debug, 查看jobbole.py中article\_item中title的值是否有所变化.

也可以调用2个函数

def add\_jobbole(value):

return value+'-bobby'

title = scrapy.Field(

input\_processor = MapCompose(lambda x:x+'-jobbole', add\_jobbole)

)

再次debug, 查看title的值. 这样就说明了input\_process可以按顺序调用MapCompose中的多个函数.

可以用类似的方法处理时间, 把传递到create\_date中的的string时间转换为python的date对象. 只需要把之前对时间处理的语句封装为一个函数即可. 注意定义的date\_convert函数和add\_jobbole函数之间要间隔2行, 以满足PEP8的规范.

import datetime

def date\_convert(value):

try:

create\_date = datetime.datetime.strptime(value, "%Y/%m/%d").date()

except Exception as e:

#如果出现异常, 就将它的时间设置为当前的时间

create\_date = datetime.datetime.now().date()

return create\_date

create\_date = scrapy.Field(

input\_processor = MapCompose(date\_convert)

)

再次debug, 查看create\_date的值. 它就变成了一个放在数组中的date对象, 而不是之前的string的对象了.

前面使用了最基本的方式来解析的文章详情页, 这样使得spider的代码十分长, 不容易维护, 因此可以采用自定义ItemLoder的方式方便对规则的管理

同样, 可以处理comment\_num的问题.

解决第二个问题, 即所有字段获取的值都是数组的形式. 需要用到scrapy中的另外一个函数TakeFirst(), 在output\_processor中调用TakeFirst()函数, 就能把列表中的第1个元素提取出来了.

from scrapy.loader.processors import TakeFirst

create\_date = scrapy.Field(

input\_processor = MapCompose(date\_convert),

output\_processor = TakeFirst()

)

再次debug, 查看create\_date的值. 它就变成了一个date的对象, 并且不再放在数组中了.

如果要对所有的字段都添加一个output\_processor = TakeFirst()的方法, 就会显得很麻烦. 为了对所有的字段都取其中的第一个值, 可以定义一个我们自己的item\_loader.

修改item.py. 自定义ItemLoader

from scrapy.loader import ItemLoader

#自定义item\_loader, 继承于ItemLoader.

class ArticleItemLoader(ItemLoader):

#查看ItemLoader的源码, 其中可以设置一个default\_output\_processor

default\_output\_processor = TakeFirst()

在jobbole.py中就不能再使用系统的ItemLoader了, 而是要使用我们这里自定义的ArticleItemLoader.

修改jobbole.py, 导入并使用自定义的ArticleItemLoader

**from** **ArticleSpider.items** **import** JobBoleArticleItem, ArticleItemLoader

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

name = "jobbole"

allowed\_domains = ["blog.jobbole.com"]

start\_urls = ['http://blog.jobbole.com/all-posts/']

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

article\_item = JobBoleArticleItem()

*#通过item loader加载item*

front\_image\_url = response.meta.get("front\_image\_url", "") *# 文章封面图*

item\_loader = ArticleItemLoader(item=JobBoleArticleItem(), response=response)

现在由于已经使用了添加过TakeFirst()的自定义的ArticleItemLoader, 在item.py中就不需要逐个的添加TakeFirst的语句了.

再次debug, 查看item中的value值, 所有的字段都不再放在一个列表中了, 都只取了原来列表中的第一个值了.

再次改进input\_processor.

可以把原来写入到item\_loader中的一些功能, 如get\_md5, 以及对几个\_num进行正则表达式的匹配, 转变create\_date的格式, 这些功能都可以在items.py中实现.

修改items.py, 完成\_num的正则表达式的匹配功能, 把之前的代码封装成一个函数即可. 这个函数可以用于praise\_num, fav\_num, comment\_num3个\_num.

import re

def get\_num(value):

match\_re = re.match('.\*?(\d+).\*',value)

if match\_re:

x\_num = int(match\_re.group(1))

else:

x\_num = 0

return x\_num

praise\_num = scrapy.Field(

input\_processor = MapCompose(get\_num)

)

fav\_num = scrapy.Field(

input\_processor = MapCompose(get\_num)

)

comment\_num = scrapy.Field(

input\_processor = MapCompose(get\_num)

)

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

title = scrapy.Field()

create\_date = scrapy.Field(

input\_processor=MapCompose(date\_convert),

)

url = scrapy.Field()

url\_object\_id = scrapy.Field()

front\_image\_url = scrapy.Field(

output\_processor=MapCompose(return\_value)

)

front\_image\_path = scrapy.Field()

praise\_nums = scrapy.Field(

input\_processor=MapCompose(get\_nums)

)

comment\_nums = scrapy.Field(

input\_processor=MapCompose(get\_nums)

)

fav\_nums = scrapy.Field(

input\_processor=MapCompose(get\_nums)

)

tags = scrapy.Field(

input\_processor=MapCompose(remove\_comment\_tags),

output\_processor=Join(",")

)

content = scrapy.Field()

进行debug, 查看三者的值.

还要对list格式的tags做一个join的操作. 因为提取出来的tags已经是一个list了, 对它进行take\_first的操作只能取到第一个值, 这样也就不合适了. 也要对它进行一个join的操作, 把list中的所有元素组合成一个字符串.

就不能再使用ArticleItemLoader中的default\_output\_processor了, 而是要使用scrapy提供的Join的功能.

from scrapy.loader.processors import Join

tags = scrapy.Field(

#查看Join()的源码, 它的参数是seperator, 即连接符, 在这里把它设置成","

output\_processor = Join(",")

)

debug查看tags的值. 把原来的列表变成了一个字符串. 但此时其中还包含着评论, 要把评论从中去除掉. 定义一个函数判断tags中的值是否是评论, 如果是评论, 把这个值返回为空即可.

def remove\_comment\_tag(value):

if "评论" in value:

return ""

else:

return value

tags = scrapy.Field(

input\_processor = MapCompose(remove\_comment\_tag),

output\_processor = Join(",")

)

还需要注意的是front\_image\_url, 在配置了下载图片的pipeline之后, 这个字段传递回去的一定要是一个列表, 在处理时也是把它放在一个列表中传递过来的. 但现在使用了ArticleItemLoader中的default\_output\_processor = TakeFirst()之后, items.py中的front\_image\_url就会变成字符串的形式. 把它交给image pipeline下载的时候就会抛出异常.

可以定义一个函数, 这个函数什么都不处理, 只返回一个value, 然后定义front\_image\_url中的output\_processor调用这个函数即可. 这样, 既没有修改其中的值又覆盖了默认的output\_processor.

def return\_value(value):

return value

front\_image\_url = scrapy.Field(

output\_processor = MapCompose(return\_value)

)

之前写过一个插入jobbole文章的sql语句, 在这个sql语句中是直接取front\_image\_url这个字段, 这时它是一个list, 要改为取这个字段的list中的第一个值.

修改pipelines.py文件

self.cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

之前在pipelines.py中写了一个ArticleImagePipeline的pipeline, 继承于ImagesPipeline

class ArticleImagePipeline(ImagesPipeline):

def item\_completed(self, results, item, info):

for key, value in results:

front\_image\_path = value["path"]

item["front\_image\_path"] = front\_image\_path

return item

这个pipeline会对所有的item都会生效, 在后面爬取知乎的问答时, 文章没有封面图片, 再使用此pipeline进行提取时就可能会出错, 所以这里可以进行一个判断, 只处理有front\_image\_url的那些item. item是一个类字典的结构, 所以if "front\_image\_url" in item这种写法也是正确的.

class ArticleImagePipeline(ImagesPipeline):

def item\_completed(self, results, item, info):

if "front\_image\_url" in item:

for key, value in results:

front\_image\_path = value["path"]

item["front\_image\_path"] = front\_image\_path

return item

在jobbole.py的yield article\_item上加上断点, 通过main.py文件进行调试, 在articel\_item的value中就可以看到tags中已经没有了评论的内容. 同时front\_iamge\_url也变成了列表的形式.

把front\_image\_url提取的语句拿出来放在item\_loader = ArticleImagePipeline之前, 把jobbole.py中之前使用article\_item的方法写的提取数据的代码全部屏蔽即可, 以后就使用item\_loader的方法来提取数据了. 这样代码更加精练, 并且在jobbole.py爬虫文件中只进行数据的提取, 数据的处理都放在items.py中进行.

spider文件的修改

import re, scrapy, datetime

from scrapy.http import Request

from urllib import parse

from scrapy.loader import ItemLoader

class JobboleSpider(scrapy.Spider):

# 爬虫的名称 后续启动爬虫是采用此名称

name = "jobbole"

# 爬取允许的域名

allowed\_domains = ["blog.jobbole.com"]

# 起始url列表 , 其中的每个URL会进入下面的parse函数进行解析

start\_urls = ['http://blog.jobbole.com/all-posts/']

# 列表页面的解析

def parse(self, response):

# 获取文章列表中的每一篇文章的url交给Scrapy下载并解析

article\_nodes = response.css('div#archive .floated-thumb .post-thumb a')

for article\_node in article\_nodes:

# 解析每个文章的封面图

front\_image\_url = article\_node.css('img::attr(src)').extract\_first("")

# 解析每个文章的url

article\_url = article\_node.css('::attr(href)').extract\_first("")

# 智能对url进行拼接, 如果url中不带有域名则会自动添加域名

# 通过在Request中设置meta信息来进行数据的传递

yield Request(url=parse.urljoin(response.url, article\_url),meta={'front\_image\_url':parse.urljoin(response.url, front\_image\_url)}, callback=self.parse\_detail)

# 获取文章的下一页url地址, 并交给自身解析

next\_url = response.css('a.next.page-numbers::attr(href)').extract\_first('')

if next\_url:

yield Request(url=parse.urljoin(response.url, next\_url), callback=self.parse)

# 详情页面的解析

def parse\_detail(self, response):

article\_item = JobBoleArticleItem()

# 从response中获取文章封面图

front\_image\_url = response.meta.get('front\_image\_url', '')

item\_loader = JobBoleArticleItemLoader(item=JobBoleArticleItem(),response=response)

item\_loader.add\_css('title', 'div.entry-header h1::text')

item\_loader.add\_css('create\_time', 'p.entry-meta-hide-on-mobile::text')

item\_loader.add\_value('url', response.url)

item\_loader.add\_value('url\_object\_id', get\_md5(response.url))

item\_loader.add\_value('front\_image\_url', [front\_image\_url])

item\_loader.add\_css('comment\_num', 'a[href="#article-comment"] span::text')

item\_loader.add\_css('content', 'div.entry')

item\_loader.add\_css('tags', 'p.entry-meta-hide-on-mobile a::text')

item\_loader.add\_css('praise\_num', '.vote-post-up h10::text')

item\_loader.add\_css('fav\_num', '.bookmark-btn::text')

article\_item = item\_loader.load\_item()

yield article\_item

自定义的ItemLoader

import datetime

import re

import scrapy

from scrapy.loader import ItemLoader

from scrapy.loader.processors import MapCompose, TakeFirst, Join

# 去除文本中的点

def remove\_dote(value):

return value.replace('·','').strip()

# 时间转换处理

def date\_convert(value):

try:

create\_time = datetime.datetime.strptime(value, "%Y/%m/%d").date()

except Exception as e:

create\_time = datetime.datetime.now().date()

return create\_time

# 获得数字

def get\_num(value):

match\_re = re.match('.\*?(\d+).\*', value)

if match\_re:

num = int(match\_re.group(1))

else:

num = 0

return int(num)

# 获取点赞数

def get\_praise\_num(value):

match\_re = re.match('<h10 id=".\*?">(\d+)</h10>', value)

if match\_re:

num = match\_re.group(1)

else:

num = 0

return int(num)

# 去掉tag中的评论

def remove\_comment\_tag(value):

if "评论" in value:

return ""

return value

# 默认返回

def return\_value(value):

return value

# 自定义ITemLoader

class JobBoleArticleItemLoader(ItemLoader):

# 改写默认的output\_processor

default\_output\_processor = TakeFirst()

# 伯乐在线Item

class JobBoleArticleItem(scrapy.Item):

title = scrapy.Field()

create\_time = scrapy.Field(

# 该传入的字段值要批量处理的函数

input\_processor=MapCompose(remove\_dote, date\_convert),

)

url = scrapy.Field()

url\_object\_id = scrapy.Field()

front\_image\_url = scrapy.Field(

output\_processor = MapCompose(return\_value)

)

front\_image\_path = scrapy.Field()

praise\_num = scrapy.Field(

input\_processor = MapCompose(get\_praise\_num)

)

fav\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

comment\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

tags = scrapy.Field(

input\_processor=MapCompose(remove\_comment\_tag),

output\_processor = Join(',')

)

content = scrapy.Field()

```

前面使用了最基本的方式来解析的文章详情页, 这样使得spider的代码十分长, 不容易维护, 因此可以采用自定义ItemLoder的方式方便对规则的管理

spider文件的修改

import re, scrapy, datetime

from scrapy.http import Request

from urllib import parse

from scrapy.loader import ItemLoader

class JobboleSpider(scrapy.Spider):

# 爬虫的名称 后续启动爬虫是采用此名称

name = "jobbole"

# 爬取允许的域名

allowed\_domains = ["blog.jobbole.com"]

# 起始url列表 , 其中的每个URL会进入下面的parse函数进行解析

start\_urls = ['http://blog.jobbole.com/all-posts/']

# 列表页面的解析

def parse(self, response):

# 获取文章列表中的每一篇文章的url交给Scrapy下载并解析

article\_nodes = response.css('div#archive .floated-thumb .post-thumb a')

for article\_node in article\_nodes:

# 解析每个文章的封面图

front\_image\_url = article\_node.css('img::attr(src)').extract\_first("")

# 解析每个文章的url

article\_url = article\_node.css('::attr(href)').extract\_first("")

# 智能对url进行拼接, 如果url中不带有域名则会自动添加域名

# 通过在Request中设置meta信息来进行数据的传递

yield Request(url=parse.urljoin(response.url, article\_url),meta={'front\_image\_url':parse.urljoin(response.url, front\_image\_url)}, callback=self.parse\_detail)

# 获取文章的下一页url地址, 并交给自身解析

next\_url = response.css('a.next.page-numbers::attr(href)').extract\_first('')

if next\_url:

yield Request(url=parse.urljoin(response.url, next\_url), callback=self.parse)

# 详情页面的解析

def parse\_detail(self, response):

article\_item = JobBoleArticleItem()

# 从response中获取文章封面图

front\_image\_url = response.meta.get('front\_image\_url', '')

item\_loader = JobBoleArticleItemLoader(item=JobBoleArticleItem(),response=response)

item\_loader.add\_css('title', 'div.entry-header h1::text')

item\_loader.add\_css('create\_time', 'p.entry-meta-hide-on-mobile::text')

item\_loader.add\_value('url', response.url)

item\_loader.add\_value('url\_object\_id', get\_md5(response.url))

item\_loader.add\_value('front\_image\_url', [front\_image\_url])

item\_loader.add\_css('comment\_num', 'a[href="#article-comment"] span::text')

item\_loader.add\_css('content', 'div.entry')

item\_loader.add\_css('tags', 'p.entry-meta-hide-on-mobile a::text')

item\_loader.add\_css('praise\_num', '.vote-post-up h10::text')

item\_loader.add\_css('fav\_num', '.bookmark-btn::text')

article\_item = item\_loader.load\_item()

yield article\_item

自定义的ItemLoader

import datetime

import re

import scrapy

from scrapy.loader import ItemLoader

from scrapy.loader.processors import MapCompose, TakeFirst, Join

# 去除文本中的点

def remove\_dote(value):

return value.replace('·','').strip()

# 时间转换处理

def date\_convert(value):

try:

create\_time = datetime.datetime.strptime(value, "%Y/%m/%d").date()

except Exception as e:

create\_time = datetime.datetime.now().date()

return create\_time

# 获得数字

def get\_num(value):

match\_re = re.match('.\*?(\d+).\*', value)

if match\_re:

num = int(match\_re.group(1))

else:

num = 0

return int(num)

# 获取点赞数

def get\_praise\_num(value):

match\_re = re.match('<h10 id=".\*?">(\d+)</h10>', value)

if match\_re:

num = match\_re.group(1)

else:

num = 0

return int(num)

# 去掉tag中的评论

def remove\_comment\_tag(value):

if "评论" in value:

return ""

return value

# 默认返回

def return\_value(value):

return value

# 自定义ITemLoader

class JobBoleArticleItemLoader(ItemLoader):

# 改写默认的output\_processor

default\_output\_processor = TakeFirst()

# 伯乐在线Item

class JobBoleArticleItem(scrapy.Item):

title = scrapy.Field()

create\_time = scrapy.Field(

# 该传入的字段值要批量处理的函数

input\_processor=MapCompose(remove\_dote, date\_convert),

)

url = scrapy.Field()

url\_object\_id = scrapy.Field()

front\_image\_url = scrapy.Field(

output\_processor = MapCompose(return\_value)

)

front\_image\_path = scrapy.Field()

praise\_num = scrapy.Field(

input\_processor = MapCompose(get\_praise\_num)

)

fav\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

comment\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

tags = scrapy.Field(

input\_processor=MapCompose(remove\_comment\_tag),

output\_processor = Join(',')

)

content = scrapy.Field()

```

# 第5章 scrapy爬取知名问答网站

## 5-1 session和cookie自动登录机制

## 5-2 requests模拟登陆知乎

常见的http code

|  |  |
| --- | --- |
| code | 说明 |
| 200 | 请求被成功处理 |
| 301/302 | 永久性重定向/临时性重定向 |
| 403 | 没有权限访问 |
| 404 | 表示没有对应的资源 |
| 500 | 服务器错误 |
| 503 | 服务器停机或正在维护 |

打开检查元素中的network, ctrl+F5强制刷新页面, 给出了每个页面的请求状态.

爬虫应该避免爬取404, 500等的页面, 而只去获取200的页面.

分别使用错误的手机号和邮箱去登陆知乎, 获取其提交的POST内容和POST提交的地址.

如果要进行模拟登陆, 必须要判断输入的账号的类型, 是手机号还是邮箱. 还需要使用request发送post请求到https://www.zhihu.com页面, 拿到csrf code.

查看网页的源码,

<div class="view view-signin" data-za-module="SignInForm">

<form method="POST">

<input type="hidden" name="\_xsrf" value="b3dc96e1f85c0e50edfa6aafc41b606b"/>

<div class="group-inputs">

其中有一个\_xsrf的选项. 可以使用正则匹配或css取出来这个值. 使用POST方法向服务器发送一个请求, 服务器会返回包含cookie值的request, 保存cookie值, 在下次登陆的时候使用此cookie就能够自动登陆了.

进入虚拟环境, 安装requests

workon xxx

pip install requests

# 第一步: 新建/utils/zhihu\_login\_requests.py

import requests, re

#在python2中是cookielib, 在python3中是http.cookiejar.

try:

import cookielib

except:

import http.cookiejar as cookielib

def zhihu\_login(account, password):

#知乎模拟登录

if re.match("^1\d{10}", account):

print("手机号码登陆")

post\_url = "https://www.zhihu.com/login/phone\_num"

# 第二步

def zhihu\_login(account, password):

#知乎登录

if re.match("^1\d{10}",account):

print ("手机号码登录")

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

}

else:

if "@" in account:

#判断用户名是否为邮箱

print("邮箱方式登录")

post\_url = "https://www.zhihu.com/login/email"

post\_data = {

"\_xsrf": get\_xsrf(),

"email": account,

"password": password

}

response\_text = session.post(post\_url, data=post\_data, headers=header)

session.cookies.save()

# 第三步

def get\_xsrf():

#获取xsrf code

#在使用request发送get请求是默认的请求头是python2和python3, 知乎会返回一个500的服务器错误信息. 要添加headers信息.

agent = "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0"

header = {

"HOST":"www.zhihu.com",

"Referer": "https://www.zhizhu.com",

'User-Agent': agent

}

response = session.get("https://www.zhihu.com", headers=header)

#<input type="hidden" name="\_xsrf" value="b3dc96e1f85c0e50edfa6aafc41b606b"/>

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response.text)

if match\_obj:

return (match\_obj.group(1))

else:

return ""

# 第四步

#模拟登陆时不使用requests, 而是使用requests.session()的方法, 表示某一次请求是一次长连接. 这样就不用每次使用requests时都去发送一个请求, 这样效率会更高.

session = requests.session()

# 第五步

response\_text = session.post(post\_url, data = post\_data, headers = header)

# 保存服务器返回的cookies, 因为cookies中已经保存了用户登陆的信息, 在以后访问知乎的首页时就不需要再走zhihu\_login这个流程了.

# 直接使用session.cookies.save()方法时会出错, 使用cookielib中的LWPCookieJar方法实例化一个session.cookies, 就可以直接调用save()方法进行cookies的保存.

# 可以把session.cookies实例化放在最前面, 这里放在这里只是为了说明的清楚.

session.cookies = cookielib.LWPCookieJar(filename="cookies.txt")

session.cookies.save()

# 第六步

#在session.cookies.save() 设置断点, 进行调试. 查看response\_text, 其中有一个text的值, 是一个unicode字符串. 删除多余的\, 把它放使用python的print函数打印, 就可以看到登陆成功的字样. F8继续进行调试, 就会在zhihu\_login\_requests.py同目录下生成一个cookies.txt的文件.

zhihu\_login("18782902568", "admin123")

# 第七步, 在前面已经保存了cookies.txt这个文件的基础上, 导入cookies.

try:

session.cookies.load(ignore\_discard=True)

except:

print ("cookie未能加载")

# 第八步 再次请求知乎的主页, 并把response的内容写入到文件中. 打开保存的这个文件查看, 是已经登陆后的主页了.

# 这样, 通过把request的cookies和session保存到本地, 在下一次请求网站的时候, 直接从cookies文件中读取cookie, 使用带有cookie的request请求页面时, 就是处于登陆状态了.

def get\_index():

response = session.get("https://www.zhihu.com", headers=header)

with open("index\_page.html", "wb") as f:

f.write(response.text.encode("utf-8"))

print ("ok")

# zhihu\_login("account", "password")

#因为之前已经把登陆成功的cookies保存下来了, 这里就可以直接获取主页, 来验证是否能够通过cookies自动登陆.

get\_index()

# 第九步

# cookie有过期时间, 它是可能会失效的, 要判断一下用户是否处于登陆状态.

def is\_login():

# 通过个人中心页面返回状态码来判断是否为登录状态, 如果是没有登陆的状态下访问个人中心的一些页面, 如私信页面, 会出现302的错误, 就可以通过这个状态码来判断用户是否登陆.

inbox\_url = "https://www.zhihu.com/inbox"

response = session.get(inbox\_url, headers=header, allow\_redirects=False)

if response.status\_code != 200:

return False

else:

return True

# 先登陆, 再验证是否登陆成功.

zhihu\_login("18782902568", "admin123")

is\_login()

进一步改进代码

如果已经有cookies, 判断是否过期, 如果没有过期, 就加载cookies并登录, 如果成功登录, 就进行一步, 如果没有成功登录, 就使用requests重新登录并更新cookies.

如果没有cookies, 就直接使用requests登录, 并保存cookies

## 5-5 scrapy模拟知乎登录

第一步, 进入到虚拟环境中

进入到项目目录中

scrapy genspider zhihu www.zhihu.com

import scrapy

class ZhihuSpider(scrapy.Spider):

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

def parse(self,response):

pass

# 打开scrapy.Spider查看, 它的入口是一个start\_requests函数. 知乎必须要登陆才能查看其中的内容, 所以首先要进行模拟登陆. 需要重写start\_requests函数.

def start\_requests(self):

# scrapy.FormRequest可以完成表单的提交. 查看FromRequest的源码, 它有两个参数, 第一个参数是url, 第二个参数是formdata的参数. 是和使用requests模拟登陆进的post\_data的值一样的.

return [srcapy.FormRequest(

url = "https://www.zhihu.com/login/phone\_num",

formdata = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

}

)]

# 先要在获取xsrf的值. 第一种方法是使用类似于requests模拟登陆时使用的get\_xsrf函数. 第二种方法是使用scrapy提供的异步UI. scrapy是基于twisted异步实现的, 要指定一个callback函数, 如果不指定callback, 它会默认调用parse函数.

def start\_requests(self):

**return [scrapy.Request('https://www.zhihu.com/#signin'), headers = headers, callback=self.login]**

return [srcapy.FormRequest(

url = "https://www.zhihu.com/login/phone\_num",

formdata = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

})]

#定义一个login函数, 把登陆的功能放在这个函数中.

def login(self,response):

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

}

# 修改ZhihuSpider为如下.

class ZhihuSpider(scrapy.Spider):

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

def parse(self,response):

pass

# 打开scrapy.Spider查看它是入口是一个start\_requests的函数. 知乎必须要登陆才能查看其中的内容, 所以首先要进行模拟登陆. 需要重写start\_requests函数.

def start\_requests(self):

#首先从登陆页面获取数据, 获取到数据之后调用login进行登陆.

return [scrapy.Request('https://www.zhihu.com/#signin'), headers=self.headers callback=self.login]

def login(self,response):

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

}

在start\_request中先使用scrapy.Request来请求到了知乎的首页, 会返回一个response, 其中包含着xsrf的信息, 就可以使用正则来获取到xsrf的值. 修改login函数.

import re

headers = {

"HOST":"www.zhihu.com",

"Referer": "https://www.zhizhu.com",

'User-Agent': "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0"

}

def login(self, request):

response\_text = response.text

# response.text返回的是整个的html页面的内容, 正则匹配默认只匹配第一行的内容, 要加上参数re.DOTALL才能进行全文匹配. 可以在下一句加上断点debug查看.

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response.text, re.DOTALL)

xsrf = ""

if match\_obj:

xsrf = match\_obj.group(1)

#如果得到xsrf的值之后 , 就可以进行模拟登陆了. 如果没有取到xsrf的值, 就无法进行模拟登陆, 也就没必要继续进行操作了.

if xsrf:

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": xsrf,

"phone\_num": account,

"password": password

}

# 使用scrapy.FormRequest构造登陆的post请求

return [srcapy.FormRequest(

url = post\_url,

formdata = post\_data,

headers = self.headers,

#scrapy中的request, 想要对它进行下一步的处理, 都要设置一个callback. 前面的requests库是同步的操作, 这里的scrapy的request是异步的操作, 需要给它设置一个callback函数, 用来验证这个login是否成功登陆了.

# 注意这里不能用check\_login(), 如果使用check\_login()了, 相当于调用这个函数, 函数会把最后的值返回到这里. 这里要传递的是check\_login这个对象.

callback = self.check\_login

)]

def check\_login(self, response):

# 验证服务器返回的数据用来判断是否登陆成功. 如果成功登陆, 就可以直接请求下一个页面, scrapy会自动把cookie值保存下来.

# 在下一句添加断点debug, 可以看到text\_json是一个dict型的, 其中有一个msg的变量, 值为登陆成功.

import json

text\_json = json.loads(request.text)

if "msg" in text\_json and text\_json["msg"] == "登录成功":

# 先进行登陆, 如果登陆成功, 再发送请求获取start\_urls中的地址.

for url in self.start\_urls:

#如果不写回调函数的话, 会自动调用parse函数进行解析.

yield scrapy.Request(url, dont\_filter=True, headers=self.headers)

在main.py中之前写的是执行jobbole, 要修改为

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

在start\_request中的return处, login的match\_obj处和check\_login的pass处打上断点, 同时把settings中的ROBOTS\_OBEY=True改为False, 进行debug, 查看debug的信息. 首先会进入start\_request函数, 按F8, 进入到login函数中, 查看response\_text, 其中就是登陆页面的文本. 按F6跳到下一个断点处, 会跳到check\_login中, 查看response, 即上一步在login函数中发送的登陆请求的响应. 在text中为登陆成功的提示.

def parse(self, response):

pass

在pass处添加断点, 进行debug, 查看response的信息, 是否是已登陆状态的. 如果验证是已登陆状态的, 就可以在此基础上进行数据提取了. 定义一个parse\_detail来进行数据的提取.

def parse\_detail(self, response):

pass

## 5-6 知乎分析以及数据表设计

爬取网站时必须要明确, 要爬取的网站是什么, 二是要爬取网站中的什么内容. 这里爬取知乎网站的问题和回答.

要把问题和回答分别提取并保存到数据库中.

知乎可能会返回两种页面. 旧版的系统是

https://www.zhihu.com/question/56320032/answer/149034527,

而新版的系统是https://www.zhihu.com/question/56320032/. 在旧版的系统中, 回答并不是完全显示出来的, 需要点击查看全部X个回答, 才能够跳转到所有的回答页面中.

要把提取到的所有的旧版的url地址都转化为新版的去解析, 这样就能减少很多的工作量.

进入虚拟环境

使用scrapy shell也要添加headers的信息, 不然服务器就会返回500的错误.

scrapy shell -s USER-AGENT="Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0" https://www.zhihu.com/question/56320032/

scarpy shell -h

Usage

=====

scrapy shell [url|file]

Interactive console for scraping the given url

Options

=======

--help, -h show this help message and exit

-c CODE evaluate the code in the shell, print the result and

exit

--spider=SPIDER use this spider

--no-redirect do not handle HTTP 3xx status codes and print response

as-is

Global Options

--------------

--logfile=FILE log file. if omitted stderr will be used

--loglevel=LEVEL, -L LEVEL

log level (default: DEBUG)

--nolog disable logging completely

--profile=FILE write python cProfile stats to FILE

--pidfile=FILE write process ID to FILE

--set=NAME=VALUE, -s NAME=VALUE

set/override setting (may be repeated)

--pdb enable pdb on failure

response.text

查看返回的网页的内容. 如果已经知道某个问题的url地址, 在未登陆状态下也是可以访问的, 但登陆之后在个人主页上能够访问到更多的内容. 为了可以爬取到更多的内容, 就需要进行登陆.

response.url

查看返回的url地址, 就能够判断它是不是老版本的页面了.

或者把网页的内容写入到一个文件中, 根据文件的格式来判断.

with open("D:\David/Desktop/zhihu.html", "wb") as file:

file.write(response.text.encode("utf-8"))

file.close()

查看文件, 看其是否是老版本的网页. 如果是, 就可以根据两种不同的版本来分别设计提取的规则了.

对于问题, 需要提取它的标题, 内容, 分类标签, 评论的数量, 关注的数量, 浏览的数量. 还有一个created\_time和updated\_time, 通过api的接口可以访问到. url以及url最后面的id. 可以把它的id作为主键. 回答的数量.

知乎问题的sql数据表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 字段名 | 类型 | 长度 | 小数点 | 不是null | 默认值 | 主键 |
| zhihu\_id | bignit | 20 | 0 | T |  | 1 |
| topics | varchar | 255 | 0 | F |  |  |
| url | varchar | 300 | 0 | T |  |  |
| title | varchar | 200 | 0 | T |  |  |
| content | longtext | 0 | 0 | T |  |  |
| create\_time | datetime | 0 | 0 | F |  |  |
| update\_time | datetime | 0 | 0 | F |  |  |
| answer\_num | int | 11 | 0 | T | 0 |  |
| comment\_num | int | 11 | 0 | T | 0 |  |
| watch\_user\_num | int | 11 | 0 | T | 0 |  |
| click\_num | int | 11 | 0 | T | 0 |  |
| crawl\_time | datetime | 0 | 0 | T |  |  |
| crawl\_update\_time | datetime | 0 | 0 | F |  |  |

知乎回答的sql数据表

对于回答. 如果回答数很多, 并不是一次性的把所有回答都加载出来, 而是有一个更多的按钮, 点击这个按钮会加载更多的回答内容.

打开网页的调试模式, 点击下一页的按钮, 在network中就可以看到请求的url地址.

Request URL:"https://www.zhihu.com/api/v4/questions/26240321/answers?sort\_by=default&include=data%5B%2A%5D.is\_normal%2Cis\_collapsed%2Ccollapse\_reason%2Cis\_sticky%2Ccollapsed\_by%2Csuggest\_edit%2Ccomment\_count%2Ccan\_comment%2Ccontent%2Ceditable\_content%2Cvoteup\_count%2Creshipment\_settings%2Ccomment\_permission%2Cmark\_infos%2Ccreated\_time%2Cupdated\_time%2Crelationship.is\_authorized%2Cis\_author%2Cvoting%2Cis\_thanked%2Cis\_nothelp%2Cupvoted\_followees%3Bdata%5B%2A%5D.author.follower\_count%2Cbadge%5B%3F%28type%3Dbest\_answerer%29%5D.topics&limit=20&offset=23"

其实把它改为

https://www.zhihu.com/api/v4/questions/26240321/answers?limit=20&offset=0

也是能够正常提取到数据的.

这里返回的是json的数据, 就不用再通过解析了, 可以直接提取其中的内容.

最重要的参数就是questions/26240321/,limit=20&offset=23, 通过改变offset的值就可以提取到所有的回答.

返回的json数据中有两个字段, 一个paging, 一个data.

paging: {

is\_end: false,

totals: 48,

previous: "http://www.zhihu.com/api/v4/questions/26240321/answers?limit=20&offset=0",

is\_start: true,

next: "http://www.zhihu.com/api/v4/questions/26240321/answers?limit=20&offset=20"

},

paging中,

is\_end是指当前页显示的回答是否是最后一页.

totals是所有回答的数量. previous是上一页的地址, 由于这里是第1页, 所以上一页和本页的地址是相同的.

is\_start是本页是否为第一页.

next是下一页的url地址.

通过is\_end和next就可以提取出下一页的url. 如果is\_end=True, 就是最后一页了, 就不用再进行提取了.

data中包含着20个字典的结构, 每个字典结构中都是一个完整的回答.

在第一层字典结构中, 有一个id, 就是此回答的id. 通过该回答的id就能访问到该问题的api

https://www.zhihu.com/api/v4/answers/32495807

created\_time: 1414337116,

id: 32495807,

voteup\_count: 2221,

url: "http://www.zhihu.com/api/v4/answers/32495807",

updated\_time: 1487921088,

content: "XXXX"

comment\_count: 180,

type: "answer",

author: {

id: "63cdb218d3018e28fd9b32b51e1ee4da"}

因为知乎允许匿名回答, author的id可能也是不存在的.

question: {

created: 1414200023,

url: "http://www.zhihu.com/api/v4/questions/26240321",

title: "如何当一个好的面试官?",

type: "question",

id: 26240321,

updated\_time: 1414200023

}

created和updated\_time是该问题的创建时间和更新时间. 这是在网页中提取不到的内容.

这里只是做了一个知乎的问题的索引和搜索, 太多的字段也没有意义.

设计数据表

其中的create\_time和update\_time在网页中无法提取, 在answer的api中可以提取到.

answer\_num, comment\_num, watch\_user\_unm, click\_num不能为空, 设置它们的默认值为0.

crawl\_time是首次爬取的时间, 因为不止要对问题爬取一次, 还要设置一个crawl\_update\_time为此问题爬取的更新时间.

问题sql数据表

author\_id可能是获取不到的, 所以可以为空

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 字段名 | 类型 | 长度 | 小数点 | 不是null | 默认值 | 主键 |
| zhihu\_id | bigint | 20 | 0 | T |  | 1 |
| url | varchar | 300 | 0 | T |  |  |
| question\_id | bigint | 20 | 0 | T |  |  |
| author\_id | varchar | 100 | 0 | F |  |  |
| content | longtext | 0 | 0 | T |  |  |
| praise\_num | int | 11 | 0 | T | 0 |  |
| comment\_num | int | 11 | 0 | T | 0 |  |
| create\_time | datetime | 0 | 0 | T |  |  |
| update\_time | datetime | 0 | 0 | T |  |  |
| crawl\_time | datetime | 0 | 0 | T |  |  |
| crawl\_update\_time | datetime | 0 | 0 | F |  |  |

## 5-8 item loder方式提取question

爬取的策略:

与之前的jobbole不同, jobbole有一个所有文章的入口, 直接从这个入口爬取数据就可以了. 但知乎没有问题的入口, 就要使用深度优先的算法来实现.

在登陆之后爬取的是知乎首页的内容, 获取首页的url之后要交给parse函数进行处理.

for url in self.start\_urls:

yield scrapy.Request(url, dont\_filter=True, headers=self.headers)

在上面的yield中没有设置callback函数, 会自动调用parse函数进行处理.

def parse(self,request):

# 提取出html页面中所有的url, 并跟踪这些url进行下一步的爬取.

# 如果提取的url格式为/question/xxx的形式, 就下载之后直接进入解析函数parse\_detail提取问题的详情.

# 所有的url都是放在a标签中的, 并且有些url没有域名, 所以要进行url的拼接.

all\_urls = response.css("a::attr(href)").extract()

try:

#python2

import urlparse as parse

except:

#python3

from urllib import parse

all\_urls = [parse.urljoin(response.url, url) for url in all\_urls]

# 使用python的filter函数过滤掉所有不符合规则的url. 把all\_urls中的每一个元素作为变量传递给lambda函数, x就是all\_urls中的所有元素. 如果all\_urls中的元素是以https开头的, 就返回为True, 否则返回为False, 返回为True的元素就会放在新的all\_urls列表中.

all\_urls = filter(lambda x:True if x.startswith("https") else False, all\_urls)

all\_urls = [url for url in all\_urls if url.startswith("https")]

for url in all\_urls:

#在此处添加断点进行调试. url中有很多不符合规则的ulr, 如非http开头的js的链接, 要把它们去除掉.

#还可以使用if url.startswith("https")这种方法.

# pass

#爬取到一页中所有的url后, 还要提取出其中问题的url, 即questions/xxx/answers/xxxx这种格式的url. 要提取出问题的id, 如果是带有answers的旧版的url, 还要提取出question的部分, 转化为新版格式的url. 如果是旧版的url, question/xxxx/后面是一个/,新版的则是一个结束符, 使用或的方法把这两种情况都提取出来. ($|/)中就是或的关系, 满足任意一个条件即可.

# print(url)

match\_obj = re.match("(.\*zhihu.com/questions/(\d+))($|/).\*", url)

if match\_obj:

request\_url = match\_obj.group(1)

# question\_id = match\_obj.group(2)

# print(request\_url, question\_id, sep='\n')

# 在match\_obj和print(request\_url)这里添加断点进行debug.

#提取到知乎的questions了, 就要把它交给parse\_question函数提取出问题的详情.

yield scrapy.Request(request\_url, headers=self.headers, callback = self.parse\_question)

def parse\_question():

# 处理question页面, 从页面中提取出具体的question item

pass

数据的传递和保存都是通过item来进行的, 所以要设置具体的question的item. items.py

class ZhihuQuestionItem(scrapy.Item):

#知乎的问题item

zhihu\_id = scrapy.Field()

topics = scrapy.Field()

url = scrapy.Field()

title = scrapy.Field()

content = scrapy.Field()

# create\_time和update\_time无法通过页面解析到, 这里可以不用定义.

# create\_time = scrapy.Field()

# update\_time = scrapy.Field()

answer\_num = scrapy.Field()

comment\_num = scrapy.Field()

watch\_user\_num = scrapy.Field()

click\_num = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

class ZhihuAnswerItem(scrapy.Item):

#知乎的回答item

zhihu\_id = scrapy.Field()

url = scrapy.Field()

question\_id = scrapy.Field()

author\_id = scrapy.Field()

content = scrapy.Field()

praise\_num = scrapy.Field()

comment\_num = scrapy.Field()

create\_time = scrapy.Field()

update\_time = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

## 5-11 知乎spider爬虫逻辑的实现以及answer的提取

def parse\_question():

#处理question页面, 从页面中提取出具体的question item.

#question的有两个版本, 旧版和新版. 优先处理新版本.

#标题的css语法不同, 可以根据此来判断是新版还是旧版.

if "QuestionHeader-title" in request.text:

# 处理新版本, 使用item\_loader的方法.

from scrapy.loader import ItemLoader

from item import ZhihuQuestionItem, ZhihuAnswerItem

# 这里的item参数传递的是实例化后的对象, 不再是传递类方法, 和Request中的callback是不同的.

item\_loader = ItemLoader(item=ZhihuQuestionItem(), response=response)

item\_loader.add\_css("title", "h1.QuestionHeader-title::text")

# 对于content, 因为要提取其所有的内容, 所以不使用伪类选择器::text提取出它的文本

item\_loader.add\_css("content", ".QuestionHeader-detail")

item\_loader.add\_value("url", response.url)

#使用正则匹配来获取zhihu\_id, 即在parse函数中的question\_id. 也可以在parse函数中通过meta传递过来.

match\_obj = re.match(".\*zhihu.com/questions/(\d+)($|/).\*", url)

if match\_obj:

question\_id = int(match\_obj.group(1))

item\_loader.add\_value("zhihu\_id", question\_id)

# 提取出来的是类似 "64个回答" 这样的字符串, 后期再进行进一步提取

item\_loader.add\_css("answer\_id", ".List-headerText span::text")

item\_loader.add\_css("comment\_num", ".QuestionHeader-actions button::text")

#有的关注者是放在button中的, 有的是放在div中的, 所以不能写出来一个统一的提取规则, 可以用 "NumberBoard-value" 这个css样式进行提取. 实际提取出来的是一个数组, 包含着关注数和被浏览数, 可以通过取第一个值得到关注数, 取第二个值得到浏览数.

item\_loader.add\_css("watch\_user\_num", ".NumberBoard-value::text")

item\_loader.add\_css("topics", ".QuestionHeader-topics .Popover div::text")

question\_item = item\_loader.load\_item()

else:

# 处理旧版本页面的item提取

from scrapy.loader import ItemLoader

from item import ZhihuQuestionItem, ZhihuAnswerItem

# 这里的item参数传递的是实例化后的对象, 不再是传递类方法, 和Request中的callback是不同的.

item\_loader = ItemLoader(item = ZhihuQuestionItem(), headers = headers)

item\_loader.add\_css("title", "#zh-question-title h2 a::text")

item\_loader.add\_css("content", "#zh-question-detail")

item\_loader.add\_value("url", response.url)

#使用正则匹配来获取zhihu\_id, 即在parse函数中的question\_id. 也可以在parse函数中通过meta传递过来.

match\_obj = re.match(".\*zhihu.com/questions/(\d+)($|/).\*", url)

if match\_obj:

question\_id = int(match\_obj.group(1))

item\_loader.add\_value("zhihu\_id", question\_id)

item\_loader.add\_css("answer\_num", "#zh-question-answer-num::text")

#comment\_num在旧版本中点击查看所有回答的页面中提取信息.

item\_loader.add\_css("comment\_num", "#zh-question-meta-wrap a[name='addcomment']::text")

#实际提取出来的是一个数组, 包含着关注数和被浏览数, 可以通过取第一个值得到关注数, 取第二个值得到浏览数.

item\_loader.add\_css("watch\_user\_num", "#zh-question-side-header-wrap::text")

item\_loader.add\_css("topics", ".zm-tag-editor-lables a::text")

question\_item = item\_loader.load\_item()

yield question\_item

pass #添加断点进行调试.

进行调试, 发现提取旧版本页面时存在2个问题, 在question\_item的values中. watch\_user\_num没有取到值, 是一个空的列表. 也没有解析出title. 复制response的text到一个新的html页面中, 查看网页的结构有什么不同. 标题在这里变成了span标签, 之前是在a标签中. 所以对于同一个页面, title可能放在span标签中, 也可能放在a标签中. a标签可以点击, span标签无法点击. 就要把标题的提取规则改变为xpath selector, 因为xpath可以使用|或的方法进行提取.

# item\_loader.add\_css("title", "#zh-question-title h2 a::text")

item\_loader.add\_xpath("title", "//\*[id='zh-question-title']/h2/a/text()|//\*[id='zh-question-title']/h2/span/text()")

# item\_loader.add\_css("watch\_user\_num", "#zh-question-side-header-wrap::text")

item\_loader.add\_css("watch\_user\_num", "//\*[@id='zh-question-side-header-wrap']/text()|//\*[@class='zh-question-followers-sidebar']/div/a/strong/text()")

继续完善parse函数, 如果提取的url符合questions/xxx这种格式, 就交给parse\_question来处理. 如果提取到的不是这种格式的url, 则继续交给parse函数进行处理. scrapy是基于twisted异步IO框架来完成的, 异步IO的下一步操作基本上都只能通过callback回调函数来实现. 在scrapy中, 如果yield出去的是Request创建的实例, 就会把这个实例交给下载器进行下载, 并把对应的响应交给Request中指定的解析函数进行解析. 如果yield出去的是item, 这个item会传递到pipeline中进行处理.

def parse(self,request):

all\_urls = response.css("a::attr(href)").extract()

try:

#python2

import urlparse as parse

except:

#python3

from urllib import parse

all\_urls = [parse.urljoin(response.url, url) for url in all\_urls]

#使用python的filter函数过滤掉所有不符合规则的url.

all\_urls = filter(lambda x:True if x.startswith("https") else False, all\_urls)

for url in all\_urls:

match\_obj = re.match("(.\*zhihu.com/questions/(\d+))($|/).\*", url)

if match\_obj:

#如果提取到question相关的url, 则把它的响应交给parse\_question函数进行处理.

request\_url = match\_obj.group(1)

yield scrapy.Request(request\_url, headers=self.headers, callback = self.parse\_question)

else:

#如果不是questin的页面, 则进行进一步的跟踪. 继续使用parse函数对它的响应进行处理. 这里的parse函数可以省略不写, 默认就是调用parse函数进行处理的.

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

在parse\_question的页面中也会提取到很多的url, 也需要进行判断, 如果它是question相关的页面交由parse\_question进行提取, 如果不是question相关的页面则交由parse函数进行进一步的跟踪. 要把parse函数中相关的语句全部复制到parse\_question中.

还要提取出所有的问题的回答. 因为回答有api接口可以直接从json中提取数据, 就会方便很多. 因为这个api的地址不能从网页中提取到, 必须要手动向这个api地址发起一个请求, 才能得到json的数据.

在zhihuSpider中添加一个回答api接口的url地址, 同时替换掉其中的question\_id和offset的值.

class ZhihuSpider(scrapy.Spider):

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

# question的第一页answer的请求url

start\_answer\_url = "https://www.zhihu.com/api/v4/questions/{0}/answers?sort\_by=default&include=data%5B%2A%5D.is\_normal%2Cis\_collapsed%2Ccollapse\_reason%2Cis\_sticky%2Ccollapsed\_by%2Csuggest\_edit%2Ccomment\_count%2Ccan\_comment%2Ccontent%2Ceditable\_content%2Cvoteup\_count%2Creshipment\_settings%2Ccomment\_permission%2Cmark\_infos%2Ccreated\_time%2Cupdated\_time%2Crelationship.is\_authorized%2Cis\_author%2Cvoting%2Cis\_thanked%2Cis\_nothelp%2Cupvoted\_followees%3Bdata%5B%2A%5D.author.follower\_count%2Cbadge%5B%3F%28type%3Dbest\_answerer%29%5D.topics&limit={1}&offset={2}"

def parse(self,response):

pass

def start\_requests(self):

return [scrapy.Request('https://www.zhihu.com/#signin'), headers=self.headers callback=self.login]

def login(self,response):

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password

}

对回答的start\_answer\_url, 要发送一个请求才能获取到其中的数据. 可以在parse\_question处理完问题之后

def parse\_question():

xxx

question\_item = item\_loader.load\_item()

yield scrapy.Request(self.start\_answer\_url.format(question\_id, 20, 0), headers = self.headers, callback = self.parse\_answer)

yield question\_item

定义一个解析answer的函数

def parse\_answer(self, response):

#处理question的answer

import json

ans\_json = json.loads(response.text)

#is\_end判断是否是最后一页.

is\_end = ans\_json["paging"]["is\_end"]

#所有的回答数量, 没有太大的用处

# totals\_answer = ans\_json["paging"]["totals"]

#下一页的链接地址

next\_url = ans\_json["paging"]["next"]

#提取answer的具体字段

for answer in ans\_json["data"]:

answer\_item = ZhihuAnswerItem()

answer\_item["zhihu\_id"] = answer["id"]

answer\_item["url"] = answer["url"]

answer\_item["question\_id"] = answer["question"]["id"]

# 如果是匿名评论, 就没有author\_id

answer\_item["author\_id"] = answer["author"]["id"] if "id" in answer["author"] else None

# content的内容也可能不存在

answer\_item["content"] = answer["content"] if "content" in answer else None

answer\_item["praise\_num"] = answer["voteup\_count"]

answer\_item["comment\_num"] = answer["comment\_count"]

answer\_item["create\_time"] = answer["created\_time"]

answer\_item["update\_time"] = answer["updated\_time"]

import datetime

answer\_item["crawl\_time"] = datetime.datetime.now()

# 把提取出的数据交给管道文件进行处理.

yield answer\_item

if not is\_end:

#判断is\_end是否为False, 如果是False, 就调用下一页的url.

yield scrapy.Request(next\_url, headers = self.headers, callback = self.parse\_answer)

进行调试, 可以先把parse函数中最后一行的yield注释掉, 添加为pass, 以减少向服务器发送的请求量. 在yield answer\_item添加断点, 在parse\_question的 if "QuestionHeader-title" in request.text处添加断点. for answer in ans\_json处添加断点, 在if not is\_end添加断点.

完整的流程:

所有spider的入口在start\_requests, 启动爬虫的时候, 会首先从start\_requests中发送一个scrapy.Request到"https://www.zhihu.com/#signin", 然后使用callback调用login函数进行登陆.

login执行的是scrapy的模拟登陆, 最后发送一个scrapy.FormRequest进行登录, 把登录请求的响应使用check\_login函数进行解析.

把爬虫遍历start\_url, 从中读取地址发送请求的行为放在check\_login函数中了. 在发送这些请求之前, 实际上已经完成了scrapy的模拟登陆. 只有在登录成功后才会进行下一步的操作, 如果登录失败, 爬虫就会直接退出. 在check\_login函数中发送start\_url中的地址, 就开始了对这些url地址进行解析的过程.

之前的过程只是模拟登陆, 现在就开始调用parse函数进行数据的提取了.

parse函数中是深度优先的爬虫逻辑. 首先提取出这个页面所有的url, 使用filter过滤到不需要的url, 然后对所有的url进行循环遍历, 如果是question的url, 就对它进行跟踪, 把它交由parse\_question函数来提取问题的详情. parse\_question函数解析完成后会先发送该问题的回答的请求到parse\_anser函数提取该问题所有的回答. 然后再使用yield把question\_item交给管道文件进行处理. (如果yield的是item, 会交给管道进行处理, 如果yeild的是request, 会调用parse函数进行解析和下载. )

## 5-13 保存数据到mysql中

把item写入到mysql数据库中. 在第4章中介绍过如何启用pipeline通过twisted api把数据保存到mysql数据库中, 这里介绍如何通过pipeline把数据分别保存到两张数据表中.

class MysqlTwistedPipline(object):

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

self.dbpool = dbpool

@classmethod

def from\_settings(cls, settings):

dbparas = dict(

host = settings["MYSQL\_HOST"],

db = settings["MYSQL\_DBNAME"],

user = settings["MYSQL\_USERNAME"],

passwd = settings["MYSQL\_PASSWORD"],

charset = "utf8",

cursorclass = MySQLdb.cursors.DictCursor,

use\_unicode = True

)

dbpool = adbapi.ConnectionPool("MySQLdb", \*\*dbparas)

return cls(dbpool)

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

query = self.dbpool.runInteraction(self.do\_insert, item)

query.addErrback(self.handle\_error, item, spider)

def handle\_error(self, failure, item, spider)

#处理异步插入的异常

print(failure)

def do\_insert(self, cursor, item):

# 执行具体的插入

insert\_sql = '''

INSERT INTO

ArticleSpider(title, create\_time, url, url\_object\_id, front\_image\_url, front\_image\_path, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["front\_image\_path"], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

通过其中的静态方法from\_settings方法从setings中读取mysql的配置, 实例化一个mysql pipeline的实例, 在这个实例中调用self.dbpool这个容器, 使do\_insert这个方法在容器中进行运行. do\_insert才是真正插入数据的逻辑. 在do\_insert中首先构建了插入数据的sql语句, 传入item, 通过item提取出sql语句的参数值, 交由cursor来执行.

在process\_item中还添加了一个addErrback方法来处理do\_insert插入sql时的异常. do\_insert和handle\_error函数都是异步的. handle\_error语句在开发和上线后都很重要, 把异常打印到日志中, 就可以分析出插入sql时的异常. 会成为找到错误的一个入口.

知乎可以提取到两个item, 一个是question的item, 一个是answer的item. 可以再定义Pipeline, 使得一个Pipeline处理一个item, 或者使一个网站有一个对应的Pipeline.使用这个方法, 一个网站对应一个Pipeline, 如果有上百个网站, 会向mysql数据库发起上百个连接, 会大大加大服务器的压力.

所以就要修改MysqlTwistedPipline, 在其中处理所有的item, 包括jobbole网站的article\_item和知乎的question\_item和answer\_item.

在实际的开发中, 可能会把不同的爬虫爬取的数据保存到不同的数据库中, 可以对每个不同的数据库建立一个Pipeline进行处理.

其中只有在do\_insert中才是需要改动的部分. 可以根据不同的item, 构建不同的sql语句插入到mysql中不同的数据表中.

def do\_insert(self, cursor, item):

# 执行具体的插入

# item.\_\_class\_\_.\_\_name\_\_取到当前的item的class的名字

if item.\_\_class\_\_.\_\_name\_\_ == "JobBoleArticleItem":

insert\_sql = '''

INSERT INTO

ArticleSpider(title, create\_time, url, url\_object\_id, front\_image\_url, front\_image\_path, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["front\_image\_path"], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

但这种方法也不妥, 因为在程序中已经把JobBoleArticleItem的名字写成固定的了, 如果后期要进行修改的话, 要修改很多处才行.

在Django中, 对数据库的插入是使用models的方法来完成的. models中定义一个class之后可以使用这个class进行save的操作. 这样就不使用do\_insert中具体的sql语句来进行数据库的插入操作了. 现在就使用django的设计思想, 把do\_insert中变化的部分拿到item的定义中来. 把构建insert\_sql语句以及sql语句对应的填充值封装成一个get\_insert\_sql的函数, 放在具体的item的定义中. 在do\_insert中就可以调用get\_insert\_sql这个方法了.

修改items.py中的class JobBoleArticleItem

class JobBoleArticleItem(scrapy.Item):

title = scrapy.Field()

create\_time = scrapy.Field(

# 该传入的字段值要批量处理的函数

input\_processor=MapCompose(remove\_dote, date\_convert),

)

url = scrapy.Field()

url\_object\_id = scrapy.Field()

front\_image\_url = scrapy.Field(

output\_processor = MapCompose(return\_value)

)

front\_image\_path = scrapy.Field()

praise\_num = scrapy.Field(

input\_processor = MapCompose(get\_praise\_num)

)

fav\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

comment\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

tags = scrapy.Field(

input\_processor=MapCompose(remove\_comment\_tag),

output\_processor = Join(',')

)

content = scrapy.Field()

def get\_insert\_sql(self):

insert\_sql = '''

INSERT INTO

ArticleSpider(title, create\_time, url, url\_object\_id, front\_image\_url, front\_image\_path, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

# cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["front\_image\_path"], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

# 因为把代码复制到items中了, 所以这里就不再是item["title"], item["create\_time"]等了, 而是self["title"], self["create\_time"]等

params = (self["title"], self["create\_time"], self["url"], self["url\_object\_id"], self["front\_image\_url"][0], self["front\_image\_path"], self["comment\_num"], self["praise\_num"], self["fav\_num"], self["tags"], self["content"])

return insert\_sql, params

修改pipelines.py中的do\_insert方法

def do\_insert(self, cursor, item):

# 执行具体的插入

insert\_sql, params = item.get\_insert\_sql()

cursor.execute(insert\_sql, params)

知乎的question item和answer item也都用上面的方法来定义. 函数名也写为get\_insert\_sql, 参数也为params, 就可以统一调用do\_insert这个方法了.

class ZhihuQuestionItem(scrapy.Item):

#知乎的问题item

zhihu\_id = scrapy.Field()

topics = scrapy.Field()

url = scrapy.Field()

title = scrapy.Field()

content = scrapy.Field()

# create\_time和update\_time无法通过页面解析到, 这里可以不用定义.

# create\_time = scrapy.Field()

# update\_time = scrapy.Field()

answer\_num = scrapy.Field()

comment\_num = scrapy.Field()

watch\_user\_num = scrapy.Field()

click\_num = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

def get\_insert\_sql(self):

# 插入知乎question表的sql语句

# 因为create\_time和update\_time在网页中无法取得, 所以这里也不能插入

insert\_sql = '''

INSERT INTO

zhihu\_question(zhihu\_id, topics, url, title, content, answer\_num, comment\_num, watch\_user\_num, click\_num, crawl\_time)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

# 是使用item\_loader的方法实现的, 得到的是列表, 要先把列表中的值取出来.

# 由于列表中只有1个元素, 所以这里可以提取出第0个元素, 也可以使用"".join()方法得到字符串.

# zhihu\_id = "".join(self["zhihu\_id"])

# 在zhihu.py中正则表达式中已经把zhihu\_id转换为int格式了, 所以这里直接提取第0个元素即可.

zhihu\_id = self["zhihu\_id"][0]

topics = ",".join(self["topics"])

# url = "".join(self["url"])

url = self["url"][0]

title = "".join(self["title"])

content = "".join(self["content"])

# 要把提出的answer\_num中的数字进行提取. 调用get\_num来进行处理. 因为这个函数使用的非常多, 可以把get\_num提取出来, 放在utils/common.py中.

"""

修改common.py

import re

def extract\_num(text):

# 从字符串中提取出数字

match\_re = re.match('.\*?(\d+).\*',value)

if match\_re:

x\_num = int(match\_re.group(1))

else:

x\_num = 0

return x\_num

"""

form ArticleSpider.utils.common import extract\_num

answer\_num = extract\_num("".join(self["answer\_num"]))

comment\_num = extract\_num("".join(self["comment\_num"]))

watch\_user\_num = extract\_num("".join(self["watch\_user\_num"]))

click\_num = extract\_num("".join(self["click\_num"]))

"""

使用sql语句插入time类型的字段时要转化为string格式的. 需要在settings中设置把它格式化成什么类型的字符串.

SQL\_DATETIME\_FORMAT = "%Y-%m-%d %H:%M:%S"

SQL\_DATE\_FORMAT = "%Y-%m-%d"

"""

from settings import SQL\_DATETIME\_FORMAT, SQL\_DATE\_FORMAT

# 需要把时间转换为字符串, 才能保存到mysql中. 使用strftime来把时间转换为字符串. 传递的参数可以在settings.py中定义, SQL\_DATETIME\_FORMAT = "%Y-%m-%d %H:%M:%S", SQL\_DATE\_FORMAT = "%Y-%m-%d"

crawl\_time = datetime.datetime.now().strftime(SQL\_DATETIME\_FORMAT)

# params中的顺序必须要与上面insert into中字段的顺序保持一致.

params = (zhihu\_id, topics, url, title, content, answer\_num, comment\_num, watch\_user\_num, click\_num, crawl\_time)

return insert\_sql, params

修改pipelines.py中的 class MysqlTwistedPipline(object) 中的 def do\_insert(self, cursor, item), 使用cursor.execute执行insert\_sql, params.

def do\_insert(self, cursor, item):

#执行具体的插入

#根据不同的item 构建不同的sql语句并插入到mysql中

insert\_sql, params = item.get\_insert\_sql()

cursor.execute(insert\_sql, params)

打断点进行调试, 一定要学会使用调试的方式对代码中的错误进行调试和分析.

如果question\_item中取出来了两个值, 则说明这是新版本中的页面.

在pipelines.py中的do\_insert处插入两个断点, debug, 查看是否插入了数据.

接下来完成answer的item的数据插入.

class ZhihuAnswerItem(scrapy.Item):

#知乎的回答item

zhihu\_id = scrapy.Field()

url = scrapy.Field()

question\_id = scrapy.Field()

author\_id = scrapy.Field()

content = scrapy.Field()

praise\_num = scrapy.Field()

comment\_num = scrapy.Field()

create\_time = scrapy.Field()

update\_time = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

def get\_insert\_sql(self):

insert\_sql = '''

INSERT INTO

zhihu\_answer(zhihu\_id, url, question\_id, author\_id, content, praise\_num, comment\_num, create\_time, update\_time, crawl\_time)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

# 因为create\_time和update\_time从json api中取出来是int类型的, 要进行单独的处理. 把int类型转换为datetime类型的. 再把datetime类型的转换为字符串.

# 从json是提取出的create\_time和update\_time是int类型的字段, 即linux的时间戳, 需要使用datetime.datetime.fromtimestamp()把int类型转换为datetime类型的. 再使用.strftime(SQL\_DATETIME\_FORMAT)把它转换为字符串类型的.

create\_time = datetime.datetime.fromtimestamp(self["create\_time"]).strftime(SQL\_DATETIME\_FORMAT)

update\_time = datetime.datetime.fromtimestamp(self["update\_time"]).strftime(SQL\_DATETIME\_FORMAT)

# 因为answer是直接从api中提取出来的数据, 这里就不需要像question中的那样进行处理了, 直接提取出它们的值就可以了.

params = (self["zhihu\_id"], self["url"], self["question\_id"], self["author\_id"], self["content"], self["praise\_num"], self["comment\_num"], create\_time, update\_time, self["crawl\_time"].strftime(SQL\_DATETIME\_FORMAT), )

return insert\_sql, params

因为在前端页面中帖子的发表时间和更新时间都是日期型的, 所以在定义zhihu\_answer表结构时, create\_time, update\_time 都是定义为date类型的, 但在api接口中获取的却是linux时间戳, 可以转换为datetime类型的时间, 所以要修改数据表的结构, 把create\_time和update\_time改为datetime类型的.

然后进行调试, 在调试时可以运用一些技巧. zhihu.py中已经写好了爬取的逻辑. 只要一开始运行, 就会不停的向服务器发送请求, 如果在调试过程中不进行Request的控制, 程序遇到一个Request就向服务器发送一个请求, 因为scrapy是基于异步处理的. 在pipelines.py中打上了断点, 可能还没有执行到这个断点处, 另外一个线程就又发送过来了, 会一直有不同的回调函数发送数据到pipeline中进行处理. 调试起来可能就会非常麻烦. 可以在zhihu.py中进行修改, for url in all\_urls的yield Request下面加一个break, 只要发送了一个请求, 就直接跳出循环, else中的yield Request也注释掉, 改为pass, 这样就只发送了一个请求. 同时, 因为要调试的是answer的item, question的item就不需要发送到pipeline中了. 可以把parse\_question中的yield question\_item注释掉.

在zhihu.py中ZhihuSpider.parse\_answer中的yield answer\_item上打上断点, 在pipelines.MysqlTwistedPipline.process\_item中的query=self.dbpool上打上断点, handle\_error的print(failure)上打上断点, do insert中的insert\_sql, params, cursor.execute上打上断点, zhihu.ZhihuSpider.parse\_answer中的 yield answer\_item处加上断点, 进行逐步调试, 程序就沿着以上的断点顺序循环进行. 进行几轮调试, 查看数据库中answer表中是否插入了数据.

调试进行过一次之后, 如果再次进行调试, 会报错. 在failure中的value中可以看到Duplicate entry for key 1的错误. 也就是主键冲突. 因为在之前的调试中已经插入了相同的数据. 但有可能在第二次爬取同样一个数据的时候, 问答已经进行了更新. update\_time, 点赞数, 收藏数等都可能已经进行了更新. 所以一个简单的向sql中插入数据就不能满足要求了. 要实现如果数据不存在, 就插入到数据库中, 如果数据已经存在, 就更新其中的字段. create\_time不用修改, update\_time可能会被修改. 把INSERT INTO语句改为如下. 只更新其中的某些字段即可.

INSERT INTO zhihu\_answer(zhihu\_id, url, question\_id, author\_id, content, praise\_num, comment\_num, create\_time, update\_time, crawl\_time)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

ON DUPLICATE KEY UPDATE content=VALUES(content), comment\_num=VALUES(comment\_num), praise\_num=VALUES(praise\_num), comment\_num=VALUES(comment\_num), update\_time=VALUES(update\_time)

jobole和question的INSERT INTO语句也要做类似的修改.

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

**def** get\_insert\_sql(self):

insert\_sql = """

insert into jobbole\_article(title, url, create\_date, fav\_nums)

VALUES (**%s**, **%s**, **%s**, **%s**) ON DUPLICATE KEY UPDATE content=VALUES(fav\_nums)

"""

params = (self["title"], self["url"], self["create\_date"], self["fav\_nums"])

**return** insert\_sql, params

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

**def** get\_insert\_sql(self):

*#插入知乎question表的sql语句*

insert\_sql = """

insert into zhihu\_question(zhihu\_id, topics, url, title, content, answer\_num, comments\_num,

watch\_user\_num, click\_num, crawl\_time

)

VALUES (**%s**, **%s**, **%s**, **%s**, **%s**, **%s**, **%s**, **%s**, **%s**, **%s**)

ON DUPLICATE KEY UPDATE content=VALUES(content), answer\_num=VALUES(answer\_num), comments\_num=VALUES(comments\_num),

watch\_user\_num=VALUES(watch\_user\_num), click\_num=VALUES(click\_num)

"""

以上就完成了从模拟登陆到提取数据保存的所有工作.

但在爬取过快时, 服务器很快就会返回一个403的错误. 这也是反爬的一种机制, 同一个USER-AGENT在同一个IP下访问过于频繁的话服务器就会启动反爬机制. 返回错误. 可以随机更换USER-AGENT来解决.

在多次登陆失败时, 需要在登陆页面输入验证吗才能进行登陆.

## 5-16 (补充小节)知乎验证码登录

知乎验证码登陆

在使用zhihu\_login\_requests.py登陆的时候, 可能会返回登陆失败的错误. 因为一直没有输入验证码的语句. 这里使用人工识别的方式来输入验证码.

在登陆页面输入错误的验证吗, 查看验证码提交时post的内容.

POST的参数中有一个叫captcha的字段, 就是我们输入的验证码的值.

然后点击验证码进行刷新, 查看验证码请求的地址:

https://www.zhihu.com/captcha.gif?r=1496921404092&type=login&lang=cn

修改zhihu\_login\_requests.py

定义一个获取验证码的函数

def get\_captcha():

import time

t = str(int(time.time()\*1000))

captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login&lang=en".format(t)

# 构建一个请求, 请求验证码

t = session.get(captcha\_url, headers = headers)

with open("captcha.jpg", "wb") as f:

f.write(t.content)

f.close()

pass #在pass处添加断点, 进行调试, 看是否能保存验证码的图片.

get\_captcha()

可以使用PIL的一个类打开这个图片. pip install pillow

from PIL import Image

def get\_captcha():

import time

t = str(int(time.time()\*1000))

captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login&lang=en".format(t)

# 构建一个请求, 请求验证码. 这里只能使用session来发送请求, 不能使用requests, 使用requests得到的验证码输入后也不会被服务器接收, 还是会被认为是错误的验证码. 一次session就是一次与服务器的会话, 使用session向服务器发送请求, 会把response中的cookie保存下来, 下次请求时会一起发送到服务器. 可以debug查看session的内容, 其中就有cookie的字段, cookie中还包含有xsrf的值, 有了这个xsrf, 服务器才会认为第二次的请求是合法的. 否则即使输入了正确的验证吗, 服务器也认为是非法的. 为什么使用浏览器能够正常登录, 是因为浏览器保存了上一次响应的cookies值, 在下一次请求时, 会把cookies发送给服务器, cookies中包含有xsrf的值, 这样就能把服务器与本地的xsrf值匹配起来. 使用requests.get就不能正确实现二者的关联.

t = session.get(captcha\_url, headers = headers)

with open("captcha.jpg", "wb") as f:

f.write(t.content)

f.close()

try:

im = Image.open("captcha.jpg")

im.show()

im.close()

except:

pass

captcha = input("输入验证码\ninput-->")

return captcha

想要对哪个变量的值进行调试, 就在它的下一行加上断点进行调试, 这样程序会在下一行中断, 会很方便的对变量进行调试. 想要查看sesseion中的内容, 就在session的下一行中打上断点.

再次进行尝试, 就会自动打开验证码图片.

在def zhihu\_login中的post\_data中添加一个captcha的字段.

post\_data = {

"\_xsrf": get\_xsrf(),

"phone\_num": account,

"password": password,

"captcha": get\_captcha(),

}

总结: 首先需要分析出验证码的参数设置. 分析出验证码请求的url, 传递一个与当前时间有关的随机的字符串, 把验证码保存到图片中, 打开文件, 提示输入验证码, 把输入的验证码传入post\_data中. 这样就完成了验证码的输入.

因为scrapy中没有session的方法, 如果在scrapy中提交验证码呢?

在zhihu.py中, 在login中的post\_data中加一个captcha的字段,

def login(self, response):

response\_text = response.text

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL)

xsrf = ''

if match\_obj:

xsrf = (match\_obj.group(1))

if xsrf:

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": xsrf,

"phone\_num": "18782902568",

"password": "admin123",

"captcha": ""

}

import time

t = str(int(time.time()\*1000))

captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login&lang=en".format(t)

yield scrapy.Request(captcha\_url, headers = self.headers, meta={"post\_data":post\_data} callback=self.login\_after\_captcha)

login中的xsrf是yield发送的scrapy.Request请求返回的结果. 和Request请求对象没有关系, 所以请求验证码的时候无法用Request请求对象来实现, 也无法使用sessions来处理, 因为scrapy的请求是基于异步框架的, 使用session发送的异步请求也会被认为是不同的请求. 如何做才能使请求的验证码与当前的Request是在一个session中的呢?

在知乎登陆之后再次发送scrapy.Request的请求中会自动带的有session和cookie, 但由于是异步的请求, 就无法通过scrapy.Request和response再去构造一个带有相同cookie的session. 在login中yield一个请求验证码图片的scrapy.Request, 定义一个新函数login\_after\_captcha, 并在scrapy.Request中使用callback来调用. 这样, 就能够保证yield出去的callback=self.login\_after\_captcha函数和login函数是具有相同的cookie的. 同时通过meta把post\_data中的参数传递到response中去. 在获取到图片验证码之后, 把验证码中的值通过input输入放到post\_data中来.

在login\_after\_captcha函数中, 把登陆的功能放在获取验证码之后, 获取并输入验证码后, 把验证码的值放在post\_data中一起发送到服务器, 这样, 既保证了与login函数使用的是同样的cookie, 又能输入验证码.

当然, 在login函数中也可以不使用yield scrapy.Request的这种方法, 可以先在login的response中取到cookies的值, 把它设置到scrapy.Request的post\_data中去, 这样也能保证这两个函数中的Request是使用的相同的cookies了.

def start\_requests(self):

return [scrapy.Request('https://www.zhihu.com/#signin', headers=self.headers, callback=self.login)]

def login(self, response):

response\_text = response.text

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL)

xsrf = ''

if match\_obj:

xsrf = (match\_obj.group(1))

if xsrf:

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": xsrf,

"phone\_num": "",

"password": "",

"captcha": ""

}

import time

t = str(int(time.time() \* 1000))

captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login".format(t)

yield scrapy.Request(captcha\_url, headers=self.headers, meta={"post\_data":post\_data}, callback=self.login\_after\_captcha)

# 把原来在login方法中完成的登录放在了获取验证码之后再登录.

def login\_after\_captcha(self, response):

# 这里的response是图片的response

with open("captcha.jpg", "wb") as f:

# 因为这里的response是图片的response, 所以向文件中写入的应该是response的body部分.

f.write(response.body)

f.close()

from PIL import Image

try:

im = Image.open("captcha.jpg")

im.show()

im.close()

except:

pass

captcha = input("输入验证码\ninput-->")

post\_data = response.meta.get("post\_data", {})

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data["captcha"] = captcha

return [scrapy.FormRequest(

url = post\_url,

formdata = post\_data,

headers=self.headers,

callback=self.check\_login

)]

def check\_login(self, response):

#验证服务器的返回数据判断是否成功

text\_json = json.loads(response.text)

if "msg" in text\_json and text\_json["msg"] == "登录成功":

for url in self.start\_urls:

yield scrapy.Request(url, dont\_filter=True, headers=self.headers)

在login方法中的最后一行yield 和 login\_after\_captcha的第一行 with open 处, return [scrapy.FormRequest 处添加断点, 进行调试. 查看login\_after\_captch 中post\_data的值. 在check\_login处的text\_json处打上断点, 查看response的值. 单步调试, 查看text\_json中msg的值. 此时就能登陆成功.

## selenium模拟登录知乎

使用selenium模拟登录, 把cookies值放在scrapy的Request中. 分析网站的登录是一件很费时的事情, 如果能使用selenium很简单的完成网站的登录, 就能节省大量的时间和精力.

可以注册多个账号, 使用selenium登录后保存cookies, 维护一个cookies池.

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

**import** **re**

**import** **json**

**import** **datetime**

**try**:

**import** **urlparse** **as** **parse**

**except**:

**from** **urllib** **import** parse

**import** **scrapy**

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

**from** **items** **import** ZhihuQuestionItem, ZhihuAnswerItem

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

name = "zhihu\_sel"

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

start\_urls = ['https://www.zhihu.com/']

*#question的第一页answer的请求url*

start\_answer\_url = "https://www.zhihu.com/api/v4/questions/{0}/answers?sort\_by=default&include=data%5B%2A%5D.is\_normal%2Cis\_sticky%2Ccollapsed\_by%2Csuggest\_edit%2Ccomment\_count%2Ccollapsed\_counts%2Creviewing\_comments\_count%2Ccan\_comment%2Ccontent%2Ceditable\_content%2Cvoteup\_count%2Creshipment\_settings%2Ccomment\_permission%2Cmark\_infos%2Ccreated\_time%2Cupdated\_time%2Crelationship.is\_author%2Cvoting%2Cis\_thanked%2Cis\_nothelp%2Cupvoted\_followees%3Bdata%5B%2A%5D.author.is\_blocking%2Cis\_blocked%2Cis\_followed%2Cvoteup\_count%2Cmessage\_thread\_token%2Cbadge%5B**%3F**%28type%3Dbest\_answerer**%29%**5D.topics&limit={1}&offset={2}"

headers = {

"HOST": "www.zhihu.com",

"Referer": "https://www.zhizhu.com",

'User-Agent': "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:51.0) Gecko/20100101 Firefox/51.0"

}

custom\_settings = {

"COOKIES\_ENABLED": True

}

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

*"""*

*提取出html页面中的所有url 并跟踪这些url进行进一步爬取*

*如果提取的url中格式为 /question/xxx 就下载之后直接进入解析函数*

*"""*

**pass**

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

*#处理question页面， 从页面中提取出具体的question item*

**pass**

**def** parse\_answer(self, reponse):

**pass**

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

**from** **selenium** **import** webdriver

browser = webdriver.Chrome(executable\_path="E:/tmp/chromedriver.exe")

browser.get("https://www.zhihu.com/signin")

# 用户名

browser.find\_element\_by\_css\_selector(".SignFlow-accountInput.Input-wrapper input").send\_keys(

"xxx")

# 密码

browser.find\_element\_by\_css\_selector(".SignFlow-password input").send\_keys(

"xxx")

browser.find\_element\_by\_css\_selector(

".Button.SignFlow-submitButton").click()

**import** **time**

# 知乎登录成功后会跳转到首页, 此时就已经是使用登录状态了.

time.sleep(10)

Cookies = browser.get\_cookies()

**print**(Cookies)

cookie\_dict = {}

**import** **pickle**

**for** cookie **in** Cookies:

*# 把cookies写入文件*

*#此处大家修改一下自己文件的所在路径*

f = open('./ArticleSpider/cookies/zhihu/' + cookie['name'] + '.zhihu', 'wb')

pickle.dump(cookie, f)

f.close()

# 从cookie中取出来有用的cookie信息, 放入到cookie\_dict中.

cookie\_dict[cookie['name']] = cookie['value']

browser.close()

# 把cookies写入到文件中之后, 再访问知乎的其它页面时可以直接从文件中读取保存的cookies, 发送带有cookies的请求即可. 只要settings.py中cookies\_enable=True, 只要在一个Request中发送了cookies, 在以后再发送Request时即使不指定cookies=cookie\_dict, 也会默认携带cookies.

**return** [scrapy.Request(url=self.start\_urls[0], dont\_filter=True, cookies=cookie\_dict)]

修改settings.py, 启用cookies

COOKIES\_ENABLE = True

## 5-17 知乎倒立文字的识别

<https://github.com/muchrooms/zheye>

下载包, 解压, 进入到虚拟环境中,

pip install -r requirements.txt

pip install -i <https://pypi.douban.com/simple> Keras==2.0.1

如果哪个包安装出错, 就去

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

上下载对应的包, 进行安装.

把下载的zheye目录复制到项目的根目录下

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

安装numpy+mkl

在tools目录下新建 zheye\_test.py进行测试

下载一张知乎的中文验证码, 修改为captcha\_cn.gif, 放在zheye\_test.py同目录下

from zheye import zheye

z = zheye()

positions = z.Recognize('captcha\_cn.gif')

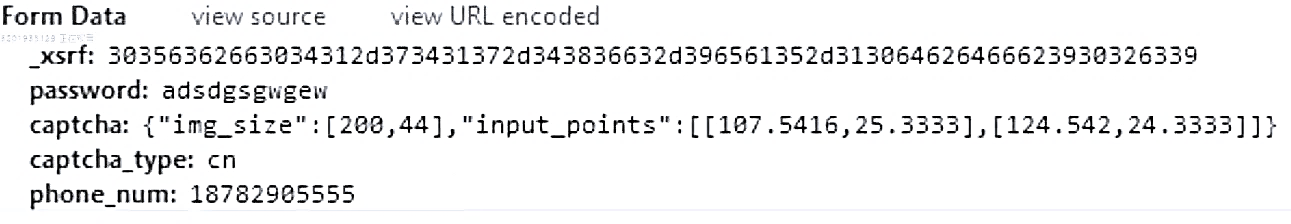
print(positions)



[(y, x), (y, x)]

列表中是两个元组, 元组中的坐标为(y, x), 并且第2个元组中的x坐标小于第1个元组中的x坐标, 即得到的坐标并不是根据倒立文字的先后进行排列的, 而是根据y轴的坐标排序的, 需要在后期进行处理.

查看登录时传递的内容.



其中的captcha为字符串格式的字典, {"img\_size": [200, 44], "input\_points": [[107.5416, 25.3333], [124.542, 24.3333]]}

注意传递的图片的大小是200, 44, 而获取到的图片的大小是400, 88, 所以使用zheye库获取的坐标除以2才得到真正传递的坐标. 坐标的格式为 [[x, y], [x, y]], 由于使用zheye得到的坐标格式为[(y, x), (y, x)], 所以要进行坐标格式的转换.

使用requests库识别中文倒立文字并进行登录

**from** **zheye** **import** zheye

z = zheye()

**import** **requests**

**import** **shutil**

**import** **time**

**import** **re**

h = {

'Accept': 'text/html, application/xhtml+xml, image/jxr, \*/\*',

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

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.10240'

}

lp = []

lpp = []

*# 构造session, 使用session登录知乎*

s = requests.session()

web\_data = s.get('http://www.zhihu.com/', headers=h).text

*# 提取出xsrf的值*

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', web\_data, re.DOTALL)

xsrf = ''

**if** match\_obj:

xsrf = (match\_obj.group(1))

*# 构造时间戳*

randomNum = str(int(time.time() \* 1000))

*# 请求*

r = s.get('https://www.zhihu.com/captcha.gif?r={}&type=login&lang=cn'.format(randomNum), headers=h, stream=True)

**if** r.status\_code == 200:

*# 把验证码保存到本地*

**with** open('pic\_captcha.gif', 'wb') **as** f:

r.raw.decode\_content = True

shutil.copyfileobj(r.raw, f)

*# 使用zheye来识别验证码*

positions = z.Recognize('pic\_captcha.gif')

**print**(positions)

captcha = {}

pos = positions

captcha['input\_points'] = []

*# pos的结构为[(y, x), (y, x)], 所以poss为元组类型, 分别取出来元组中的第1个元素和第2个元素, 即y, x的坐标, 除以2, 再格式化为1位小数的形式, 然后转换为float型, 再使用append添加到tmp列表中, 所以tmp是以列表形式保存的每个字的[y,x]的坐标. 再把tmp添加到captcha['input\_points']中, 得到[[y,x],[y,x]]的形式*

*# 输入错误的验证码, 查看提交的post请求的内容, 图片大小为保存的验证码的一半, 所以这里要除以2, 并且是字典嵌套字典的格式.*

*# pos = ((107.5416, 25.3333), (124.542, 24.3333))*

*# 把元组转换为列表*

**for** poss **in** pos:

tmp = []

tmp.append(float(format(poss[0] / 2, '0.2f')))

tmp.append(float(format(poss[1] / 2, '0.2f')))

captcha['input\_points'].append(tmp)

*# print(tmp)*

*# print(captcha)*

tmp = []

*# print str(captcha)*

params = {

'\_xsrf': xsrf,

'password': 'admin321',

*# 把传递的captcha转换为字符串. 默认有2个倒立的文字, 所以有4个坐标. 先使用%.2f进行格式化占位, 再进行填充*

'captcha': '{"img\_size": [200, 44], "input\_points": [[**%.2f**, **%f**], [**%.2f**, **%f**]]}' % (pos[0][1] / 2, pos[0][0] / 2, pos[1][1] / 2, pos[1][0] / 2),

'captcha\_type': 'cn',

'phone\_num': '2222222222'

}

**print** (params)

r = s.post('https://www.zhihu.com/login/phone\_num', headers=h, params=params)

**import** **json**

re\_text = json.loads(r.text)

**print** (r.text)

把倒立文字识别的程序集成到scrapy中的zhihu.py中的login中.

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

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

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

response\_text = response.text

match\_obj = re.match('.\*name="\_xsrf" value="(.\*?)"', response\_text, re.DOTALL)

xsrf = ''

**if** match\_obj:

xsrf = (match\_obj.group(1))

**if** xsrf:

post\_url = "https://www.zhihu.com/login/phone\_num"

post\_data = {

"\_xsrf": xsrf,

"phone\_num": "",

"password": "",

"captcha": ""

}

**import** **time**

t = str(int(time.time() \* 1000))

*# captcha\_url = "https://www.zhihu.com/captcha.gif?r={0}&type=login".format(t)*

*# yield scrapy.Request(captcha\_url, headers=self.headers, meta={"post\_data":post\_data}, callback=self.login\_after\_captcha)*

captcha\_url\_cn = "https://www.zhihu.com/captcha.gif?r={0}&type=login&lang=cn".format(t)

**yield** scrapy.Request(captcha\_url\_cn, headers=self.headers, meta={"post\_data":post\_data}, callback=self.login\_after\_captcha\_cn)

*# 知乎倒立汉字验证识别登录*

**def** login\_after\_captcha\_cn(self, response):

**with** open("captcha.jpg", "wb") **as** f:

f.write(response.body)

f.close()

**from** **zheye** **import** zheye

z = zheye()

positions = z.Recognize('captcha.gif')

post\_data = response.meta.get("post\_data", {})

*# 生成的positions中, 如果第2个元素的x坐标小到第个元素的x坐标时, 需要把前后两个元素进行调换*

pos\_arr = []

*# 在实际中, 倒立文字可能有1个, 也可能有2个, 要进行判断*

**if** len(positions) == 2:

**if** positions[0][1] > positions[1][1]:

pos\_arr.append([positions[1][1], positions[1][0]])

pos\_arr.append([positions[0][1], positions[0][0]])

**else**:

pos\_arr.append([positions[0][1], positions[0][0]])

pos\_arr.append([positions[1][1], positions[1][0]])

post\_data["captcha"] = '{"img\_size": [200, 44], "input\_points": [[**%.2f**, **%f**], [**%.2f**, **%f**]]}' % (pos\_arr[0][0] / 2, pos\_arr[0][1] / 2, pos\_arr[1][0] / 2, pos\_arr[1][1] / 2)

*# 如果倒立的文字只有1个*

**elif** len(positions) == 1:

pos\_arr.append([positions[0][1], positions[0][0]])

post\_data["captcha"] = '{"img\_size": [200, 44], "input\_points": [[**%.2f**, **%f**]}' % (pos\_arr[0][0] / 2, pos\_arr[0][1] / 2)

post\_data["captcha\_type"] = "cn"

post\_url = "https://www.zhihu.com/login/phone\_num"

**return** [scrapy.FormRequest(

url=post\_url,

formdata=post\_data,

headers=self.headers,

callback=self.check\_login

)]

在def login\_after\_captcha\_cn(self, response)中的post\_data["captcha\_type"] = "cn" 打上断点, 在check\_login中的text\_json中打上断点, 进行调试, 看能否成功登录.

如果一直登录不成功, 可以在settings.py中设置cookies\_enabled = True, 再进行尝试.

在login中需要再加上进一步的判断, 从登录页提取出captcha\_url, 如果没有验证码, 就发送一个不带验证码的直接登录请求, 如果使用的是英文的验证码, 就发送带英文验证码的登录请求, 如果使用的是中文的验证码, 就发送一个带有中文验证码的登录请求.

# 第6章 通过CrawlSpider对lagou网进行整站爬取

工具cmder

把cmder所在的目录添加到系统环境变量的path中.

## 6-1 数据表结构设计 (15:33)

了解需要爬取的数据的字段

在拉勾网中任意搜索一个职位, 打开职位详情页查看有哪些字段.

招聘公司, 在右边也能看到招聘的公司.

职位的title

职位薪资 薪资有一个最低薪资和最高薪资, 可以对字段进行进一步的提取.

工作地点

工作经验 有一个最低年限和最高年限的要求. 可以对字段进行进一步的处理, 如果是没有要求, 就设为最低为0, 最高为100. 如果有类似1-3年的要求, 就设置为最低为1, 最高为3, 如果有至少3年的要求, 就设置为最低为3, 最高为100. 为之后搜索时的高级查询作准备.

学历

工作类别

职位tag

发布时间 有1天前, 2小时前, 或者是具体的日期, 可以把发布时间统一转换为日期的格式.

职位诱惑

职位描述

工作地址

url

https://www.lagou.com/jobs/3036982.html

拉勾网的url中有一串数字, 可以把这个数字提取出来作为id并设置为主键. 这种方法与知乎中的方法类似. 现在不使用这种方法, 因为如果后期拉勾网url配置发生了变化, 使用了字符串来代替数字, 这种使用数字的配置就要进行修改. 这里使用jobbole中的方法, 获取url后使用url的md5值作为它的主键.

设计数据表.

新建一个lagou\_job的数据表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字段名 | 类型 | 长度 | 小数点 | 不是null | 主键 |
| url | varchar | 300 | 0 | T |  |
| url\_object\_id | varchar | 50 | 0 | T | 1 |
| title | varchar | 100 | 0 | T |  |
| salary | varchar | 20 | 0 | F |  |
| job\_city | varchar | 10 | 0 | F |  |
| work\_years | varchar | 100 | 0 | F |  |
| degree\_need | varchar | 30 | 0 | F |  |
| job\_type | varchar | 20 | 0 | F |  |
| publish\_time | varchar | 20 | 0 | T |  |
| tags | varchar | 100 | 0 | F |  |
| job\_advantage | varchar | 1000 | 0 | F |  |
| job\_desc | longtext | 0 | 0 | T |  |
| job\_addr | varchar | 50 | 0 | F |  |
| company\_url | varchar | 300 | 0 | F |  |
| company\_name | varchar | 100 | 0 | F |  |
| crawl\_time | datetime | 0 | 0 | T |  |
| crawl\_update\_time | datetime | 0 | 0 | F |  |

## 6-2 CrawlSpider源码分析-新建CrawlSpider与settings配置

之前是通过scrapy genspider来创建爬虫的, 默认是使用basic模板来生成spider的. 可以指定为其它的模板.

进入虚拟环境.

# 进入到项目目录中.

cd ArticleSpider

# 查看spider的可用模板

scrapy genspider --list

scrapy genspider -t crawl lagou www.lagou.com

虽然创建成功了, 但出了一个错误, No module named 'utils'

因为在pycharm中设置了ArticleSpider/ArticleSpider为Resource Root, 所以在程序中, 可以直接使用from utils.xxx import xxx来导入 ArticleSpider中的utiles中的内容. 如果取消了这个设置, 项目根目录才是默认的Resource Root, 再直接使用from utils.xxx import xxx这种方法就会出错了. 这时就要使用from ArticleSpider.utils.xxx import xxx. 这里如果设置了ArticleSpider为Resource Root, 这是在pycharm中的设置, 使用命令行来生成spider时并不会生效, 所以上面会出现错误.

打开Run/Debug Configurations

在Python中, 添加对于main.py的Script设置.

Name: main

Script: E:\linuxShare\ArticleSpider\main.py

Python interpreter: Python 3 virtualenv at …

Working directory: E:\linuxShare\ArticleSpider # 项目根目录

同时勾选 Add content roots to PYTHONPATH和Add source roots to PYTHONPATH. # 把content和source添加到python的根搜索路径中.

因为设置了ArticleSpider为Sources Root, pycharm会自动把这个路径, 即source的路径添加到PYTHONPATH中.

把ArticleSpider设置为Sources Root的操作是在pycharm中设置的, 在命令行中新建爬虫时程序就不知道这个设置了, 所以就会出错. 可以把ArticleSpider设置为Source Root的操作加入到settings文件中. 因为settings是所有应用的主入口. 修改settings.py文件.

import os, sys

sys.path.insert(0, "D:\David\Desktop\imooc\_project\ArticleSpider\ArticleSpider")

这个设置和上面在pycharm中的设置一样, 都是把ArticleSpider这个文件夹插入到PYTHONPATH中. 因为这里设置为第0个, 在使用scrapy genspider -t crawl创建爬虫的时候, 它会优先在这个文件夹下找相应的文件.

再使用scrpy genspider -t crawl lagou www.lagou.com来创建爬虫, 就不会出现错误了.

因为使用上面这种绝对的路径时, 如果把项目复制到其它地方, 就会出错, 所以要使用程序来自动获取当前文件的路径.

import sys

BASE\_DIR = os.path.dirname(os.path.abspth(os.path.dirname(\_\_file\_\_)))

# 把BASE\_DIR加入到python path中, 加入到python path中的路径中的文件可以直接import 来使用

sys.path.insert(0, os.path.join(BASE\_DIR, "ArticleSpider"))

**import** **os**

**print**(\_\_file\_\_)

"test.py"

**print**(os.path.dirname(\_\_file\_\_))

"D:/David/Desktop/IT学习文件夹/00 爬虫专题/04\_python分布式爬虫打造搜索引擎/mukepanchong/project/s0vkaq/ArticleSpider/ArticleSpider"

**print**(os.path.abspath(os.path.dirname(\_\_file\_\_)))

"D:/David/Desktop/IT学习文件夹/00 爬虫专题/04\_python分布式爬虫打造搜索引擎/mukepanchong/project/s0vkaq/ArticleSpider/ArticleSpider"

**print**(os.path.dirname(os.path.abspath(os.path.dirname(\_\_file\_\_))))

"D:/David/Desktop/IT学习文件夹/00 爬虫专题/04\_python分布式爬虫打造搜索引擎/mukepanchong/project/s0vkaq/ArticleSpider"

## 6-3 CrawlSpider源码分析 (25:29)

查看生成的lagou.py文件, 修改为如下.

import scrapy

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

class LagouSpider(CrawlSpider):

name = 'lagou'

allowed\_domains = ['www.lagou.com']

start\_urls = ['https://www.lagou.com/']

rules = (

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

)

def parse\_job(self, response):

# 解析拉勾网的职位

i = {}

return i

这里的LagouSpider不再继承于scrapy.Spider了, 而是继承于CrawlSpider了.

查看官方文档对crawlspider的解释.

搜索scrapy document

docs.scrapy.org/en/latest/topics/spiders.html

> Spider > CrawlSpider

对官方源码进行分析

点CrawlSpider, 查看其源码.

class CrawlSpider(Spider):

它也是继承于Spider这个类的. 再查看Spider的源码.

class Spider(object\_ref):

在其中有一个def start\_requests(self)的函数.

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)

def parse(self, response):

raise NotImplementedError

会对start\_urls进行遍历. make\_requests并使用yield返回.

因为CrawlSpider继承于Spider类, 它的入口也是start\_requests.

alt + 左右箭头切换pycharm中打开的文件, 这个快捷键是基于Eclipse的.

在class CrawlSpider中查看, 它的回调函数是parse函数. 在生成的lagou.py文件中, 只有parse\_item函数, 没有parse函数. 在之后的开发中, 都不能再使用parse函数, 这样会重写CrawlSpider中的parse函数, 就会出错. 这里与jobbole和zhihu的爬虫都不一样.

Spider的入口是start\_requests, 调用make\_requests\_from\_url函数. 其中又有parse函数.

CrawlSpider继承于Spider这个类, 但是又重写了parse函数.

def parse(self, response):

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

其中的\_parse\_response函数是crawlSpider的核心函数. parse函数的参数也很重要, 一个是response, 一个是parse\_start\_urls,

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

if callback:

cb\_res = callback(response, \*\*cb\_kwargs) or ()

cb\_res = self.process\_results(response, cb\_res)

for requests\_or\_item in iterate\_spider\_output(cb\_res):

yield requests\_or\_item

if follow and self.\_follow\_links:

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

yield request\_or\_item

\_parse\_response中首先判断是否有callback函数, 这个callback函数就是parse函数中的self.parse\_start\_url函数, parse\_start\_url函数虽然存在, 但其中未做任何的处理. 我们可以重写parse\_start\_url函数, 加入我们自己的处理逻辑. 这个parse\_start\_url与之前的parse函数的逻辑是相似的. 注意不能重写parse函数, 否则就会出错.

\_parse\_response函数会把cb\_kwargs参数传递给callback函数, parse\_start\_url就会接收到这个参数. 因为在parse\_start\_url中直接返回了一个空数组, 所以cb\_res就是一个空数组. cb\_res会接着调用一个process\_results的函数.

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

return results

process\_results这个函数什么也不处理, 直接把传递过来的results返回. 这里process\_results中接收的results参数就是上一步中parse\_start\_url中返回的一个空数组.

所以现在如果不重写parse\_start\_url和process\_results. \_parse\_response函数中cb\_res这两句, 它们就没有进行任何处理.

然后对cb\_res进行迭代, 即对parse\_start\_url和process\_results这两个函数处理的结果进行迭代, 如果cb\_res中有一个item, 则会直接通过yield requests\_or\_item yield出去返回给scrapy.

process\_results处理的是parse\_start\_url返回的数据. 可以根据需要对这两个函数进行重写.

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

if follow and self.\_follow\_links:

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

yield request\_or\_item

查看\_follow\_links,

def set\_crawler(self, crawler):

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

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

会在当前settings中查找CRAWLSPIDER\_FOLLOW\_LINKS这个参数, 如果这个参数不存在, 就会使用默认的Ture, 又由于def \_parse\_response(self, response, callback, cb\_kwargs, follow=True) 中的follow=True, 所以默认会调用for语句, 使用yield发送request\_or\_item, 也就是默认会对链接进行跟进提取.

查看\_requests\_to\_follow的源码

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

if not isinstance(response, HtmlResponse):

return

seen = set()

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

links = [lnk for lnk in rule.link\_extractor.extract\_links(response)

if lnk not in seen]

if links and rule.process\_links:

links = rule.process\_links(links)

for link in links:

seen.add(link)

r = self.\_build\_request(n, link)

yield rule.process\_request(r)

先判断传递过来的response是不是HtmlResponse类型的, 如果不是, 就直接return出去了.

\_requests\_to\_follow这个参数就是处理rule规则的. 那么parse\_response函数会不起作用, lagou.py中设置的rules也不会起作用.

但因为CRAWLSPIDER\_FOLLOW\_LINKS为True, 所以会继续向下执行代码. 开始执行\_requests\_to\_follow这个函数. 如果这个函数传递过来的参数不是response, 会不做任何处理直接返回, 如果是response则继续执行.

设置一个seen=set(), 这个set会记录response中所有的rul, 并进行去重处理. for n, rule in enumerate(self.\_rules)把lagou.py中设置的rules变成一个可迭代的对象. 查看\_rules. 这是在函数\_compile\_rules中.

def \_compile\_rules(self):

def get\_method(method):

if callable(method):

return method

elif isinstance(method, six.string\_types):

return getattr(self, method, None)

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

for rule in self.\_rules:

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

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

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

在CrawlSpider这个类初始化的时候会调用\_compile\_rules, 对lagou.py中设置的rules进行编译.

class CrawlSpider(Spider):

rules = ()

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

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

self.\_compile\_rules()

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

for rule in self.\_rules:

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

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

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

在对rules规则进行处理时, 先进行拷贝, 再调用get\_method方法, get\_method方法的参数就是在lagou.py中rule中设置的callback函数. 处理完成后再调用rule中的process\_links方法. 在lagou.py中并没有设置process\_links方法, 但在后期可能要对url进行一个初步的处理, 如在link前面加上一个域名, 或者取到拉勾网中r'Items/'的link时, 前面加一个不是https://www.lagou.com/的域名等操作, 就使用这里的process\_links方法.

\_compile\_rules在处理完前面两句后, 接着处理rule.process\_request = get\_method(rule.process\_request)

## 6-4 Rule和LinkExtractor使用 (14:28)

import scrapy

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

class LagouSpider(CrawlSpider):

name = 'lagou'

allowed\_domains = ['www.lagou.com']

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

rules = (

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

)

def parse\_job(self, response):

i = {}

return i

查看Rules的源码

**class** Rule(object):

**def** \_\_init\_\_(self, link\_extractor, callback=**None**, cb\_kwargs=**None**, follow=**None**, process\_links=**None**, process\_request=identity):

self.link\_extractor = link\_extractor

self.callback = callback

self.cb\_kwargs = cb\_kwargs **or** {}

self.process\_links = process\_links

self.process\_request = process\_request

**if** follow **is None**:

self.follow = **False if** callback **else True**

**else**:

self.follow = follow

查看LinkExtractor的源码

**class** LxmlLinkExtractor(FilteringLinkExtractor):

**def** \_\_init\_\_(self, allow=(), deny=(), allow\_domains=(), deny\_domains=(), restrict\_xpaths=(),

tags=(**'a'**, **'area'**), attrs=(**'href'**,), canonicalize=**False**,

unique=**True**, process\_value=**None**, deny\_extensions=**None**, restrict\_css=(),

strip=**True**):

restrict\_xpaths和restrict\_css可以对url进行进一步的限定, 如对于拉勾网链接的提取, 可以从列表中提取出职位的链接, 也可以在右边相似的职位中提取出链接, 如果只想从列表中提取出链接, 就可以使用restrict\_xpaths来进行限定.

tags, 默认从a标签和area标签来进行查找.

**def** extract\_links(self, response):

base\_url = get\_base\_url(response)

**if** self.restrict\_xpaths:

docs = [subdoc

**for** x **in** self.restrict\_xpaths

**for** subdoc **in** response.xpath(x)]

**else**:

docs = [response.selector]

all\_links = []

**for** doc **in** docs:

links = self.\_extract\_links(doc, response.url, response.encoding, base\_url)

all\_links.extend(self.\_process\_links(links))

**return** unique\_list(all\_links)

**def** get\_base\_url(response):

*"""Return the base url of the given response, joined with the response url"""*

**if** response **not in** \_baseurl\_cache:

text = response.text[0:4096]

\_baseurl\_cache[response] = html.get\_base\_url(text, response.url,

response.encoding)

**return** \_baseurl\_cache[response]

html.get\_base\_url

**def** get\_base\_url(text, baseurl=**''**, encoding=**'utf-8'**):

*"""Return the base url if declared in the given HTML `text`,*

*relative to the given base url.*

*If no base url is found, the given `baseurl` is returned.*

*"""*

text = to\_unicode(text, encoding)

m = \_baseurl\_re.search(text)

**if** m:

**return** moves.urllib.parse.urljoin(

safe\_url\_string(baseurl),

safe\_url\_string(m.group(1), encoding=encoding)

)

**else**:

**return** safe\_url\_string(baseurl)

在lagou.py中, Rule(LinkExtractor(allow=r'Items/')中的allow中是一个正则表达式. allow中可以传递字符串形式的正则表达式, 也可以传递tuple格式的多个正则表达式.

allow=(r'Items/', r'Links/')

对于拉勾网的招聘信息

https://www.lagou.com/jobs/3036982.html

这里只需要匹配jobs及后面的内容即可, 前面的内容已经通过allowed\_domains过滤了.

rules = (

Rule(LinkExtractor(allow=r'jobs/\d+.hmtl'), callback='parse\_job', follow=True),

)

其中设置的follow=True, 使得它可以对本页面中的url再次进行跟进, 再次提取其中符合本规则的所有的url.

除了类似于https://www.lagou.com/jobs/3036982.html

这要的url地址之外, 还有很多url可以跳转到这种具体的职位的页面. 如

https://www.lagou.com/zhaopin/Python/?labelWords=label

https://www.lagou.com/gongsi/j120576.html

就要设置两个Rule来处理这两个规则 .

rules = (

Rule(LinkExtractor(allow=r'zhaopin/.\*'), follow=True),

Rule(LinkExtractor(allow=r'gongsi/j\d+.html'), follow=True),

Rule(LinkExtractor(allow=r'jobs/\d+.hmtl'), callback='parse\_job', follow=True),

)

也可以把提取的规则写为元组的形式, 每个元组中还可以定义多个提取规则.

rules = (

Rule(LinkExtractor(allow=(r'zhaopin/.\*', )), follow=True),

Rule(LinkExtractor(allow=(r'gongsi/j\d+.html', )), follow=True),

Rule(LinkExtractor(allow=(r'jobs/\d+.hmtl', )), callback='parse\_job', follow=True),

)

爬取到https://www.lagou.com/jobs/3036982.html之后, 就可以来调用parse\_job函数来提取招聘信息的具体内容了.

在main.py中写一个lagou.py的启动命令:

from scrapy.cmdline import execute

import sys, os

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

# execute(["scrapy", "crawl", "jobbole"])

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

在parse\_job中的i = {}处打上断点, 进行debug, 查看是否有链接进入parse\_job这个函数.

下一步就是对职位的信息进行解析并入库

## 6-5 item loader方式解析职位 (24:46)

item\_loader方式解析职位

设计关于lagou的item, 字段的设置尽量和数据库中的字段保持一致. 修改items.py.

class LagouJobItem(scrapy.Item):

#拉勾网职位信息

url = scrapy.Field()

url\_object\_id = scrapy.Field()

title = scrapy.Field()

salary = scrapy.Field()

job\_city = scrapy.Field()

work\_years = scrapy.Field()

degree\_need = scrapy.Field()

job\_type = scrapy.Field()

publish\_time = scrapy.Field()

tags = scrapy.Field()

job\_advantage = scrapy.Field()

job\_desc = scrapy.Field()

job\_addr = scrapy.Field()

company\_url = scrapy.Field()

company\_name = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

要使用item\_loader的方法来完成, 在items.py中定义一个lagou的item\_loader

class LagouJobItemLoader(ItemLoader):

pass

首先分析lagou招聘页面, 找到职位信息的css或xpath提取规则.

使用scrapy shell来进行分析.

scrapy shell <https://www.lagou.com/jobs/478355.html>

在lagou.py中解析职位信息.

from items import LagouJobItemLoader, LagouJob

from ArticleSpider.utils.common import get\_md5

def parse\_job(self, response):

item\_loader = LagouJobItemLoader(item=LagouJobItem(), response=response)

item\_loader.add\_css("title", ".job-name span::text")

item\_loader.add\_value("url", response.url)

item\_loader.add\_value("url\_object\_id", get\_md5(response.url))

item\_loader.add\_css("salary", ".job\_request .salary::text")

# 使用xpath提取第几个元素比较方便.

item\_loader.add\_xpath("job\_city", "//\*[@class='job\_request']/p/span[2]/text()")

item\_loader.add\_xpath("work\_years", "//\*[@class='job\_request']/p/span[3]/text()")

item\_loader.add\_xpath("degree\_need", "//\*[@class='job\_request']/p/span[4]/text()")

item\_loader.add\_xpath("job\_type", "//\*[@class='job\_request']/p/span[5]/text()")

item\_loader.add\_css("tags", ".position-label li::text")

item\_loader.add\_css("publish\_time", ".publish\_time::text")

item\_loader.add\_css("job\_advantage", ".job-advantage p::text")

# 以下两个字段都先提取出所有的包含标签的内容, 然后再在itemloader中进行处理. 在itemloader中尽量只提取内容, 而把对内容的处理都放在items.py中进行.

item\_loader.add\_css("job\_desc", ".job\_bt div")

item\_loader.add\_css("job\_addr", ".work\_addr")

item\_loader.add\_css("company\_name", "#job\_company dt a img::attr(alt)")

#公司有两个url, 一个是在拉勾网上的url, 一个是公司本身的主页. 可以分别进行提取.

item\_loader.add\_css("company\_url", "#job\_company dt a::attr(href)")

from datetime import datetime

item\_loader.add\_value("crawl\_time", datetime.now())

job\_item = item\_loader.load\_item()

return job\_item

在最后一句和parse\_job中第一句打上断点, 进行调试.

## 6-6 职位数据入库

因为之前已经在pipelines.py中的do\_insert函数中完成了把insert\_sql的语句转入到items.py中处理. 现在就不需要去修改pipelines.py文件了, 只需要修改items.py文件就可以了.

修改items.py中的LagouJobItem. 因为提取到的很多字段都是字符串的格式, 要在这里进行处理, 进一步的提取其中的信息.

定义一个函数来处理工作城市中的两个斜杠 "/"

def remove\_splash(value):

#去掉工作城市中的斜杠.

return value.replace("/","")

class LagouJobItem(scrapy.Item):

#拉勾网职位信息

url = scrapy.Field()

url\_object\_id = scrapy.Field()

title = scrapy.Field()

salary = scrapy.Field()

job\_city = scrapy.Field(

input\_processor = MapCompose(remove\_splash),

)

work\_years = scrapy.Field(

input\_processor = MapCompose(remove\_splash),

)

degree\_need = scrapy.Field(

input\_processor = MapCompose(remove\_splash),

)

job\_type = scrapy.Field()

publish\_time = scrapy.Field()

# 把列表形式的标签拼接成字符串

tags = scrapy.Field(

input\_processor = Join(",")

)

job\_advantage = scrapy.Field()

job\_desc = scrapy.Field()

处理工作地点, 把其中的所有html标签都去掉.

from w3lib.html import remove\_tags

job\_addr = scrapy.Field(

input\_processor = MapCompose(remove\_tags),

)

company\_url = scrapy.Field()

company\_name = scrapy.Field()

crawl\_time = scrapy.Field()

crawl\_update\_time = scrapy.Field()

debug, 查看信息是否正确处理. 发现job\_addr中有很多空格没有处理, 并且有"查看地址". 定义一个函数处理这个问题.

def handle\_jobaddr(value):

addr\_list = value.split("\n")

addr\_list = [item.strip() for item in addr\_list if item.strip()!="查看地图"]

return "".join(addr\_list)

修改class LagouJobItem(scrapy.Item)

job\_addr = scrapy.Field(

input\_processor = MapCompose(remove\_tags, handle\_jobaddr),

)

在items.py中定义插入sql的函数, 这里的sql语句必须要与class LagouJobItem中的字段保持一致.

def get\_insert\_sql(self):

insert\_sql = """

insert into lagou\_job(title, url, salary, job\_city, work\_years, degree\_need, job\_type, publish\_time, job\_advantage, job\_desc, job\_addr, company\_name, company\_url, tags, crawl\_time)

VALUES(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

ON DUPLICATE KEY UPDATE salary=VALUES(salary), job\_desc=VALUES(job\_desc)

"""

params = (self["title"], self["url"], self["salary"], self["job\_city"], self["work\_years"], self["degree\_need"], self["job\_type"], self["publish\_time"], self["job\_advantage"], self["job\_desc"], self["job\_addr"], self["company\_name"], self["company\_url"], self["tags"], self["crawl\_time"].strftime(SQL\_DATETIME\_FORMAT)

)

return insert\_sql, params

debug调试. 在pipelines中的process\_item中的query = self.dbpool处打断点, 在do\_inser中的insert\_sql, params和cursor.execute中都打上断点. 一定要在handle\_error中的print处打上断点, 查看error的信息.

但是现在数据库中只有一条数据, 并数据并没有不断的添加进来. 查看数据库, 发现没有url\_object\_id这个主键, 并没有在上面的sql语句中插入进来. 除了第一条数据外, 第二条开始的数据传递进来时都没有主键, 就都和第一条的主键相同, 就不同的冲突, 冲突, 不停的更新更新. 只要在insert\_sql中插入url\_object\_id即可了.

def get\_insert\_sql(self):

insert\_sql = """

insert into lagou\_job(title, url, url\_object\_id, salary, job\_city, work\_years, degree\_need, job\_type, publish\_time, job\_advantage, job\_desc, job\_addr, company\_name, company\_url, tags, crawl\_time)

VALUES(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

ON DUPLICATE KEY UPDATE salary=VALUES(salary), job\_desc=VALUES(job\_desc)

"""

params = (self["title"], self["url"], self["url\_object\_id"], self["salary"], self["job\_city"], self["work\_years"], self["degree\_need"], self["job\_type"], self["publish\_time"], self["job\_advantage"], self["job\_desc"], self["job\_addr"], self["company\_name"], self["company\_url"], self["tags"], self["crawl\_time"].strftime(SQL\_DATETIME\_FORMAT)

)

return insert\_sql, params

再次开始调试, 发现因为爬取过快, 服务器把本机的ip封锁了, 返回了一个错误的信息.

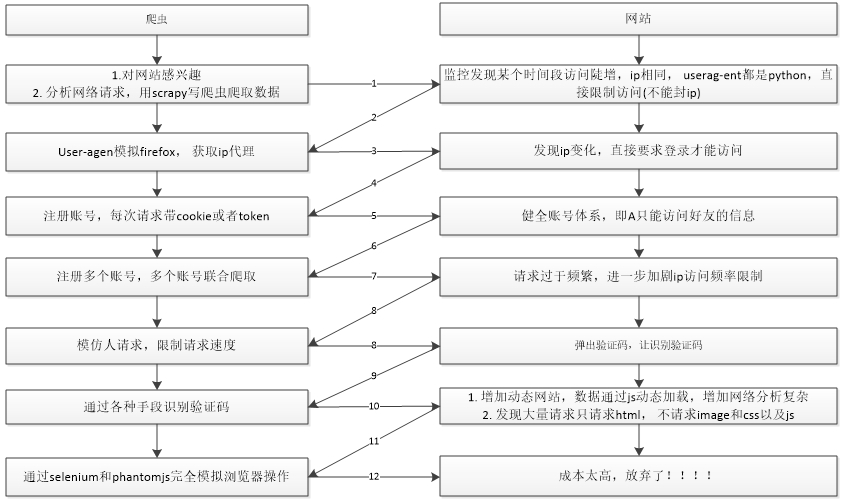
显示https://forbidden.lagou.com/forbidden/fb.hmtl?ip=112.123.322.333这样的错误.

访问任何页面都会跳转到这个forbidden的页面. 其中的ip就是本机的ip地址.

打开这个地址, 发现返回的状态码是200, 这是不合适的, 因为200是请求正常时返回的状态码, 这时是出现了forbidden, 就应该出现返回类似于403这种表示错误的状态码. 搜索引擎在爬取到这个页面时不会停止, 会把这个页面爬下来在搜索引擎中收录. 当seo爬取到多个这样的异常页面时, 会有大量的重复页面, 会认为本网站存在恶意seo的行为, 会在seo中会给本网站降权, 把本网站的排名降低. 如果是403的页面, 百度等搜索引擎根本不会收录这些页面.

# 第7章 Scrapy突破反爬虫的限制

## 7-1 爬虫和反爬的对抗过程以及策略 (20:17)



基本概念

爬虫-自动获取网站数据的程序, 关键是批量的获取.

反爬虫-使用技术手段防止爬虫程序的方法.

误伤-反爬虫技术将普通用户识别为爬虫, 如果误伤过高, 效果再好也不能用.

成本-反爬虫需要的人力和机器成本

拦截-成功拦截爬虫, 一般拦截率越高, 误伤率越高

反爬虫的目的

初始爬虫-简单粗暴, 不管服务器的压力, 容易弄挂网站

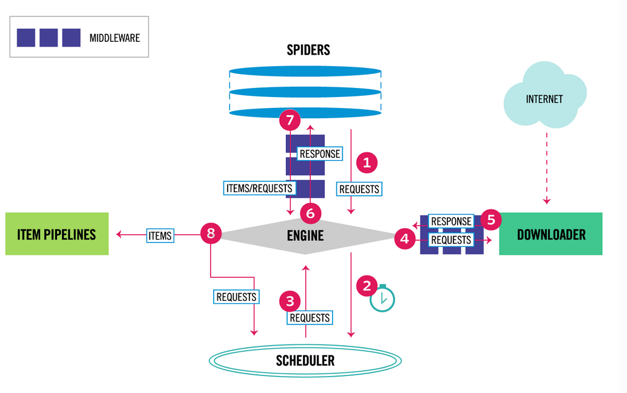
数据保护-

失控的爬虫-由于某些情况下, 忘记或者无法关闭的爬虫

商业竞争的对手.

如果一个ip地址只请求html, 不请求js, css, 图片等静态文件, 它一般都是爬虫.

## 7-2 scrapy架构源码分析 (10:45)



默认情况下, 所有的Request都是由Spider产生的, 而所有的Response都是由Downloader产生的.

Spider产出Request, 处理Response, 使用yield发出Request, 直接接收从Engine传递过来的Response对象.

## 7-3 Requests和Response介绍 (10:18)

<https://doc.scrapy.org/en/latest/topics/request-response.html>

### Request objects

**classscrapy.http.Request(url[, callback, method='GET', headers, body, cookies, meta, encoding='utf-8', priority=0, dont\_filter=False, errback, flags])**

A [**Request**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request) object represents an HTTP request, which is usually generated in the Spider and executed by the Downloader, and thus generating a [**Response**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Response).

|  |  |
| --- | --- |
| **Parameters:** | * **url** (string) – the URL of this request * **callback** (callable) – the function that will be called with the response of this request (once its downloaded) as its first parameter. For more information see [Passing additional data to callback functions](https://doc.scrapy.org/en/latest/topics/request-response.html#topics-request-response-ref-request-callback-arguments) below. If a Request doesn’t specify a callback, the spider’s [**parse()**](https://doc.scrapy.org/en/latest/topics/spiders.html#scrapy.spiders.Spider.parse) method will be used. Note that if exceptions are raised during processing, errback is called instead. * **method** (string) – the HTTP method of this request. Defaults to **'GET'**. * **meta** (dict) – the initial values for the [**Request.meta**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request.meta) attribute. If given, the dict passed in this parameter will be shallow copied. * **body** (str or unicode) – the request body. If a **unicode** is passed, then it’s encoded to**str** using the encoding passed (which defaults to **utf-8**). If **body** is not given, an empty string is stored. Regardless of the type of this argument, the final value stored will be a **str** (never **unicode** or **None**). * **headers** (dict) – the headers of this request. The dict values can be strings (for single valued headers) or lists (for multi-valued headers). If **None** is passed as value, the HTTP header will not be sent at all. * **cookies** (dict or [list](https://doc.scrapy.org/en/latest/topics/api.html#scrapy.loader.SpiderLoader.list)) –   the request cookies. These can be sent in two forms.   * 1. Using a dict:   2. request\_with\_cookies = Request(url="http://www.example.com",   3. cookies={'currency': 'USD', 'country': 'UY'})   4. Using a list of dicts:   5. request\_with\_cookies = Request(url="http://www.example.com",   6. cookies=[{'name': 'currency',   7. 'value': 'USD',   8. 'domain': 'example.com',   9. 'path': '/currency'}])   The latter form allows for customizing the **domain** and **path** attributes of the cookie. This is only useful if the cookies are saved for later requests.  When some site returns cookies (in a response) those are stored in the cookies for that domain and will be sent again in future requests. That’s the typical behaviour of any regular web browser. However, if, for some reason, you want to avoid merging with existing cookies you can instruct Scrapy to do so by setting the **dont\_merge\_cookies** key to True in the [**Request.meta**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request.meta).  Example of request without merging cookies:  request\_with\_cookies = Request(url="http://www.example.com",  cookies={'currency': 'USD', 'country': 'UY'},  meta={'dont\_merge\_cookies': True})  For more info see [CookiesMiddleware](https://doc.scrapy.org/en/latest/topics/downloader-middleware.html#cookies-mw).   * **encoding** (string) – the encoding of this request (defaults to **'utf-8'**). This encoding will be used to percent-encode the URL and to convert the body to **str** (if given as **unicode**). * **priority** (int) – the priority of this request (defaults to **0**). The priority is used by the scheduler to define the order used to process requests. Requests with a higher priority value will execute earlier. Negative values are allowed in order to indicate relatively low-priority. * **dont\_filter** (boolean) – indicates that this request should not be filtered by the scheduler. This is used when you want to perform an identical request multiple times, to ignore the duplicates filter. Use it with care, or you will get into crawling loops. Default to **False**. * **errback** (callable) – a function that will be called if any exception was raised while processing the request. This includes pages that failed with 404 HTTP errors and such. It receives a [Twisted Failure](https://twistedmatrix.com/documents/current/api/twisted.python.failure.Failure.html) instance as first parameter. For more information, see [Using errbacks to catch exceptions in request processing](https://doc.scrapy.org/en/latest/topics/request-response.html#topics-request-response-ref-errbacks) below. * **flags** ([list](https://doc.scrapy.org/en/latest/topics/api.html#scrapy.loader.SpiderLoader.list)) – Flags sent to the request, can be used for logging or similar purposes. |

**url**

A string containing the URL of this request. Keep in mind that this attribute contains the escaped URL, so it can differ from the URL passed in the constructor.

This attribute is read-only. To change the URL of a Request use [**replace()**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request.replace).

**method**

A string representing the HTTP method in the request. This is guaranteed to be uppercase. Example: **"GET"**, **"POST"**, **"PUT"**, etc

**headers**

A dictionary-like object which contains the request headers.

**body**

A str that contains the request body.

This attribute is read-only. To change the body of a Request use [**replace()**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request.replace).

**meta**

A dict that contains arbitrary metadata for this request. This dict is empty for new Requests, and is usually populated by different Scrapy components (extensions, middlewares, etc). So the data contained in this dict depends on the extensions you have enabled.

See [Request.meta special keys](https://doc.scrapy.org/en/latest/topics/request-response.html#topics-request-meta) for a list of special meta keys recognized by Scrapy.

This dict is [shallow copied](https://docs.python.org/2/library/copy.html) when the request is cloned using the **copy()** or **replace()**methods, and can also be accessed, in your spider, from the **response.meta** attribute.

**copy()**

Return a new Request which is a copy of this Request. See also: [Passing additional data to callback functions](https://doc.scrapy.org/en/latest/topics/request-response.html#topics-request-response-ref-request-callback-arguments).

**replace([url**, **method**, **headers**, **body**, **cookies**, **meta**, **encoding**, **dont\_filter**, **callback**, **errback])**

Return a Request object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute [**Request.meta**](https://doc.scrapy.org/en/latest/topics/request-response.html#scrapy.http.Request.meta) is copied by default (unless a new value is given in the **meta** argument). See also [Passing additional data to callback functions](https://doc.scrapy.org/en/latest/topics/request-response.html#topics-request-response-ref-request-callback-arguments).

### Using errbacks to catch exceptions in request processing

The errback of a request is a function that will be called when an exception is raise while processing it.

It receives a [Twisted Failure](https://twistedmatrix.com/documents/current/api/twisted.python.failure.Failure.html) instance as first parameter and can be used to track connection establishment timeouts, DNS errors etc.

Here’s an example spider logging all errors and catching some specific errors if needed:

import scrapy

from scrapy.spidermiddlewares.httperror import HttpError

from twisted.internet.error import DNSLookupError

from twisted.internet.error import TimeoutError, TCPTimedOutError

class ErrbackSpider(scrapy.Spider):

name = "errback\_example"

start\_urls = [

"http://www.httpbin.org/", # HTTP 200 expected

"http://www.httpbin.org/status/404", # Not found error

"http://www.httpbin.org/status/500", # server issue

"http://www.httpbin.org:12345/", # non-responding host, timeout expected

"http://www.httphttpbinbin.org/", # DNS error expected

]

def start\_requests(self):

for u in self.start\_urls:

yield scrapy.Request(u, callback=self.parse\_httpbin,

errback=self.errback\_httpbin,

dont\_filter=True)

def parse\_httpbin(self, response):

self.logger.info('Got successful response from {}'.format(response.url))

# do something useful here...

def errback\_httpbin(self, failure):

# log all failures

self.logger.error(repr(failure))

# in case you want to do something special for some errors,

# you may need the failure's type:

if failure.check(HttpError):

# these exceptions come from HttpError spider middleware

# you can get the non-200 response

response = failure.value.response

self.logger.error('HttpError on %s', response.url)

elif failure.check(DNSLookupError):

# this is the original request

request = failure.request

self.logger.error('DNSLookupError on %s', request.url)

elif failure.check(TimeoutError, TCPTimedOutError):

request = failure.request

self.logger.error('TimeoutError on %s', request.url)

## 7-4 通过downloadmiddleware随机更换user-agent

如何实现user-agent的自动更换呢.

最简单的方法就是爬虫中设置一个全局性的参数headers, 每次yield发送Request时都使用headers=headers把这个请求头添上, 这也是之前spider中使用的方法,

第二种方法是在settings中添加设置, 把所能想到的全部的user-agent都写在这个列表中.

user\_agent\_list = [

"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2227.1 Safari/537.36",

]

修改zhihu.py

from settings import user\_agent\_list

class ZhihuSpider(scrapy.Spider):

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

start\_answer\_url = ""

import random

random\_index = random.randint(0, len(user\_agent\_list)-1)

random\_agent = user\_agent\_list[random\_index]

headers = {

"HOST": "www.zhihu.com",

"Referer": "https://www.zhizhu.com",

'User-Agent': random\_agent

}

但是上面定义的headers是在类之中的, 在想要在类方法中使用, 必须要把以上的生成随机user-agent的代码复制到每一个需要发送Request的类方法中.

def parse(self, response):

...

import random

random\_index = random.randint(0, len(user\_agent\_list)-1)

random\_agent = user\_agent\_list[random\_index]

self.headers["User-Agent"] = random\_agent

yield scrapy.Request(request\_url, headers=self.headers, callback=self.parse\_question)

else:

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

def parse\_question(self, response):

...

import random

random\_index = random.randint(0, len(user\_agent\_list)-1)

random\_agent = user\_agent\_list[random\_index]

self.headers["User-Agent"] = random\_agent

yield scrapy.Request(self.start\_answer\_url.format(question\_id, 20, 0), headers=self.headers, callback=self.parse\_answer)

yield question\_item

def parse\_answer(self, reponse):

...

yield scrapy.Request(next\_url, headers=self.headers, callback=self.parse\_answer)

def start\_requests(self):

...

return [scrapy.Request('https://www.zhihu.com/#signin', headers=self.headers, callback=self.login)]

def login(self, response):

...

return [scrapy.FormRequest(

url = post\_url,

formdata = post\_data,

headers=self.headers,

callback=self.check\_login

)]

def check\_login(self, response):

...

yield scrapy.Request(url, dont\_filter=True, headers=self.headers)

在知乎这个例子中, 以上所有的类方法中都要加入取随机user-agent的代码, 这样做就显得很麻烦, 并且如果爬虫太多, 一个个的进行修改就有些不现实.

在downloader和engine之间有一层middleware, 叫做download middleware, 是requests从engine发送到downloaer和downloader下载数据返回到engine时都必须要经过的middleware. 这是一个全局的middleware, 只要在这个downoad midddleware中设置user-agent, 就不用把随机更换user-agent的代码写到spider中了.

在settings中可以配置download middleware,

DOWNLOADER\_MIDDLEWARES = {

'ArticleSpider.middlewares.MyCustomDownloaderMiddleware': 543,

}

只要定义一个downloadermiddleware的类, 来随机的更换user-agent, 并把它放在settings中的DOWNLOADER\_MIDDLEWARES中去, 就能实现对所有的请求都使用随机的user-agent.

scrapy本身提供了一个user-agent的mdiddleware, scrapy.downloadermiddlewares.useragent.py,

from scrapy import signals

class UserAgentMiddleware(object):

def \_\_init\_\_(self, user\_agent='Scrapy'):

self.user\_agent = user\_agent

@classmethod

def from\_crawler(cls, crawler):

o = cls(crawler.settings['USER\_AGENT'])

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

return o

def spider\_opened(self, spider):

self.user\_agent = getattr(spider, 'user\_agent', self.user\_agent)

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

if self.user\_agent:

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

在其中可以看到, user-agent实际上有一个Scrapy的默认值. 然后设置了一个静态的方法from\_crawler, 可以直接通过类来调用这个方法. 在这个from\_crawler的方法中, 会把当前的爬虫crawler传递进来, 然后从settings中读取USER\_AGENT的值. 如果读取不到, 就会使用默认的"Scrapy". 所以在settings中要单独设置一个USER\_AGENT的变量. 接下来process\_request的函数, 如果在自定义UserAgentMiddleware或downloader middleware的时候, 想要对request进行进一步的处理, 就要使用process\_request这个函数. 它对每一个request中的headers都设置了一个默认的self.user\_agent的值.

查看官方文档, Activating a downloader middleware中如何去自定义一个middleware. 这个downlaoder middleware既可以处理request又可以处理response.

如果要自己定义user-agent的download middleware, 就要把原来scrapy中的useragentmiddleware设置为None或注释掉或者把自定义的useragent downloadmiddleware的值设置的比系统的大, 这样自定义的后执行, 就覆盖掉了系统的useragent download midddleware.

重写了process\_request(request, spider)函数, 就会处理request

重写了process\_response(request, response, spider), 就会处理response

重写了process\_exception(request, exception, spider), 就会处理异常

修改middlewares.py文件, 定义一个生成随机的user-agent的middleware.

class RandomUserAgentMiddleware(object):

#随机更换user-agent

# 第2步, 在初始化方法中读取settings中设置的user\_agent\_list

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

#调用父类的初始化方法进行初始化.

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

#把crawler传递过来, 读取它的设置中的user\_agent\_list.

self.user\_agent\_list = crawler.settings.get("user\_agent\_list", [])

# 第1步, 定义from\_crawler这个静态方法

@classmethod

def from\_crawler(cls, crawler):

# 如果是静态的方法, class要改为cls

# 把爬虫传递到RandomUserAgentMiddleware这个类中.

return cls(crawler)

import random

random\_index = random.randint(0, len(user\_agent\_list)-1)

random\_agent = user\_agent\_list[random\_index]

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

#处理所有的request, 把它的默认的headers中的user-agent设置为randowm\_agent

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

使用Fake Useragent

以上这样写是可以实现的, 但在后期还会需要手动去维护settings中的user\_agent\_list的列表. 是否还有更便携的方法.

https://github.com/hellysmile/fake-useragent

http://useragentstring.com/

https://www.w3schools.com/browsers/default.asp

Installation

pip install fake-useragent

Usage

from fake\_useragent import UserAgent

#使用UserAgent()的类生成实例.

ua = UserAgent()

就可以通过以下命令来取到相应浏览器的随机的user-agent.

ua.ie

ua.msie

ua['Internet Explorer']

ua.opera

ua.chrome

ua.google

ua['google chrome']

ua.firefox

ua.ff

ua.safari

或者使用这个命令来取得随机的user-agent

ua.random

在github的源码中并没有看到useragent的列表, 它实际上在下面的地址上维护了一些useragent的列表

https://fake-useragent.herokuapp.com/browsers/0.1.8

<http://d2g6u4gh6d9rq0.cloudfront.net/browsers/fake_useragent_0.1.10.json>

修改middlewares.py

from fake\_useragent import UserAgent

class RandomUserAgentMiddleware(object):

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

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

self.ua = UserAgent()

@classmethod

def from\_crawler(self, crawler):

return cls(crawler)

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

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

如果希望在这种方法的基础上进行一些扩展, 在settings中设置一项, 指定middlewares中随机选择的是哪一个浏览器的user-agent.

修改settings.py, 添加一行

# RANDOM\_UA\_TYPE = "random"

RANDOM\_UA\_TYPE = "ie"

修改middlewares.py

from fake\_useragent import UserAgent

class RandomUserAgentMiddleware(object):

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

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

self.ua = UserAgent()

self.ua\_type = crawler.settings.get("RANDOM\_UA\_TYPE", "random")

@classmethod

def from\_crawler(self, crawler):

return cls(crawler)

但现在就不能通过self.ua.random来取任意的user-agent了, 而是要根据settings中设置的RANDOM\_UA\_TYPE来取对应的类型的user-agent. 因为没有slef.ua.self.ua\_type的方法, 可以在def process\_request中再定义一个函数来获取ua\_type.

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

def get\_ua():

#getattr这种方法, 就是取ua这个对象中的ua\_type值

return getattr(self.ua, self.ua\_type)

random\_agent = get\_ua() #为了调试的方法, 可以把get\_ua()的结果保存为一个变量.

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

getattr这种方法, 就是取ua这个类中ua\_type的值. 如果settings中设置为RANDOM\_UA\_TYPE = "random", 就取到了random这个值. get\_ua()返回的结果就是self.ua.random.

如果settings中设置为RANDOM\_UA\_TYPE = "ie", 就取到了random这个值. get\_ua()返回的结果就是self.ua.ie.

把RandomUserAgentMiddleware添加到settings中的DOWNLOADER\_MIDDLEWARES中.

DOWNLOADER\_MIDDLEWARES = {

'ArticleSpider.middlewares.DOWNLOADER\_MIDDLEWARES': 543,

}

在random\_agent = get\_ua()处打上断点, 进行调试.

## 7-6 scrapy实现ip代理池

爬取时尽量不要让本机的ip被禁止掉.

在middlewares.py中的process\_request中添加一行使用代理的代码即可使用代理地址.

class RandomUserAgentMiddleware(object):

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

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

self.ua = UserAgent()

self.ua\_type = crawler.settings.get("RANDOM\_UA\_TYPE", "random")

@classmethod

def from\_crawler(self, crawler):

return cls(crawler)

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

def get\_ua():

return getattr(self.ua, self.ua\_type)

random\_agent = get\_ua()

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

request.meta["proxy"] = "http://221.238.67.231:8081"

simple\_http\_server.py 是一个微型的服务器, 可以打印出访问这个服务器的主机的ip地址, 在远程主机上运行这个微型服务器, 在本机访问这个ip地址, 端口号为8088, 就会返回Hi, 然后在服务器运行窗口中打印出来本机的ip地址,

把scrapy中的spider的start\_urls改为服务器的ip地址和8088端口, 然后在scrapy中使用代理地址. 在parse和parse\_detail中添加断点, 调试爬虫, 服务器中打印的ip地址就会发生变化.

关键是如何设置一个ip的代理池, 从代理池中随机的取出一些ip地址.

首先要写一个爬虫, 来爬取xicidaili.com中的高匿代理.

爬取代理的ip地址, 商品, 类型, 是否匿名中只取高匿的. 把爬取的数据放在文件或数据库中. 再从中随机取代理即可.

在项目根目录中新建一个tools的python package文件夹, 专门存在一些脚本文件. 在其中新建crawl\_xici\_ip.py

import requests

from scrapy.selector import Selector

def crawl\_ips():

#爬取xici的免费ip代理.

headers = {

"User-Agent": "Mozilla/5.0 (Windows NT 6.4; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2225.0 Safari/537.36"

}

#获取所有页面中的ip

for i in range(1568):

response = requests.get("http://www.xicidaili.com/nn/{0}".format(i), headers = headers)

# print(re.text)

# 在scrapy爬虫中都是直接使用response.xpath和response.css, 其实是把scrapy的selector封装到response中了, 这里是直接调用scrapy中的selector

selector = Selector(text=response.text)

all\_trs = selector.css("#ip\_list tr")

# 保存爬取到的ip地址

ip\_list = []

# all\_trs中还包括表头, 要把表头去掉.

for tr in all\_trs[1:]:

# tr是一个selector, 可以进行嵌套的选择

# 连接时间, 在后期可以选择连接时间比较短的代理

speed\_str = tr.css(".bar::attr(title)").extract()[0]

if speed\_str:

speed = float(speed\_str.split("秒")[0])

# 得到的all\_text()中有很多的空格和换行符, 删除后就成了标准格式了. 也可以使用这种方法来选取.

all\_texts = tr.css("td::text").extract()

# print(all\_texts)

ip = all\_texts[0]

port = all\_texts[1]

proxy\_type = all\_texts[5]

#以tuple的方式把ip的信息存入到ip\_list中.

ip\_list.append((ip, port, proxy\_type, speed))

#每提取完一页的数据, 就把它保存到数据库中.

for ip\_info in ip\_list:

#以下三行要放在开始, 放在这里只是为了说明的方便.

import MySQLdb

#数据库的连接

conn = MySQLdb.connect(host="127.0.0.1", user="root", password="mysql", db="article\_spider", charset="utf8")

cursor = conn.cursor()

"""

新建一张数据表proxy\_ip

字段名称 类型 长度 是否为null 主键

ip varchar 20 T primary key

port varchar 255 T

speed float 0 F

proxy\_type varchar 5 F

"""

#这里只提取的http的, 所以可以直接把所有的proxy\_type设置为'HTTP', 注意在传递值的时候字符串一定要传递单引号

cursor.execute(

"insert proxy\_ip(ip, port, speed, proxy\_type) VALUES('{0}', '{1}', {2}, 'HTTP')".format(ip\_info[0], ip\_info[1], ip\_info[3]

)

)

conn.commit()

在cursor.execute处打上断点, 进行调试. 在scrapy中使用代理地址时还需要把代理地址从数据库中取出来, 修改crawl\_xici\_ip.py, 定义一个类从数据库中取出代理地址.

#从数据库中取出ip代理

class GetIP(object):

def delete\_ip(self, ip):

#从数据库中删除无效的ip, 也可以使用django来操作数据库.

delete\_sql = """

delete from proxy\_ip where ip='{0}'

""".format(ip)

cursor.execute(delete\_sql)

conn.commit()

return True

def judge\_ip(self, ip, port)

#判断ip是否可用

http\_url = "http://www.baidu.com"

#这里使用的是http的代理

proxy\_url = "http://{0}:{1}".format(ip, port)

try:

proxy\_dict = {

# http格式的代理

"http": proxy\_url

# https格式的代理

# "https": xxxx

}

response = request.get(http\_url, proxies=proxy\_dict)

#如果访问百度没有出现异常, 就认为ip是可用的

return True

except Exception as e:

print("invalid ip and port")

# 如果使用代理访问百度异常, 就调用delet\_ip把它删除掉. 其实这种方法还不太保险, 能正常访问百度, 并不一定能够正常访问我们要爬取的网站.

# ip代理不可用

self.delete\_ip(ip)

return False

else:

code = response.status\_code()

# 如果返回的状态码在200-300之间, 就认为是可用的ip

if code >=200 and code < 300:

print("effective ip")

return True

else:

print("invalid ip and port")

self.delete\_ip(ip)

return False

def get\_random\_ip(self):

#从数据库中随机获取一个可用的ip

random\_sql = """

SELECT ip, port FROM proxy\_ip

ORDER BY RAND()

LIMIT 1

"""

result = cursor.execute(random\_sql)

#ip的格式是tuple, 可以像数组一样取值.

for ip\_info in cursor.fetchall():

ip = ip\_info[0]

port = ip\_info[1]

# 判断ip地址是否可用.

judge\_result = self.judge\_ip(ip, port)

if judge\_result:

#如果ip可用, 把这个ip代理返回

return "http://{0}:{1}".format(ip, port)

else:

#如果ip不可用, 再次调用get\_random\_ip获取随机ip

return self.get\_random\_ip()

# print(crawl\_ips())

# 这样, 在其它文件import这个函数时, 就不会执行这两句了.

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

get\_ip = GetIP()

get\_ip.get\_random\_ip()

在proxy\_url = "http://{0}:{1}".format(ip, port)处打上断点, result = cursor.execute打上断点. judge\_result = self.judge\_ip处打上断点. get\_ip = GetIP()和get\_ip.get\_random\_ip打上断点.

在middlewares.py中定义一个RandomProxy的middleware

from tools.crawl\_xici\_ip import GetIP

class RandomProxyMiddleware(object):

#动态设置ip代理

def process\_request(self, request, spider)

get\_ip = GetIP()

request.meta["proxy"] = get\_ip.get\_random\_ip()

以上就完成了爬取ip地址并保存到数据库中, 并且从数据库中取出ip使用.

还可以使用scrapy-proxies这个库

https://github.com/aivarsk/scrapy-proxies

但这里的ip代理是写在settings中的, 不方便进行维护, 如无法像数据库中一样进行添加和删除, 可以根据文件中的内容结合课程中的代码改成更符合自己要求的代码.

最稳定的ip还是自己本机的ip, 一定要注意限速, 限量的去爬取.

scrapy官网也提供了一个工具: scrapy crawlera, 只是一个收费的服务.

https://github.com/scrapy-plugins/scrapy-crawlera

https://scrapinghub.com/crawlera

## 7-9 云打码实现验证码识别

验证码识别方法

1. 编码实现(tesseract-ocr)

2. 在线打码

3. 人工打码

http://www.uxwang.com/codelist.asp

yundama.com

注册两个账号, 一个普通用户, 一个开发者账户.

调用示例

http://yundama.com/apidoc/YDM\_SDK.html

python调用示例中是调用dll文件实现打码, 只能在windows中使用, linux中的动态连接库中ios文件, python HTTP调用示例是纯python调用实现打码. 这里使用python HTTP方法.

yundama\_requests.py

import json

import requests

class YDMHttp(object):

# 调用api

apiurl = 'http://api.yundama.com/api.php'

username = ''

password = ''

appid = ''

appkey = ''

# 初始化时传递用户名, 密码, appid, appkey四个参数

def \_\_init\_\_(self, username, password, appid, appkey):

self.username = username

self.password = password

self.appid = str(appid)

self.appkey = appkey

# balance, login和decode三个函数都调用的是apiurl这个接口, 只是传递的data不同.

def balance(self):

data = {'method': 'balance', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey}

response\_data = requests.post(self.apiurl, data=data)

ret\_data = json.loads(response\_data.text)

#获取积分成功

if ret\_data["ret"] == 0:

print ("获取剩余积分", ret\_data["balance"])

return ret\_data["balance"]

#获取积分失败

else:

return None

def login(self):

data = {'method': 'login', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey}

response\_data = requests.post(self.apiurl, data=data)

ret\_data = json.loads(response\_data.text)

if ret\_data["ret"] == 0:

print ("登录成功", ret\_data["uid"])

return ret\_data["uid"]

else:

return None

#验证码解码

def decode(self, filename, codetype, timeout):

data = {'method': 'upload', 'username': self.username, 'password': self.password, 'appid': self.appid, 'appkey': self.appkey, 'codetype': str(codetype), 'timeout': str(timeout)}

files = {'file': open(filename, 'rb')}

response\_data = requests.post(self.apiurl, files=files, data=data)

ret\_data = json.loads(response\_data.text)

if ret\_data["ret"] == 0: #在此处添加断点调试.

print ("识别成功", ret\_data["text"])

return ret\_data["text"]

else:

return None

if \_\_name\_\_ == "\_\_main\_\_":

# 用户名, 普通用户的用户名

username = 'da\_ge\_da1'

# 密码, 普通用户的密码

password = 'da\_ge\_da'

# 软件ID, 开发者分成必要参数. 登录开发者后台【我的软件】>软件代码获得！开发者注意之后开发软件, 提供给普通用户使用, 普通用户收费使用开发者提供的软件.

appid = 3129

# 软件密钥, 开发者分成必要参数. 登录开发者后台【我的软件】> 通讯密钥获得！

appkey = '40d5ad41c047179fc797631e3b9c3025'

# 图片文件

filename = 'getimage.jpg'

# 验证码类型, # 例: 1004表示4位字母数字, 不同类型收费不同. 请准确填写, 否则影响识别率. 在此查询所有类型 http://www.yundama.com/price.html, 如果不知道, 可以设置为5000

codetype = 1004

# 超时时间, 秒

timeout = 60

# 检查

if (username == 'username'):

print ('请设置好相关参数再测试')

else:

# 初始化, 传递username, password, appid, appkey, appid和appkey在登录之后就可以得到.

yundama = YDMHttp(username, password, appid, appkey)

# 登陆云打码

uid = yundama.login();

print ('uid: %s' % uid)

# 查询余额

balance = yundama.balance();

print ('balance: %s' % balance)

# 开始识别, 图片路径, 验证码类型ID, 超时时间（秒）, 识别结果

text = yundama.decode(filename, codetype, timeout);

在decode()中的if ret\_data["ret"] == 0 中添加断点, 进行测试.

## 7-10自定义settings

自动限速

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

<https://doc.scrapy.org/en/latest/topics/autothrottle.html>

<https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/autothrottle.html?highlight=%E9%99%90%E9%80%9F>

DOWNLOAD\_DELAY

默认: 0

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

DOWNLOAD\_DELAY = 0.25 # 250 ms of delay

该设定影响(默认启用的) [RANDOMIZE\_DOWNLOAD\_DELAY](https://scrapy-chs.readthedocs.io/zh_CN/latest/topics/settings.html?highlight=download%20delay#std:setting-RANDOMIZE_DOWNLOAD_DELAY) 设定。 默认情况下，Scrapy在两个请求间不等待一个固定的值， 而是使用0.5到1.5之间的一个随机值 \* [DOWNLOAD\_DELAY](https://scrapy-chs.readthedocs.io/zh_CN/latest/topics/settings.html?highlight=download%20delay#std:setting-DOWNLOAD_DELAY) 的结果作为等待间隔。

当 [CONCURRENT\_REQUESTS\_PER\_IP](https://scrapy-chs.readthedocs.io/zh_CN/latest/topics/settings.html?highlight=download%20delay#std:setting-CONCURRENT_REQUESTS_PER_IP) 非0时，延迟针对的是每个ip而不是网站。

另外您可以通过spider的 download\_delay 属性为每个spider设置该设定。

# Disable cookies (enabled by default)

COOKIES\_ENABLED = False

# Disable Telnet Console (enabled by default)

# scrapy默认我们每个网页下载的间隔即下载延迟为0, scrapy提供了一种可以动态设置下载速度的扩展. AutoThrottle扩展

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

TELNETCONSOLE\_ENABLED = False

DOWNLOAD\_DELAY = 10

AUTOTHROTTLE\_ENABLED = True

### 为不同的spider设置不同的settings

如知乎是不能禁用cookies的, jobbole是可以禁用cookies的.

查看spider的源码, scrapy.spiders.\_\_init\_\_.py

class Spider(object\_ref):

name = None

custom\_settings = None

@classmethod

def update\_settings(cls, settings):

settings.setdict(cls.custom\_settings or {}, priority='spider')

这里的custom\_settings是放在spider爬虫中的, 如果在爬虫中设置了custom\_settings, 就会使用spider自己的设置, 从而覆盖掉settings.py中默认的设置. cls.custom\_settings注意这是class的settings, 不是对象的属性, 所以不能放到\_\_init\_\_中去.

在settings.py中, 设置

COOKIES\_ENABLED = False

在zhihu.py中, 添加一句设置即可.

class ZhihuSpider(scrapy.Spider):

name = "zhihu"

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

start\_urls = ['https://www.zhihu.com/']

start\_answer\_url = ""

headers = {}

custom\_settings = {

"COOKIES\_ENABLED": True

}

def parse(self, response):

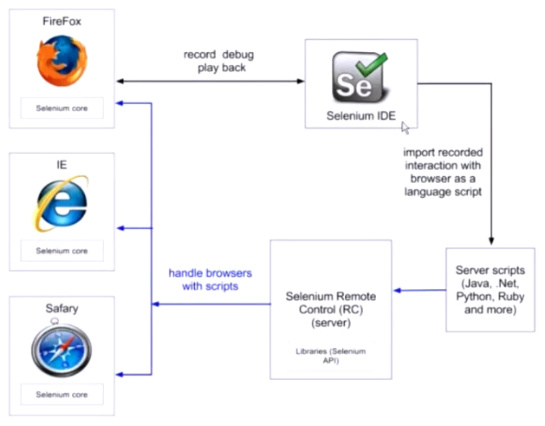
这样, 知乎的spider就会使用custom\_settings中的值.

# 第8章 scrapy进阶开发

## 8-1 selenium动态网页请求与模拟登录知乎 (21:24)

淘宝的详情页是动态加载的页面，如果在浏览器中查看源码，会发现很多字段都是空的，都是使用js加载的效果。

selenium是一个自动化测试框架，它的核心是使用javascript来操作浏览器。selenium来操作每一个浏览器都需要有一个driver



进入虚拟环境

pip install selenium

查看官方文档：

http://selenium-python.readthedocs.io/api.html

selenium实际上只是一个api接口，它还是要调用浏览器，必须要有对应的浏览器的driver才能完成与浏览器的交互。

selenium也支持分布式。

新建tools/selenium\_spider.py

from selenium import webdriver

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://detail.tmall.com/item.htm?id=537951179371")

print(browser.page\_source)

#关闭浏览器

browser.quit()

这里的page\_source是在浏览器中通过F12或者检查元素时看到的源码。是js解析过之后的页面对应的源码。会打开一个chrome的页面，加载完成后就会打印html页面。

使用selenuim自带的元素提取工具

browser.find\_element\_by\_css\_selector

但由于selenuim是使用python开发的，速度上会有所限制，所以并不建议使用其自带的元素选择方法, find\_element\_by\_id, css\_selector。但对页面元素进行操作时, 还是要使用selenuim自带的选择方法. 而对页面元素进行提取时,还是推荐使用scrapy的selector

scrapy的selector是使用lxml这个库来实现元素的查的的, lxml是使用c语言完成的库, 所以速度会更快一些.

使用scrapy的selector来进行元素的选取。

from selenium import webdriver

from scrapy.selector import Selector

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://detail.tmall.com/item.htm?id=537951179371")

s\_selector = Selector(text=browser.page\_source)

#选取天猫中的价格

promo\_price = s\_selector.css(".tm-promo-price .tm-price::text").extract()

print(promo\_price)

browser.quit()

使用selenuim完成知乎的模拟登陆。

from selenium import webdriver

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://www.zhihu.com/#signin")

browser.find\_element\_by\_css\_selector(".view-signin input[name='account']").send\_keys("13423423422")

browser.find\_element\_by\_css\_selector(".view-signin input[name='password']").send\_keys("23f@fsaf2")

browser.find\_element\_by\_css\_selector(".view-signin button.sign-button]").click()

# browser.quit()

会打开一个浏览器，然后自动输入用户名和密码, 如果需要输入验证码，则手动输入即可。

## 8-2 selenium模拟登录微博，模拟鼠标下拉

如果想要获取微博的内容，可以使用open.weibo.com这个开放平台。

selenium模拟登录微博

from selenium import webdriver

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://www.weibo.com")

browser.find\_element\_by\_css\_selector("#loginname").send\_keys("")

browser.find\_element\_by\_css\_selector(".info\_list.password input[name='password']").send\_keys("")

browser.find\_element\_by\_css\_selector(".info\_list.login\_btn a[node-type='submitBtn']").click()

# browser.quit()

打开浏览器后并没有自动登陆，出现了一个错误，找不到#loginname

这是因为这个页面还没有加载完成，登陆的代码还没有加载出来，就跳转到登陆的操作了。只需要等待一段时间即可.

from selenium import webdriver

import time

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://www.weibo.com")

time.sleep(10)

browser.find\_element\_by\_css\_selector("#loginname").send\_keys("")

browser.find\_element\_by\_css\_selector(".info\_list.password input[name='password']").send\_keys("")

browser.find\_element\_by\_css\_selector(".info\_list.login\_btn a[node-type='submitBtn']").click()

# browser.quit()

再进行登陆，就可以正常跳转了。

在微博中，鼠标下拉会加载更多的内容，在类似页面是动态加载的时候，就可以使用此操作。

selenuim可以直接执行javascript代码完成功能, 就可以控制页面的动作.

from selenium import webdriver

import time

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe")

browser.get("https://www.oschina.net/blog")

time.sleep(5)

# 下拉3次

for i in range(3):

# 执行原生的js代码

js = "window.scrollTo(0, document.body.scrollHeight); var lenOfPage = document.body.scrollHeight; return lenOfPage;"

browser.execute\_script(js)

time.sleep(3)

# browser.quit()

## 8-3 chromedriver不加载图片、phantomjs获取动态网页

不加载图片加以加快页面的加载速度。

from selenium import webdriver

import time

# 生成chrome选项

chrome\_opt = webdriver.ChromeOptions()

# 设置不加载图片

prefs = {"profile.managed\_default\_content\_settings.images": 2}

chrome\_opt.add\_experimental\_option("prefs", prefs)

browser = webdriver.Chrome(excutable\_path="D:/python\_packages/chromedriver.exe", chrome\_options=chrome\_opt)

browser.get("https://www.taobao.com")

# browser.quit()

phantomjs是一个无界面的浏览器。多进程的情况下phantomjs的性能会下降很严重, 并且页面渲染可能出问题。在没有图形界面的centos或ubuntu中，phantomjs的优势就能体验出来了。

而在windows下，还是使用chrome，chrome的性能比phantomjs更好。

http://phantomjs.org/

from selenium import webdriver

browser = webdriver.Phantomjs(excutable\_path="D:\python\_packages\phantomjs-2.1.1-windows\bin\phantomjs.exe")

browser.get("https://detail.tmall.com/item.htm?id=537951179371")

print(browser.page\_source)

#phantomjs是一个无界面的浏览器，一定要记得把它退出

browser.quit()

## 8-4 把selenium集成到scrapy中

可以设置一个中间件，中间件的process\_request方法会对所有的request进行处理。

修改middlewares.py, 添加一个中间件

from selenium import webdriver

import time

class JSPageMiddleware(object):

"""通过chrome请求动态网页"""

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

# 不是对所有的爬虫都使用slenium来进行解析, 只对jobbole爬虫使用slenium来解析.

if spider.name == "jobbole":

browser = webdriver.Chrome(executable\_path="D:/python\_packages/chromedriver.exe")

browser.get(request.url)

time.sleep(3)

print("访问{0}".format(request.url))

#因为此处已经通过browser发送了reqeust请求，就不用再发送到scrapy的下载器进行下载了。

from scrapy.http import HtmlResponse

#只要遇到了htmlResponse，scracpy就不会再向下载器发送请求了，而是直接返回到spider。查看htmlResponse的源码。默认的编码是ascii, 需要根据具体的页面来设置encoding编码

return HtmlResponse(url=browser.current\_url, body=browser.page\_source, encoding="utf-8", request=request)

修改settings.py, 启用JSPageMiddleware

DOWNLOADER\_MIDDLEWARES = {

'ArticleSpider.middlewares.JSPageMiddleware': 100

}

在if spider.name处，return HtmlResponse处打上断点，在jobbole.py中post\_nodes=response.css处打上断点。看能否返回到spider中。

以上方法虽然可以通过chrome来请求到页面，但每次请求一个页面都会打开一个chrome，这样就会特别慢。需要设置在请求页面时不打开一个新的chrome。只需要在\_\_init\_\_初始化方法中把browser定义为一个类属性，这样就不会每次请求页面都打开chrome了。

class JSPageMiddleware(object):

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

self.browser = webdriver.Chrome(executable\_path="D:/python\_packages/chromedriver.exe")

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

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

if spider.name == "jobbole":

# browser = webdriver.Chrome(executable\_path="D:/python\_packages/chromedriver.exe")

self.browser.get(request.url)

import time

time.sleep(3)

print("访问{0}".format(request.url))

from scrapy.http import HtmlResponse

return HtmlResponse(url=self.browser.current\_url, body=self.browser.page\_source, encoding="utf-8", request=request)

打上断点，在main.py中再次进行调试，查看打开的chrome，第一次请求的是blog.jobbole.com/all-posts/，按F8继续调试, 第二次请求的是blog.jobbole.com/110774/这个页面，两个页面都是在同一个chrome窗口中打开的。并且return之后不会关闭chrome。手动关闭调试时chrome才会关闭。

在jobbole.py中把parse函数中post\_nodes以下的内容都注释掉，以pass来替换，这样就只请求一个all\_post页面，请求完成后爬虫就会自动关闭。再次运行，但是爬虫关闭后浏览器却没有自动关闭。

因为在中间件中只能重写process\_request和process\_response，没有spider close时调用的内容的设置。

上面的方法是把browser放在JSPageMiddleware的\_\_init\_\_中，每个请求都会生成一个browser。想要整个爬虫运行的过程中只生成一个browser，且随着spider的打开而打开，随着spider的关闭而关闭。可以把browser放在spider中，每个spider都会启动一个chrome，这样对并发也是有益的。

修改jobbole.py，把browser放在JobboleSpider的\_\_init\_\_方法中。

from selenium import webdriver

# 导入分发器

from scrapy.xlib.pydispatch import dispatcher

# 导入信息

from scrapy import signals

class JobboleSpider(scrapy.Spider):

name = "jobbole"

allowed\_domains = ["blog.jobbole.com"]

start\_urls = ['http://blog.jobbole.com/all-posts/']

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

self.browser = webdriver.Chrome(executable\_path="D:/python\_packages/chromedriver.exe")

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

#scrapy允许程序在任何地方都可以加入一个信号, 可以做一个信号的映射，这里调用spider\_closed的信号，关闭爬虫的时候执行spider\_closed这个事件。scrpay的信号可以使我们在添加自己的处理时不用进入到scrapy本身的代码中.

dispatcher.connect(self.spider\_closed, signals.spider\_closed)

def spider\_closed(self, spider):

#当爬虫退出的时候关闭chrome

print("spider closed")

self.browser.quit()

def parse(self, response):

信号量允许我们在执行很多操作的时候不会侵入到scrapy的代码之中，这与django的很多设计思想是很相似的。

修改middlewares.py文件。

class JSPageMiddleware(object):

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

if spider.name == "jobbole":

# browser = webdriver.Chrome(executable\_path="D:/python\_packages/chromedriver.exe")

# 因为browser是在spider中初始化的, 所以这里就可以调用spider的browser了

spider.browser.get(request.url)

import time

time.sleep(3)

print("访问{0}".format(request.url))

from scrapy.http import HtmlResponse

return HtmlResponse(url=spider.browser.current\_url, body=spider.browser.page\_source, encoding="utf-8", request=request)

在jobbole.py中self.browser.quit()处，def parse pass处，添加断点，进行调试。当执行到self.browser.quit()时, 打印出来INFO: Closing spider (finished), 按F8继续执行代码, 就会关闭到浏览器.

最后，需要注意的是，整个scrapy都是一个异步的框架，它的性能是很高的，由于在middlewares.py中使用了chrome的diver，已经成为同步的请求，整体的性能会受到影响。可以重写scrapy downloader的方法把这个操作也变成异步的操作，要熟悉twisted和它的api。

github搜索scrapy downloader，打开scrapy-phantomjs-downloader进行学习。

https://github.com/flisky/scrapy-phantomjs-downloader

## 8-5 其余动态网页获取技术介绍-chrome无界面运行、scrapy-splash、selenium-grid, splinter

chrome的无界面运行

pip install pyvirtualdisplay

from pyvirtualdisplay import Display

# visible

display = Display(visible=0, size=(800, 600))

display.start()

# 只要加入以上几行，下面的操作都与之间的相同。

browser = webdriver.Chrome()

browser.get()

在linux下运行时可能会出错，

cmd=['Xvfb', '-help']

No such file or directory

查看xvfb是否已经安装

xvfb --help

sudo apt install xvfb

pip install xvfbwrapper

在windows下无法运行。

scrapy也提供了一种针对这种动态网页的方法scrapy splash，它实际上是自己运行了一个server，通过http的请求方式去执行js，它的执行效率会比使用chrome driver的方式要高。因为是运行在一个server上，scrapy splash也支持分布式。稳定性没有chrome高。

https://github.com/scrapy-plugins/scrapy-splash

https://pypi.python.org/pypi/scrapy-splash

也可以使用selenium的grid。也是支持分布式的。与scrapy splash方法类似，都是运行一个server，通过api的方式向它发送一个请求。

splinter也是一种可以通过scrapy操作浏览器的方案。

## 8-6 scrapy的暂停与重启

要想暂停爬虫，就要保存很多中间状态，要设置一个目录。用来保存这些中间状态的文件。不同的spider是不能共用同一个目录的，还要给每一个spider单独设置一个文件夹。想要暂停，在每次运行spider时都要事先设置一个不同的目录，每次运行的临时文件也不能保存在同一个目录下。

在当前项目下新建一个文件夹job\_info

进入虚拟环境，进入项目的根目录

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

-s表示 set的意思

这时不能使用pycharm来运行和调试了，因为spider的暂停命令是接收ctrl+C的命令，在pycharm中，停止debug或停止运行是直接结束进程的运行，程度不会接收到ctrl+C的命令，自然也就不能保存中间状态的文件。

linux中使用kill -f main.py

在linux中使用kill来结束某个进程时，这个进程会收到一个中断信号，有了这个中断信息，scrapy就可以做一些后期的处理。

kill -f -9 main.py则是强制kill某个进程，这时程序就不会收到中断信号。与通过windows任务管理器中结束进程时类似, 程序也不会接收到中断信号。

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

使用上面这个命令时windows和linux中都能接收到ctrl+C命令。

也可以在settings.py中设置JOBDIR。

JOBDIR=job\_info/001

每个spider都可以通过custom\_settings的方法来设置一个自己的设置。

可以在每个爬虫文件的spider类中设置一个JOBDIR

custom\_settings = {

"JOBDIR": "job\_info/001"

}

注意，爬虫开始运行后，想要暂停运行，必须只能按一下ctrl+C命令，按两下ctrl+C程序就会强制退出了。

按一下ctrl+C之后程序会继续执行，完成已经发送的请求，并把未发送的请求保存到本地文件中。

爬虫运行结束时，会在job\_info/001中保存有临时的文件。

request.seen就是已经访问过的url。

spider.state是spider的状态信息。

request.queue中保存的是队列信息

再次运行爬虫的命令与首次运行的命令相同

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

注意, 在连续按2次ctrl+C后会强制退出爬虫, 如果需要再次从头开始运行爬虫, 需要切换运行目录.

scrapy crawl lagou -s JOBDIR=job\_info/002

## 8-7 scrapy url去重原理

使用伯乐在线进行调试, 注释掉settings.py中的downloader middlewares, 注释掉jobbole.py中所有selenium相关的代码.

"C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\dupefilters.py"

去重的中间件在xlib/dupefilters.py中。RFPDupeFilter即是scrapy默认的去重和url过滤器. class RFPDupeFilter中有一个request\_seen函数，用来处理重复的url。

在core/scheduler.py即调度器函数中。qnqueue\_request函数来调用xlib.dupefilters.RFPDupeFilter中的request\_seen函数。每一个request都要放在调度器schedulter中。把request放进schedulter之前都会调用request\_seen方法，如果要自定义一个去除器，就要重写request\_seen函数。

可以在settings.py中设置DUPEFILTER\_DEBUG = True来打印去重信息

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)

## 8-8 scrapy telnet服务

使用telnet服务要先打开windowsr telnet服务。在打开或关闭windows功能中，勾选Telnet服务器和Telnet客户端。

scrapy中的telnet服务默认是开启的。

启动一个spider，scrapy crawl lagou

它会在127.0.0.1:6023上监听，这实际上就是一个telnet服务。

查看scrapy的官方文档，在Telnet终端，Telnet Console中查看说明。

在命令行或cmder中输入telnet localhost 6023

使用est()命令来查看当前的spider的状态。

连接telnet服务后就相当于一个python的终端，可以使用python终端的形式获取其中变量的值。

输入spider就会打印出spider这个类。

spider.settings

spider.settings["COOKIES\_ENABLED"]

https://doc.scrapy.org/en/latest/topics/telnetconsole.html#available-variables-in-the-telnet-console

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/topics/shell.html

scrapy的telnet源码位于scrapy/extensions/telnet.py中。

### Telnet终端(Telnet Console)

Scrapy提供了内置的telnet终端，以供检查，控制Scrapy运行的进程。 telnet仅仅是一个运行在Scrapy进程中的普通python终端。因此您可以在其中做任何事。

telnet终端是一个 [自带的Scrapy扩展](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/extensions.html#topics-extensions-ref) 。 该扩展默认为启用，不过您也可以关闭。 关于扩展的更多内容请参考 [Telnet console 扩展](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/extensions.html#topics-extensions-ref-telnetconsole) 。

#### 如何访问telnet终端

telnet终端监听设置中定义的 [**TELNETCONSOLE\_PORT**](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/telnetconsole.html#std:setting-TELNETCONSOLE_PORT) ，默认为 6023 。 访问telnet请输入:

telnet localhost 6023

>>>

Windows及大多数Linux发行版都自带了所需的telnet程序。

#### telnet终端中可用的变量

telnet仅仅是一个运行在Scrapy进程中的普通python终端。因此您可以做任何事情，甚至是导入新终端。

telnet为了方便提供了一些默认定义的变量:

|  |  |
| --- | --- |
| 快捷名称 | 描述 |
| crawler | Scrapy Crawler ([scrapy.crawler.Crawler](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/api.html#scrapy.crawler.Crawler) 对象) |
| engine | Crawler.engine属性 |
| spider | 当前激活的爬虫(spider) |
| slot | the engine slot |
| extensions | 扩展管理器(manager) (Crawler.extensions属性) |
| stats | 状态收集器 (Crawler.stats属性) |
| settings | Scrapy设置(setting)对象 (Crawler.settings属性) |
| est | 打印引擎状态的报告 |
| prefs | 针对内存调试 (参考 [调试内存溢出](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/leaks.html#topics-leaks)) |
| p | [pprint.pprint](https://docs.python.org/library/pprint.html#pprint.pprint) 函数的简写 |
| hpy | 针对内存调试 (参考 [调试内存溢出](https://scrapy-chs.readthedocs.io/zh_CN/1.0/topics/leaks.html#topics-leaks)) |

#### Telnet console usage examples

下面是使用telnet终端的一些例子:

##### 查看引擎状态

在终端中您可以使用Scrapy引擎的 est() 方法来快速查看状态:

telnet localhost 6023

>>> est()

Execution engine status

time()-engine.start\_time : 8.62972998619

engine.has\_capacity() : False

len(engine.downloader.active) : 16

engine.scraper.is\_idle() : False

engine.spider.name : followall

engine.spider\_is\_idle(engine.spider) : False

engine.slot.closing : False

len(engine.slot.inprogress) : 16

len(engine.slot.scheduler.dqs or []) : 0

len(engine.slot.scheduler.mqs) : 92

len(engine.scraper.slot.queue) : 0

len(engine.scraper.slot.active) : 0

engine.scraper.slot.active\_size : 0

engine.scraper.slot.itemproc\_size : 0

engine.scraper.slot.needs\_backout() : False

##### 暂停，恢复和停止Scrapy引擎

暂停:

telnet localhost 6023

>>> engine.pause()

>>>

恢复:

telnet localhost 6023

>>> engine.unpause()

>>>

停止:

telnet localhost 6023

>>> engine.stop()

Connection closed by foreign host.

#### Telnet终端信号

**scrapy.telnet.update\_telnet\_vars(telnet\_vars)**

在telnet终端开启前发送该信号。您可以挂载(hook up)该信号来添加，移除或更新 telnet本地命名空间可用的变量。 您可以通过在您的处理函数(handler)中更新 **telnet\_vars** 字典来实现该修改。

|  |  |
| --- | --- |
| **参数:** | **telnet\_vars** (dict) – telnet变量的字典 |

#### Telnet设定

以下是终端的一些设定:

##### TELNETCONSOLE\_PORT

Default: [6023, 6073]

telnet终端使用的端口范围。如果设为 None 或 0 ， 则动态分配端口。

##### TELNETCONSOLE\_HOST

默认: '127.0.0.1'

telnet终端监听的接口(interface)。

## 8-9 spider middleware 详解

scrapy有两种middleware，一种是download middleware，一种是spider middleware。

spider middleware是界于spider和engine之间的。

查看官方文档：

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

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/topics/spider-middleware.html

要在settings中设置并激活自定义的spider middleware

SPIDER\_MIDDLEWARES = {

'ArticleSpider.middlewares.ArticlespiderSpiderMiddleware': 543,

}

class ArticlespiderSpiderMiddleware(object):

@classmethod

def from\_crawler(cls, crawler):

s = cls()

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

return s

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

return None

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

for i in result:

yield i

middleware是有manager的，这个manager会统一管理download midddleware和spider middleware。其中的from\_crawler是会被middleware中的manager调用的。定义的所有middleware中，只要重写了from\_crawler这个函数，都会被middleware的manager调用。

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

在from\_crawler中会调用crawler，发送一个信号。这个信号会在signals.spider\_opened即爬虫打开的时候连接到spider\_opened的这个函数中。

def spider\_opened(self, spider):

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

spider\_opened只是打印了登陆的信息。可以在这个函数中添加自定义的信息。

spider middleware的处理函数是

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

return None

即spider的input处理函数。engine拿到response之后会发送给spider。process\_spider\_input函数就是处理这个过程的。可以在这里添加一些自定义的处理逻辑。scrapy架构图中的6。

process\_spider\_output这个函数处理的则是spider把response解析后的result送到engine这个过程。

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

for i in result:

yield i

在scrapy.spidermiddlewares中有一些scrapy官方自定义的middleware。

deepth.py中的DepthMiddleware可以监控spider爬取的层数即爬取的深度。

## 8-10 scrapy的数据收集stats collection

可以在spider中定义一些全局变量，来做数据收集

sccrpy也提供了数据收集的方法

https://doc.scrapy.org/en/latest/topics/stats.html

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/topics/stats.html

数据收集器对每个spider保持一个状态表。当spider启动时，该表自动打开，当spider关闭时，自动关闭。

如在spider运行时，记录已经发送了多少个request或返了多少response，在spider中，yield了多少的item等。

可以在statscollectors.py中可以查看相关的源码。

如要收集jobbole中所有的404的url以及404的页面数。

spider默认情况下只会处理200-300之间的页面，为了统计404的页面，可以设置handle\_httpstatus\_list。在httperror.py中可以查看相关的源码。

class JobboleSpider(scrapy.Spider):

name = 'jobbole'

allowed\_domains = ['blog.jobbole.com']

start\_urls = ['http://blog.jobbole.com/']

#收集jobbole中所有的404的url以及404的页面数。

handle\_httpstatus\_list = [404]

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

#把所有404页面的url保存到一个列表中，为什么不使用系统的数据收集器呢，因为系统的数据收集器只能采集到一些数字。

self.fail\_urls = []

def parse(self, response):

# 在parse函数中定义对404页面的处理

if response.status == 404:

self.fail\_urls.append(response.url)

#如果出现了404页面，就把failed\_url的数据增加1。increase\_value的缩写。

self.crawler.stats.inc\_value("failed\_url")

post\_urls = response.css("#archive div.floated-thumb .post-thumb a::attr(href)").extract()

但在爬取网站时看到的页面大部分都是200的页面，可以先把404改为200，看能否收集到数据。或者修改start\_urls中的url地址，使用一些错误的地址。

在settings中把spider\_middlewares注释掉，然后进行debug。

在self--> fail\_urls中就能看到错误的url地址。

self--> crawler --> stats --> failed\_url中就能看到错误的url的数量。

## 8-11 scrapy信号详解

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/topics/signals.html

信号是中间件扩展的桥梁，scrapy的所有组件和扩展都是基于scrapy的信号来设计的。

Scrapy使用信号来通知事情发生。您可以在您的Scrapy项目中捕捉一些信号(使用 extension)来完成额外的工作或添加额外的功能，扩展Scrapy。

middleware也可以看成是scrapy中扩展的一部分，只是这些middleware只是用来处理一些特定信号的扩展。

在jobbole.py中写过一个当爬虫退出时关闭chrome的功能，就是使用了scrapy的spider\_closed的信号。

dispatcher.connect(self.spider\_closed, signals.spider\_closed)

延迟的信号处理器(Deferred signal handlers)

有些信号支持从处理器返回 Twisted deferreds ，参考下边的 内置信号参考手册(Built-in signals reference) 来了解哪些支持。

在pipelines.py中定义的

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

query = self.dbpool.runInteraction(self.do\_insert, item)

query.addErrback(self.handle\_error, item, spider)

query就是一个deferred对象。

可以给deferred对象加上一个回调函数或错误处理的函数

继续来完善jobbole.py。增加当spider closed的时候的处理。

class JobboleSpider(scrapy.Spider):

name = 'jobbole'

allowed\_domains = ['blog.jobbole.com']

start\_urls = ['http://blog.jobbole.com/']

handle\_httpstatus\_list = [404]

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

self.fail\_urls = []

#使用dispatcher来连接handle\_spider\_closed函数和signals.spider\_closed的信号连接起来。

dispatcher.connect(self.handle\_spider\_closed, signals.spider\_closed)

#增加当spider结束的时候的把所有错误的url放在stats中。因为stats中没有列表，可以把所有错误的url拼接成一个字符串放在stats中。

def handle\_spider\_closed(self, spider, reason):

self.crawler.stats.set\_value("failed\_urls", ",".join(self.fail\_urls))

pass

def parse(self, response):

if response.status == 404:

self.fail\_urls.append(response.url)

self.crawler.stats.inc\_value("failed\_url")

post\_urls = response.css("#archive div.floated-thumb .post-thumb a::attr(href)").extract()

查看statscollectors.py中，在close\_spider函数中也进行了信号的绑定，可以把stats中的所有内容都打印出来。

把所有错误的url放在stats中，运行spider，爬虫运行结束时在运行窗口中就会打印出来这个failed\_urls的值。

运行debug，在jobbole.py中，handle\_spider\_closed中的self就能够看到self中传递的内容。

## 8-12 scrapy扩展开发 (13:16)

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

http://scrapy-chs.readthedocs.io/zh\_CN/1.0/topics/extensions.html

scrapy/downloadermiddlewares

downloader中间件中只有以下三个函数可以重写。

process\_request, process\_response，process\_exception

spider中间件只有以下四个函数可以重写。

process\_spider\_input, process\_spider\_output, process\_spider\_exception, process\_spider\_requests

中间件和扩展的关系：中间件都是一些缩减版的扩展。

系统提供了3个manager来管理中间件和pipeline:

与downlaodermiddleware对应的downlaodermiddleware manager

与spidermiddleware对应的spidermiddleware manager

还有一个与pipeline对应的manager

以上3个manager都属于extention manager。

downloader middleware，spidermiddleware和pipeline都可以看成是缩减版的扩展。

middleware中的函数是和信号量绑定的。

所有extention的实现机制都是通过信号量来实现的。

要使得扩展可用，需要在settings中把它添加到Scrapy的EXTENSIONS配置中。

EXTENSIONS = {

'scrapy.extensions.corestats.CoreStats': 500,

'scrapy.telnet.TelnetConsole': 500,

}

每个扩展是一个单一的Python class.Scrapy扩展(包括middlewares和pipelines)的主要入口是 from\_crawler 类方法，它接收一个Crawler 类的实例.通过这个对象访问settings，signals，stats，控制爬虫的行为。

通常来说，扩展关联到 signals 并执行它们触发的任务。

扩展并不像middleware和pipeline中只能定义有限个特定名称的函数，而是可以编写任意的函数，然后把函数与信号进行绑定，在信号被触发的时候就会调用这些函数进行处理。

官方提供了一些扩展scrpy/extensions/

corestats.py记录了一些重要的统计信息。

import datetime

from scrapy import signals

class CoreStats(object):

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

self.stats = stats

@classmethod

def from\_crawler(cls, crawler):

o = cls(crawler.stats)

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

crawler.signals.connect(o.spider\_closed, signal=signals.spider\_closed)

crawler.signals.connect(o.item\_scraped, signal=signals.item\_scraped)

crawler.signals.connect(o.item\_dropped, signal=signals.item\_dropped)

crawler.signals.connect(o.response\_received, signal=signals.response\_received)

return o

def spider\_opened(self, spider):

self.stats.set\_value('start\_time', datetime.datetime.utcnow(), spider=spider)

def spider\_closed(self, spider, reason):

self.stats.set\_value('finish\_time', datetime.datetime.utcnow(), spider=spider)

self.stats.set\_value('finish\_reason', reason, spider=spider)

def item\_scraped(self, item, spider):

self.stats.inc\_value('item\_scraped\_count', spider=spider)

def response\_received(self, spider):

self.stats.inc\_value('response\_received\_count', spider=spider)

def item\_dropped(self, item, spider, exception):

reason = exception.\_\_class\_\_.\_\_name\_\_

self.stats.inc\_value('item\_dropped\_count', spider=spider)

self.stats.inc\_value('item\_dropped\_reasons\_count/%s' % reason, spider=spider)

在from\_crawler中绑定了很多的信号，crawler.signals.connect(o.spider\_opened, signal=signals.spider\_opened)，在spider open的时候会调用spider\_opened的函数，设置start\_time为系统当前的utc时间。

同时，在spider\_closed的时候会设置finish\_time和finish\_reason。

当item被scraped的时候，会把item\_scraped\_count加1。

response\_received，即engine从downoader中接收到response的时候，会把response\_received\_count加1。

memusage.py

监控内存使用的扩展，关键要先找到函数的入口。

class MemoryUsage(object):

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

if not crawler.settings.getbool('MEMUSAGE\_ENABLED'):

raise NotConfigured

try:

# stdlib's resource module is only available on unix platforms.

self.resource = import\_module('resource')

except ImportError:

raise NotConfigured

self.crawler = crawler

self.warned = False

self.notify\_mails = crawler.settings.getlist('MEMUSAGE\_NOTIFY\_MAIL')

self.limit = crawler.settings.getint('MEMUSAGE\_LIMIT\_MB')\*1024\*1024

self.warning = crawler.settings.getint('MEMUSAGE\_WARNING\_MB')\*1024\*1024

self.check\_interval = crawler.settings.getfloat('MEMUSAGE\_CHECK\_INTERVAL\_SECONDS')

self.mail = MailSender.from\_settings(crawler.settings)

crawler.signals.connect(self.engine\_started, signal=signals.engine\_started)

crawler.signals.connect(self.engine\_stopped, signal=signals.engine\_stopped)

先去settings中查找是MEMUSAGE\_ENABLED是否为True，如果为True，则进一步查找详细的设置，如发送邮件notify\_mails，内存使用的限制limit，多少秒检查一次check\_interval。

在\_\_init\_\_中绑定了两个信号，从这里入口就容易理解。

crawler.signals.connect(self.engine\_started, signal=signals.engine\_started)

crawler.signals.connect(self.engine\_stopped, signal=signals.engine\_stopped)

其它代码都是围绕着这两个信号进行处理的。

整个extention的思路就是通过信号来绑定函数，在函数中可以根据需要任意定义处理的逻辑。只需要知道在什么信号触发的时候要去运行哪些函数，完成什么样的处理即可。

# 第9章 scrapy-redis分布式爬虫

## 9-1 分布式爬虫要点

分布式爬虫的优点

充分利用多机器的带宽加速爬取

充分利用多机器的ip加速爬取速度，网站的反爬机器，会封禁爬取过快的ip

分布式需要解决的问题

reqeust队列集中管理

去重集中管理

## 9-2 redis基础知识

redis实际上是一个内容数据库，把所有的数据放在内存中。速度很快。

很多系统都把redis当成缓存系统。

### 9.2.1 Redis安装

https://github.com/antirez/redis

**linux**

**命令行安装**

sudo apt install redis-server redis\_client

sudo apt install redis-tools

可以把redis注册为一个开机启动的服务, 每次重启电脑可以自动启动redis server

**源码安装**

Installation

Download, extract and compile Redis with:

$ wget http://download.redis.io/releases/redis-3.2.9.tar.gz

$ tar xzf redis-3.2.9.tar.gz

$ cd redis-3.2.9

$ make

The binaries that are now compiled are available in the src directory. Run Redis with:

$ src/redis-server

You can interact with Redis using the built-in client:

$ src/redis-cli

redis> set foo bar

OK

redis> get foo

优点，安装了redis-cli，缺点，每次重启电脑都要手动启动redis服务。

**windows安装**

<https://github.com/MSOpenTech/redis/releases>

没有redis-cli客户端, 安装的redis server是以服务的形式自动启动的.

搜索redis for windows，githup网站。

# https://github.com/ServiceStack/redis-windows

<https://github.com/ServiceStack/redis-windows/blob/master/downloads/redis-latest.zip>

既有客户端又有服务器

下载后解压到一文件夹中，在命令行中运行redis-server

如果之前使用过msi文件安装过redis，会出现TCP listening socket \*:6379: bind: No such file or directory的错误。

这是因为之前安装redis后，服务已经启动，6379端口已经被占用，再次启动就会失败。

在服务中停止redis服务就可以了。

再打开一个命令行窗口，输入redis-cli就可以连接到redis数据库了。

http://redisdoc.com/

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

### 9.2.2 Reids数据类型

字符串

散列/哈希(类似python的字典)

列表

集合

可排序集合

### 9.2.3 redis命令

<http://redisdoc.com/>

字符串命令

set course\_name "scrapy-redis"

get course\_name

getrange 获取字符串的子串

getrange course\_name start end

获取course\_name的子串，start和end分别是子串的开始位置和结束位置

getrange course\_name 2 6

strlen 获得字符串的长度

strlen course\_name

set mykey 2

incr/decr mykey

对字符串进行加1或减1的操作，此时的变量mykey必须为int或者字符串格式的int

set course\_count "3"

incr course\_count

get course\_count

append course\_name ".cn"

更多命令:

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

哈希命令

hset myhash name "imooc.com"

myhash类似于字典的变量名，name类似于字典的key，imooc.com为字典的value

hset course\_dict bobby "python scrpy"

hget course\_dict bobby

hgetall course\_dict # 返回哈希的key 和value

hexists myhash name # 检查myhash这个哈希中 name的key是否存在

myhash表示字典的变量名，name表示字典的key。

hexists course\_dict bobby

删除键

hdel course\_dict bobby

# 返回所有的keys

hkeys course\_dict --> bobby

# 返回所有的values

hvals course\_dict --> python scrapy

HMSET runoobkey name "redis tutorial" description "redis basic commands for caching" likes 20 visitors 23000

**列表命令**

lpush/rpush mylist "imooc.com"

lpush从列表的左侧插入，rpush从列表的右侧插入

lpush imooc\_courses "scrapy"

lpush imooc\_courses "django"

lrange imooc\_courses 0 10

blpop/brpop key1 [key2] timeout

blpop从列表的左侧删除一个元素

brpop从列表的右侧删除一个元素

b表示阻塞，timeout为时间，单位为秒。如果列表为空，就会等待timeout中设置的时间, 如果在等待的时间内列表还为空, 就会返回nil, 如果在等待的时间内列表中出现元素, 就会立即执行操作并返回。如果列表不为空，就立即执行操作。

blpop imooc\_course 3 # 列表为空, 等待3s会返回nil

blpop imooc\_courses 3 # 列表不为空, 不会等待

lpop/rpop key 从列表左侧或右侧删除1个元素, 这里没有b的参数, 不会再等待，而是直接返回

llen key

列表的长度

llen imooc\_courses

lindex key index

取列表中的第index个元素

lindex imooc\_courses 1

**集合命令**

sadd course\_set "django"

sadd course\_set "django"

添加元素, 返回1表示操作成功, 返回0表示数据已经存在, 操作失败.

scard key

集合中元素的个数

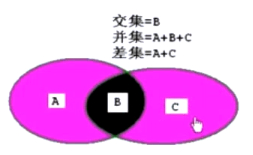
scard course\_set

sadd course\_set "scrapy"

scard course\_set

sdiff key1 [key2]

key1和key2表示从key1这个set中减去key1和key2的交集



sadd course\_set2 "scrapy"

sadd course\_set2 "redis"

sdiff course\_set course\_set2

sdiff course\_set2 course\_set

sinter key1 [key2]

求key1和key2的交集

sinter course\_set2 course\_set

spop key

spop是随机删除并返回集合中的元素

spop course\_set

srandmember key member # 从set中随机的获取member个元素

srandmember key随机获取一个元素

smembers获取set中的所有元素

**可排序集合命令**

zadd myset 0 "projectsedu.com" 5 "imooc.com" 8 "163.com"

每个元素都添加一个score的值。

zadd zcourses\_set 0 "django" 2 "python" 5 "scrapy" 10 "scrapy-redis"

zrangebyscore myset 1 100

获取score分数在1到100之间的所有元素

zrangebyscore zcourses\_set 0 5

zcount myset 1 100

获取score在1到100之间的元素个数

zcount zcourses\_set 0 5

## 9-4 scrapy-redis编写分布式爬虫代码

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

进入虚拟环境

pip install scrapy-redis

Usage

Use the following settings in your project:

1. 在settings.py中进行设置

# Enables scheduling storing requests queue in redis.

#调度器

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# Ensure all spiders share same duplicates filter through redis.

#去重的方法

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

# Default requests serializer is pickle, but it can be changed to any module

# with loads and dumps functions. Note that pickle is not compatible between

# python versions.

# Caveat: In python 3.x, the serializer must return strings keys and support

# bytes as values. Because of this reason the json or msgpack module will not

# work by default. In python 2.x there is no such issue and you can use

# 'json' or 'msgpack' as serializers.

#SCHEDULER\_SERIALIZER = "scrapy\_redis.picklecompat"

# Don't cleanup redis queues, allows to pause/resume crawls.

#SCHEDULER\_PERSIST = True

# Schedule requests using a priority queue. (default)

#SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.PriorityQueue'

# Alternative queues.

#SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.FifoQueue'

#SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.LifoQueue'

# Max idle time to prevent the spider from being closed when distributed crawling.

# This only works if queue class is SpiderQueue or SpiderStack,

# and may also block the same time when your spider start at the first time (because the queue is empty).

#SCHEDULER\_IDLE\_BEFORE\_CLOSE = 10

# Store scraped item in redis for post-processing.

# item pipelines要添加redis的pipeline

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

# The item pipeline serializes and stores the items in this redis key.

#REDIS\_ITEMS\_KEY = '%(spider)s:items'

# The items serializer is by default ScrapyJSONEncoder. You can use any

# importable path to a callable object.

#REDIS\_ITEMS\_SERIALIZER = 'json.dumps'

# Specify the host and port to use when connecting to Redis (optional).

#REDIS\_HOST = 'localhost'

#REDIS\_PORT = 6379

# Specify the full Redis URL for connecting (optional).

# If set, this takes precedence over the REDIS\_HOST and REDIS\_PORT settings.

#REDIS\_URL = 'redis://user:pass@hostname:9001'

# Custom redis client parameters (i.e.: socket timeout, etc.)

#REDIS\_PARAMS = {}

# Use custom redis client class.

#REDIS\_PARAMS['redis\_cls'] = 'myproject.RedisClient'

# If True, it uses redis' ``SPOP`` operation. You have to use the ``SADD``

# command to add URLs to the redis queue. This could be useful if you

# want to avoid duplicates in your start urls list and the order of

# processing does not matter.

#REDIS\_START\_URLS\_AS\_SET = False

# Default start urls key for RedisSpider and RedisCrawlSpider.

#REDIS\_START\_URLS\_KEY = '%(name)s:start\_urls'

# Use other encoding than utf-8 for redis.

#REDIS\_ENCODING = 'latin1'

2. Running the example project

This example illustrates how to share a spider's requests queue across multiple spider instances, highly suitable for broad crawls.

Setup scrapy\_redis package in your PYTHONPATH

Run the crawler for first time then stop it:

$ cd example-project

$ scrapy crawl dmoz

... [dmoz] ...

^C

Run the crawler again to resume stopped crawling:

$ scrapy crawl dmoz

... [dmoz] DEBUG: Resuming crawl (9019 requests scheduled)

Start one or more additional scrapy crawlers:

$ scrapy crawl dmoz

... [dmoz] DEBUG: Resuming crawl (8712 requests scheduled)

Start one or more post-processing workers:

$ python process\_items.py dmoz:items -v

...

Processing: Kilani Giftware (http://www.dmoz.org/Computers/Shopping/Gifts/)

Processing: NinjaGizmos.com (http://www.dmoz.org/Computers/Shopping/Gifts/)

...

3. 写一个基于Redis的spider

Feeding a Spider from Redis

The class scrapy\_redis.spiders.RedisSpider enables a spider to read the urls from redis. The urls in the redis queue will be processed one after another, if the first request yields more requests, the spider will process those requests before fetching another url from redis.

For example, create a file myspider.py with the code below:

from scrapy\_redis.spiders import RedisSpider

# 要继承于RedisSpider

class MySpider(RedisSpider):

name = 'myspider'

def parse(self, response):

# do stuff

pass

4. run the spider:

#启动时不再写爬虫名，而是使用spider的文件来启动

scrapy runspider myspider.py

5. push urls to redis:

#启动spider之后，还要往队列中放入start\_urls，爬虫才会开始爬取数据

redis-cli lpush myspider:start\_urls http://google.com

Note

These spiders rely on the spider idle signal to fetch start urls, hence it may have a few seconds of delay between the time you push a new url and the spider starts crawling it.

### 新建scrapy-redis项目

新建项目时还是新建的scrapy的项目，只是把scrapy项目中的settings设置为redis的，爬虫以scrapy redis的格式进行书写。

cd D:\Daivd\Desktop\imooc\_projects

workon进入虚拟环境

scrapy startproject ScrapyRedisTest

使用pycharm来打开此项目

然后把reids官方github的项目复制下来

可以使用TortoiseGit

https://tortoisegit.org/

在scrapy-redis/example-project/example/spiders中有两种类型的spider

mycrawler\_redis.py是基于crawl spider实现的

myspider\_redis.py是基于spider来实现的

复制scrapy-reids/src/scrapy-redis文件夹复制到项目的根目录中。这样, 新建的爬虫项目中导入的内容就是从项目根目录中的scrapy-redis中导入的了. 就不再是使用的系统安装的scrapy-redis了.

新建/spiders/jobbole.py

from scrapy\_redis.spiders import RedisSpider

#要继承于RedisSpider

class JobboleSpider(RedisSpider):

name = 'jobbole'

allowed\_domains = ["blog.jobbole.com"]

redis\_key = 'jobbole:start\_urls'

def parse(self, response):

# do stuff

pass

redis scrapy的爬虫只有以上部分的内容与scrapy的不同，其它的内容都可以直接复制过来。start\_urls不需要了。只要把allowed\_domains复制过来就可以了。可以删除所有的parse\_detail函数的内容。

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

**from** **scrapy.http** **import** Request

**from** **urllib** **import** parse

**from** **scrapy\_redis.spiders** **import** RedisSpider

**class** **JobboleSpider**(RedisSpider):

name = 'jobbole'

allowed\_domains = ["blog.jobbole.com"]

redis\_key = 'jobbole:start\_urls'

*# 收集伯乐在线所有404的url以及404页面数*

handle\_httpstatus\_list = [404]

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

*"""*

*1. 获取文章列表页中的文章url并交给scrapy下载后并进行解析*

*2. 获取下一页的url并交给scrapy进行下载， 下载完成后交给parse*

*"""*

*# 解析列表页中的所有文章url并交给scrapy下载后并进行解析*

**if** response.status == 404:

self.fail\_urls.append(response.url)

self.crawler.stats.inc\_value("failed\_url")

post\_nodes = response.css("#archive .floated-thumb .post-thumb a")

**for** post\_node **in** post\_nodes:

image\_url = post\_node.css("img::attr(src)").extract\_first("")

post\_url = post\_node.css("::attr(href)").extract\_first("")

**yield** Request(url=parse.urljoin(response.url, post\_url), meta={"front\_image\_url": image\_url}, callback=self.parse\_detail)

*# 提取下一页并交给scrapy进行下载*

next\_url = response.css(".next.page-numbers::attr(href)").extract\_first("")

**if** next\_url:

**yield** Request(url=parse.urljoin(response.url, post\_url), callback=self.parse)

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

**pass**

根据scrapy-redis的使用说明设置settings.py

BOT\_NAME = 'ScrapyRedisTest'

SPIDER\_MODULES = ['ScrapyRedisTest.spiders']

NEWSPIDER\_MODULE = 'ScrapyRedisTest.spiders'

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = False

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

复制main.py文件, 放在项目的根目录中.

在pycharm中设置项目的project interpreter为之前创建项目时的虚拟环境

在parse函数中的if response.status处打上断点，在for post\_nodes中的yield Request处打上断点，parse\_detail函数中的pass处打上断点，进行调试, 但由于没有start\_urls, 所以程序会一直处于等待状态中.

进入redis-cli

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

需要把下一个request放到redis的队列中去，才能查看reids中队列中的内容，但一旦把Request放到reids的队列中，scrapy就会把这个Request pop出来。需要把scrapy\_redis/scheduler.py中的next\_request函数中的request = self.queue.pop这里也打上断点。这样就不会把request pop出来了，就可以看到redis中数据的情况了。

在redis-cli中输入keys \*查看所有的变量

其中有jobbole.dupefilter和jobbole.requests的变量。

凡是在jobbole.py中使用Request发送的request都会进入到scheduler.py中的enqueue\_request中。enqueue\_request把request都放在jobbole:requests的变量中。并且调用jobbole:dupefilter来进行去重。

type jobbole:requests查看其类型，为zset，也就是说其中的request是有一个优先级的排序的。

type jobbole:dupefilter --> set

zrangebyscore jobbole:requests 0 100

request在内存中是一个类，redis是不能够保存类的。使用picklecompat.py来把request这个类序列化为一个字符串，保存到redis数据库中。

## 9-5 scrapy源码解析-connection.py、defaults.py- (11:05)

## 9-6 scrapy-redis源码剖析-dupefilter.py- (05:29)

## 9-7 scrapy-redis源码剖析- pipelines.py、 queue.py- (10:41)

## 9-8 scrapy-redis源码分析- scheduler.py、spider.py- (11:52)

## 9-9 集成bloomfilter到scrapy-redis中 (19:30)

1个bit位

10101001

使用电压的高低来判断, 高1, 低0

1byte = 8 bit

"a" ===> 1byte

"a" utf8 ===> 1byte

"中" utf8 ===> 3bytes

set("abc") 使用1bit来进行去重的话会节省大量的空间.

在scrapyredisTest/scrapyredisTest下面新建一个utils的python package，新建一个bloomfilter.py

把从https://github.com/liyaopinner/BloomFilter\_imooc

下载的py\_bloomfilter.py中的内容全部复制到bloomfilter中即可。

依赖mmh3，安装依赖包：

pip install mmh3

if \_\_name\_\_ == "\_\_main\_\_":

start = time.time()

bf = PyBloomFilter(conn=conn)

bf.add('www.jobbole.com')

bf.add('www.zhihu.com')

print(bf.is\_exist('www.zhihu.com'))

print(bf.is\_exist('www.lagou.com'))

如何把bloomfilter集成到redis的去重器中。

bloomfilter的用法是bf.is\_exist，但先要生成一个bf的对象。但不可能在每次调用request的时候都会成生一个bf对象。所以要在\_\_init\_\_中生成一个全局的bf对象。

修改dupefilter.py，在\_\_init\_\_中添加如下内容

from ScrapyRedisTest.utils.bloomfilter import conn, PyBloomFilter

class RFPDupeFilter(BaseDupefilter):

def \_\_init\_\_(self, server, key, debug=False):

...

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

修改request\_seen函数，使用bf的去重方法。

def request\_seen(self, request):

fp = self.request\_fingerprint(request)

if self.bf.is\_exist(fp):

return True

else:

self.bf.add(fp)

return False

# added = self.server.sadd(self.key, fp)

# return added == 0

先要清空数据，在redis-cli中输入

flushall

lpush一个初始的url

在if self.bf.is\_exist()处加上断点，进行调试。只要数据进入到了dupefilter.py中的bf，就可以停止了。

再次lpush同样的初始url。查看返回的值是True还是False

# 第10章 elasticsearch搜索引擎的使用

## 10-1 elasticsearch介绍 (18:21)

我们建立一个网站或者程序，希望添加搜索功能，发现搜索工作很难：

1 我们希望搜索解决方案要高效

2 我们希望零配置和完全免费的搜索方案

3 我们希望能够简单的通过json和http与搜索引擎交互

4 我们希望我们的搜索服务器稳定

5 我们希望能够简单的将一台服务器扩展到上百台

http://www.elastic.co/use-cases

关系数据库搜索缺点

1 无法打分，无法排序

2 无分布式

3 无法解析搜索请求

4 效率低

5 分词

no sql not only sql

一个文档数据库

mongodb redis elasticsearch

在某些情况之下，甚至可以将elasticsearch当成是mongodb来使用，如不会去频繁的update数据库，因为elasticsearch数据库的更新操作是比较慢的，会远低于mongodb数据库。而mongodb的更新又慢于关系数据库的更新。但是mongodb的插入和查询操作在大多数的情况下都是优于mysql的。elasticsearch虽然可以当成mongodb来使用，但它又不能取代mysql或nosql数据库。

## 10-2 elasticsearch安装 (13:24)

1 安装elasticsearch-rtf

首先要安装java-jdk

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Java SE Development Kit 8 Downloads

安装完成后可以在cmder中输入java -version来查看java的版本

在有的ubuntu系统中会默认安装open-jdk 1.7的java版本，最好卸载掉open-jdk的版本。

https://www.elastic.co/cn/downloads/elasticsearch

elasticsearch需要安装很多插件以更好的在中文的环境下使用。插件的安装过程是比较繁琐的。这里使用elasticsearch-rtf

https://github.com/medcl/elasticsearch-rtf

/bin/中存在的是可执行文件

/config/elasticsearch.yml是配置文件

/data/中保存的是elasticsearch的数据文件

/lib/是elasticsearch依赖的jar包

logs是ES的日志文件

modules中是ES的模块

plugins是安装的插件

在bin文件夹中打开命令行，直接运行elasticsearch.bat就能启动elastisearch服务了。

会监听127.0.0.1:9300端口，这是java的端口。

可以在浏览器中通过127.0.0.1:9200访问elasticsearch

只要输出以下内容就表示elasticsearch正确启动了。

{

name: "a1fDLXl",

cluster\_name: "elasticsearch",

cluster\_uuid: "rdRQoOF8T822umhE1iiing",

version: {

number: "5.1.1",

build\_hash: "5395e21",

build\_date: "2016-12-06T12:36:15.409Z",

build\_snapshot: false,

lucene\_version: "6.3.0"

},

tagline: "You Know, for Search"

}

## 10-3 elasticsearch-head插件以及kibana的安装 (24:09)

2 head插件和kibana的安装

安装了mysql之后，可以通过navicat来查看数据库中的数据。head是一个基于浏览器的插件，可以完成类似于navicat的功能，可以执行ES的查询语句，对其中的数据进行查看。

https://github.com/mobz/elasticsearch-head

http://mobz.github.io/elasticsearch-head/

Running

There are multiple ways of running elasticsearch-head.

Running with built in server

git clone git://github.com/mobz/elasticsearch-head.git

cd elasticsearch-head

npm install

npm run start

open http://localhost:9100/

npm是node.js中提出的包管理器，类似于java中的maven，或者类似于python中的pip。可以使用类似的命令来安装node.js的依赖文件

https://nodejs.org/zh-cn/

安装nodejs会自动安装npm包管理器。

在命令行中输入npm来验证安装是否生效。

npm类似于java中的maven，在下载时会从国外的中央仓库中下载，速度会非常慢

使用cnpm来安装。cnpm是淘宝NPM镜像。

http://npm.taobao.org/

使用

npm install -g cnpm --registry=https://registry.npm.taobao.org

来安装cnpm

安装模块

从 registry.npm.taobao.org 安装所有模块. 当安装的时候发现安装的模块还没有同步过来, 淘宝 NPM 会自动在后台进行同步, 并且会让你从官方 NPM registry.npmjs.org 进行安装. 下次你再安装这个模块的时候, 就会直接从 淘宝 NPM 安装了.

$ cnpm install [name]

同步模块

直接通过 sync 命令马上同步一个模块, 只有 cnpm 命令行才有此功能:

$ cnpm sync connect

当然, 你可以直接通过 web 方式来同步: /sync/connect

$ open https://npm.taobao.org/sync/connect

其它命令

支持 npm 除了 publish 之外的所有命令, 如:

$ cnpm info connect

安装完cnpm后，就能够使用cnpm来代替npm来执行操作了。

cd elasticsearch-head

cnpm install

会把所有的第三方包放在node\_modules这个文件夹中。

cnpm run start

elasticsearch-head@0.0.0 start D:\python\_packages\elasticsearch-head-master

> grunt server

Running "connect:server" (connect) task

Waiting forever...

Started connect web server on http://localhost:9100

open http://localhost:9100/

使用浏览器访问http://localhost:9100/

会出现错误提示

Connectcluster health: not connected

但是如果直接在浏览器中使用localhost:9200访问elasticsearch，是成功的。

但是使用head这个插件来访问却提示没有连接，这是因为elasticserach为了安全性在默认情况下是不允许使用第三方的服务来进行连接的。head可以看成是一个代理，使用这个代理来连接就出现了错误。

想要使用代理来访问ES，必须要进行设置

打开

D:\python\_packages\elasticsearch-rtf-master\config\elasticsearch.yml

这个是ES的配置文件，添加以下代码

http.cors.enabled: true

http.cors.allow-origin: "\*"

http.cors.allow-methods: OPTIONS, HEAD, GET, POST, PUT, DELETE

http.cors.allow-headers: "X-Requested-With, Content-Type, Content-Length, X-User"

然后重新启动ES

再次访问http://localhost:9100/

在其中进行的操作实际上还是head调用了ES提供的rees接口来完成的，把这个接口用界面的形式显示出来了。

kibana插件的安装

kibana实际上是一个html页面，它完成了和ES的交互，使ES的操作变得更加简单。

主要是使用kibana中的sense功能。

https://www.elastic.co/products/kibana

要下载与使用的ES对应的5.1.1或5.1.2版本。

使用命令行打开

kibana-5.1.2-windows-x86\bin\kibana.bat

通过访问

http://localhost:5601

来使用kibana

在Dev Tools中点击Get to work

在Console中会把以前运行过的ES命令都保存着。

## 10-4 elasticsearch的基本概念 (12:15)

1 集群：一个或者多个节点组织在一起。 ES是一个分布式

2 节点：一个节点是集群中的一个服务器，由一个名字来标识，默认是一个随机的漫威角色的名字。

3 分片：将索引划分为多份的能力，允许水平分割和扩展，多个分片响应请求，提高性能和吞吐量

ES可以把一个索引分成多份放在不同的分片中，在查询时会自动把查询路由到所在的一个或多个分片上。

4 副本：创建分片的一份或多份的能力，在一个节点失败其余节点可以顶上。

分片是把一个过大的数据分成多份放在不同的地方，副本是多少份复制的数据。一个数据的副本是2，它就会保存2份。

ES是一个搜索引擎，但这并不是说ES可以对其它数据库中的数据进行查询，ES为了达到自己搜索的目的，数据是保存在ES自己的服务器中的。它不是一个中间库，而是集成了数据保存和数据分析功能的一个搜索引擎。为了存储数据库，ES就要具有同其它数据库相同的功能和字段。

ES与mysql中对应的概念

ES mysql

index(索引) 数据库

type(类型) 表

documents(文档) 行

fields 列

HTTP方法

HTTP1.0定义了三种请求方式：GET，POST和HEAD方法

HTTP1.1新增了五种方法：OPTIONS, PUT, DELETE, TRACE, CONNECT

方法 描述

GET 请求指定的页面信息，并返回实体主体。

POST 向指定的资源提交数据库进行处理请求。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改

PUT 向服务器传闯祸的数据取代指定的文档的内容。

DELETE 请求服务器删除指定的页面

使用post方法也能完成delete的操作。服务器可以根据post中的参数来做相应的操作。如post一个opration参数，服务器判断operation，如果是delete的话就删除相应的数据。

通过这些http请求方法，才能实现rees的接口，而ES又是通过rees的接口来完成的。

## 10-5 倒排索引 (11:24)

目前搜索引擎中底层的索引都是使用倒排索引。倒排索引也是搜索引擎区分于关系型数据库或nosql数据库的核心。

倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值 的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。

文件A：通过python django搭建网站

文件B：python scrapy爬取网站数据

文件C：scrapy-redis分布式爬虫

这里的文件A，文件B和文件C都可以看成是ES数据库中的documents(文档)，也就是mysql中的行的意思。

如果要查询哪些文档包含有python这个关键词。按照正常的查询方式，要完成这个查询，需要对每个文档进行遍历查询，当文档数很多时，如达到上亿的级别时，第一个查询都要遍历上亿次，这肯定是不现实的。

而使用倒排索引的方式，在文件进行存储之前，都对每个文档进行遍历分析。如对下面这句话：

python写各大聊天系统屏蔽脏话功能原理

使用分词，进行倒排索引处理之后，就有了下面的关键词与文章的对应关系。如python这个关键词在文章1和文章3中存在。这样把整句话都进行分词的处理并以下面的结构进行存储。类似于dict的结构，存储的key就是关键词，value就是链表，链表中存放的是关键词所在的文章。

关键词 文章

python 文章1，文章3

聊天 文章2

系统 文章3，文章4

屏蔽脏话 文章5

功能原理 文章6，文章7，文章8

还有一个问题是关键词在某篇文章中出现的频次。如python这个关键词，如果它在文章1中出现了30次，在文章3中只出现了5次，也必须要把关键词在某篇文章中出现的频次记录下来，后期才能实现搜索的打分功能。在某一篇文章中此关键词出现的频次越高，它的权重就越高，按照上面的结构就无法存储关键词的权重。倒排索引还有更加精确的索引结构。

关键词 倒排列表

python (文章1,<2,10>,2)

聊天 (文章2,<12,25,100>,3)

系统 (文章3,<10>,1)

屏蔽脏话 (文章5,<50,60>,2)

功能原理 (文章6,<56,57,58>,3)

在倒排列表中，文章1表示python在文章1中是关键词，<>中的数字表示python这个关键词在文章1中出现的位置。2表示出现的次数。次数称为词频TF，对ES的打分来说是很关键的因素。

可以通过了解TF-IDF的原理来了解ES的打分的具体实现方法。

倒排索引待解决的问题

1 大小写转换问题，如python和PYTHON应该视为同一个词

2 词干抽取，looking和look应该处理为一个词

3 分词，如"屏蔽系统"，应该分词为"屏蔽"、"系统"还是应该为"屏蔽系统"

4 倒排索引文件过大，要进行压缩编码

## 10-6 elasticsearch 基本的索引和文档CRUD操作 (18:44)

CRUD操作即ES的增删改查操作

后面的操作都是在kibana > DevTools > Console中完成的

#索引初始化操作，索引类似于关系数据库中的数据库，可以看成是创建一个数据库，这里使用http的PUT方法。要指定索引的名称lagou，有一个settings，settings中有一个index，index中有两个值，number\_of\_shared是指索引的分片的数量，number\_of\_replicas是索引的副本的数量。

#指定分片和副本的数量

#shards一旦设置就不能修改

PUT lagou

{

"settings": {

"index":{

"number\_of\_shards":5,

"number\_of\_replicas":1

}

}

}

点击运行。出现以下提示，就表示创建成功了。

{

"acknowledged": true,

"shards\_acknowledged": true

}

使用head插件，在浏览器中输入http://localhost:9100/，查看ES数据库，其中就多出了一个lagou的数据库

lagou数据库中的Unassigned表示没有给lagou指定索引的分片。

可以在info中看到lagou的索引的信息。

也可以在head中新建索引

在Indices标签中点击new index就可以新建索引了。输入索引名称jobbole，分片数5和副本数1即可。

也可以在head中Overview标签下每个索引的actions中选择delete来删除某个索引。

在head中和kibana中进行的操作是等价的。在kibana中使用的是rees的接口，可以理解成关系数据库中的sql语句。

一旦索引创建完成，shards的数量是不能够修改的。replicas的数量是可以修改的。

获取索引settings的方法

可以使用GET lagou/\_settings来获取lagou这个索引的settings信息。点运行，就能看到lagou的settings信息。

使用GET \_all/\_settings可以获取所有索引的settings。

使用GET .kibana,lagou/\_settings来获取指定的索引的settings信息。注意这里的逗号后面不能有空格。

使用GET \_settings了可以获取所有索引的settings。

更新索引settings的方法

PUT lagou/\_settings

{

"number\_of\_replicas":2

}

获取所有的索引信息

GET \_all

获取lagou的索引

GET lagou

保存一篇文章或一个职位信息到索引中

在关系数据库中，必须要在数据库中创建数据表，才能向表中存储数据。

但在包括ES在内的很多nosql的非关系型数据库中，不用设置type，即不用创建表就可以存储数据。

PUT lagou/job/1

{

"title":"python分布式爬虫开发",

"salary\_min":15000,

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园"

},

"publish\_date":"2017-4-16",

"comments":15

}

job即索引的type，即表。1表示数据的id，即主键。

运行后在以下信息中出现successful:1即表示添加成功。

{

"\_index": "lagou",

"\_type": "job",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 3,

"successful": 1,

"failed": 0

},

"created": true

}

在head插件中的Browser标签中就能浏览到数据了。

如果在添加数据时不指定数据的id，ES会自动生成此id。

POST lagou/job/

{

"title":"python django开发工程师",

"salary\_min":30000,

"city":"上海",

"company":{

"name":"美团科技",

"company\_addr":"北京市软件园A区"

},

"publish\_date":"2017-4-16",

"comments":20

}

运行成功后在head中查看。会自动生成一个类似于如下的id。

"\_id": "AVypiskRJoQhHPKeMEPt",

获取数据的方法

GET lagou/job/1

{

"\_index": "lagou",

"\_type": "job",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"title": "python分布式爬虫开发",

"salary\_min": 15000,

"city": "北京",

"company": {

"name": "百度",

"company\_addr": "北京市软件园"

},

"publish\_date": "2017-4-16",

"comments": 15

}

}

\_source中就是保存的数据信息。上面几行是此数据的描述信息。

可以只获取\_source中的某些字段。多个字段之间以逗号分隔开且不能有空格。

GET lagou/job/1?\_source=title

GET lagou/job/1?\_source=title,city

也可以使用

GET lagou/job/1?\_source

来获取所有的字段

修改ES中的数据，修改文档

PUT lagou/job/1

{

"title":"python分布式爬虫开发",

"salary\_min":15000,

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园"

},

"publish\_date":"2017-4-16",

"comments":15

}

PUT这种方式与创建的方式相同，必须要指明要修改的id，且是覆盖式的修改。

POST lagou/job/1/\_update

{

"doc":{

"comments":20

}

}

删除文档

DELETE lagou/job/1

删除index

DELETE lagou

## 10-7 elasticsearch的mget和bulk批量操作 (12:36)

批量获取和bulk批量操作

bulk操作可以实现增删改的批量操作

添加\_index testdb

添加\_type job1,job2

job1和job2中分别有id为1和2的文档，

PUT testdb

{

"settings": {

"index":{

"number\_of\_shards":5,

"number\_of\_replicas":1

}

}

}

PUT lagou/job/3

{

"title":"python分布式爬虫开发",

"salary\_min":15000,

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园"

},

"publish\_date":"2017-4-16",

"comments":15

}

PUT testdb/job1/1

{

"title":"job1\_1"

}

PUT testdb/job1/2

{

"title":"job1\_2"

}

PUT testdb/job2/1

{

"title":"job2\_1"

}

PUT testdb/job2/2

{

"title":"job2\_2"

}

注意ES中的逗号，既不能多写也不能少写。

查询job1中id=1的数据，job2中id=2的数据

GET \_mget

{

"docs":[

{"\_index":"testdb",

"\_type":"job1",

"\_id":1

},

{"\_index":"testdb",

"\_type":"job2",

"\_id":2

}

]

}

结果为

{

"docs": [

{

"\_index": "testdb",

"\_type": "job1",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"title": "job1\_1"

}

},

{

"\_index": "testdb",

"\_type": "job2",

"\_id": "2",

"\_version": 1,

"found": true,

"\_source": {

"title": "job2\_2"

}

}

]

}

批量查询时，可以同时查询不同的index中的数据。

要查询同一个index中不同的type的数据

GET testdb/\_mget

{

"docs":[

{

"\_type":"job1",

"\_id":1

},

{

"\_type":"job2",

"\_id":2

}

]

}

结果为

{

"docs": [

{

"\_index": "testdb",

"\_type": "job1",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"title": "job1\_1"

}

},

{

"\_index": "testdb",

"\_type": "job2",

"\_id": "2",

"\_version": 1,

"found": true,

"\_source": {

"title": "job2\_2"

}

}

]

}

查询同一个index中的同一个type下不同id的数据

GET testdb/job1/\_mget

{

"docs":[

{

"\_id":1

},

{

"\_id":2

}

]

}

结果为

{

"docs": [

{

"\_index": "testdb",

"\_type": "job1",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"title": "job1\_1"

}

},

{

"\_index": "testdb",

"\_type": "job1",

"\_id": "2",

"\_version": 1,

"found": true,

"\_source": {

"title": "job1\_2"

}

}

]

}

上面的查询还可以简写为

GET testdb/job1/\_mget

{

"ids":[1,2]

}

bulk批量操作

批量导入可以合并多个操作，比如index, delete, update, create等等，可以把以上多个操作一次提交上去，同时进行执行。也可以帮助从一个索引导入到另一个索引。

bulk批量操作的格式如下：

action\_and\_meta\_data\n

optional\_source\n

action\_and\_meta\_data\n

optional\_source\n

...

action\_and\_meta\_data\n

optional\_source\n

需要注意的是，除delete之外，每一条数据都由两行构成构成，其他的命令比如index和create都是由元信息和数据行组成，update比较特殊，它的数据行可能是doc也可能是upsert或者script，可以参考前面的update的翻译。

一对

action\_and\_meta\_data\n

optional\_source\n

完成一个操作，action\_and\_meta\_data指明它的操作，optional\_source指明它的数据。

多个action与optional在一起就可以完成批量操作。

bulk批量操作示例

{"index":{"\_index":"text","\_type":"type1","\_id":"1"}}

{"field1":"value1"}

第一行中的index指明这个操作是index操作。后面是meta的信息，指明了这个操作是针对什么索引以及索引的什么type和type的什么id来完成的。

也就是说要创建一条数据，这个数据是放在test这个\_index中的，\_type为type1，\_id为1。

第二行就是指定了文档的内容。

把上面对lagou的操作变成是bulk批量操作。数据只能放在一行中，不能使用类似上面的美化方式。

POST \_bulk

{"index":{"\_index":"lagou","\_type":"job1","\_id":"1"}}

{"title":"python分布式爬虫开发","salary\_min":15000,"city":"北京","company":{"name":"百度","company\_addr":"北京市软件园"},"publish\_date":"2017-4-16","comments":15}

{"index":{"\_index":"lagou","\_type":"job2","\_id":"2"}}

{"title":"python\_django开发工程师","salary\_min":30000,"city":"上海","company":{"name":"美团科技","company\_addr":"北京市软件园A区"}, "publish\_date":"2017-4-16", "comments":20}

POST \_bulk

{"index":{"\_index":"test","\_type":"type1","\_id":"1"}}

{"field1":"value1"}

{"create":{"\_index":"test","\_type":"type1","\_id":"3"}}

{"field1":"value3"}

{"update":{"\_id":"1","\_type":"type1","\_index":"index1"}}

{"doc":{"field2":"value2"}}

{"delete":{"\_index":"test","\_type":"type1","\_id":"2"}}

因为bulk是一次性提交很多数据，会一次性的把数据发送到一个节点，然后由这个节点解析元数据。元数据即{"\_index":"test","\_type":"type1","\_id":"1"}}中的内容。这个节点会解析这些元数据，并把它分发给其他节点的分片进行操作。这些命令完成之后，再统一返回结果。

数据量太大时可能会造成延迟，所以一次也不能提交过多的数据。

## 10-8 elasticsearch的mapping映射管理 (21:03)

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 适合类型 |
| store | 值为yes表示存储, 为no表示不存储, 默认为no | all |
| index | yes表示分析, no表示不分析, 默认值为true | string |
| null\_value | 如果字段为空, 可以设置一个默认值, 比如"NA" | all |
| analyzer | 可以设置索引和搜索时用的分析器, 默认使用的是standard分析器, 还可以使用whitespace, simple, english | all |
| include\_in\_all | 默认ES为每个文档定义一个特殊域\_all, 它的作用是让每个字段被搜索到, 哪颗不想某个字段被搜索到, 可以设置为false | all |
| format | 时间格式字符串的模式 | date |

映射(mapping)

映射：创建索引的时候，可以预先定义字段的类型以及相关的属性。这里的属性比mysql中的字段的属性要丰富。mysql中只能定义是否为Null。

映射是创建在type中的，可以理解为在定义数据表的时候，每一列都要指定一种数据类型。如jobbole article中评论数就要定义为int的类型。

对ES也是如此，在插入数据的时候，也可以对每一个字段定义一种类型。

如对于lagou这个\_index，title是字符串的类型，salary则是int类型的。

ES会根据JSON源数据的基础类型猜测你想要的字段映射。将输入的数据转变成可搜索的索引项。

mapping就是我们自己定义的字段的数据类型，同时告诉ES如果索引数据以及是否可以被搜索，都是通过属性的设置来实现的。

作用：会让索引建立的更加细致和完善。大多数情况下，都不用自己去定义映射。

类型：静态映射和动态映射。

ES中内置的类型

string类型：text, keyword(第三种sting类型在ES5中已经废弃)

如果把string的类型的数据设置为text，这个字段传过来的数据就会被分析器分析，包括分词，抽取词干，去除停用词。如"在"是个介词， 一般没有明确的意义，可以进行去除。如果设置为keyword，就不会进行分析不会建立索引，只是当成一个字符串来存储，这种情况下只有完全匹配时才能找到这个sting。

数字类型：长整数long, 整数integer, 短整数short, byte, double, float

日期类型：date。不仅可以解析日期，还可以解析datetime类型的数据。

bool类型：boolean

binary类型：binary，这种类型的数据是不会被检索的。

复杂类型：object，nested。

geo类型：geo-point, geo-shape，地理位置。geo-point是使用经纬度来标记一个位置，geo-shape是通过多个点来标记一片区域。

专业类型：ip，competion，ip即ip地址。competion是用来作搜索建议的。

复杂类型：object，nested。

如在之前对ES数据库的操作，company的信息不是一个字段，而是一个字典，就构成了company的一个实例，就是一个object对象。把object的对象放在数组中，就成为nested。如company中有一个emplyee，就是nested类型的。

{

"title":"python分布式爬虫开发",

"salary\_min":15000,

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园",

"emplyee":[{

"name":"David",

"age":28

},{

"name":"Olina",

"age":29

}]

},

"publish\_date":"2017-4-16",

"comments":15

}

对于内置类型，可以设置的属性。

null\_value，为空的字段可以设置默认值为NA，这样在做搜索或者filter的时候就会很方便，如果字段为空，就无法对它进行索引。

analyzer，在中文的时候一般使用IK这个分析器。建立索引和分词都是用分析器来完成的。

include\_in\_all=false，就不会被搜索到。

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-params.html

在ES的官方文档中可以找到更多的属性。

下面演示一下如果给一个索引中的文档建立mapping以及如何它是如何影响数据插入的。

#创建索引

PUT lagou

{

"mappings":{

"job":{

"properties":{

"title":{

"type":"text"

},

"salary\_min":{

"type":"integer"

},

"city":{

"type":"keyword"

},

"company":{

"properties":{

"name":{

"type":"text"

},

"company\_addr":{

"type":"text"

},

"employee\_count":{

"type":"integer"

}

}

},

"publish\_date":{

"type":"date",

"format":"yyyy-MM-dd"

},

"comments":{

"type":"integer"

}

}

}

}

}

#放入数据

PUT lagou/job/1

{

"title":"python分布式爬虫开发",

"salary\_min":15000,

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园",

"employee\_count":50

},

"publish\_date":"2017-4-18",

"comments":15

}

在head中把之前建立的索引都删除，进入上面两步的操作。

如果把salary写为字符串形式的数字，再次执行，还可以成功。在写入的时候会尝试去转换，如果转换成功，就进行插入的操作。但如果改为"abc"，就会出错了。

PUT lagou/job/2

{

"title":"python分布式爬虫开发",

"salary\_min":"15000",

"city":"北京",

"company":{

"name":"百度",

"company\_addr":"北京市软件园",

"employee\_count":50

},

"publish\_date":"2017-4-18",

"comments":15

}

获取定义的mapping

获取lagou这个index下的所有mapping

GET lagou/\_mapping

获取lagou这个index下job中的mapping

GET lagou/\_mapping/job

获得集群中所有的mapping

GET \_all/\_mapping

获得集群中job下的mapping

GET \_all/\_mapping/job

注意，索引中一旦给某个列创建了类型，就不能再修改了。想要修改，只能将它的索引删除，再新建索引，导入以前的数据。而关系数据库中是可以修改列的类型的。

## 10-9 elasticsearch的简单查询 - 1 (14:56)

elasticsearch是功能非常强大的搜索引擎，使用它的目的就是为了快速的查询到需要的数据

ES查询分类：

基本查询：使用ES内置查询条件进行查询

组合查询：把多个查询组合在一起进行复合查询

过滤：查询的同时，通过filter条件在不影响打分的情况下筛选数据

过滤不参与打分，只有基本查询和组合查询才涉及到打分。

基本查询

#添加映射

#type字段会加入倒排索引，加入倒排索引之前要使用分析器对它进行分析，这里使用ik\_max\_word分析器。

#https://github.com/medcl/elasticsearch-analysis-ik，ES已经安装了ik，可以直接使用。

#desc没有设置分析器，ES就会使用内置的分析器来对中文进行分词，但效果没有ik好。

PUT lagou

{

"mappings":{

"job":{

"properties":{

"title":{

"store":true,

"type":"text",

"analyzer":"ik\_max\_word"

},

"company\_name":{

"store":true,

"type":"keyword"

},

"desc":{

"type":"text"

},

"comments":{

"type":"integer"

},

"add\_time":{

"type":"date",

"format":"yyyy-MM-dd"

}

}

}

}

}

DELETE lagou

POST lagou/job/

{

"title":"python django开发工程师",

"company\_name":"美团科技有限公司",

"desc":"对django的概念熟悉，熟悉python基础知识",

"comments":20,

"add\_time":"2017-4-1"

}

POST lagou/job/

{

"title":"python scrapy redis分布式爬虫基本",

"company\_name":"百度科技有限公司",

"desc":"对scrapy的概念熟悉，熟悉redis的基本操作",

"comments":5,

"add\_time":"2017-4-15"

}

POST lagou/job/

{

"title":"elasticsearch打造搜索引擎",

"company\_name":"阿里巴巴科技有限公司",

"desc":"熟悉数据结构算法，熟悉python的基本开发",

"comments":15,

"add\_time":"2017-6-20"

}

POST lagou/job/

{

"title":"python打造推荐引擎系统",

"company\_name":"阿里巴巴科技有限公司",

"desc":"熟悉推荐引擎的原理以及算法，掌握C语言",

"comments":60,

"add\_time":"2016-10-20"

}

#match查询，会对输入进入分词，会去查询match中输入的字段中单词，把python改为大写的也能搜索出来。会使用上面设置的ik\_max\_word对要查询的内容进行分词。会自动进行大小写的转换。搜索"python网站"，会把python网站分为python和网站两个词，分别在ES数据库中进行匹配。只要匹配到任意一个，都会返回查询的数据。搜索"爬取"，也会搜索出结果。

#但是这里如果查询"company\_name":"公司"，是无法查询到结果的，因为在ES数据库中，company\_name是以keyword的方式存储的，不会进行分词处理。

GET lagou/\_search

{

"query":{

"match":{

"title":"python"

}

}

}

#term查询，不会对传入的要查询的关键词进行分词处理，类似于ES数据库中的keyword一样。如果term中查询"title":"python"，或查询"title":"打造"，都能得出结果，但如果查询"python打造"，就不会搜索到结果了。但在match查询中，因为有分词，搜索"python爬虫"也能搜索出结果。

#如果term查询"company\_name":"公司"，因为不会进行分词处理，就查不到结果。但如果查询"company\_name":"阿里巴巴科技有限公司"，这样能得出结果。

GET lagou/\_search

{

"query":{

"term":{

"company\_name":"阿里巴巴科技有限公司"

}

}

}

#terms查询，只要符合terms中的任何一项，都会视为匹配查询规则

GET lagou/\_search

{

"query":{

"terms":{

"title":["工程师","django","系统"]

}

}

}

#控制查询的返回数量，可以用来进行分页。from：1表示从所有查询结构的第1个开始返回，size：2表示只返回2个数据。数据过多时很有用。from:0, size:3可以返回所有的数据。

GET lagou/\_search

{

"query":{

"match":{

"title":"python"

}

},

"from":1,

"size":2

}

#match\_all查询

GET lagou/\_search

{

"query":{

"match\_all":{}

}

}

#match\_phrase查询，短语查询，ES会对查询的内容进行分词，分成一个词条，即词的数组。python系统分为python和系统，并且必须要同时满足python和系统的才满足查询的条件。slop表示title中python和系统这两个词之间的距离要小于等于slop中的数字才满足查询条件。如果设置slop为5，则查询不到结果。因为title中python和系统这两个词之间的距离为6，这里的slop要设置为6以上的数字才行。

GET /lagou/\_search

{

"query":{

"match\_phrase":{

"title":{

"query":"python系统",

"slop":6

}

}

}

}

#multi\_match查询

#比如可以在field中指定多个字段

#比如查询title和desc这两个字段里面包含python的关键词文档

#title^3表示权重，title中的权重是desc中的3倍。权重会影响到打分，当不使用权重时，title和desc中出现python时它们处于同等重要的地位。

GET lagou/\_search

{

"query":{

"multi\_match":{

"query":"python",

"fields":["title^3","desc"]

}

}

}

#指定返回的字段，在stored\_fields中指定要返回的字段。

GET lagou/\_search

{

"stored\_fields":["title","company\_name"],

"query":{

"match":{

"title":"python"

}

}

}

但如果在stored\_fields中再添加一个desc，就搜索不到结果了。因为这里指定的是stored\_fields，desc的属性中没有包含stored，所以没有返回任何结果。

GET lagou/\_search

{

"stored\_fields":["title","company\_name","desc"],

"query":{

"match":{

"title":"python"

}

}

}

#通过sort把结果排序，sort中是一个数组，数组中是一个字典，字典的key就是需要sort的字段，这里是comments。依据评论数降序desc排列,asc为升序排列。

GET lagou/\_search

{

"query":{

"match\_all":{}

},

"sort":[{

"comments":{

"order":"desc"

}

}]

}

#查询范围

#range查询，range是放在query中的，是query的条件。在range中可以指定限定查询范围的字段，在字段中再设置范围。boost表示权重。

GET lagou/\_search

{

"query":{

"range":{

"comments":{

"gte":10,

"lte":20,

"boost":2.0

}

}

}

}

#这里是对时间的范围进行限定，gte是大于等于，gt是大于。

GET lagou/\_search

{

"query":{

"range":{

"add\_time":{

"gte":"2017-04-01",

"lte":"now"

}

}

}

}

#wildcard查询，可以简单的理解成一个模糊查询。wildcard查询即支持查询条件中通配符的查询语句。

GET lagou/\_search

{

"query":{

"wildcard":{

"title":{

"value":"pyth\*n",

"boost":2.0

}

}

}

}

以上查询都既可以会对某个index，也可以针对某个type。GET lagou/job/\_match就可以对job这个type进行查询了。

中国人学习的很大的一个障碍就是英文，如果把英文的技术性文档全都翻译过来，很多课程都没有存在的必要了。

## 10-11 elasticsearch的bool组合查询 (22:58)

bool查询。

老版本中的filtered已经被bool替代。

bool查询包括 must should must\_not filter四种，每一种查询中都可以填写多种查询条件，把多种查询条件放在数组中即可。如果只有一种查询条件，使用字典即可。格式如下：

bool:{

"must":[],

"should":[],

"must\_not":[],

"filter":[],

}

must表示数组中的所有查询都必须要同时满足，

should表示数组中的查询条件满足一个或多个，

must\_not表示数组中的查询条件必须全部不满足，或一个也不能满足，

filter实现对字段的过滤，并且filter不参与打分。

删除lagou中所有的数据，重新建立测试数据

POST lagou/testjob/\_bulk

{"index":{"\_id":1}}

{"salary":10, "title":"Python"}

{"index":{"\_id":2}}

{"salary":20, "title":"Scrapy"}

{"index":{"\_id":3}}

{"salary":30, "title":"Django"}

{"index":{"\_id":4}}

{"salary":30, "title":"Elasticsearch"}

DELETE lagou/testjob

简单过滤查询

最简单的filter查询

对应的sql语句

select \* from testjob where salary=20

查询薪资为20的工作，用term来完成对某个字段取特定的值。

GET lagou/testjob/\_search

{

"query":{

"bool":{

"must":{

"match\_all":{}

},

"filter":{

"term":{

"salary":20

}

}

}

}

}

不使用must，直接过滤出薪资为20的工作也可以

GET lagou/testjob/\_search

{

"query":{

"bool":{

"filter":{

"term":{

"salary":20

}

}

}

}

}

也可以使用match来查询，salary为int的类型，int类型是不会分词的，使用match在salary中查找时也是要完全匹配为20的documents文档。但在filter中，特别是某些精确的值，最好使用term来查询。

GET lagou/testjob/\_search

{

"query":{

"bool":{

"filter":{

"match":{

"salary":20

}

}

}

}

}

也可以指定多个值，如查询salary为10和20的documents文档。

GET lagou/testjob/\_search

{

"query":{

"bool":{

"must":{

"match\_all":{}

},

"filter":{

"terms":{

"salary":[10,20]

}

}

}

}

}

通过title进行查询，查询title=Python的documents文档

类似于如下sql语句的功能

select \* from testjob where title="Python"

GET lagou/testjob/\_search

{

"query":{

"bool":{

"must":{

"match\_all":{}

},

"filter":{

"term":{

"title":"Python"

}

}

}

}

}

但这里却没有查询到任何内容。这里查询title=Python，ES中存储时也是写的Python，为什么查询不到内容呢？

title是text类型的字符串，text字段在作索引的时候会先进行分词，也会进行大小写的转换，所以上面向数据库中插入数据的时候，ES会把Python转化为小写来存储。而这里使用term查询大写的Python，不会进行分词操作，也不会进行大小写的转换，所以就查询不到内容了。

这里把term改为match，就可以查询到内容了。使用term查询小写的python，也能查询到数据。

如果想要使用term查询Python，在进行数据插入的时候，可以在mapping中把title设置为text类型的，同时设置不对它进行倒排索引即index=false。但一般情况下是不会这样处理的。

查看分析器解析的结果

GET /\_analyze

{

"analyzer":"ik\_max\_word",

"text":"python网络开发工程师"

}

或

GET \_analyze

{

"analyzer":"ik\_smart",

"text":"python网络开发工程师"

}

使用ik\_max\_word可能会存在大量的重复。如果设置为ik\_smart，就会使用最小的数据集把分词分出来。

https://github.com/medcl/elasticsearch-analysis-ik

Tips:

ik\_max\_word: 会将文本做最细粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,中华人民,中华,华人,人民共和国,人民,人,民,共和国,共和,和,国国,国歌”，会穷尽各种可能的组合；

ik\_smart: 会做最粗粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,国歌”。

bool组合过滤查询

查询薪资等于20或者工作为python的工作，排除薪资为30的

select \* from testjob where (salary=20 OR title=python) AND (salary !=30)

GET lagou/testjob/\_search

{

"query":{

"bool":{

"should":[

{"term":{"salary":20}},

{"term":{"title":"python"}}

],

"must\_not":{

"term":{"salary":30}

}

}

}

}

查询薪资等于20或者工作为python的工作，排除薪资为30和10的。

GET lagou/testjob/\_search

{

"query":{

"bool":{

"should":[

{"term":{"salary":20}},

{"term":{"title":"python"}}

],

"must\_not":[

{"term":{"salary":30}},

{"term":{"salary":10}}

]

}

}

}

嵌套查询

查询title=python或者tile=elasticsearch同时salary=30的工作。

select \* from testjob where title="elasticsearch" or (title="django" AND salary=30)

GET lagou/testjob/\_search

{

"query":{

"bool":{

"should":[

{"term":{"title":"python"}},

{"bool":{

"must":[

{"term":{"title":"elasticsearch"}},

{"term":{"salary":30}}

]

}

}

]

}

}

}

过滤空和非空的数据

建立测试数据

POST lagou/testjob2/\_bulk

{"index":{"\_id":1}}

{"tags":["search"]}

{"index":{"\_id":2}}

{"tags":["search","python"]}

{"index":{"\_id":3}}

{"other\_field":["some data"]}

{"index":{"\_id":4}}

{"tags":null}

{"index":{"\_id":5}}

{"tags":["search",null]}

处理null空值的方法

select tags from testjob2 where tags is not NULL

查询tags字段不为空的内容

GET lagou/testjob2/\_search

{

"query":{

"bool":{

"filter":{

"exists":{

"field":"tags"

}

}

}

}

}

查询tags字段为空的内容

GET lagou/testjob2/\_search

{

"query":{

"bool":{

"must\_not":{

"exists":{

"field":"tags"

}

}

}

}

}

## 10-12 scrapy写入数据到elasticsearch中 - 1 (14:16)

使用ES的python接口dsl

https://github.com/elastic/elasticsearch-dsl-py

进入虚拟环境

pip install elasticsearch-dsl

查看dsl实例

http://elasticsearch-dsl.readthedocs.io/en/latest/persistence.html#doctype

from datetime import datetime

from elasticsearch\_dsl import DocType, Date, Nested, Boolean, \

analyzer, InnerObjectWrapper, Completion, Keyword, Text

html\_strip = analyzer('html\_strip',

tokenizer="standard",

filter=["standard", "lowercase", "stop", "snowball"],

char\_filter=["html\_strip"]

)

class Comment(InnerObjectWrapper):

def age(self):

return datetime.now() - self.created\_at

class Post(DocType):

title = Text()

title\_suggest = Completion()

created\_at = Date()

published = Boolean()

category = Text(

analyzer=html\_strip,

fields={'raw': Keyword()}

)

comments = Nested(

doc\_class=Comment,

properties={

'author': Text(fields={'raw': Keyword()}),

'content': Text(analyzer='snowball'),

'created\_at': Date()

}

)

class Meta:

index = 'blog'

def add\_comment(self, author, content):

self.comments.append(

{'author': author, 'content': content})

def save(self, \*\* kwargs):

self.created\_at = datetime.now()

return super().save(\*\* kwargs)

就像django的models一样，可以定义一个DocType类Post，这个class可以像定义item一样进行定义，其中的字段对应于item中的字段，只需要定义好这里的字段，给它赋上类型的值，这里的类型与ES中的类型是对应的，从ES\_dsl中引入进来这些类型。

class Post(DocType):

title = Text()

title\_suggest = Completion()

created\_at = Date()

published = Boolean()

category = Text(

analyzer=html\_strip,

fields={'raw': Keyword()}

)

comments = Nested(

doc\_class=Comment,

properties={

'author': Text(fields={'raw': Keyword()}),

'content': Text(analyzer='snowball'),

'created\_at': Date()

}

)

为了与django的设计理念保持一致，新建一个名为models的python package，新建/models/es\_types.py

from datetime import datetime

from elasticsearch\_dsl import DocType, Date, Nested, Boolean, analyzer, InnerObjectWrapper, Completion, Keyword, Text, Integer

#通过ES-dsl提供的connections连接ES的数据库

from elasticsearch\_dsl.connections import connections

#这里允许连接到多个服务器

connections.create\_connection(hosts=["localhost"])

class ArticleType(DocType):

#jobbole的文章类型，可以参照items.py中的定义来做。

#定义Text()类型的字段的分析器

title = Text(analyzer="ik\_max\_word")

create\_time = Date()

#url和url\_object\_id不作分词处理，只需要全量保存即可

url = Keyword()

url\_object\_id = Keyword()

front\_image\_url = Keyword()

front\_image\_path = Keyword()

praise\_num = Integer()

fav\_num = Integer()

comment\_num = Integer()

tags = Text(analyzer="ik\_max\_word")

content = Text(analyzer="ik\_max\_word")

#通过Meta来定义ArticleType这个类所属的index和type

class Meta:

index = 'jobbole'

doc\_type = "article"

#下面的问题是如何通过定义的item直接生成mapping

#调用ArticleType的init()方法直接生成mapping

#使用这种方法就不会在每次调用时都重新生成mapping

if \_\_name\_\_ == "\_\_main\_\_":

ArticleType.init()

在本文件中运行，查看是否会生成名为jobbole的index和名为article的type。

这样，就通过以上的方法把生成mapping的操作简单化了。后期在改变了字段的类型后，只要把ES数据库中的内容删除，重新运行此文件就可以了。比手动写mapping要简单很多。

在pipelines.py中新建一个，通过pipeline把数据写入到ES中。

class ElasticsearchPipeline(object):

#将数据写入到ES中

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

#将item转换为ES的数据。可以像操作类一样直接把数据保存到ES中。

from models.es\_types import ArticleType

#生成ArticleType的对象

article = ArticleType()

article.title = item["title"]

article.create\_date = item["create\_date"]

#之前提取的content中包含html的标签，这里要先把html标签去掉。

from w3lib.html import remove\_tags

article.content = remove\_tags(item["content"])

article.front\_image\_url = item["front\_image\_url"]

if "front\_image\_path" in item:

article.front\_image\_path = item["front\_image\_path"]

article.praise\_num = item["praise\_num"]

article.fav\_num = item["fav\_num"]

article.comment\_num = item["comment\_num"]

article.url = item["url"]

article.tags = item["tags"]

#使用url\_object\_id作为ES数据库中的id

article.meta.id = item["url\_object\_id"]

article.save()

return item

在settings中添加配置，把ElasticsearchPipeline添加到ITEM\_PIPELINES列表中

ITEM\_PIPELINES = [

'ArticleSpider.pipelines.ElasticsearchPipeline': 100

]

查看ES的head插件页面，ES中现在没有jobbole的数据。

在pipelines.py中article=ArticleType()处添加断点，在return处添加断点，进行调试。查看ES数据库中的数据，并进行搜索

GET jobbole/article/\_search

{

"query":{

"match":{

"tags":"持续部署"

}

}

}

以上就完成了把scrapy爬取的数据保存到ES中的工作。

但还存在一个问题，在爬取不同的网站时，可能会存在与pipelines.py中定义的字段不符合的情况。如知乎的字段和jobbole的字段就相差很多，传递到ElasticsearchPipeline中的item就会不同。所以要像之前的mysql的处理一样，把item的逻辑和在items.py本身中定义。

class JobBoleArticleItem(scrapy.Item):

title = scrapy.Field()

create\_time = scrapy.Field(

# 该传入的字段值要批量处理的函数

input\_processor=MapCompose(remove\_dote, date\_convert),

)

url = scrapy.Field()

url\_object\_id = scrapy.Field()

front\_image\_url = scrapy.Field(

output\_processor = MapCompose(return\_value)

)

front\_image\_path = scrapy.Field()

praise\_num = scrapy.Field(

input\_processor = MapCompose(get\_praise\_num)

)

fav\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

comment\_num = scrapy.Field(

input\_processor=MapCompose(get\_num),

)

tags = scrapy.Field(

input\_processor=MapCompose(remove\_comment\_tag),

output\_processor = Join(',')

)

content = scrapy.Field()

def get\_insert\_sql(self):

insert\_sql = '''

INSERT INTO

ArticleSpider(title, create\_time, url, url\_object\_id, front\_image\_url, front\_image\_path, comment\_num, praise\_num, fav\_num, tags, content)

VALUES

(%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)

'''

# cursor.execute(insert\_sql, (item["title"], item["create\_time"], item["url"], item["url\_object\_id"], item["front\_image\_url"][0], item["front\_image\_path"], item["comment\_num"], item["praise\_num"], item["fav\_num"], item["tags"], item["content"]))

params = (self["title"], self["create\_time"], self["url"], self["url\_object\_id"], self["front\_image\_url"][0], self["front\_image\_path"], self["comment\_num"], self["praise\_num"], self["fav\_num"], self["tags"], self["content"])

return insert\_sql, params

from models.es\_types import ArticleType

def save\_to\_es(self):

article = ArticleType()

article.title = self["title"]

article.create\_date = self["create\_date"]

#之前提取的content中包含html的标签，这里要先把html标签去掉。

from w3lib.html import remove\_tags

article.content = remove\_tags(self["content"])

article.front\_image\_url = self["front\_image\_url"]

if "front\_image\_path" in self:

article.front\_image\_path = self["front\_image\_path"]

article.praise\_num = self["praise\_num"]

article.fav\_num = self["fav\_num"]

article.comment\_num = self["comment\_num"]

article.url = self["url"]

article.tags = self["tags"]

#使用url\_object\_id作为ES数据库中的id

article.meta.id = self["url\_object\_id"]

article.save()

return

class ZhihuQuestionItem(scrapy.Item):

...

修改pipelines.py

class ElasticsearchPipeline(object):

#将数据写入到ES中

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

item.save\_to\_es()

return item

在测试之前删除原数据库中的所有内容。

再次运行es\_types.py，生成ES数据库的mappings。设置断点并运行，查看数据是否会写入到ES数据库中。

# 第11章 django搭建搜索网站

## 11-1 es完成搜索建议-搜索建议字段保存

首先了解网站的功能，在搜索页面中有3个tag，文章，问答和职位，分别对应于3个网站爬取到的数据。

输入一个关键词，如python，会自动进行搜索内容的提示。

搜索页面会有热门搜索和我的搜索

在搜索结果列表页面中，分为3个部分，左导航为网站的列表和网站数据的爬取量，中间为搜索结果的列表显示，右边栏为热门搜索和我的搜索。每次搜索之后都会把搜索记录保存到我的搜索中。点击我的搜索中的内容，就会跳转到对应的搜索结果页。搜索结果页最上面是搜索栏，最下面是分页。

第1个要解决的问题是输入内容的智能提示。

ES提供了搜索建议的接口，如何来使用此接口完成输入内容的智能提示。

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-suggesters-completion.html

要使用suggesters-completion，首先要在mapping中新增一个字段，这个字段的类型要设置为"completion"

PUT music

{

"mappings": {

"song" : {

"properties" : {

"suggest" : {

"type" : "completion"

},

"title" : {

"type": "keyword"

}

}

}

}

}

由于mapping中字段的实现是在models中实现的，现在要做的就是如何在models中实现以上功能。

在ArticleSpider/models/es\_types.py中定义了一个ArticleType的类。

from datetime import datetime

from elasticsearch\_dsl import DocType, Date, Nested, Boolean, analyzer, InnerObjectWrapper, Completion, Keyword, Text, Integer

from elasticsearch\_dsl.connections import connections

connections.create\_connection(hosts=["localhost"])

class ArticleType(DocType):

title = Text(analyzer="ik\_max\_word")

create\_time = Date()

url = Keyword()

url\_object\_id = Keyword()

front\_image\_url = Keyword()

front\_image\_path = Keyword()

praise\_num = Integer()

fav\_num = Integer()

comment\_num = Integer()

tags = Text(analyzer="ik\_max\_word")

content = Text(analyzer="ik\_max\_word")

class Meta:

index = 'jobbole'

doc\_type = "article"

if \_\_name\_\_ == "\_\_main\_\_":

ArticleType.init()

为了完成自动补全的功能，需要在每一个models中也就是ES的type中添加一个Completion字段，现在需要在ArticleType中添加一个字段

class ArticleType(DocType):

suggest = Completion(analyzer="ik\_max\_word")

本来是可以使用这种与Text中相同的方法来定义的，但是由于ES-dsl官方源码存在bug，这样做会报错，要自己去定义一个CustomAnalyzer，这个自定义的CustomAnalyzer继承ES默认的CustomAnalyzer。

from elasticsearch\_dsl.analysis import CustomAnalyzer as \_CustomAnalyzer

class CustomAnalyzer(\_CustomAnalyzer):

def get\_analysis\_definition(self):

return {}

#生成一个自定义的CustomAnalyzer对象ik\_analyzer，filter中可以传入多个值，lowercase的作用是全部转换为小写

ik\_analyzer = CustomAnalyzer("ik\_max\_word", filter=["lowercase"])

全部代码如下

from datetime import datetime

from elasticsearch\_dsl import DocType, Date, Nested, Boolean, analyzer, InnerObjectWrapper, Completion, Keyword, Text, Integer

from elasticsearch\_dsl.connections import connections

connections.create\_connection(hosts=["localhost"])

from elasticsearch\_dsl.analysis import CustomAnalyzer as \_CustomAnalyzer

class CustomAnalyzer(\_CustomAnalyzer):

def get\_analysis\_definition(self):

return {}

ik\_analyzer = CustomAnalyzer("ik\_max\_word", filter=["lowercase"])

class ArticleType(DocType):

title = Text(analyzer="ik\_max\_word")

suggest = Completion(analyzer="ik\_analyzer")

create\_time = Date()

url = Keyword()

url\_object\_id = Keyword()

front\_image\_url = Keyword()

front\_image\_path = Keyword()

praise\_num = Integer()

fav\_num = Integer()

comment\_num = Integer()

tags = Text(analyzer="ik\_max\_word")

content = Text(analyzer="ik\_max\_word")

class Meta:

index = 'jobbole'

doc\_type = "article"

if \_\_name\_\_ == "\_\_main\_\_":

ArticleType.init()

现在运行此文件，查看生成的mapping。

生成的suggest中添加了一个max\_input\_length的字段。其analyzer为"ik\_max\_word"，type变为了"completion"，说明以上代码是可用的。

上面已经根据文档生成了一个mapping，每爬取到一条数据库的时候，如何来生成suggest的值呢？

PUT music/song/1?refresh

{

"suggest" : {

"input": [ "Nevermind", "Nirvana" ],

"weight" : 34

}

}

查看官方文档说明，在做数据保存的时候，可以放入一个字典，字典中的input是很多单词组成的数组，可以给这一组词语赋一个权重。会对很多字段都生成这个suggest，如果是对title生成suggest，可以使用上面的方法。但如果要对tags生成suggest，因为tags的值可能为多个，用上面的方法就不行了。要使用下面的方法来实现。

#You can index multiple suggestions for a document as follows:

PUT music/song/1?refresh

{

"suggest" : [

{

"input": "Nevermind",

"weight" : 10

},

{

"input": "Nirvana",

"weight" : 3

}

]

}

如何来获取suggest中的input值呢。可以使用\_analyze的方法。

GET /\_analyze

{

"analyzer":"ik\_max\_word",

"text":"python网络开发工程师"

}

这个analyze这个接口可以在传入字符串以后自动生成分词。需要通过这个接口自己定义一个方法生成input中的值。

之前是在items.save\_to\_es中定义的数据的保存。需要在其中生成搜索建议词。

因为之后的lagou和zhihu都可能会用到这个函数，要把它定义为全局的函数。

def gen\_suggests(index,info\_tuple)

###这个函数也具有通用性，lagou,zhihu都能直接使用这个函数。

# 这里为什么要接收tuple类型的数据呢，因为希望这个函数可以指定传递过来的字符串的权重，可以接收多个字符串，tuple是有顺序的。

# 根据传来的字符串和它的权重生成类似[{"input":[],"weigh":2}]这样结构的搜索建议数组

# 为什么要定义一个used\_words的set呢，因为后面要进行去除，那为什么要去除。如"title"中包含"python重要性"，如果tags中也包含了"python重要性"这串字符，二者的权重是不同的。如果title解析到"重要性"的权重为10，tags解析到"重要性"的权重为3。当分词时在title中分出一个权重为10的"重要性"，后面又在tags中又解析到权重为3的"重要性"，因为分词是不会重复的，它会把之前在title中解析到的权重给覆盖掉。这样是不合理的，要把它过滤掉，即只使用第一次分词得出的权重。所以要定义一个used\_words的集合用来去重。

###是否可以对title和tags相同的关键词分别进行权重的处理呢。

used\_words = set()

#以数组的形式返回最终的搜索建议

suggests = []

#这里使用tuple，就可以传递过来多个的值和它们的权重。

#这里的text就是文章的标题或者tags。

for text, weight in info\_tuple:

#传递过来的可能是空字符串，如果是空的，则不进行处理。

if text:

#调用ES的analyze的接口分析字符串，进行分词及大小写的转换。

#获取ES的连接。

from elasticsearch\_dsl.connections import connections

es = connections.create\_connection(ArticleType.\_doc\_type.using)

#调用处理器对text即文章的标题或tags进行分词处理，并返回处理后的词。

#analyze的第1个参数就是index，也就是它所属的ES数据库的名称，第2个参数是分析器，第3个参数是params，通过filter把lowercase传递进来完成大小写的转换。第4个参数就是要处理的内容text。

words = es.indices.analyze(index=index, analyzer="ik\_max\_word", params={"filter":["lowercase"]}, body=text)

"""

words的结构与GET \_analyze{"analyzer":"ik\_max\_word","text":"python网络开发工程师"}的结果的结构类似。

{

"tokens": [

{

"token": "python",

"start\_offset": 0,

"end\_offset": 6,

"type": "ENGLISH",

"position": 0

},

{},

{}

]}

要使用word["token"]的方法来取出其中的值。

"""

#使用列表的解析式来完成

#len(r["token"])>用来过滤单个字的分词结果，单个字的分词一般没有特定的意义

analyzed\_words = set([r["token"] for r in words["tokens"] if len(r["token"])>1])

#过滤掉已经存在的分词

new\_words = analyzed\_words - used\_words

else:

new\_words = set()

#info\_tuple中可以有多个值，处理完一个之后就将处理后的set放到要返回的suggests中。

if new\_words:

#在添加到suggests中之前首先要处理成类似于[{"input":[],"weigh":2}]的格式。

suggests.append({"input":list(new\_words), "weight":weight})

return suggests

class JobBoleArticleItem(scrapy.Item):

...

def save\_to\_es(self):

article = ArticleType()

article.title = self["title"]

article.create\_date = self["create\_date"]

#之前提取的content中包含html的标签，这里要先把html标签去掉。

from w3lib.html import remove\_tags

article.content = remove\_tags(self["content"])

article.front\_image\_url = self["front\_image\_url"]

if "front\_image\_path" in self:

article.front\_image\_path = self["front\_image\_path"]

article.praise\_num = self["praise\_num"]

article.fav\_num = self["fav\_num"]

article.comment\_num = self["comment\_num"]

article.url = self["url"]

article.tags = self["tags"]

#使用url\_object\_id作为ES数据库中的id

article.meta.id = self["url\_object\_id"]

# article.suggest = {{"input":[],"weight":2}}

# gen\_suggests中要传递2个参数，第一个是index，第二个是一个tuple。tuple中分别传递title及tags和它们的权重。把title的权重设置的高一些。

article.suggest = gen\_suggests(ArticleType.\_doc\_type.index, ((article.title,10),(article.tags,7)))

article.save()

return

在article\_item = JobBoleArticleItem()和article.suggest处添加断点，进行调试。

在head插件中查看jobbole中的数据。就生成了suggest的信息。

以上就完成了搜索建议前期的准备工作，生成了suggest的信息，并同爬虫爬取到的信息一起保存在ES数据中。

还需要对zhihu的question和answer进行类似的处理。

## 11-3 django实现elasticsearch的搜索建议

本节中开始通过django来搭建搜索网站。需要新建一个基于python3.5名为lcv\_search的虚拟环境。

进入虚拟环境，安装所需要的开发包。

pip install django -i https://pypi.douban.com/simple/

通过pycharm来新建一个名为LcvSearch的django项目，选择Interpreter为上面新建的lcv\_search的虚拟环境。同时在More Settings中新建一个名为search的App

使用这种方法新建的app会自动添加到settings中的INSTALLED\_APPS列表中。

可以直接运行项目，访问127.0.0.1:8000来查看是否能正常运行。

在项目根目录中新建一个static文件夹来存放静态文件，css, img, js

要把前端的index和result页面复制到django的templates文件夹中。

在urls.py中设置一个可以访问index和result的url匹配规则

from django.views.generic import TemplateView

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'^$', TemplateView.as\_view(template\_name="index.html"),name="index"),

]

再次访问127.0.0.1:8000，就能访问到index这个主页了。但是css,js和图片都没有正常加载出来。因为还没有配置static静态文件夹。

在settings中添加静态文件夹路径。

#这里是用数组的形式传递参数的，说明可以传递多个static文件夹的地址。

STATICFILES\_DIRS = [

os.path.join(BASE\_DIR, "static")

]

#这里也可以使用tuple来完成，但注意一定要添加一个逗号。更建议使用数组来完成。

STATICFILES\_DIRS = (

os.path.join(BASE\_DIR, "static"),

)

修改模板文件，在文件开头{% load staticfiles %}，同时把所有的引用静态文件的地址都修改为类似{% static 'css/style.css' %}的形式。

在运行时，会首先在settings中读取STATIC\_URL这个变量，把模板文件中所有类似{% static 'css/style.css' %}的静态文件引用都替换为/static/css/style.css这样的形式。

重新访问127.0.0.1:8000，页面就能正常显示了。

下面要完成搜索建议的功能。

访问百度，打开检查元素，切换到network中，在搜索栏中任意输入内容，如python，都会向后台发送搜索建议的请求，把输入的内容传回服务器中，服务器再返回自动补全的内容。

在index.html页面中已经使用ajax完成了搜索请求的代码。

在搜索建议中定义了一个searchInput的函数，会bind绑定input的事件，在input变化的时候，就会向suggest\_url发送一个请求，suggest\_url会调用suggest这个变量，变量中就是要在后台处理的url。suggest\_url = "/suggest/"。只要在后台urls中配置好这个url匹配规则和对应的视图函数，就能完成相应的搜索建议功能了。

搜索建议中会获取用户在搜索框中输入的内容，使用get方法向suggest\_url发送一个请求，把用户输入的内容通过searchText这个参数发送到服务器。在发送的请求中有一个s\_type的值，对应于文章、问答和职位三种类型，选择每一种类型时会提供不同类型的搜索建议。

<script type="text/javascript">

var suggest\_url = "/suggest/"

var search\_url = "/search/"

$('.searchList').on('click', '.searchItem', function(){

$('.searchList .searchItem').removeClass('current');

$(this).addClass('current');

});

function removeByValue(arr, val) {

for(var i=0; i<arr.length; i++) {

if(arr[i] == val) {

arr.splice(i, 1);

break;

}

}

}

// 搜索建议

$(function(){

$('.searchInput').bind(' input propertychange ',function(){

var searchText = $(this).val();

var tmpHtml = ""

$.ajax({

cache: false,

type: 'get',

dataType:'json',

url:suggest\_url+"?s="+searchText+"&s\_type="+$(".searchItem.current").attr('data-type'),

async: true,

success: function(data) {

for (var i=0;i<data.length;i++){

tmpHtml += '<li><a href="'+search\_url+'?q='+data[i]+'">'+data[i]+'</a></li>'

}

$(".dataList").html("")

$(".dataList").append(tmpHtml);

if (data.length == 0){

$('.dataList').hide()

}else {

$('.dataList').show()

}

}

});

} );

})

hideElement($('.dataList'), $('.searchInput'));

</script>

在search/views.py中新建一个类，用来处理搜索建议。一般是使用函数来完成的，但在这里使用类来完成，这也是django官方推荐的方法。空两行是PEP8的规范。

from django.views.generic.base import View

class SearchSuggest(View):

#重写View中的get方法。

def get(self, request):

#获取用户传递过来的关键词，同时设置其默认值为空字符串

key\_words = request.GET.get('s','')

#搜索建议的数组，因为搜索建议不止一个，使用数组来存储。

re\_datas = []

#如果取到了用户输入的关键词，就可以使用上一节中在es\_types.py中定义的ArticleType来获取关键词。

"""

把上一节中在es\_types.py中定义的ArticleType的内容全部复制到本项目中的search/models.py中。

首先要在虚拟环境中安装elasticsearch-dsl。

pip install elasticsearch-dsl

修改search/models.py文件

from datetime import datetime

from elasticsearch\_dsl import DocType, Date, Nested, Boolean, \

analyzer, InnerObjectWrapper, Completion, Keyword, Text, Integer

from elasticsearch\_dsl.analysis import CustomAnalyzer as \_CustomAnalyzer

from elasticsearch\_dsl.connections import connections

connections.create\_connection(hosts=["localhost"])

class CustomAnalyzer(\_CustomAnalyzer):

def get\_analysis\_definition(self):

return {}

ik\_analyzer = CustomAnalyzer("ik\_max\_word", filter=["lowercase"])

class ArticleType(DocType):

#伯乐在线文章类型

suggest = Completion(analyzer=ik\_analyzer)

title = Text(analyzer="ik\_max\_word")

create\_date = Date()

url = Keyword()

url\_object\_id = Keyword()

front\_image\_url = Keyword()

front\_image\_path = Keyword()

praise\_nums = Integer()

comment\_nums = Integer()

fav\_nums = Integer()

tags = Text(analyzer="ik\_max\_word")

content = Text(analyzer="ik\_max\_word")

class Meta:

index = "jobbole"

doc\_type = "article"

if \_\_name\_\_ == "\_\_main\_\_":

ArticleType.init()

"""

from search.models import ArticleType

if key\_words:

#使用ArticleType.search的方法生成一个对象。

s = ArticleType.search()

#使用ArticleType.search的方法生成的对象有一个suggest方法，

"""

首先要介绍ES的其它使用方法。

Fuzzy Query是ES提供的一个模糊搜索的功能。

先在ES的jobbole中添加100左右的数据。

fuzzy模糊搜索的方法。

GET jobbole/\_search

{

"query":{

"fuzzy":{"title":"linux"}

},

"\_source":["title"]

}

以上的搜索能返回大约20条数据。如果把搜索关键词改为"linx"或"linxx"，还是能搜索到linux的内容，这就是fuzzy搜索的功能。

如果把这里的fuzzy改为match，并且搜索linx，是无法搜索到linux的内容的。

fuzzy搜索的高级用法，在title中传递了一个字典，字典中的fuzziness是最小编辑距离，

什么是编辑距离

编辑距离是一种字符串之间相似程度的计算方法。

两个字符串之间的编辑距离等于使一个字符串变成另外一个字符串而进行的(1)插入, (2)删除, (3)替换, (4)相邻字符交换位置而进行操作的最少次数。

比如：ed("recoginze","recognize") == 1,这里需要交换两个相邻字符i和n的位置。

ed("sailn","failing") == 3, 需要将s换成f, 在字母l后面插入i, n后面插入g.

编辑距离的求法，普遍采用的是动态规划的方法。

在本处fuzzy搜索中，设置的编辑距离是2，因为linx与linux间的编辑距离为1，所以使用linx来搜索时也能够搜索到linux的内容。

prefix\_length是定义输入的value中的值在编辑距离匹配过程中前面不参与变化的字符的个数。如果设置prefix\_length=2，那么linux中的li就不参与到编辑距离的变化中，只有后面的nux在编辑距离匹配过程中才发生变化。如果value=linx，prefix\_length=4，就不会搜索到任何结果了。

GET /\_search

{

"query":{

"fuzzy":{

"title":{

"value":"linux",

"fuzziness":2,

"prefix\_length":0

}

}

},

"\_source":["title"]

}

completion suggester

如何使用suggest title来完成自动补全。

Fuzzy queriesedit

The completion suggester also supports fuzzy queries - this means, you can have a typo in your search and still get results back.

其中的song-suggest为变量名称，可以任意指定。prefix为前面的值，可以直接用"text":"linux"。completion中field的不能改变，表示从suggest这个字段中进行搜索，也可以指定fuzziness的值。为了显示的方便，可以指定只显示title的字段。运行搜索，就可以提取出满足要求的title了。只要把这里的搜索结果返回给前端就可以了。

POST jobbole/\_search?pretty

{

"suggest": {

"my-suggest" : {

"prefix" : "nor",

"completion" : {

"field" : "suggest",

"fuzzy" : {

"fuzziness" : 2

}

}

}

},

"\_source":"title"

}

COPY AS CURLVIEW IN CONSOLE

Suggestions that share the longest prefix to the query prefix will be scored higher.

The fuzzy query can take specific fuzzy parameters. The following parameters are supported:

fuzziness

The fuzziness factor, defaults to AUTO. See the section called “Fuzzinessedit” for allowed settings.

transpositions

if set to true, transpositions are counted as one change instead of two, defaults to true

min\_length

Minimum length of the input before fuzzy suggestions are returned, defaults 3

prefix\_length

Minimum length of the input, which is not checked for fuzzy alternatives, defaults to 1

unicode\_aware

If true, all measurements (like fuzzy edit distance, transpositions, and lengths) are measured in Unicode code points instead of in bytes. This is slightly slower than raw bytes, so it is set to false by default.

Note

If you want to stick with the default values, but still use fuzzy, you can either use fuzzy: {} or fuzzy: true.

"""

与上面的用法类似，suggest中首先传入一个变量名，第2个变量就是搜索关键词key\_words，第3个参数是completion。size表示要返回10个结果。

s = s.suggest('my\_suggest', key\_words, completion={

"field":"suggest", "fuzzy":{

"fuzziness":2

},

"size": 10

})

#执行搜索建议，并返回suggestions。suggestions的格式与上面运行POST jobbole/\_search?pretty得出的结果相似。返回的是一个封装后的SuggestResponse对象，这个对象中有一个my-suggest的值。my-suggest的第1个元素是一个list，list中又有一个字典，字典中又有一个options的数组，数组中就保存着运行POST jobbole/\_search?pretty得出的所有搜索建议的结果，可以使用suggestions.my\_suggest[0].options来得到这个保存着搜索建议的列表。

suggestions = s.execute\_suggest()

#这里使用my-suggest会出错，不符合python变量的命名规则。要改为下划线，或者使用for match in getattr(suggestions,"my-suggest")[0].options这种写法。注意这里要与s=s.suggest中的第一个参数my\_suggest保持一致。

for match in suggestions.my\_suggest[0].options:

# 列表中的每一个元素也是一个列表，列表中有一个名为sourse的字典，字典中保存着具体的搜索结果。只取其中的title即可。

source = match.\_source

re\_datas.append(source["title"])

from django.http import HttpResponse

import json

#使用HttpResponse把re\_datas这个list转换为json格式返回给用户。content\_type指定编码。

return HttpResponse(json.dumps(re\_datas), content\_type="application/json")

在search/urls.py中添加与上面的视图函数对应的url匹配规则。

from search.views import SearchSuggest

url(r'^suggest/$', SearchSuggest.as\_view(), name="suggest"),

在index.html中修改suggest\_url的值，可以使用urls.py中定义的名字suggest

var suggest\_url = "{% url 'suggest' %}"

重启服务器，访问127.0.0.1:8000，输入任意一个关键字，就会自动出现搜索提示了。可以通过检查元素看到netword中请求的内容。每变化一个字符就会发送一个请求。点击一个请求，就可以看到其对应的response的内容了。

## 11-5 django实现elasticsearch的搜索功能 -1 (14:06)

首先在search/views.py中定义搜索功能的视图函数。

class SearchView(View):

#重定View的get方法。

def get(self, request):

#获取用户输入的关键词。在index.hmtl中有一个add\_search()的方法，可以获取用户输入的关键词。并把搜索内容保存为搜索记录。

"""

<script>

var searchArr;

//定义一个search的，判断浏览器有无数据存储（搜索历史）

if(localStorage.search){

//如果有，转换成 数组的形式存放到searchArr的数组里（localStorage以字符串的形式存储，所以要把它转换成数组的形式）

searchArr= localStorage.search.split(",")

}else{

//如果没有，则定义searchArr为一个空的数组

searchArr = [];

}

//把存储的数据显示出来作为搜索历史

MapSearchArr();

function add\_search(){

var val = $(".searchInput").val();

if (val.length>=2){

//点击搜索按钮时，去重

KillRepeat(val);

//去重后把数组存储到浏览器localStorage

localStorage.search = searchArr;

//然后再把搜索内容显示出来

MapSearchArr();

}

#通过window.location.href=search\_url直接跳转到search\_url中。search\_url会调用search\_url这个变量，并把用户输入的参数和s\_type的值传送var search\_url = "{% url 'search' %}"

suggest\_url会调用suggest这个变量，变量中就是要在后台处理的url。suggest\_url = "/suggest/"。只要在后台urls中配置好这个url匹配规则和对应的视图函数，就能完成相应的搜索建议功能了。

使用get方法向suggest\_url发送一个请求，把用户输入的内容通过searchText这个参数发送到服务器。在发送的请求中有一个s\_type的值，对应于文章、问答和职位三种类型，选择每一种类型时会提供不同类型的搜索建议。

window.location.href=search\_url+'?q='+val+"&s\_type="+$(".searchItem.current").attr('data-type')

}

"""

#如果取不到关键词，就使其默认为空字符串。

key\_words = request.GET.get("q","")

from elasticsearch import Elasticsearch

#初始化一个ES的连接

client = Elasticsearch(host=["127.0.0.1"])

#使用client.search方法，可以使用ES原始的查询语句。

#这里的response的结构与使用ES原始的查询语句中得出的结构是相似的。可以使用debug信息来查看。

response = client.search(

index= "jobbole",

body={

"query":{

#使用multi\_match对多个字段进行搜索。

"multi\_match":{

"query":key\_words,

#fields中传递要搜索字段。

"fields":["tags", "title", "content"]

}

},

#进行分页，每一页的数据不能过多。

"from":0,

"size":10,

#当搜索到某个词的时候，要对搜索结果中的此关键词进行高亮处理

"highlight": {

#给高亮的词加一个html的标签。在css样式中对此keyword进行高亮变红处理。

"pre\_tags": ['<span class="keyWord">'],

"post\_tags": ['</span>'],

"fields": {

"title": {},

"content": {},

}

}

}

)

#一共搜索到多少条记录

total\_nums = response["hits"]["total"]

hit\_list = []

#从response中取出来其中的值，传递到前端页面中去。response的结构与直接在ES中执行查询操作得出的结果相似。可以根据ES搜索的结果来写获取搜索结果的代码。

for hit in response["hits"]["hits"]:

hit\_dict = {}

#可能会搜索不到输入的关键词，所以就要进行判断。如果存在了，从highlight中去取title。

if "title" in hit["highlight"]:

#如果存在highlight标签，是会放在一个数组中的，所以要取数组中的第1个元素，或者使用join把它转化为字符串。

hit\_dict["title"] = "".join(hit["highlight"]["title"])

#如果在highlight中搜索不到title的值，就从\_source中去取。

else:

hit\_dict["title"] = hit["\_source"]["title"]

if "content" in hit["highlight"]:

#content的内容很长，只取前500个字符

hit\_dict["content"] = "".join(hit["highlight"]["content"])[:500]

else:

hit\_dict["content"] = hit["\_source"]["content"][:500]

#取出来发布时间

hit\_dict["create\_date"] = hit["\_source"]["create\_date"]

#对应的url

hit\_dict["url"] = hit["\_source"]["url"]

#得分

hit\_dict["score"] = hit["\_score"]

hit\_list.append(hit\_dict)

return render(request, "result.html", {

"all\_hits":hit\_list,

"key\_words":key\_words, #搜索关键词也要传递回去，显示在搜索栏中。

})

修改result.html页面，把静态文件使用static标签来处理。

搜索的结果是放在div class="resultItem"中的。原始页面中为了展示，有8个resultItem的记录。删除其中的7个，只留下1个resultItem的结构，使用循环接收从views.py中传递过来的搜索结果数据。

<div class="resultList">

{% for hit in all\_hits %}

<div class="resultItem">

<div class="itemHead">

#要把django的autoescape给关闭掉，否则django会对搜索结果页面中html标签进行转译，会显示为html标签，而不是其效果。

<a href="{{ hit.url }}" target="\_blank" class="title">{% autoescape off %}{{ hit.title }}{% endautoescape %}</a>

<span class="divsion">-</span>

<span class="fileType">

<span class="label">来源：</span>

<span class="value">伯乐在线</span>

</span>

<span class="dependValue">

<span class="label">得分：</span>

<span class="value">{{ hit.score }}</span>

</span>

</div>

<div class="itemBody">

#关闭autoescape

{% autoescape off %}{{ hit.content }}{% endautoescape %}

</div>

<div class="itemFoot">

<span class="info">

<label>网站：</label>

<span class="value">伯乐在线</span>

</span>

<span class="info">

<label>发布时间：</label>

<span class="value">2017-04-23</span>

</span>

</div>

</div>

{% endfor %}

</div>

在search/urls.py中添加search对应的url匹配规则

from search.views import SearchView

url(r'^search/$', SearchView.as\_view(), name="search"),

在index.html中把var search\_url = "/search/"修改为{% url 'search' %}

为了在搜索页面中实现把输入的搜索关键词放在输入框中，需要把searchInput中的value改为key\_words

<div class="inputArea">

<input type="text" class="searchInput" value="{{ key\_words }}"/>

<input type="button" class="searchButton" onclick="add\_search()"/>

</div>

在搜索结果页中也是可以进行搜索的，所以要把var search\_url修改为var search\_url = "{% url 'search' %}"

打开主页，进行修改，发现content中的内容很乱，有很多的空格，和未解析出来的html标签。

## 11-7 django实现搜索结果分页 (09:12)

搜索结果中的分页是通过js来完成的。

//分页

$(".pagination").pagination(56, {

current\_page :1, //当前页码

items\_per\_page :10,

display\_msg :true,

callback :pageselectCallback

});

function pageselectCallback(page\_id, jq) {

window.location.href=search\_url+'?q='+key\_words+'&p='+page\_id

}

获取到.pagination之后，调用pagination这个函数，第一个参数是数据的总量，第二个参数是当前的页码，第三个参数是每页显示的数据数。callback函数会调用pageselectCallback函数，在pageselectCallback函数中实现了url的跳转。在点击某一个页面的时候，会跳转到对应的页面中去。

在search/views.py中修改SearchView。

class SearchView(View):

#重定View的get方法。

def get(self, request):

#获取用户输入的关键词。在index.hmtl中有一个add\_search()的方法，可以获取用户输入的关键词。并把搜索内容保存为搜索记录。

"""

<script>

var searchArr;

//定义一个search的，判断浏览器有无数据存储（搜索历史）

if(localStorage.search){

//如果有，转换成 数组的形式存放到searchArr的数组里（localStorage以字符串的形式存储，所以要把它转换成数组的形式）

searchArr= localStorage.search.split(",")

}else{

//如果没有，则定义searchArr为一个空的数组

searchArr = [];

}

//把存储的数据显示出来作为搜索历史

MapSearchArr();

function add\_search(){

var val = $(".searchInput").val();

if (val.length>=2){

//点击搜索按钮时，去重

KillRepeat(val);

//去重后把数组存储到浏览器localStorage

localStorage.search = searchArr;

//然后再把搜索内容显示出来

MapSearchArr();

}

#通过window.location.href=search\_url直接跳转到search\_url中。search\_url会调用search\_url这个变量，并把用户输入的参数和s\_type的值传送var search\_url = "{% url 'search' %}"

suggest\_url会调用suggest这个变量，变量中就是要在后台处理的url。suggest\_url = "/suggest/"。只要在后台urls中配置好这个url匹配规则和对应的视图函数，就能完成相应的搜索建议功能了。

使用get方法向suggest\_url发送一个请求，把用户输入的内容通过searchText这个参数发送到服务器。在发送的请求中有一个s\_type的值，对应于文章、问答和职位三种类型，选择每一种类型时会提供不同类型的搜索建议。

window.location.href=search\_url+'?q='+val+"&s\_type="+$(".searchItem.current").attr('data-type')

}

"""

#如果取不到关键词，就使其默认为空字符串。

key\_words = request.GET.get("q","")

#要获取第几页的数据，默认是第1页的数据

page = request.GET.get("p", "1")

#如果p传递的是一个字符，不是一个数字，就会出现异常。如果出现异常，就跳转到第1页中。

try:

page = int(page)

except:

page = 1

jobbole\_count = redis\_cli.get("jobbole\_count")

#设置一个start\_time和后面的end\_time，来计算搜索过程一共使用了多少时间。

start\_time = datetime.now()

from elasticsearch import Elasticsearch

#初始化一个ES的连接

client = Elasticsearch(host=["127.0.0.1"])

#使用client.search方法，可以使用ES原始的查询语句。

#这里的response的结构与使用ES原始的查询语句中得出的结构是相似的。可以使用debug信息来查看。

response = client.search(

index= "jobbole",

body={

"query":{

#使用multi\_match对多个字段进行搜索。

"multi\_match":{

"query":key\_words,

#fields中传递要搜索字段。

"fields":["tags", "title", "content"]

}

},

#进行分页，每一页的数据不能过多。

#由于下标是从0开始的，如果是第1页，from就为0，如果是第2页，from就要从10开始。

"from":(page-1)\*10,

"size":10,

#当搜索到某个词的时候，要对搜索结果中的此关键词进行高亮处理

"highlight": {

#给高亮的词加一个html的标签。在css样式中对此keyword进行高亮变红处理。

"pre\_tags": ['<span class="keyWord">'],

"post\_tags": ['</span>'],

"fields": {

"title": {},

"content": {},

}

}

}

)

end\_time = datetime.now()

#搜索过程一共使用了多少秒。

last\_seconds = (end\_time-start\_time).total\_seconds()

#一共搜索到多少条记录

total\_nums = response["hits"]["total"]

#通过total\_nums来计算一共搜索到多少页的数据

if (page%10) > 0:

page\_nums = int(total\_nums/10) +1

else:

page\_nums = int(total\_nums/10)

hit\_list = []

#从response中取出来其中的值，传递到前端页面中去。response的结构与直接在ES中执行查询操作得出的结果相似。可以根据ES搜索的结果来写获取搜索结果的代码。

for hit in response["hits"]["hits"]:

hit\_dict = {}

#可能会搜索不到输入的关键词，所以就要进行判断。如果存在了，从highlight中去取title。

if "title" in hit["highlight"]:

#如果存在highlight标签，是会放在一个数组中的，所以要取数组中的第1个元素，或者使用join把它转化为字符串。

hit\_dict["title"] = "".join(hit["highlight"]["title"])

#如果在highlight中搜索不到title的值，就从\_source中去取。

else:

hit\_dict["title"] = hit["\_source"]["title"]

if "content" in hit["highlight"]:

#content的内容很长，只取前500个字符

hit\_dict["content"] = "".join(hit["highlight"]["content"])[:500]

else:

hit\_dict["content"] = hit["\_source"]["content"][:500]

#取出来发布时间

hit\_dict["create\_date"] = hit["\_source"]["create\_date"]

#对应的url

hit\_dict["url"] = hit["\_source"]["url"]

#得分

hit\_dict["score"] = hit["\_score"]

hit\_list.append(hit\_dict)

return render(request, "result.html", {"page":page, #获取某一页的数据

"all\_hits":hit\_list,

"key\_words":key\_words, #搜索关键词也要传递回去，显示在搜索栏中。

"total\_nums":total\_nums, #搜索到的总的数量，要传到result页面中。

"page\_nums":page\_nums, #搜索结果可以分成多少页

"last\_seconds":last\_seconds, #搜索过程一共使用了多少秒。

"jobbole\_count":jobbole\_count,

"topn\_search":topn\_search})

修改result.html中的分布数据，使用django标签来进行填充

//分页

$(".pagination").pagination({{ total\_nums }}, {

current\_page :{{ page|add:'-1' }}, //当前页码，第1页是page=0，这里要调用django filter中的add方法减1。

items\_per\_page :10,

display\_msg :true,

callback :pageselectCallback

});

function pageselectCallback(page\_id, jq) {

window.location.href=search\_url+'?q='+key\_words+'&p='+page\_id

}

修改result.html中，把找到的结果数填充到页面中

<p class="resultTotal">

<span class="info">找到约&nbsp;<span class="totalResult">{{ total\_nums }}</span>&nbsp;条结果(用时<span class="time">{{ last\_seconds }}</span>秒)，共约<span class="totalPage">{{ page\_nums }}</span>页</span>

</p>

## 11-8 搜索记录、热门搜索功能实现 - 1 (14:34)

实现我的搜索中的搜索记录，每次进行一次搜索，都会记录这个搜索并把它排列到我的搜索的第一位。

这里是在前端中使用js来实现的，并不是通过后端实现的，也就是说，并不是把用户的搜索记录都保存到服务器中，而是通过本地浏览器的缓存实现的。

在用户输入内容，点击搜索的时候，会触发一个事件，叫add\_search()，在add\_search中通过一个数组把搜索记录保存在本地。

如果搜索了同样的内容，需要先把之间相同的搜索记录给删除。同时把最新的搜索记录放在队列中的第1个的位置上。

function add\_search(){

var val = $(".searchInput").val();

if (val.length>=2){

//点击搜索按钮时，去重

KillRepeat(val);

//去重后把数组存储到浏览器localStorage

localStorage.search = searchArr;

//然后再把搜索内容显示出来

MapSearchArr();

}

//去重。

function KillRepeat(val){

var kill = 0;

for (var i=0;i<searchArr.length;i++){

#判断数组中的值是否存在。如果存在，就把kill的值加1

if(val===searchArr[i]){

kill ++;

}

}

#如果kill==0，也就是搜索了一个在缓存中不存在的新词，把它放到搜索记录的队列的头部。

if(kill<1){

searchArr.unshift(val);

#如果kill>0，表示搜索记录已经存在。就要把记录从数组中删除掉，同时把这个词放到搜索记录的头部。

}else {

removeByValue(searchArr, val)

searchArr.unshift(val)

}

}

# removeByValue就是通过遍历数组来查找这个词，如果找到了，就用arr.splice把它删除掉。

function removeByValue(arr, val) {

for(var i=0; i<arr.length; i++) {

if(arr[i] == val) {

arr.splice(i, 1);

break;

}

}

}

这样就实现了搜索记录的保存，还要给每一个搜索记录添加一个url，点击相应的url会对这个记录进行搜索并跳转到它对应的搜索结果页面中，在add\_search中调用MapSearchArr这个方法。在MapSearchArr这个方法中添加一些a标签，增加一些html的内容，在a标签中把search\_url这个变量和数组中的值searchArr[i]拼接到一起，成为每个搜索记录对应的url。

function MapSearchArr(){

var tmpHtml = "";

var arrLen = 0

if (searchArr.length > 6){

arrLen = 6

}else {

arrLen = searchArr.length

}

for (var i=0;i<arrLen;i++){

tmpHtml += '<li><a href="'+search\_url+'?q='+searchArr[i]+'">'+searchArr[i]+'</a></li>'

}

$(".mySearch .historyList").append(tmpHtml);

}

在左边栏中的网站中，展示出爬虫在每个网站上爬取的数据量。如何来实现这个统计功能呢。

有很多方法可以实现这个功能，如在search/views.py中的SearchView中添加一个ES的query查询，就可以查询到每个index中存在多少数据量。

这里要使用redis来完成这个功能。

左边栏中的数据是每个请求页面都要加载的内容，并且它不是经常变动的，就可以把它放在redis的缓存中，这样不但能提高效率，还能减少对ES数据库的查询量，从而减小服务器的压力。

在utils/items.py中，以jobbole为例来进行说明。

在jobBoleArticleItem中，爬虫每爬取到一个值之后，就通过save\_to\_es保存到ES数据库中，可以在save\_to\_es中添加一个处理，每向ES中添加一个数据，就使用一个Redis中全局的变量jobbole\_count向

redis中可以对变量进行赋值，进行加1操作。

首先要在虚拟环境中安装redis。

pip install redis

修改ArticleSpider项目中的utils/items.py

import redis

#创建一个redis的连接

redis\_cli = redis.StrictRedis()

#修改save\_to\_es

def save\_to\_es(self):

article = ArticleType()

article.title = self['title']

article.create\_date = self["create\_date"]

article.content = remove\_tags(self["content"])

article.front\_image\_url = self["front\_image\_url"]

if "front\_image\_path" in self:

article.front\_image\_path = self["front\_image\_path"]

article.praise\_nums = self["praise\_nums"]

article.fav\_nums = self["fav\_nums"]

article.comment\_nums = self["comment\_nums"]

article.url = self["url"]

article.tags = self["tags"]

article.meta.id = self["url\_object\_id"]

article.suggest = gen\_suggests(ArticleType.\_doc\_type.index, ((article.title,10),(article.tags, 7)))

article.save()

#使用redis的incr方法给jobbole\_count这个值加1。

redis\_cli.incr("jobbole\_count")

return

新建tools/test.py，测试上面的redis代码是否可用。

import redis

redis\_cli = redis.StrictRedis()

redis\_cli.incr("jobbole\_count")

直接运行此代码，没有报错，在命令行中打开redis-cli，查看redis中是否有数据。

keys \* > "jobbole\_count"

type jobbole\_count > string

get jobbole\_count > "1"

再次运行test.py，再次查看jobbole\_count的值。

get jobbole\_count > "2"

在LcvSearch的views.py中就可以修改SearchView来获取redis中的数据了。

在本虚拟环境中安装redis

import redis

redis\_cli = redis.StrictRedis()

class SearchView(View):

def get(self, request):

key\_words = request.GET.get("q","")

page = request.GET.get("p", "1")

try:

page = int(page)

except:

page = 1

#获取redis中的数据

jobbole\_count = redis\_cli.get("jobbole\_count")

start\_time = datetime.now()

response = client.search(...

...

return render(request, "result.html", {"page":page,

"all\_hits":hit\_list,

"key\_words":key\_words,

"total\_nums":total\_nums,

"page\_nums":page\_nums,

"last\_seconds":last\_seconds,

"jobbole\_count":jobbole\_count, #获取jobbole中的记录数。

)

修改result.html，把获取的jobbole\_count的值放在result.html中。

<div class="sideBar">

<div class="subfield">网站</div>

<ul class="subfieldContext">

<li>

<span class="name">伯乐在线</span>

<span class="unit">({{ jobbole\_count }})</span>

</li>

<li>

<span class="name">知乎</span>

<span class="unit">(9862)</span>

</li>

<li>

<span class="name">拉勾网</span>

<span class="unit">(9862)</span>

</li>

<li class="more">

<a href="javascript:;">

<span class="text">更多</span>

<i class="moreIcon"></i>

</a>

</li>

</ul>

<div class="sideBarShowHide">

<a href="javascript:;" class="icon"></a>

</div>

</div>

重启服务器，访问127.0.0.1:8000，就能看到伯乐在线中的数据量了。

可以使用同样的方法来完善知乎和拉勾中的代码。

## 11-9 热门搜索功能的实现

也可以使用redis来完成热门搜索的功能。是一种典型的top N的功能，使用redis可以简单高效的实现。

在redis中有一种可以排序的sort即SortedSet

http://redisdoc.com/sorted\_set/index.html

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

一个是ZINCRBY,

Redis Zincrby 命令对有序集合中指定成员的分数加上增量 increment

每进行一次关键词的搜索，就可以使用Zincrby命令对相同的搜索关键词加上1，然后只取分数最高的top N就可以实现热门搜索的功能了。

一个是zrevrangebyscore

Redis Zrevrangebyscore 返回有序集中指定分数区间内的所有的成员。有序集成员按分数值递减(从大到小)的次序排列。

具有相同分数值的成员按字典序的逆序(reverse lexicographical order )排列。

ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

ZREVRANGEBYSCORE salary +inf -inf # 逆序排列所有成员，因为不知道最大的搜索和最少的搜索都是哪些

修改views.py中的SearchView。

class SearchView(View):

def get(self, request):

key\_words = request.GET.get("q","")

#完成搜索关键词的加1操作。

#对搜索关键词key\_words做一个zincrby，第一个变量是搜索关键词在redis中的变量名，第2个参数是搜索关键词。

redis\_cli.zincrby("search\_keywords\_set", key\_words)

#取出搜索频率最高的前5个词。

topn\_search = redis\_cli.zrevrangebyscore("search\_keywords\_set", "+inf", "-inf", start=0, num=5)

page = request.GET.get("p", "1")

...

return render(request, "result.html", {"page":page,

"all\_hits":hit\_list,

"key\_words":key\_words,

"total\_nums":total\_nums,

"page\_nums":page\_nums,

"last\_seconds":last\_seconds,

"jobbole\_count":jobbole\_count,

"topn\_search":topn\_search}) #搜索最多的5个关键词。

之间在urls.py中写了一个静态的view，即TemplateView.as\_view，现在要添加功能，就不能再使用django默认的view函数了，需要自定义一个view视图函数。

修改views.py，添加一个IndexView的类。

class IndexView(View):

#首页的热闹搜索函数

def get(self, request):

topn\_search = redis\_cli.zrevrangebyscore("search\_keywords\_set", "+inf", "-inf", start=0, num=5)

return render(request, "index.html", {"topn\_search":topn\_search})

修改urls.py文件，添加IndexView的url匹配规则。

from search.views import SearchSuggest, SearchView, IndexView

urlpatterns = [

url(r'^admin/', admin.site.urls),

url(r'^$', IndexView.as\_view(), name="index"),

url(r'^suggest/$', SearchSuggest.as\_view(), name="suggest"),

url(r'^search/$', SearchView.as\_view(), name="search"),

]

修改result.html和index.html，遍历topn，把取到的数据放在右边栏的热门搜索中。

<div class="hotSearch">

<h6>热门搜索</h6>

<ul class="historyList">

{% for search\_word in topn\_search %}

<li><a href="/search?q={{ search\_word }}">{{ search\_word }}</a></li>

{% endfor %}

</ul>

</div>

现在打开首页，进行几次搜索，查看热门搜索和我的搜索的变化。

最后一步是实现文章，问答和职位搜索。

在首页中点击问答，进行搜索，可以看到url的变化。

http://127.0.0.1:8000/search/?q=linux&s\_type=question

使用s\_type来区分这3种不同的搜索。这里选择的内容会传递到views.py中的SearchView中。需要在其中添加字段来接收传递过去的s\_type这个变量。

class SearchView(View):

def get(self, request):

key\_words = request.GET.get("q","")

#接收搜索的分类，默认为对article进行搜索。

s\_type = request.GET.get("s\_type", "article")

redis\_cli.zincrby("search\_keywords\_set", key\_words)

...

有了s\_type这个参数，就可以对不同的index进行设置，jobbole，zhihu和lagou。设置完成后统一将搜索到的内容中的关键词进行高亮显示。

在github有这个项目，

https://github.com/holgerd77/django-dynamic-scraper

这个项目可以通过django来控制scrapy。

# 第12章 scrapyd部署scrapy爬虫

https://github.com/scrapy/scrapyd

scrapyd是scrapy官方提供的部署scrapy的一个系统，它可以用来部署scrapy的项目并使用http json api去控制spider

安装scrapyd

进入开发ArticleSpider的虚拟环境，必须要保证这个虚拟环境能够正常运行scrapy项目。

在部署爬虫的时候，需要两个包，一个是scrapyd，它是一个服务器。二是scrapyd-client。scrapy-client允许将本地的一个scrapy项目打包发送到scrapyd的服务器中。scrapyd也是将scrapy-client上传的scrapy项目放在虚拟环境中运行的。所以首先要保证虚拟环境能够运行scrapy项目。

pip install scrapyd

scrapyd是安装在虚拟环境下的Scripts目录中。windows中有一个scrapyd.exe文件。

想要运行scrapyd，就要进入这个虚拟环境。

新建一个文件夹scrapyd，进入到这个文件夹中。运行scrapyd。会在127.0.0.1:6800地址上启动一个server。

访问这个地址，在jobs中就可以看到当前运行的所有任务。由于没有部署任务，所以jobs中所有项目都为空。

安装scrapyd-client。

因为scrapyd-client只是用来打包项目到服务器的，所以可以不用安装到虚拟环境中。但还是建议安装到虚拟环境中去，方便对某个项目的包进行统一管理。

进入到项目根目录ArticleSpider中。

在scrapy项目根目录下有一个scrapy.cfg文件。其中的[deploy]就是为scrapyd服务的。把url前的注释删除掉。可以给deploy定义一个名称。

[deploy:davidscrapy]

url = http://localhost:6800/

project = ArticleSpider

使用pip安装过scrapyd-client之后，会在python的Scripts目录下添加一个scrapyd-deploy的文件。如果是在linux下，这个文件可以直接运行。在windows下需要做如下处理。

在scrapyd-deploy同目录下新建一个scrapyd-deploy.bat文件。第一个路径是虚拟环境中的python.exe，第二个路径是scrapyd-deploy的全路径。

@echo off

"D:\Envs\article\_spider\Scripts\python.exe" "D:\Envs\article\_spider\Scripts\scrapyd-deploy" %1 %2 %3 %4 %5 %6 %7 %8 %9

然后再在此目录下运行scrapyd-deploy，就可以使用此命令了。

scrapyd-deploy -l

显示deploy的名字和服务器的地址

davidscrapy http://localhost:6800/

在部署之前必须要确定scrapy list这个命令可用。

如果在运行这个命令时遇到了import的问题，就要确认一下settings中是否加入了以下代码，即要把ArticleSpider项目根目录中的同名文件夹ArticleSpider设置为python可以搜索的路径。

import sys, os

BASE\_DIR = os.path.dirname(os.path.abspath(os.path.dirname(\_\_file\_\_)))

sys.path.insert(0, os.path.join(BASE\_DIR, 'ArticleSpider'))

开始部署项目

scrapyd-deploy davidscrapy -p ArticleSpider

其中第一个参数就是在scrapy.cfg中定义的deploy的名称，-p表示project，-p后面的名称要与scrapy.cfg中的project=中的名称保持一致。

会出现部署的一些信息。出现Server response 200，就表示部署成功了。

在之前运行scrapyd的目录中就会出现一个eggs/ArticleSpider/1234114.egg的文件。

scrapyd-client部署项目完成的工作就是将本地的scrapy项目打包成一个egg文件传递到scrapyd服务器中去。

以上只是把项目部署到scrapyd服务器中了，还没有开始运行项目。

使用scrapyd的命令来启动项目。

http://scrapyd.readthedocs.io/en/stable/api.html

这个网页中包含scrapyd的命令

daemonstatus.json命令可以显示scrapy项目运行的状态。

curl http://localhost:6800/daemonstatus.json

注意这个命令只能在linux下运行，在windows中要在cmder中运行，或者安装windows版本的curl。

schedule.json命令可以运行scrapyd服务器中某个项目下的某个spider。scrapyd是可以同时管理多个scrapy项目的。

curl http://localhost:6800/schedule.json -d project=ArticleSpider -d spider=jobbole

{"status": "ok", "node\_name": "PC201403152008", "jobid": "785738ge2a9611e79730e03f491b30e2"}

要记下返回的jobid参数。

在127.0.0.1:6800/jobs中查看运行的状态。

运行的时间很短，说明出现了问题，可以查看log信息中的错误。

\_\_init\_\_() got an unexpected keyword argument '\_job'

在jobbole.py这个spider中声明了一个\_\_init\_\_方法，要添加一个\*\*kwargs参数才行。

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

为了重新运行此项目，要删除掉原来部署的项目重新部署。

delproject.json

curl http://localhost:6800/delproject.json -d project=ArticleSpider

重新部署项目

scrapyd-deploy davidscrapy -p ArticleSpider

重新运行项目

curl http://localhost:6800/schedule.json -d project=ArticleSpider -d spider=jobbole

在127.0.0.1:6800/jobs中查看运行的状态。可以看到jobbole这个spider正在运行。查看log信息，就是之前在运行spider时出现的信息。说明spider的运行成功了。

取消项目运行

cancel.json

curl http://localhost:6800/cancel.json -d project=ArticleSpider -d job=ef23cdfasfd2dcvdfasdf

其中的job就是之前在运行时返回的jobid。

在127.0.0.1:6800/jobs中查看运行的状态。可以看到jobbole这个spider已经停止运行了。

查看所有项目

listprojects.json

curl http://localhost:6800/listprojects.json

查看所有的spiders

listspiders.json

curl http://localhost:6800/listspiders.json?project=ArticleSpider

查看所有的job

listjobs.json

curl http://localhost:6800/listjobs.json?project=ArticleSpider