# [1. 第一章：网络请求](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\chapter1.html)

## [1.1. 爬虫前奏](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\chapter1\13001-pa-chong-qian-zou.html)

### 爬虫的实际例子：

1. 搜索引擎 (百度、谷歌、360搜索等) 。
2. 伯乐在线。
3. 惠惠购物助手。
4. 数据分析与研究 (数据冰山知乎专栏) 。
5. 抢票软件等。

### 什么是网络爬虫：

1. 通俗理解：爬虫是一个模拟人类请求网站行为的程序。可以自动请求网页、并数据抓取下来, 然后使用一定的规则提取有价值的数据。
2. 专业介绍：[百度百科](https://baike.baidu.com/item/网络爬虫/5162711?fr=aladdin)。

### 通用爬虫和聚焦爬虫：

1. 通用爬虫：通用爬虫是搜索引擎抓取系统 (百度、谷歌、搜狗等) 的重要组成部分。主要是将互联网上的网页下载到本地, 形成一个互联网内容的镜像备份。
2. 聚焦爬虫：是面向特定需求的一种网络爬虫程序, 他与通用爬虫的区别在于：聚焦爬虫在实施网页抓取的时候会对内容进行筛选和处理, 尽量保证只抓取与需求相关的网页信息。

### 为什么用Python写爬虫程序：

1. PHP：PHP是世界是最好的语言, 但他天生不是做这个的, 而且对多线程、异步支持不是很好, 并发处理能力弱。爬虫是工具性程序, 对速度和效率要求比较高。
2. Java：生态圈很完善, 是Python爬虫最大的竞争对手。但是Java语言本身很笨重, 代码量很大。重构成本比较高, 任何修改会导致代码大量改动。爬虫经常要修改采集代码。
3. C/C++：运行效率是无敌的。但是学习和开发成本高。写个小爬虫程序可能要大半天时间。
4. Python：语法优美、代码简洁、开发效率高、支持的模块多。相关的HTTP请求模块和HTML解析模块非常丰富。还有Scrapy和Scrapy-redis框架让我们开发爬虫变得异常简单。

### 准备工具：

1. Python3.6开发环境。
2. Pycharm 2017 professional版。
3. 虚拟环境。`virtualenv/virtualenvwrapper`。

## [1.2. http协议和chrome浏览器](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\chapter1\2-httpxie-yi-hechrome-liu-lan-qi.html)

### 什么是http和https协议：

HTTP协议：全称是HyperText Transfer Protocol, 中文意思是超文本传输协议, 是一种发布和接收HTML页面的方法。服务器端口号是80端口。 HTTPS协议：是HTTP协议的加密版本, 在HTTP下加入了SSL层。服务器端口号是443端口。

### 在浏览器中发送一个http请求的过程：

1. 当用户在浏览器的地址栏中输入一个URL并按回车键之后, 浏览器会向HTTP服务器发送HTTP请求。HTTP请求主要分为"Get"和"Post"两种方法。
2. 当我们在浏览器输入URL <http://www.baidu.com> 的时候, 浏览器发送一个Request请求去获取 <http://www.baidu.com> 的html文件, 服务器把Response文件对象发送回给浏览器。
3. 浏览器分析Response中的 HTML, 发现其中引用了很多其他文件, 比如Images文件, CSS文件, JS文件。 浏览器会自动再次发送Request去获取图片, CSS文件, 或者JS文件。
4. 当所有的文件都下载成功后, 网页会根据HTML语法结构, 完整的显示出来了。

### url详解：

URL是Uniform Resource Locator的简写, 统一资源定位符。 一个URL由以下几部分组成：

scheme://host:port/path/?query-string=xxx#anchor

* **scheme**：代表的是访问的协议, 一般为http或者https以及ftp等。
* **host**：主机名, 域名, 比如www.baidu.com。
* **port**：端口号。当你访问一个网站的时候, 浏览器默认使用80端口。
* **path**：查找路径。比如：www.jianshu.com/trending/now, 后面的trending/now就是path。
* **query-string**：查询字符串, 比如：www.baidu.com/s?wd=python, 后面的wd=python就是查询字符串。
* **anchor**：锚点, 后台一般不用管, 前端用来做页面定位的。

在浏览器中请求一个url, 浏览器会对这个url进行一个编码。除英文字母, 数字和部分符号外, 其他的全部使用百分号+十六进制码值进行编码。

### 常用的请求方法：

在Http协议中, 定义了八种请求方法。这里介绍两种常用的请求方法, 分别是get请求和post请求。

1. get请求：一般情况下, 只从服务器获取数据下来, 并不会对服务器资源产生任何影响的时候会使用get请求。
2. post请求：向服务器发送数据 (登录) 、上传文件等, 会对服务器资源产生影响的时候会使用post请求。 以上是在网站开发中常用的两种方法。并且一般情况下都会遵循使用的原则。但是有的网站和服务器为了做反爬虫机制, 也经常会不按常理出牌, 有可能一个应该使用get方法的请求就一定要改成post请求, 这个要视情况而定。

### 请求头常见参数：

在http协议中, 向服务器发送一个请求, 数据分为三部分, 第一个是把数据放在url中, 第二个是把数据放在body中 (在post请求中) , 第三个就是把数据放在head中。这里介绍在网络爬虫中经常会用到的一些请求头参数：

1. User-Agent：浏览器名称。这个在网络爬虫中经常会被使用到。请求一个网页的时候, 服务器通过这个参数就可以知道这个请求是由哪种浏览器发送的。如果我们是通过爬虫发送请求, 那么我们的User-Agent就是Python, 这对于那些有反爬虫机制的网站来说, 可以轻易的判断你这个请求是爬虫。因此我们要经常设置这个值为一些浏览器的值, 来伪装我们的爬虫。
2. Referer：表明当前这个请求是从哪个url过来的。这个一般也可以用来做反爬虫技术。如果不是从指定页面过来的, 那么就不做相关的响应。
3. Cookie：http协议是无状态的。也就是同一个人发送了两次请求, 服务器没有能力知道这两个请求是否来自同一个人。因此这时候就用cookie来做标识。一般如果想要做登录后才能访问的网站, 那么就需要发送cookie信息了。

### 常见响应状态码：

1. 200：请求正常, 服务器正常的返回数据。
2. 301：永久重定向。比如在访问www.jingdong.com的时候会重定向到www.jd.com。
3. 302：临时重定向。比如在访问一个需要登录的页面的时候, 而此时没有登录, 那么就会重定向到登录页面。
4. 400：请求的url在服务器上找不到。换句话说就是请求url错误。
5. 403：服务器拒绝访问, 权限不够。
6. 500：服务器内部错误。可能是服务器出现bug了。

### Chrome抓包工具：

Chrome浏览器是一个非常亲近开发者的浏览器。可以方便的查看网络请求以及发送的参数。对着网页右键->检查。然后就可以打开开发者选项。以下用图片来说明。

## [1.3. urllib库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\chapter1\3-urllibhe-urllib2-ku.html)

urllib库是Python中一个最基本的网络请求库。可以模拟浏览器的行为, 向指定的服务器发送一个请求, 并可以保存服务器返回的数据。

### urlopen函数：

在Python3的urllib库中, 所有和网络请求相关的方法, 都被集到urllib.request模块下面了, 以先来看下urlopen函数基本的使用：

from urllib import request

resp = request.urlopen('http://www.baidu.com')

print(resp.read())

实际上, 使用浏览器访问百度, 右键查看源代码。你会发现, 跟我们刚才打印出来的数据是一模一样的。也就是说, 上面的三行代码就已经帮我们把百度的首页的全部代码爬下来了。一个基本的url请求对应的python代码真的非常简单。

以下对urlopen函数的进行详细讲解：

1. url：请求的url。
2. data：请求的data, 如果设置了这个值, 那么将变成post请求。
3. 返回值：返回值是一个http.client.HTTPResponse对象, 这个对象是一个类文件句柄对象。有read(size)、readline、readlines以及getcode等方法。

### urlretrieve函数：

这个函数可以方便的将网页上的一个文件保存到本地。以下代码可以非常方便的将百度的首页下载到本地：

from urllib import request

request.urlretrieve('http://www.baidu.com/','baidu.html')

### urlencode函数：

用浏览器发送请求的时候, 如果url中包含了中文或者其他特殊字符, 那么浏览器会自动的给我们进行编码。而如果使用代码发送请求, 那么就必须手动的进行编码, 这时候就应该使用urlencode函数来实现。urlencode可以把字典数据转换为URL编码的数据。示例代码如下：

from urllib import parse

data = {'name':'爬虫基础','greet':'hello world','age':100}

qs = parse.urlencode(data)

print(qs)

### parse\_qs函数：

可以将经过编码后的url参数进行解码。示例代码如下：

from urllib import parse

qs = "name=%E7%88%AC%E8%99%AB%E5%9F%BA%E7%A1%80&greet=hello+world&age=100"

print(parse.parse\_qs(qs))

### urlparse和urlsplit：

有时候拿到一个url, 想要对这个url中的各个组成部分进行分割, 那么这时候就可以使用urlparse或者是urlsplit来进行分割。示例代码如下：

from urllib import request,parse

url = 'http://www.baidu.com/s?username=zhiliao'

result = parse.urlsplit(url)

# result = parse.urlparse(url)

print('scheme:',result.scheme)

print('netloc:',result.netloc)

print('path:',result.path)

print('query:',result.query)

urlparse和urlsplit基本上是一模一样的。唯一不一样的地方是, urlparse里面多了一个params属性, 而urlsplit没有这个params属性。比如有一个url为：url = 'http://www.baidu.com/s;hello?wd=python&username=abc#1',

那么urlparse可以获取到hello, 而urlsplit不可以获取到。url中的params也用得比较少。

### request.Request类：

如果想要在请求的时候增加一些请求头, 那么就必须使用request.Request类来实现。比如要增加一个User-Agent, 示例代码如下：

from urllib import request

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

req = request.Request("http://www.baidu.com/",headers=headers)

resp = request.urlopen(req)

print(resp.read())

### 内涵段子爬虫实战作业：

1. url链接：<http://neihanshequ.com/bar/1/>
2. 要求：能爬取一页的数据就可以了。

### ProxyHandler处理器 (代理设置)

很多网站会检测某一段时间某个IP的访问次数(通过流量统计, 系统日志等), 如果访问次数多的不像正常人, 它会禁止这个IP的访问。

所以我们可以设置一些代理服务器, 每隔一段时间换一个代理, 就算IP被禁止, 依然可以换个IP继续爬取。

urllib中通过ProxyHandler来设置使用代理服务器, 下面代码说明如何使用自定义opener来使用代理：

from urllib import request

# 这个是没有使用代理的

# resp = request.urlopen('http://httpbin.org/get')

# print(resp.read().decode("utf-8"))

# 这个是使用了代理的

handler = request.ProxyHandler({"http":"218.66.161.88:31769"})

opener = request.build\_opener(handler)

req = request.Request("http://httpbin.org/ip")

resp = opener.open(req)

print(resp.read())

常用的代理有：

* 西刺免费代理IP：<http://www.xicidaili.com/>
* 快代理：<http://www.kuaidaili.com/>
* 代理云：<http://www.dailiyun.com/>

### 什么是cookie：

在网站中, http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后, 第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题, 第一次登录后服务器返回一些数据 (cookie) 给浏览器, 然后浏览器保存在本地, 当该用户发送第二次请求的时候, 就会自动的把上次请求存储的cookie数据自动的携带给服务器, 服务器通过浏览器携带的数据就能判断当前用户是哪个了。cookie存储的数据量有限, 不同的浏览器有不同的存储大小, 但一般不超过4KB。因此使用cookie只能存储一些小量的数据。

#### cookie的格式：

Set-Cookie: NAME=VALUE；Expires/Max-age=DATE；Path=PATH；Domain=DOMAIN\_NAME；SECURE

参数意义：

* NAME：cookie的名字。
* VALUE：cookie的值。
* Expires：cookie的过期时间。
* Path：cookie作用的路径。
* Domain：cookie作用的域名。
* SECURE：是否只在https协议下起作用。

### 使用cookielib库和HTTPCookieProcessor模拟登录：

Cookie 是指网站服务器为了辨别用户身份和进行Session跟踪, 而储存在用户浏览器上的文本文件, Cookie可以保持登录信息到用户下次与服务器的会话。

这里以人人网为例。人人网中, 要访问某个人的主页, 必须先登录才能访问, 登录说白了就是要有cookie信息。那么如果我们想要用代码的方式访问, 就必须要有正确的cookie信息才能访问。解决方案有两种, 第一种是使用浏览器访问, 然后将cookie信息复制下来, 放到headers中。示例代码如下：

from urllib import request

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36',

'Cookie': 'anonymid=jacdwz2x-8bjldx; depovince=GW; \_r01\_=1; \_ga=GA1.2.1455063316.1511436360; \_gid=GA1.2.862627163.1511436360; wp=1; JSESSIONID=abczwY8ecd4xz8RJcyP-v; jebecookies=d4497791-9d41-4269-9e2b-3858d4989785|||||; ick\_login=884e75d4-f361-4cff-94bb-81fe6c42b220; \_de=EA5778F44555C091303554EBBEB4676C696BF75400CE19CC; p=61a3c7d0d4b2d1e991095353f83fa2141; first\_login\_flag=1; ln\_uact=970138074@qq.com; ln\_hurl=http://hdn.xnimg.cn/photos/hdn121/20170428/1700/main\_nhiB\_aebd0000854a1986.jpg; t=3dd84a3117737e819dd2c32f1cdb91d01; societyguester=3dd84a3117737e819dd2c32f1cdb91d01; id=443362311; xnsid=169efdc0; loginfrom=syshome; ch\_id=10016; jebe\_key=9c062f5a-4335-4a91-bf7a-970f8b86a64e%7Ca022c303305d1b2ab6b5089643e4b5de%7C1511449232839%7C1; wp\_fold=0'

}

url = 'http://www.renren.com/880151247/profile'

req = request.Request(url,headers=headers)

resp = request.urlopen(req)

with open('renren.html','w') as fp:

fp.write(resp.read().decode('utf-8'))

但是每次在访问需要cookie的页面都要从浏览器中复制cookie比较麻烦。在Python处理Cookie, 一般是通过http.cookiejar模块和urllib模块的HTTPCookieProcessor处理器类一起使用。http.cookiejar模块主要作用是提供用于存储cookie的对象。而HTTPCookieProcessor处理器主要作用是处理这些cookie对象, 并构建handler对象。

#### http.cookiejar模块：

该模块主要的类有CookieJar、FileCookieJar、MozillaCookieJar、LWPCookieJar。这四个类的作用分别如下：

1. CookieJar：管理HTTP cookie值、存储HTTP请求生成的cookie、向传出的HTTP请求添加cookie的对象。整个cookie都存储在内存中, 对CookieJar实例进行垃圾回收后cookie也将丢失。
2. FileCookieJar (filename,delayload=None,policy=None)：从CookieJar派生而来, 用来创建FileCookieJar实例, 检索cookie信息并将cookie存储到文件中。filename是存储cookie的文件名。delayload为True时支持延迟访问访问文件, 即只有在需要时才读取文件或在文件中存储数据。
3. MozillaCookieJar (filename,delayload=None,policy=None)：从FileCookieJar派生而来, 创建与Mozilla浏览器 cookies.txt兼容的FileCookieJar实例。
4. LWPCookieJar (filename,delayload=None,policy=None)：从FileCookieJar派生而来, 创建与libwww-perl标准的 Set-Cookie3 文件格式兼容的FileCookieJar实例。

#### 登录人人网：

利用http.cookiejar和request.HTTPCookieProcessor登录人人网。相关示例代码如下：

from urllib import request, parse

from http.cookiejar import CookieJar

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

def get\_opener():

cookiejar = CookieJar()

handler = request.HTTPCookieProcessor(cookiejar)

opener = request.build\_opener(handler)

return opener

def login\_renren(opener):

data = {"email": "970138074@qq.com", "password": "pythonspider"}

data = parse.urlencode(data).encode('utf-8')

login\_url = "http://www.renren.com/PLogin.do"

req = request.Request(login\_url, headers=headers, data=data)

opener.open(req)

def visit\_profile(opener):

url = 'http://www.renren.com/880151247/profile'

req = request.Request(url,headers=headers)

resp = opener.open(req)

with open('renren.html','w') as fp:

fp.write(resp.read().decode("utf-8"))

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

opener = get\_opener()

login\_renren(opener)

visit\_profile(opener)

#### 保存cookie到本地：

保存cookie到本地, 可以使用cookiejar的save方法, 并且需要指定一个文件名：

from urllib import request

from http.cookiejar import MozillaCookieJar

cookiejar = MozillaCookieJar("cookie.txt")

handler = request.HTTPCookieProcessor(cookiejar)

opener = request.build\_opener(handler)

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

req = request.Request('http://httpbin.org/cookies',headers=headers)

resp = opener.open(req)

print(resp.read())

cookiejar.save(ignore\_discard=True,ignore\_expires=True)

#### 从本地加载cookie：

从本地加载cookie, 需要使用cookiejar的load方法, 并且也需要指定方法：

from urllib import request

from http.cookiejar import MozillaCookieJar

cookiejar = MozillaCookieJar("cookie.txt")

cookiejar.load(ignore\_expires=True,ignore\_discard=True)

handler = request.HTTPCookieProcessor(cookiejar)

opener = request.build\_opener(handler)

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

req = request.Request('http://httpbin.org/cookies',headers=headers)

resp = opener.open(req)

print(resp.read())

## [1.4. requests库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\chapter1\4-requestsmo-kuai.html)

虽然Python的标准库中 urllib模块已经包含了平常我们使用的大多数功能, 但是它的 API 使用起来让人感觉不太好, 而 Requests宣传是 "HTTP for Humans", 说明使用更简洁方便。

### 安装和文档地址：

利用pip可以非常方便的安装：

pip install requests

中文文档：<http://docs.python-requests.org/zh_CN/latest/index.html>

github地址：<https://github.com/requests/requests>

### 发送GET请求：

1. 最简单的发送get请求就是通过requests.get来调用：

response = requests.get("http://www.baidu.com/")

1. 添加headers和查询参数：

如果想添加 headers, 可以传入headers参数来增加请求头中的headers信息。如果要将参数放在url中传递, 可以利用 params 参数。相关示例代码如下：

import requests

kw = {'wd':'中国'}

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36"}

# params 接收一个字典或者字符串的查询参数, 字典类型自动转换为url编码, 不需要urlencode()

response = requests.get("http://www.baidu.com/s", params = kw, headers = headers)

# 查看响应内容, response.text 返回的是Unicode格式的数据

print(response.text)

# 查看响应内容, response.content返回的字节流数据

print(response.content)

# 查看完整url地址

print(response.url)

# 查看响应头部字符编码

print(response.encoding)

# 查看响应码

print(response.status\_code)

### 发送POST请求：

1. 最基本的POST请求可以使用post方法：

response = requests.post("http://www.baidu.com/",data=data)

1. 传入data数据：

这时候就不要再使用urlencode进行编码了, 直接传入一个字典进去就可以了。比如请求拉勾网的数据的代码：

import requests

url = "https://www.lagou.com/jobs/positionAjax.json?city=%E6%B7%B1%E5%9C%B3&needAddtionalResult=false&isSchoolJob=0"

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36',

'Referer': 'https://www.lagou.com/jobs/list\_python?labelWords=&fromSearch=true&suginput='

}

data = {

'first': 'true',

'pn': 1,

'kd': 'python'

}

resp = requests.post(url,headers=headers,data=data)

# 如果是json数据, 直接可以调用json方法

print(resp.json())

### 使用代理：

使用requests添加代理也非常简单, 只要在请求的方法中 (比如get或者post) 传递proxies参数就可以了。示例代码如下：

import requests

url = "http://httpbin.org/get"

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36',

}

proxy = {

'http': '171.14.209.180:27829'

}

resp = requests.get(url,headers=headers,proxies=proxy)

with open('xx.html','w',encoding='utf-8') as fp:

fp.write(resp.text)

### cookie：

如果在一个响应中包含了cookie, 那么可以利用cookies属性拿到这个返回的cookie值：

import requests

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

data = {"email":"970138074@qq.com",'password':"pythonspider"}

resp = requests.get('http://www.baidu.com/')

print(resp.cookies)

print(resp.cookies.get\_dict())

### session：

之前使用urllib库, 是可以使用opener发送多个请求, 多个请求之间是可以共享cookie的。那么如果使用requests, 也要达到共享cookie的目的, 那么可以使用requests库给我们提供的session对象。注意, 这里的session不是web开发中的那个session, 这个地方只是一个会话的对象而已。还是以登录人人网为例, 使用requests来实现。示例代码如下：

import requests

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

data = {"email":"970138074@qq.com",'password':"pythonspider"}

headers = {

'User-Agent': "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36"

}

# 登录

session = requests.session()

session.post(url,data=data,headers=headers)

# 访问大鹏个人中心

resp = session.get('http://www.renren.com/880151247/profile')

print(resp.text)

### 处理不信任的SSL证书：

对于那些已经被信任的SSL整数的网站, 比如https://www.baidu.com/, 那么使用requests直接就可以正常的返回响应。示例代码如下：

resp = requests.get('http://www.12306.cn/mormhweb/',verify=False)

print(resp.content.decode('utf-8'))

# [2. 第二章：数据提取](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-er-zhang-ff1a-shu-ju-ti-qu.html)

## [2.1. xpath语法与lxml库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\1xpathyu-fa-yu-lxml-ku.html)

### 什么是XPath？

xpath (XML Path Language) 是一门在XML和HTML文档中查找信息的语言, 可用来在XML和HTML文档中对元素和属性进行遍历。

### XPath开发工具

1. Chrome插件XPath Helper。
2. Firefox插件Try XPath。

### XPath语法

#### 选取节点：

XPath 使用路径表达式来选取 XML 文档中的节点或者节点集。这些路径表达式和我们在常规的电脑文件系统中看到的表达式非常相似。

|  |  |  |  |
| --- | --- | --- | --- |
| **表达式** | **描述** | **示例** | **结果** |
| nodename | 选取此节点的所有子节点 | bookstore | 选取bookstore下所有的子节点 |
| / | 如果是在最前面, 代表从根节点选取。否则选择某节点下的某个节点 | /bookstore | 选取根元素下所有的bookstore节点 |
| // | 从全局节点中选择节点, 随便在哪个位置 | //book | 从全局节点中找到所有的book节点 |
| @ | 选取某个节点的属性 | //book[@price] | 选择所有拥有price属性的book节点 |
| . | 当前节点 | ./a | 选取当前节点下的a标签 |

#### 谓语：

谓语用来查找某个特定的节点或者包含某个指定的值的节点, 被嵌在方括号中。

在下面的表格中, 我们列出了带有谓语的一些路径表达式, 以及表达式的结果：

|  |  |
| --- | --- |
| **路径表达式** | **描述** |
| /bookstore/book[1] | 选取bookstore下的第一个子元素 |
| /bookstore/book[last()] | 选取bookstore下的倒数第二个book元素。 |
| bookstore/book[position()<3] | 选取bookstore下前面两个子元素。 |
| //book[@price] | 选取拥有price属性的book元素 |
| //book[@price=10] | 选取所有属性price等于10的book元素 |

#### 通配符

\*表示通配符。

|  |  |  |  |
| --- | --- | --- | --- |
| **通配符** | **描述** | **示例** | **结果** |
| \* | 匹配任意节点 | /bookstore/\* | 选取bookstore下的所有子元素。 |
| @\* | 匹配节点中的任何属性 | //book[@\*] | 选取所有带有属性的book元素。 |

#### 选取多个路径：

通过在路径表达式中使用"|"运算符, 可以选取若干个路径。

示例如下：

//bookstore/book | //book/title

# 选取所有book元素以及book元素下所有的title元素

#### 运算符：

### lxml库

lxml 是 一个HTML/XML的解析器, 主要的功能是如何解析和提取 HTML/XML 数据。

lxml和正则一样, 也是用 C 实现的, 是一款高性能的 Python HTML/XML 解析器, 我们可以利用之前学习的XPath语法, 来快速的定位特定元素以及节点信息。

lxml python 官方文档：<http://lxml.de/index.html>

需要安装C语言库, 可使用 pip 安装：

pip install lxml

#### 基本使用：

我们可以利用他来解析HTML代码, 并且在解析HTML代码的时候, 如果HTML代码不规范, 他会自动的进行补全。示例代码如下：

# 使用 lxml 的 etree 库

from lxml import etree

text = '''

<div>

<ul>

<li class="item-0"><a href="link1.html">first item</a></li>

<li class="item-1"><a href="link2.html">second item</a></li>

<li class="item-inactive"><a href="link3.html">third item</a></li>

<li class="item-1"><a href="link4.html">fourth item</a></li>

<li class="item-0"><a href="link5.html">fifth item</a> # 注意, 此处缺少一个 </li> 闭合标签

</ul>

</div>

'''

#利用etree.HTML, 将字符串解析为HTML文档

html = etree.HTML(text)

# 按字符串序列化HTML文档

result = etree.tostring(html)

print(result)

输入结果如下：

<html><body>

<div>

<ul>

<li class="item-0"><a href="link1.html">first item</a></li>

<li class="item-1"><a href="link2.html">second item</a></li>

<li class="item-inactive"><a href="link3.html">third item</a></li>

<li class="item-1"><a href="link4.html">fourth item</a></li>

<li class="item-0"><a href="link5.html">fifth item</a></li>

</ul>

</div>

</body></html>

可以看到。lxml会自动修改HTML代码。例子中不仅补全了li标签, 还添加了body, html标签。

#### 从文件中读取html代码：

除了直接使用字符串进行解析, lxml还支持从文件中读取内容。我们新建一个hello.html文件：

<!-- hello.html -->

<div>

<ul>

<li class="item-0"><a href="link1.html">first item</a></li>

<li class="item-1"><a href="link2.html">second item</a></li>

<li class="item-inactive"><a href="link3.html"><span class="bold">third item</span></a></li>

<li class="item-1"><a href="link4.html">fourth item</a></li>

<li class="item-0"><a href="link5.html">fifth item</a></li>

</ul>

</div>

然后利用etree.parse()方法来读取文件。示例代码如下：

from lxml import etree

# 读取外部文件 hello.html

html = etree.parse('hello.html')

result = etree.tostring(html, pretty\_print=True)

print(result)

输入结果和之前是相同的。

#### 在lxml中使用XPath语法：

1. 获取所有li标签：

from lxml import etree

html = etree.parse('hello.html')

print type(html) # 显示etree.parse() 返回类型

result = html.xpath('//li')

print(result) # 打印<li>标签的元素集合

1. 获取所有li元素下的所有class属性的值：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li/@class')

print(result)

1. 获取li标签下href为www.baidu.com的a标签：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li/a[@href="www.baidu.com"]')

print(result)

1. 获取li标签下所有span标签：

from lxml import etree

html = etree.parse('hello.html')

#result = html.xpath('//li/span')

#注意这么写是不对的：

#因为 / 是用来获取子元素的, 而 <span> 并不是 <li> 的子元素, 所以, 要用双斜杠

result = html.xpath('//li//span')

print(result)

1. 获取li标签下的a标签里的所有class：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li/a//@class')

print(result)

1. 获取最后一个li的a的href属性对应的值：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li[last()]/a/@href')

# 谓语 [last()] 可以找到最后一个元素

print(result)

1. 获取倒数第二个li元素的内容：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li[last()-1]/a')

# text 方法可以获取元素内容

print(result[0].text)

1. 获取倒数第二个li元素的内容的第二种方式：

from lxml import etree

html = etree.parse('hello.html')

result = html.xpath('//li[last()-1]/a/text()')

print(result)

### 使用requests和xpath爬取电影天堂

示例代码如下：

import requests

from lxml import etree

BASE\_DOMAIN = 'http://www.dytt8.net'

HEADERS = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36',

'Referer': 'http://www.dytt8.net/html/gndy/dyzz/list\_23\_2.html'

}

def spider():

url = 'http://www.dytt8.net/html/gndy/dyzz/list\_23\_1.html'

resp = requests.get(url,headers=HEADERS)

# resp.content：经过编码后的字符串

# resp.text：没有经过编码, 也就是unicode字符串

# text：相当于是网页中的源代码了

text = resp.content.decode('gbk')

# tree：经过lxml解析后的一个对象, 以后使用这个对象的xpath方法, 就可以

# 提取一些想要的数据了

tree = etree.HTML(text)

# xpath/beautifulsou4

all\_a = tree.xpath("//div[@class='co\_content8']//a")

for a in all\_a:

title = a.xpath("text()")[0]

href = a.xpath("@href")[0]

if href.startswith('/'):

detail\_url = BASE\_DOMAIN + href

crawl\_detail(detail\_url)

break

def crawl\_detail(url):

resp = requests.get(url,headers=HEADERS)

text = resp.content.decode('gbk')

tree = etree.HTML(text)

create\_time = tree.xpath("//div[@class='co\_content8']/ul/text()")[0].strip()

imgs = tree.xpath("//div[@id='Zoom']//img/@src")

# 电影海报

cover = imgs[0]

# 电影截图

screenshoot = imgs[1]

# 获取span标签下所有的文本

infos = tree.xpath("//div[@id='Zoom']//text()")

for index,info in enumerate(infos):

if info.startswith("◎年　　代"):

year = info.replace("◎年　　代","").strip()

if info.startswith("◎豆瓣评分"):

douban\_rating = info.replace("◎豆瓣评分",'').strip()

print(douban\_rating)

if info.startswith("◎主　　演"):

# 从当前位置, 一直往下面遍历

actors = [info]

for x in range(index+1,len(infos)):

actor = infos[x]

if actor.startswith("◎"):

break

actors.append(actor.strip())

print(",".join(actors))

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

spider()

### chrome相关问题：

在62版本 (目前最新) 中有一个bug, 在页面302重定向的时候不能记录FormData数据。这个是这个版本的一个bug。详细见以下链接：<https://stackoverflow.com/questions/34015735/http-post-payload-not-visible-in-chrome-debugger。>

在金丝雀版本中已经解决了这个问题, 可以下载这个版本继续, 链接如下：<https://www.google.com/chrome/browser/canary.html>

### 作业：

使用requests和xpath爬取腾讯招聘网信息。要求为获取每个职位的详情信息。

## [2.2. BeautifulSoup4库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\1-beautifulsoup4jie-xi-qi.html)

和 lxml 一样, Beautiful Soup 也是一个HTML/XML的解析器, 主要的功能也是如何解析和提取 HTML/XML 数据。

lxml 只会局部遍历, 而Beautiful Soup 是基于HTML DOM (Document Object Model) 的, 会载入整个文档, 解析整个DOM树, 因此时间和内存开销都会大很多, 所以性能要低于lxml。

BeautifulSoup 用来解析 HTML 比较简单, API非常人性化, 支持CSS选择器、Python标准库中的HTML解析器, 也支持 lxml 的 XML解析器。

Beautiful Soup 3 目前已经停止开发, 推荐现在的项目使用Beautiful Soup 4。

### 安装和文档：

1. 安装：pip install bs4。
2. 中文文档：<https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html>

### 几大解析工具对比：

|  |  |  |
| --- | --- | --- |
| **解析工具** | **解析速度** | **使用难度** |
| BeautifulSoup | 最慢 | 最简单 |
| lxml | 快 | 简单 |
| 正则 | 最快 | 最难 |

### 简单使用：

from bs4 import BeautifulSoup

html = """

<html><head><title>The Dormouse's story</title></head>

<body>

<p class="title" name="dromouse"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,

<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

"""

#创建 Beautiful Soup 对象

# 使用lxml来进行解析

soup = BeautifulSoup(html,"lxml")

print(soup.prettify())

### 四个常用的对象：

Beautiful Soup将复杂HTML文档转换成一个复杂的树形结构,每个节点都是Python对象,所有对象可以归纳为4种:

1. Tag
2. NavigatableString
3. BeautifulSoup
4. Comment

#### 1. Tag：

Tag 通俗点讲就是 HTML 中的一个个标签。示例代码如下：

from bs4 import BeautifulSoup

html = """

<html><head><title>The Dormouse's story</title></head>

<body>

<p class="title" name="dromouse"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,

<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

"""

#创建 Beautiful Soup 对象

soup = BeautifulSoup(html,'lxml')

print soup.title

# <title>The Dormouse's story</title>

print soup.head

# <head><title>The Dormouse's story</title></head>

print soup.a

# <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>

print soup.p

# <p class="title" name="dromouse"><b>The Dormouse's story</b></p>

print type(soup.p)

# <class 'bs4.element.Tag'>

我们可以利用 soup 加标签名轻松地获取这些标签的内容, 这些对象的类型是bs4.element.Tag。但是注意, 它查找的是在所有内容中的第一个符合要求的标签。如果要查询所有的标签, 后面会进行介绍。

对于Tag, 它有两个重要的属性, 分别是name和attrs。示例代码如下：

print soup.name

# [document] #soup 对象本身比较特殊, 它的 name 即为 [document]

print soup.head.name

# head #对于其他内部标签, 输出的值便为标签本身的名称

print soup.p.attrs

# {'class': ['title'], 'name': 'dromouse'}

# 在这里, 我们把 p 标签的所有属性打印输出了出来, 得到的类型是一个字典。

print soup.p['class'] # soup.p.get('class')

# ['title'] #还可以利用get方法, 传入属性的名称, 二者是等价的

soup.p['class'] = "newClass"

print soup.p # 可以对这些属性和内容等等进行修改

# <p class="newClass" name="dromouse"><b>The Dormouse's story</b></p>

#### 2. NavigableString：

如果拿到标签后, 还想获取标签中的内容。那么可以通过tag.string获取标签中的文字。示例代码如下：

print soup.p.string

# The Dormouse's story

print type(soup.p.string)

# <class 'bs4.element.NavigableString'>thon

#### 3. BeautifulSoup：

BeautifulSoup 对象表示的是一个文档的全部内容.大部分时候,可以把它当作 Tag 对象,它支持 遍历文档树 和 搜索文档树 中描述的大部分的方法.

因为 BeautifulSoup 对象并不是真正的HTML或XML的tag,所以它没有name和attribute属性.但有时查看它的 .name 属性是很方便的,所以 BeautifulSoup 对象包含了一个值为 "[document]" 的特殊属性 .name

soup.name

# '[document]'

#### 4. Comment：

Tag , NavigableString , BeautifulSoup 几乎覆盖了html和xml中的所有内容,但是还有一些特殊对象.容易让人担心的内容是文档的注释部分:

markup = "<b><!--Hey, buddy. Want to buy a used parser?--></b>"

soup = BeautifulSoup(markup)

comment = soup.b.string

type(comment)

# <class 'bs4.element.Comment'>

Comment 对象是一个特殊类型的 NavigableString 对象:

comment

# 'Hey, buddy. Want to buy a used parser'

### 遍历文档树：

#### 1. contents和children：

html\_doc = """

<html><head><title>The Dormouse's story</title></head>

<p class="title"><b>The Dormouse's story</b></p>

<p class="story">Once upon a time there were three little sisters; and their names were

<a href="http://example.com/elsie" class="sister" id="link1">Elsie</a>,

<a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and

<a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;

and they lived at the bottom of a well.</p>

<p class="story">...</p>

"""

from bs4 import BeautifulSoup

soup = BeautifulSoup(html\_doc,'lxml')

head\_tag = soup.head

# 返回所有子节点的列表

print(head\_tag.contents)

# 返回所有子节点的迭代器

for child in head\_tag.children:

print(child)

#### 2. strings 和 stripped\_strings

如果tag中包含多个字符串 [2] ,可以使用 .strings 来循环获取：

for string in soup.strings:

print(repr(string))

# u"The Dormouse's story"

# u'\n\n'

# u"The Dormouse's story"

# u'\n\n'

# u'Once upon a time there were three little sisters; and their names were\n'

# u'Elsie'

# u',\n'

# u'Lacie'

# u' and\n'

# u'Tillie'

# u';\nand they lived at the bottom of a well.'

# u'\n\n'

# u'...'

# u'\n'

输出的字符串中可能包含了很多空格或空行,使用 .stripped\_strings 可以去除多余空白内容：

for string in soup.stripped\_strings:

print(repr(string))

# u"The Dormouse's story"

# u"The Dormouse's story"

# u'Once upon a time there were three little sisters; and their names were'

# u'Elsie'

# u','

# u'Lacie'

# u'and'

# u'Tillie'

# u';\nand they lived at the bottom of a well.'

# u'...'

### 搜索文档树：

#### 1. find和find\_all方法：

搜索文档树, 一般用得比较多的就是两个方法, 一个是find, 一个是find\_all。find方法是找到第一个满足条件的标签后就立即返回, 只返回一个元素。find\_all方法是把所有满足条件的标签都选到, 然后返回回去。使用这两个方法, 最常用的用法是出入name以及attr参数找出符合要求的标签。

soup.find\_all("a",attrs={"id":"link2"})

或者是直接传入属性的的名字作为关键字参数：

soup.find\_all("a",id='link2')

#### 2. select方法：

使用以上方法可以方便的找出元素。但有时候使用css选择器的方式可以更加的方便。使用css选择器的语法, 应该使用select方法。以下列出几种常用的css选择器方法：

##### (1) 通过标签名查找：

print(soup.select('a'))

##### (2) 通过类名查找：

通过类名, 则应该在类的前面加一个.。比如要查找class=sister的标签。示例代码如下：

print(soup.select('.sister'))

##### (3) 通过id查找：

通过id查找, 应该在id的名字前面加一个＃号。示例代码如下：

print(soup.select("#link1"))

##### (4) 组合查找：

组合查找即和写 class 文件时, 标签名与类名、id名进行的组合原理是一样的, 例如查找 p 标签中, id 等于 link1的内容, 二者需要用空格分开：

print(soup.select("p #link1"))

直接子标签查找, 则使用 > 分隔：

print(soup.select("head > title"))

##### (5) 通过属性查找：

查找时还可以加入属性元素, 属性需要用中括号括起来, 注意属性和标签属于同一节点, 所以中间不能加空格, 否则会无法匹配到。示例代码如下：

print(soup.select('a[href="http://example.com/elsie"]'))

##### (6) 获取内容

以上的 select 方法返回的结果都是列表形式, 可以遍历形式输出, 然后用 get\_text() 方法来获取它的内容。

soup = BeautifulSoup(html, 'lxml')

print type(soup.select('title'))

print soup.select('title')[0].get\_text()

for title in soup.select('title'):

print title.get\_text()

## [2.3. 正则表达式和re模块](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\3zheng-ze-biao-da-shi.html)

### 什么是正则表达式

通俗理解：按照一定的规则, 从某个字符串中匹配出想要的数据。这个规则就是正则表达式。

标准答案：<https://baike.baidu.com/item/正则表达式/1700215?fr=aladdin>

### 一个段子

世界是分为两种人, 一种是懂正则表达式的, 一种是不懂正则表达式的。

### 正则表达式常用匹配规则

#### 匹配某个字符串

text = 'hello'

ret = re.match('he',text)

print(ret.group())

>> he

以上便可以在hello中, 匹配出he。

#### 点 (.) 匹配除换行符之外的任意字符

text = "ab"

ret = re.match('.',text)

print(ret.group())

>> a

但是点 (.) 不能匹配不到换行符。示例代码如下：

text = "ab"

ret = re.match('.',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute 'group'

#### \d匹配任意的数字

text = "123"

ret = re.match('\d',text)

print(ret.group())

>> 1

#### \D匹配任意的非数字

text = "a"

ret = re.match('\D',text)

print(ret.group())

>> a

而如果text是等于一个数字, 那么就匹配不成功了。示例代码如下：

text = "1"

ret = re.match('\D',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute 'group'

#### \s匹配的是空白字符 (包括：\n, \t, \r和空格)

text = "\t"

ret = re.match('\s',text)

print(ret.group())

>> 空白

#### \w匹配的是a-z和A-Z以及数字和下划线

text = "\_"

ret = re.match('\w',text)

print(ret.group())

>> \_

而如果要匹配一个其他的字符, 那么就匹配不到。示例代码如下：

text = "+"

ret = re.match('\w',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute

#### \W匹配的是和\w相反的

text = "+"

ret = re.match('\W',text)

print(ret.group())

>> +

而如果你的text是一个下划线或者英文字符, 那么就匹配不到了。示例代码如下：

text = "\_"

ret = re.match('\W',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute

#### []组合的方式, 只要满足中括号中的某一项都算匹配成功

text = "0731-88888888"

ret = re.match('[\d\-]+',text)

print(ret.group())

>> 0731-88888888

之前讲到的几种匹配规则, 其实可以使用中括号的形式来进行替代：

* \d：[0-9]
* \D：[^0-9]
* \w：[0-9a-zA-Z\_]
* \W：[^0-9a-zA-Z\_]

#### 匹配多个字符

1. \*：可以匹配0或者任意多个字符。示例代码如下：

text = "0731"

ret = re.match('\d\*',text)

print(ret.group())

>> 0731

以上因为匹配的要求是\d, 那么就要求是数字, 后面跟了一个星号, 就可以匹配到0731这四个字符。

1. +：可以匹配1个或者多个字符。最少一个。示例代码如下：

text = "abc"

ret = re.match('\w+',text)

print(ret.group())

>> abc

因为匹配的是\w, 那么就要求是英文字符, 后面跟了一个加号, 意味着最少要有一个满足\w的字符才能够匹配到。如果text是一个空白字符或者是一个不满足\w的字符, 那么就会报错。示例代码如下：

text = ""

ret = re.match('\w+',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute

1. ?：匹配的字符可以出现一次或者不出现 (0或者1) 。示例代码如下：

text = "123"

ret = re.match('\d?',text)

print(ret.group())

>> 1

1. {m}：匹配m个字符。示例代码如下：

text = "123"

ret = re.match('\d{2}',text)

print(ret.group())

>> 12

1. {m,n}：匹配m-n个字符。在这中间的字符都可以匹配到。示例代码如下：

text = "123"

ret = re.match('\d{1,2}',text)

prit(ret.group())

>> 12

如果text只有一个字符, 那么也可以匹配出来。示例代码如下：

text = "1"

ret = re.match('\d{1,2}',text)

prit(ret.group())

>> 1

#### 小案例：

1. 验证手机号码：手机号码的规则是以1开头, 第二位可以是34587, 后面那9位就可以随意了。示例代码如下：

text = "18570631587"

ret = re.match('1[34587]\d{9}',text)

print(ret.group())

>> 18570631587

而如果是个不满足条件的手机号码。那么就匹配不到了。示例代码如下：

text = "1857063158"

ret = re.match('1[34587]\d{9}',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute

1. 验证邮箱：邮箱的规则是邮箱名称是用数字、数字、下划线组成的, 然后是@符号, 后面就是域名了。示例代码如下：

text = "hynever@163.com"

ret = re.match('\w+@\w+\.[a-zA-Z\.]+',text)

print(ret.group())

1. 验证URL：URL的规则是前面是http或者https或者是ftp然后再加上一个冒号, 再加上一个斜杠, 再后面就是可以出现任意非空白字符了。示例代码如下：

text = "http://www.baidu.com/"

ret = re.match('(http|https|ftp)://[^\s]+',text)

print(ret.group())

1. 验证身份证：身份证的规则是, 总共有18位, 前面17位都是数字, 后面一位可以是数字, 也可以是小写的x, 也可以是大写的X。示例代码如下：

text = "3113111890812323X"

ret = re.match('\d{17}[\dxX]',text)

print(ret.group())

#### ^ (脱字号) ：表示以...开始：

text = "hello"

ret = re.match('^h',text)

print(ret.group())

如果是在中括号中, 那么代表的是取反操作.

#### $：表示以...结束：

# 匹配163.com的邮箱

text = "xxx@163.com"

ret = re.search('\w+@163\.com$',text)

print(ret.group())

>> xxx@163.com

#### |：匹配多个表达式或者字符串：

text = "hello|world"

ret = re.search('hello',text)

print(ret.group())

>> hello

#### 贪婪模式和非贪婪模式：

贪婪模式：正则表达式会匹配尽量多的字符。默认是贪婪模式。

非贪婪模式：正则表达式会尽量少的匹配字符。

示例代码如下：

text = "0123456"

ret = re.match('\d+',text)

print(ret.group())

# 因为默认采用贪婪模式, 所以会输出0123456

>> 0123456

可以改成非贪婪模式, 那么就只会匹配到0。示例代码如下：

text = "0123456"

ret = re.match('\d+?',text)

print(ret.group())

#### 案例：匹配0-100之间的数字：

text = '99'

ret = re.match('[1-9]?\d$|100$',text)

print(ret.group())

>> 99

而如果text=101, 那么就会抛出一个异常。示例代码如下：

text = '101'

ret = re.match('[1-9]?\d$|100$',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute 'group'

#### 转义字符和原生字符串：

在正则表达式中, 有些字符是有特殊意义的字符。因此如果想要匹配这些字符, 那么就必须使用反斜杠进行转义。比如$代表的是以...结尾, 如果想要匹配$, 那么就必须使用\$。示例代码如下：

text = "apple price is \$99,orange paice is $88"

ret = re.search('\$(\d+)',text)

print(ret.group())

>> $99

原生字符串：

在正则表达式中, \是专门用来做转义的。在Python中\也是用来做转义的。因此如果想要在普通的字符串中匹配出\, 那么要给出四个\。示例代码如下：

text = "apple \c"

ret = re.search('\\\\c',text)

print(ret.group())

因此要使用原生字符串就可以解决这个问题：

text = "apple \c"

ret = re.search(r'\\c',text)

print(ret.group())

### re模块中常用函数：

#### match：

从开始的位置进行匹配。如果开始的位置没有匹配到。就直接失败了。示例代码如下：

text = 'hello'

ret = re.match('h',text)

print(ret.group())

>> h

如果第一个字母不是h, 那么就会失败。示例代码如下：

text = 'ahello'

ret = re.match('h',text)

print(ret.group())

>> AttributeError: 'NoneType' object has no attribute 'group'

如果想要匹配换行的数据, 那么就要传入一个flag=re.DOTALL, 就可以匹配换行符了。示例代码如下：

text = "abc\nabc"

ret = re.match('abc.\*abc',text,re.DOTALL)

print(ret.group())

#### search：

在字符串中找满足条件的字符。如果找到, 就返回。说白了, 就是只会找到第一个满足条件的。

text = 'apple price $99 orange price $88'

ret = re.search('\d+',text)

print(ret.group())

>> 99

#### 分组：

在正则表达式中, 可以对过滤到的字符串进行分组。分组使用圆括号的方式。

1. group：和group(0)是等价的, 返回的是整个满足条件的字符串。
2. groups：返回的是里面的子组。索引从1开始。
3. group(1)：返回的是第一个子组, 可以传入多个。

示例代码如下：

text = "apple price is $99,orange price is $10"

ret = re.search(r".\*(\$\d+).\*(\$\d+)",text)

print(ret.group())

print(ret.group(0))

print(ret.group(1))

print(ret.group(2))

print(ret.groups())

#### findall：

找出所有满足条件的, 返回的是一个列表。

text = 'apple price $99 orange price $88'

ret = re.findall('\d+',text)

print(ret)

>> ['99', '88']

#### sub：

用来替换字符串。将匹配到的字符串替换为其他字符串。

text = 'apple price $99 orange price $88'

ret = re.sub('\d+','0',text)

print(ret)

>> apple price $0 orange price $0

sub函数的案例, 获取拉勾网中的数据：

html = """

<div>

<p>基本要求：</p>

<p>1、精通HTML5、CSS3、 JavaScript等Web前端开发技术, 对html5页面适配充分了解, 熟悉不同浏览器间的差异, 熟练写出兼容各种浏览器的代码；</p>

<p>2、熟悉运用常见JS开发框架, 如JQuery、vue、angular, 能快速高效实现各种交互效果；</p>

<p>3、熟悉编写能够自动适应HTML5界面, 能让网页格式自动适应各款各大小的手机；</p>

<p>4、利用HTML5相关技术开发移动平台、PC终端的前端页面, 实现HTML5模板化；</p>

<p>5、熟悉手机端和PC端web实现的差异, 有移动平台web前端开发经验, 了解移动互联网产品和行业, 有在Android,iOS等平台下HTML5+CSS+JavaScript (或移动JS框架) 开发经验者优先考虑；6、良好的沟通能力和团队协作精神, 对移动互联网行业有浓厚兴趣, 有较强的研究能力和学习能力；</p>

<p>7、能够承担公司前端培训工作, 对公司各业务线的前端 (HTML5\CSS3) 工作进行支撑和指导。</p>

<p><br></p>

<p>岗位职责：</p>

<p>1、利用html5及相关技术开发移动平台、微信、APP等前端页面, 各类交互的实现；</p>

<p>2、持续的优化前端体验和页面响应速度, 并保证兼容性和执行效率；</p>

<p>3、根据产品需求, 分析并给出最优的页面前端结构解决方案；</p>

<p>4、协助后台及客户端开发人员完成功能开发和调试；</p>

<p>5、移动端主流浏览器的适配、移动端界面自适应研发。</p>

</div>

"""

ret = re.sub('</?[a-zA-Z0-9]+>',"",html)

print(ret)

#### split：

使用正则表达式来分割字符串。

text = "hello world ni hao"

ret = re.split('\W',text)

print(ret)

>> ["hello","world","ni","hao"]

#### compile：

对于一些经常要用到的正则表达式, 可以使用compile进行编译, 后期再使用的时候可以直接拿过来用, 执行效率会更快。而且compile还可以指定flag=re.VERBOSE, 在写正则表达式的时候可以做好注释。示例代码如下：

text = "the number is 20.50"

r = re.compile(r"""

\d+ # 小数点前面的数字

\.? # 小数点

\d\* # 小数点后面的数字

""",re.VERBOSE)

ret = re.search(r,text)

print(ret.group())

# [3. 第三章：数据存储](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-san-zhang-ff1a-shu-ju-cun-chu.html)

## [3.1. json文件处理](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\1-jsonwen-jian-chu-li.html)

### 什么是json：

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 ECMAScript (w3c制定的js规范)的一个子集, 采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。 易于人阅读和编写, 同时也易于机器解析和生成, 并有效地提升网络传输效率。更多解释请见：<https://baike.baidu.com/item/JSON/2462549?fr=aladdin>

### JSON支持数据格式：

1. 对象 (字典) 。使用花括号。
2. 数组 (列表) 。使用方括号。
3. 整形、浮点型、布尔类型还有null类型。
4. 字符串类型 (字符串必须要用双引号, 不能用单引号) 。

多个数据之间使用逗号分开。

**注意：json本质上就是一个字符串。**

### 字典和列表转JSON：

import json

books = [

{

'title': '钢铁是怎样练成的',

'price': 9.8

},

{

'title': '红楼梦',

'price': 9.9

}

]

json\_str = json.dumps(books,ensure\_ascii=False)

print(json\_str)

因为json在dump的时候, 只能存放ascii的字符, 因此会将中文进行转义, 这时候我们可以使用ensure\_ascii=False关闭这个特性。

在Python中。只有基本数据类型才能转换成JSON格式的字符串。也即：int、float、str、list、dict、tuple。

#### 将json数据直接dump到文件中：

json模块中除了dumps函数, 还有一个dump函数, 这个函数可以传入一个文件指针, 直接将字符串dump到文件中。示例代码如下：

books = [

{

'title': '钢铁是怎样练成的',

'price': 9.8

},

{

'title': '红楼梦',

'price': 9.9

}

]

with open('a.json','w') as fp:

json.dump(books,fp)

### 将一个json字符串load成Python对象：

json\_str = '[{"title": "钢铁是怎样练成的", "price": 9.8}, {"title": "红楼梦", "price": 9.9}]'

books = json.loads(json\_str,encoding='utf-8')

print(type(books))

print(books)

#### 直接从文件中读取json：

import json

with open('a.json','r',encoding='utf-8') as fp:

json\_str = json.load(fp)

print(json\_str)

## [3.2. csv文件处理](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\2-csvwen-jian-chu-li.html)

### 读取csv文件：

import csv

with open('stock.csv','r') as fp:

reader = csv.reader(fp)

titles = next(reader)

for x in reader:

print(x)

这样操作, 以后获取数据的时候, 就要通过下表来获取数据。如果想要在获取数据的时候通过标题来获取。那么可以使用DictReader。示例代码如下：

import csv

with open('stock.csv','r') as fp:

reader = csv.DictReader(fp)

for x in reader:

print(x['turnoverVol'])

### 写入数据到csv文件：

写入数据到csv文件, 需要创建一个writer对象, 主要用到两个方法。一个是writerow, 这个是写入一行。一个是writerows, 这个是写入多行。示例代码如下：

import csv

headers = ['name','age','classroom']

values = [

('zhiliao',18,'111'),

('wena',20,'222'),

('bbc',21,'111')

]

with open('test.csv','w',newline='') as fp:

writer = csv.writer(fp)

writer.writerow(headers)

writer.writerows(values)

也可以使用字典的方式把数据写入进去。这时候就需要使用DictWriter了。示例代码如下：

import csv

headers = ['name','age','classroom']

values = [

{"name":'wenn',"age":20,"classroom":'222'},

{"name":'abc',"age":30,"classroom":'333'}

]

with open('test.csv','w',newline='') as fp:

writer = csv.DictWriter(fp,headers)

writer = csv.writeheader()

writer.writerow({'name':'zhiliao',"age":18,"classroom":'111'})

writer.writerows(values)

## [3.3. excel文件处理](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\3-excelwen-jian-chu-li.html)

## [3.4. MySQL数据库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\04-mysqlcao-zuo.html)

### 安装mysql：

1. 在官网：<https://dev.mysql.com/downloads/windows/installer/5.7.html>
2. 如果提示没有.NET Framework框架。那么就在提示框中找到下载链接, 下载一个就可以了。
3. 如果提示没有Microsoft Virtual C++ x64(x86), 那么百度或者谷歌这个软件安装即可。
4. 如果没有找到。那么私聊我。
5. ubuntu apt-get install mysql-server mysql-client

### navicat：

navicat是一个操作mysql数据库非常方便的软件。使用他操作数据库, 就跟使用excel操作数据是一样的。

### 安装驱动程序：

Python要想操作MySQL。必须要有一个中间件, 或者叫做驱动程序。驱动程序有很多。比如有mysqldb、mysqlclient、pymysql等。在这里, 我们选择用pymysql。安装方式也是非常简单, 通过命令pip install pymysql即可安装。

### 数据库连接：

数据库连接之前。首先先确认以下工作完成, 这里我们以一个pymysql\_test数据库.以下将介绍连接mysql的示例代码：

import pymysql

db = pymysql.connect(

host="127.0.0.1",

user='root',

password='root',

database='pymysql\_test',

port=3306

)

cursor = db.cursor()

cursor.execute("select 1")

data = cursor.fetchone()

print(data)

db.close()

### 插入数据：

import pymysql

db = pymysql.connect(

host="127.0.0.1",

user='root',

password='root',

database='pymysql\_test',

port=3306

)

cursor = db.cursor()

sql = """

insert into user(

id,username,gender,age,password

)

values(null,'abc',1,18,'111111');

"""

cursor.execute(sql)

db.commit()

db.close()

如果在数据还不能保证的情况下, 可以使用以下方式来插入数据：

sql = """

insert into user(

id,username,gender,age,password

)

values(null,%s,%s,%s,%s);

"""

cursor.execute(sql,('spider',1,20,'222222'))

### 查找数据：

使用pymysql查询数据。可以使用fetch\*方法。

1. fetchone()：这个方法每次之获取一条数据。
2. fetchall()：这个方法接收全部的返回结果。
3. fetchmany(size)：可以获取指定条数的数据。
4. 示例代码如下：

cursor = db.cursor()

sql = """

select \* from user

"""

cursor.execute(sql)

while True:

result = cursor.fetchone()

if not result:

break

print(result)

db.close()

或者是直接使用fetchall, 一次性可以把所有满足条件的数据都取出来：

cursor = db.cursor()

sql = """

select \* from user

"""

cursor.execute(sql)

results = cursor.fetchall()

for result in results:

print(result)

db.close()

或者是使用fetchmany, 指定获取多少条数据：

cursor = db.cursor()

sql = """

select \* from user

"""

cursor.execute(sql)

results = cursor.fetchmany(1)

for result in results:

print(result)

db.close()

### 删除数据：

cursor = db.cursor()

sql = """

delete from user where id=1

"""

cursor.execute(sql)

db.commit()

db.close()

### 更新数据：

conn = pymysql.connect(host='localhost',user='root',password='root',database='pymysql\_demo',port=3306)

cursor = conn.cursor()

sql = """

update user set username='aaa' where id=1

"""

cursor.execute(sql)

conn.commit()

conn.close()

## [3.5. MongoDB数据库](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\5-mongodbshu-ju-ku.html)

MongoDB是一个基于分布式文件存储的NoSQL数据库。在处理海量数据的时候会比MySQL更有优势。爬虫如果上了一个量级, 可能就会比较推荐使用MongoDB, 当然没有上量的数据也完全可以使用MongoDB来存储数据。因此学会使用MongoDB也是爬虫开发工程师必须掌握的一个技能。

### Windows下安装MongoDB数据库：

在官网下载MongoDB数据库。是一个msi文件。官网地址如下：<https://www.mongodb.com/download-center?ct=atlasheader#community>

然后双击进行安装：

选择安装的地址。不要包含中文、空格。

在安装的时候选择安装Compass。这个软件是用图形化的界面操作MongoDB, 使用起来非常方便。

### 运行MongoDB：

1. 创建数据目录： 启动MongoDB之前, 首先要给他指定一个数据存储的路径。比如我在MongoDB的安装路径下创建一个data文件夹, 专门用来存储数据的。D:\ProgramApp\mongodb\data。
2. 把mongodb的bin目录加入到环境变量中。方便后期调用。
3. 执行命令mongod --dbpath D:\ProgramApp\mongodb\data启动。

### 连接MongoDB：

在环境变量设置好的前提下, 使用以下命令mongo就可以进入到mongo的操作终端了。

### 使用Compass软件连接MongoDB：

Compass是一个图形化的操作MongoDB的客户端。使用他来操作会更加方便。

### 将MongoDB制作成windows服务：

启动mongodb后, 如果想让mongodb一直运行, 那么这个终端便不能关闭, 而且每次运行的时候还需要指定data的路径。因此我们可以将mongodb制作成一个服务, 以后就通过一行命令就可以运行了。以下将讲解如何制作服务：

1. 创建配置文件：

在mongodb安装的路径下创建配置文件mongod.cfg (路径和名字不是必须和我这的一样) , 然后在配置文件中添加以下代码：

logpath=D:\ProgramApp\mongodb\data\log\mongod.log

dbpath=D:\ProgramApp\mongodb\data\db

logpath是日志的路径。dbpath是mongodb数据库的存储路径。

1. 安装mongodb服务：

使用以下命令即可将mongodb安装成一个服务：

mongod --config "cfg配置文件所在路径" --install

比如：

mongod --config "D:\ProgramApp\mongodb\mongod.cfg" --install

1. 启动mongodb服务：net start mongodb
2. 关闭mongodb服务：net stop mongodb
3. 移除mongodb服务："D:\ProgramApp\mongodb\bin\mongod.exe" --remove

### MongoDB概念介绍：

|  |  |  |
| --- | --- | --- |
| SQL术语/概念 | MongoDB术语/概念 | 解释/说明 |
| database | database | 数据库 |
| table | collection | 数据库表/集合 |
| row | document | 数据记录行/文档 |
| column | field | 数据字段/域 |
| index | index | 索引 |
| joins | ~~joins~~ | 表连接,MongoDB不支持 |
| primary key | primary key | 主键,MongoDB自动将\_id字段设置为主键 |

### MongoDB三元素：

三元素：数据库、集合、文档。

1. 文档 (document) ：就是关系型数据库中的一行。文档是一个对象, 由键值对构成, 是json的扩展形式：

{'name':'abc', 'gender':'1'}

1. 集合 (collection) ：就是关系型数据库中的表。可以存储多个文档, 结构可以不固定。如可以存储如下文档在一个集合中：

{"name":"abc", "gender":"1"}

{"name":"xxx", "age":18}

{"title":'yyy', 'price':20.9}

### MongoDB基本操作命令：

1. db：查看当前的数据库。
2. show dbs：查看所有的数据库。
3. use 数据库名：切换数据库。如果数据库不存在, 则创建一个。 (如果数据库不存在, 会先在内存中创建, 创建完成后需要向数据库中插入数据才算创建成功, 才会把数据库保存到硬盘上)
4. db.dropDatabase()：删除当前指向的数据库。
5. db.集合名.insert(value)：添加数据到指定的集合中。

use zhihu

db.qa.insert({'username':'aaa', 'age': '18'})

1. db.集合名.find()：从指定的集合中查找数据。 更多命令请见：<http://www.runoob.com/mongodb/mongodb-tutorial.html>

### Python操作MongoDB：

#### 安装pymongo：

要用python操作mongodb, 必须下载一个驱动程序, 这个驱动程序就是pymongo

pip install pymongo

#### 连接MongoDB：

import pymongo

# 获取连接的对象

client = pymongo.MongoClient('127.0.0.1',port=27017)

# 获取数据库

db = client.zhihu

# 获取集合 (表)

collection = db.qa

# 插入一条数据到集合中

collection.insert\_one({

"username":"abc",

"password":'hello'

})

#### 数据类型：

|  |  |
| --- | --- |
| 类型 | 说明 |
| Object ID | 文档ID |
| String | 字符串, 最常用, 必须是有效的UTF-8 |
| Boolean | 存储一个布尔值, true或false |
| Integer | 整数可以是32位或64位, 这取决于服务器 |
| Double | 存储浮点值 |
| Arrays | 数组或列表, 多个值存储到一个键 |
| Object | 用于嵌入式的文档, 即一个值为一个文档, 即一个python中的字典 |
| Null | 存储Null值 |
| Timestamp | 时间戳, 表示从1970-1-1到现在的总秒数 |
| Date | 存储当前日期或时间的UNIX时间格式 |

#### 操作MongoDB：

操作MongoDB的主要方法如下：

1. insert\_one：与insert功能相同. 插入一条文档数据到集合中。如：

collection.insert\_one({

"username":"abc",

"password":'hello'

})

1. insert\_many：加入多条文档数据到集合中。

collection.insert\_many([

{

"username":'abc',

'password':'111111'

},

{

"username":'bbb',

'password':'222222'

},

])

1. find\_one：查找一条文档对象。

result = collection.find\_one()

print(result)

# 或者是指定条件

result = collection.find\_one({"username":"abc"})

print(result)

1. update\_one：更新一条文档对象。

collection.update\_one({"username":"abc"},{"$set":{"username":"aaa"}})

1. update\_many：更新多条文档对象。

collection.update\_many({"username":"abc"},{"$set":{"username":"aaa"}})

1. delete\_one：删除一条文档对象。

collection.delete\_one({"username":"abc"})

1. delete\_many：删除多条文档对象。

collection.delete\_one({"username":"abc"})

# 4. 第四章：爬虫进阶

## [4.1. 多线程爬虫](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\1duo-xian-cheng-pa-chong.html)

有些时候, 比如下载图片, 因为下载图片是一个耗时的操作。如果采用之前那种同步的方式下载。那效率肯会特别慢。这时候我们就可以考虑使用多线程的方式来下载图片。

### 多线程介绍

多线程是为了同步完成多项任务, 通过提高资源使用效率来提高系统的效率. 线程是在同一时间需要完成多项任务的时候实现的.

最简单的比喻多线程就像火车的每一节车厢, 而进程则是火车. 车厢离开火车是无法跑动的, 同理火车也可以有多节车厢. 多线程的出现就是为了提高效率. 同时它的出现也带来了一些问题. 更多介绍请参考：<https://baike.baidu.com/item/多线程/1190404?fr=aladdin>

### threading模块介绍：

threading模块是python中专门提供用来做多线程编程的模块. threading模块中最常用的类是Thread. 以下看一个简单的多线程程序：

# 传统的程序执行方式

*# encoding: utf-8*

**import** **time**

*# 传统的方式*

**def** coding():

**for** x **in** range(3):

**print**('正在写代码**%s**' % x)

time.sleep(1)

**def** drawing():

**for** x **in** range(3):

**print**('正在画图**%s**' % x)

time.sleep(1)

**def** main():

coding()

drawing()

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

main()

# 多线程方式执行程序----函数式

# encoding: utf-8

**import** **time**

**import** **threading**

# 采用多线程的方式：

**def** coding():

**for** x **in** range(3):

# threading.current\_thread()打印出当前的进程号

**print**('正在写代码**%s**' % threading.current\_thread())

time.sleep(1)

**def** drawing():

**for** x **in** range(3):

**print**('正在画图**%s**' % threading.current\_thread())

time.sleep(1)

**def** multi\_thread():

# 定义一个新的线程, 并指向coding这个函数. 线程开始运行时就会在其中运行coding中的代码.

t1 = threading.Thread(target=coding)

t2 = threading.Thread(target=drawing)

# 开始运行两个线程, 这时两个函数就开始在不同的线程中执行了, 彼此之间不会受影响, 也不会处于堵塞状态.

t1.start()

t2.start()

# 查看当前进程的数量, 一共有3个线程, 1是执行main函数的线程, 另外2个是创建的2个子线程.

**print**(threading.enumerate())

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

multi\_thread()

#### 查看线程数：

使用threading.enumerate()函数便可以看到当前线程的数量.

#### 查看当前线程的名字：

使用threading.current\_thread()可以看到当前线程的信息.

#### 继承自threading.Thread类：

为了让线程代码更好的封装. 可以使用threading模块下的Thread类, 继承自这个类, 然后实现run方法, 线程就会自动运行run方法中的代码.

# 多线程方式执行程序----定义类的方式

*# encoding: utf-8*

**import** **threading**

**import** **time**

*# 使用threading.Thread定义两个子类*

**class** **CodingThread**(threading.Thread):

**def** run(self):

**for** x **in** range(3):

**print**('正在写代码**%s**' % threading.current\_thread())

time.sleep(1)

**class** **DrawingThread**(threading.Thread):

**def** run(self):

**for** x **in** range(3):

**print**('正在画图**%s**' % threading.current\_thread())

time.sleep(1)

**def** multi\_thread():

t1 = CodingThread()

t2 = DrawingThread()

t1.start()

t2.start()

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

multi\_thread()

#### 多线程共享全局变量的问题：

多线程都是在同一个进程中运行的. 因此在进程中的全局变量所有线程都是可共享的. 这就造成了一个问题, 因为线程执行的顺序是无序的. 有可能会造成数据错误. 比如以下代码：

import threading

tickets = 0

def get\_ticket():

global tickets

for x in range(1000000):

tickets += 1

print('tickets:%d'%tickets)

def main():

for x in range(2):

t = threading.Thread(target=get\_ticket)

t.start()

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

main()

以上结果正常来讲应该是6, 但是因为多线程运行的不确定性. 因此最后的结果可能是随机的.

# 多线程共享全局变量的问题

*# encoding: utf-8*

**import** **threading**

VALUE = 0

**def** add\_value():

**global** VALUE

*# 在循环的次数少时, 2个线程执行的结果可能不会出错, 循环的次数越多, 出错的可能性就越大.*

**for** x **in** range(1000000):

VALUE += 1

**print**('value：**%d**' % VALUE)

**def** main():

**for** x **in** range(2):

t = threading.Thread(target=add\_value)

t.start()

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

main()

#### 锁机制：

为了解决以上使用共享全局变量的问题. threading提供了一个Lock类, 这个类可以在某个线程访问某个变量的时候加锁, 其他线程此时就不能进来, 直到当前线程处理完后, 把锁释放了, 其他线程才能进来处理.

import threading

VALUE = 0

gLock = threading.Lock()

def add\_value():

global VALUE

gLock.acquire()

for x in range(1000000):

VALUE += 1

gLock.release()

print('value：%d'%VALUE)

def main():

for x in range(2):

t = threading.Thread(target=add\_value)

t.start()

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

main()

# 使用加锁的方式解决多线程共享全局变量的问题

*# encoding: utf-8*

**import** **threading**

VALUE = 0

*# 定义一个线程锁, 一个线程在获取到全局变量的修改权之后, 其它变量就不能修改全局变量. 注意线程锁只能用在修改全局变量时, 访问全局变量时不用加锁.*

*# 要定义一个全局性的锁， 不同的线程使用同一把锁才能保证一个线程加锁时另一个线程处于等待状态．*

gLock = threading.Lock()

**def** add\_value():

**global** VALUE

*# 在修改全局变量之前就加锁, 这样一个线程在修改时就是独占的, 其它的线程就不能对全局变量进行修改.*

**print**("initial VALUE in threading **%s** is **%d**"% (threading.current\_thread().name, VALUE))

**for** x **in** range(1000000):

gLock.acquire()

VALUE += 1

*# 执行完全局变量的修改后就解锁*

gLock.release()

**print**('value：**%d**' % VALUE)

**def** main():

**for** x **in** range(2):

t = threading.Thread(target=add\_value)

t.start()

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

main()

### Lock版本生产者和消费者模式：

生产者和消费者模式是多线程开发中经常见到的一种模式. 生产者的线程专门用来生产一些数据, 然后存放到一个中间的变量中. 消费者再从这个中间的变量中取出数据进行消费. 但是因为要使用中间变量, 中间变量经常是一些全局变量, 因此需要使用锁来保证数据完整性. 以下是使用threading.Lock锁实现的"生产者与消费者模式"的一个例子：

import threading

import random

import time

gMoney = 1000

gLock = threading.Lock()

# 记录生产者生产的次数, 达到10次就不再生产

gTimes = 0

class Producer(threading.Thread):

def run(self):

global gMoney

global gLock

global gTimes

while True:

money = random.randint(100, 1000)

gLock.acquire()

# 如果已经达到10次了, 就不再生产了

if gTimes >= 10:

gLock.release()

break

gMoney += money

print('%s当前存入%s元钱, 剩余%s元钱' % (threading.current\_thread(), money, gMoney))

gTimes += 1

time.sleep(0.5)

gLock.release()

class Consumer(threading.Thread):

def run(self):

global gMoney

global gLock

global gTimes

while True:

money = random.randint(100, 500)

gLock.acquire()

if gMoney > money:

gMoney -= money

print('%s当前取出%s元钱, 剩余%s元钱' % (threading.current\_thread(), money, gMoney))

time.sleep(0.5)

else:

# 如果钱不够了, 有可能是已经超过了次数, 这时候就判断一下

if gTimes >= 10:

gLock.release()

break

print("%s当前想取%s元钱, 剩余%s元钱, 不足！" % (threading.current\_thread(),money,gMoney))

gLock.release()

def main():

for x in range(5):

Consumer(name='消费者线程%d'%x).start()

for x in range(5):

Producer(name='生产者线程%d'%x).start()

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

main()

*# Lock版生产者和消费者模式*

*# encoding: utf-8*

**import** **threading**

**import** **random**

**import** **time**

gMoney = 1000

gLock = threading.Lock()

*# 总的生产次数*

gTotalTimes = 10

*# 已经进行了生产的次数, 由于会在多个线程中进行生产, 不能在线程中定义生产的次数, 要把生产的次数定义为全局变量*

gTimes = 0

**class** **Producer**(threading.Thread):

**def** run(self):

**global** gMoney

**global** gTimes

*# 使用whild True来不断的生产金钱, 否则只会执行main函数中的3次循环进行3次生产.*

**while** True:

*# 生产者随机生成100到1000之间的整数, 加到原始的gMoney上.*

money = random.randint(100, 1000)

*# 要在生产者和消费者中使用同一把锁, 才能保证一个线程锁定时另一个线程处于等待状态.*

gLock.acquire()

*# 在每次生产之前都进行判断, 如果已经进行了10次生产, 就进行解锁, 同时中止循环, 即不再进行生产. 但是消费者是一直在消费的, 所以总会出现钱不足的情况.*

**if** gTimes >= gTotalTimes:

*# 必须要在break前把锁进行释放, 如果这里不进行释放, 下面的程序不会再执行下去了, 程序会一直卡在这里等待解锁, 程序也不会结束.*

gLock.release()

**break**

gMoney += money

**print**('**%s**生产了**%d**元钱，剩余**%d**元钱' % (threading.current\_thread(), money, gMoney))

gTimes += 1

*# 生产一次后就解锁, 进行消费或下一次的生产*

gLock.release()

time.sleep(0.5)

**class** **Consumer**(threading.Thread):

**def** run(self):

**global** gMoney

**while** True:

*# 消费者也随机消费100到1000之间的数量*

money = random.randint(100, 1000)

gLock.acquire()

*# 如果总的钱的数量大于要消费的钱的数量, 就进行消费.*

**if** gMoney >= money:

gMoney -= money

**print**('**%s**消费者消费了**%d**元钱，剩余**%d**元钱' % (threading.current\_thread(), money, gMoney))

*# 如果总的钱的数量小于要消费的钱的数量, 并且经过了10次产生, 即生产已经结束, 意味着以后总的钱的数量都不会增加, 此时就解锁并退出循环.*

**else**:

**if** gTimes >= gTotalTimes:

gLock.release()

**break**

**print**('**%s**消费者准备消费**%d**元钱，剩余**%d**元钱，无法进行消费！' % (threading.current\_thread(), money, gMoney))

*# 消费之后就解锁, 进行下一次的消费或生产*

gLock.release()

time.sleep(0.5)

**def** main():

*# 因为起始阶段有1000块钱, 可以把消费者线程放在前面.*

*# 定义3个消费者线程*

**for** x **in** range(3):

*# 实例化线程时指定线程的名字*

t = Consumer(name='消费者线程**%d**' % x)

t.start()

*# 定义5个生产者线程*

**for** x **in** range(5):

t = Producer(name="生产者线程**%d**" % x)

t.start()

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

main()

### Condition版的生产者与消费者模式：

Lock版本的生产者与消费者模式可以正常的运行. 但是存在一个不足, 在消费者中, 总是通过while True死循环并且上锁的方式去判断钱够不够. 上锁是一个很耗费CPU资源的行为. 因此这种方式不是最好的. 还有一种更好的方式便是使用threading.Condition来实现. threading.Condition可以在没有数据的时候处于阻塞等待状态. 一旦有合适的数据了, 还可以使用notify相关的函数来通知其他处于等待状态的线程. 这样就可以不用做一些无用的上锁和解锁的操作. 可以提高程序的性能. 首先对threading.Condition相关的函数做个介绍, threading.Condition类似threading.Lock, 可以在修改全局数据的时候进行上锁, 也可以在修改完毕后进行解锁. 以下将一些常用的函数做个简单的介绍：

1. acquire：上锁.
2. release：解锁.
3. wait：将当前线程处于等待状态, 并且会释放锁. 可以被其他线程使用notify和notify\_all函数唤醒. 被唤醒后会继续等待上锁, 上锁后继续执行下面的代码.

如在消费时如果发现钱不足了, 就使用wait()处于等待状态, 同时把锁释放掉. 这样其它处于等待的线程就可以上锁并执行代码, 这样就不用使用while True死循环来一直进行检测了, 这样就能减少cpu的使用, 提高程序的性能.

在某个线程中中调用wait, 就相当于把当前的线程修改为睡眠状态, 阻塞到了这里, 并且不会去排队抢Condition的锁, 处于wait状态的线程可以被其它线程使用notify和notify\_all唤醒, 然后进入排队状态去抢Condition的锁, 只要抢到了锁并加锁, 就可以再向下执行本线程中的代码.

1. notify：通知某个正在等待的线程, 默认是第1个等待的线程.
2. notify\_all：通知所有正在等待的线程. notify和notify\_all不会释放锁. 并且需要在release之前调用.

如在生产者进行了一次生产之后, 就可以使用notify或notify\_all来通知消费线程来进行消费. 但只是进行通知, 通知其它处于等待的进程, 可以准备好去排除抢锁的操作了, 并不会释放当前进程的锁, 还需要使用release手动进行锁的释放. 所以一般把nofity和nofity\_all放在release的前面, 否则就会出现异常.

Condition版的生产者与消费者模式代码如下：

import threading

import random

import time

gMoney = 1000

gCondition = threading.Condition()

gTimes = 0

gTotalTimes = 5

class Producer(threading.Thread):

def run(self):

global gMoney

global gCondition

global gTimes

while True:

money = random.randint(100, 1000)

gCondition.acquire()

if gTimes >= gTotalTimes:

gCondition.release()

print('当前生产者总共生产了%s次'%gTimes)

break

gMoney += money

print('%s当前存入%s元钱, 剩余%s元钱' % (threading.current\_thread(), money, gMoney))

gTimes += 1

time.sleep(0.5)

gCondition.notify\_all()

gCondition.release()

class Consumer(threading.Thread):

def run(self):

global gMoney

global gCondition

while True:

money = random.randint(100, 500)

gCondition.acquire()

# 这里要给个while循环判断, 因为等轮到这个线程的时候

# 条件有可能又不满足了

while gMoney < money:

if gTimes >= gTotalTimes:

gCondition.release()

return

print('%s准备取%s元钱, 剩余%s元钱, 不足！'%(threading.current\_thread(),money,gMoney))

gCondition.wait()

gMoney -= money

print('%s当前取出%s元钱, 剩余%s元钱' % (threading.current\_thread(), money, gMoney))

time.sleep(0.5)

gCondition.release()

def main():

for x in range(5):

Consumer(name='消费者线程%d'%x).start()

for x in range(2):

Producer(name='生产者线程%d'%x).start()

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

main()

*# encoding: utf-8*

**import** **threading**

**import** **random**

**import** **time**

gMoney = 1000

*# threading.Condition继承自threading.Lock*

gCondition = threading.Condition()

gTotalTimes = 10

gTimes = 0

**class** **Producer**(threading.Thread):

**def** run(self):

**global** gMoney

**global** gTimes

**while** True:

money = random.randint(100, 1000)

gCondition.acquire()

**if** gTimes >= gTotalTimes:

gCondition.release()

**break**

gMoney += money

**print**('**%s**生产了**%d**元钱，剩余**%d**元钱' % (threading.current\_thread(), money, gMoney))

gTimes += 1

*# 生产过1次之后就通知其它所有处于休眠状态的线程, 然后把锁释放.*

gCondition.notify\_all()

gCondition.release()

time.sleep(0.5)

**class** **Consumer**(threading.Thread):

**def** run(self):

**global** gMoney

**while** True:

money = random.randint(100, 1000)

gCondition.acquire()

*# 使用条件循环, 每次在消费之前都进行判断, 如果所有的钱小于要消费的钱时就使用gCondition.wait把当前的进程置于休眠的状态. 直到有其它的进程通知解除休眠状态, 进入到等待获取锁的状态. 一旦获取了锁并加锁之后, 还会进行一次判断, 如果总的钱数还是小于要消费的钱数, 还会进行到这个循环中, 处于休眠状态并等待被唤醒. 直到某一次被唤醒并加锁之后判断总的钱数大于要消费的钱数, 才会跳出这个循环, 继续执行下面进行消费的代码. 如果总的钱数总是小于要消费的钱数, 就会一直处于这个循环中.*

**while** gMoney < money:

*# 如果钱已经不足, 并且已经生产的次数大于10次, 就解锁并结束本函数的执行. 注意这里不能使用break, break只是结束了内层的循环, 外层的循环还会继续执行, 还会进行消费.*

**if** gTimes >= gTotalTimes:

gCondition.release()

**return**

**print**('**%s**准备消费**%d**元钱，剩余**%d**元钱，无法进行消费！' % (threading.current\_thread(), money, gMoney))

*# 处于休眠状态时, 一旦被其它线程通知之后, 就会解除休眠, 进入等待获取锁的状态. 这样在钱不足时就不会一直执行while死循环进行一次次的判断, 一次次的加锁解锁, 而是等待到其它线程通知它时才解除等待状态, 排除进行加锁解锁的操作.*

gCondition.wait()

gMoney -= money

**print**('**%s**消费了**%d**元钱，剩余**%d**元钱' % (threading.current\_thread(), money, gMoney))

gCondition.release()

time.sleep(0.5)

**def** main():

**for** x **in** range(3):

t = Consumer(name='消费者线程**%d**' % x)

t.start()

**for** x **in** range(5):

t = Producer(name="生产者线程**%d**" % x)

t.start()

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

main()

### Queue线程安全队列

Python3.5 queue模块详解

<http://www.cnblogs.com/CongZhang/p/5274486.html>

在python中, 多个线程之间的数据是共享的, 多个线程进行数据交换的时候, 不能够保证数据的安全性和一致性, 所以当多个线程需要进行数据交换的时候, 队列就出现了, 队列可以完美解决线程间的数据交换, 保证线程间数据的安全性和一致性

对于资源, 加锁是个重要的环节. 因为python原生的list,dict等, 都是not thread safe的. 而Queue, 是线程安全的, 因此在满足使用条件下, 建议使用队列

在线程中, 访问一些全局变量, 加锁是一个经常的过程. 如果你是想把一些数据存储到某个队列中, 那么Python内置了一个线程安全的模块叫做queue模块. Python中的queue模块中提供了同步的, 线程安全的队列类, 包括FIFO (先进先出) 队列Queue, LIFO (后入先出) 队列LifoQueue. 这些队列都实现了锁原语 (可以理解为原子操作, 即要么不做, 要么都做完) , 能够在多线程中直接使用. 可以使用队列来实现线程间的同步.

queue模块有三种队列及构造函数:

1. Python queue模块的FIFO队列先进先出. class queue.Queue(maxsize)

2. LIFO类似于堆, 即先进后出. class queue.LifoQueue(maxsize)

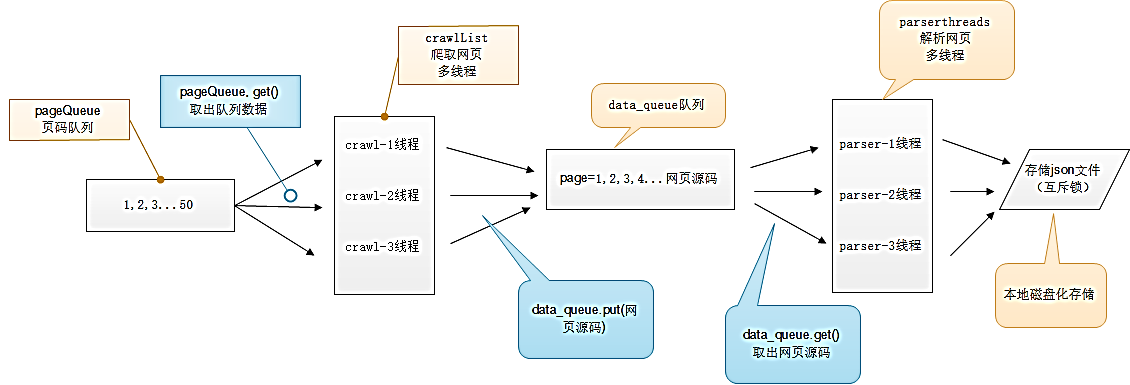
3. 还有一种是优先级队列级别越低越先出来. class queue.PriorityQueue(maxsize)

队列与列表的操作类似, 可以向其中添加元素, 也可以从中删除元素. 但列表不是一个线程安全的队列. 可以对列表中任意位置的元素进行插入和删除操作, 但是Queue队列中的元素只能按照一定的顺序执行, 先进先出和先进后出. 队列FIFO, 先进入的元素先取出, 如排队买票, 栈LIFO, 后进入的元素先取出, 弹夹中的子弹.

### queue模块中的常用方法:

* queue.qsize() 返回队列的大小
* queue.empty() 如果队列为空, 返回True,反之False
* queue.full() 如果队列满了, 返回True,反之False
* queue.full 与 maxsize 大小对应
* queue.get([block[, timeout]])获取队列中最后一个元素, timeout等待时间
* queue.get\_nowait() 相当queue.get(False)
* queue.put(item) 把一个元素写入队列, timeout等待时间
* queue.put\_nowait(item) 相当queue.put(item, False)
* queue.task\_done() 在完成一项工作之后, queue.task\_done()函数向任务已经完成的队列发送一个信号
* queue.join() 实际上意味着等到队列为空, 再执行别的操作

### 多线程示意图



Queue队列的基本使用

*# encoding: utf-8*

**from** **queue** **import** Queue

**import** **time**

**import** **threading**

*# 创建一个大小为4的队列*

q = Queue(4)

*# put向队列中添加元素, 元素可以是任意类型的python对象, 放入什么, 取出来时就取出什么.*

q.put(1)

*# q.qsize()查看队列的大小. 或其中元素的个数*

**print**(q.qsize())

*# 1*

q.put(1)

**print**(q.qsize())

*# 2*

*# 查看队列是否为空, 为空时返回True*

**print**(q.empty())

*# False*

q.put(2)

*# 查看队列是否已满*

**print**(q.full())

*# True*

**print**(q.qsize())

*# 4*

*# 队列中元素的放入和取出*

**from** **queue** **import** Queue

q1 = Queue(4)

**for** x **in** range(4):

*# put向队列中添加元素*

q1.put(x)

**print**(q1.full())

*# True*

**for** x **in** range(4):

*# 从队列中取出来最先放进去的元素. 放入的时候按0,1,2,3的顺序放入的, 取的时候是按0,1,2,3的顺序取出的. 注意此时并不是打印的x, 而是q.get()*

**print**(q.get())

*# put和get操作的block参数.*

*# 其中的block参数指当前的操作是否是阻塞式的. 默认是True, 即是阻塞式的. 当使用get从队列中取出元素时, 如果队列为空, 会一直阻塞在get操作处, 直到队列中有元素存在并取出元素后才解除阻塞状态.*

q.get(block = True)

*# 当使用put向队列中放入元素时, 如果队列已满, 会一直阻塞在put操作处, 直到队列变得不满并把元素放入队列后才解除阻塞状态*

q.put(block = True)

# block在程序中的用法

**from** **queue** **import** Queue

**import** **time**

**import** **threading**

**def** set\_value(q):

index = 0

*# 在本函数中, 每隔3s会向其中放入一个值, 而在get\_value中, 就不用等待一直从中取出来值.*

**while** True:

q.put(index)

print("put value %d" %index)

index += 1

time.sleep(3)

**def** get\_value(q):

**while** True:

*# q.get()的操作默认是阻塞状态的, 即如果q这个队列中没有值, 会一直阻塞在这里等待着队列中有值. 如果有值, 会立即把它取出来.*

**print**("get value %d" %q.get())

**def** main():

*# 创建一个4个元素容量的队列*

q = Queue(4)

*# 创建两个线程, 指定目标函数并把创建的列队作为参数传递进去.*

*# threading.Thread的参数使用args来传递, 其格式为元组或列表.*

t1 = threading.Thread(target=set\_value, args=[q])

t2 = threading.Thread(target=get\_value, args=[q])

t1.start()

t2.start()

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

main()

### 使用生产者与消费者模式多线程下载表情包：

import threading

import requests

from lxml import etree

from urllib import request

import os

import re

from queue import Queue

class Producer(threading.Thread):

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

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

super(Producer, self).\_\_init\_\_(\*args,\*\*kwargs)

self.page\_queue = page\_queue

self.img\_queue = img\_queue

def run(self):

while True:

if self.page\_queue.empty():

break

url = self.page\_queue.get()

self.parse\_page(url)

def parse\_page(self,url):

response = requests.get(url,headers=self.headers)

text = response.text

html = etree.HTML(text)

imgs = html.xpath("//div[@class='page-content text-center']//a//img")

for img in imgs:

if img.get('class') == 'gif':

continue

img\_url = img.xpath(".//@data-original")[0]

suffix = os.path.splitext(img\_url)[1]

alt = img.xpath(".//@alt")[0]

alt = re.sub(r'[, 。？?,/\\·]','',alt)

img\_name = alt + suffix

self.img\_queue.put((img\_url,img\_name))

class Consumer(threading.Thread):

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

super(Consumer, self).\_\_init\_\_(\*args,\*\*kwargs)

self.page\_queue = page\_queue

self.img\_queue = img\_queue

def run(self):

while True:

if self.img\_queue.empty():

if self.page\_queue.empty():

return

img = self.img\_queue.get(block=True)

url,filename = img

request.urlretrieve(url,'images/'+filename)

print(filename+' 下载完成！')

def main():

page\_queue = Queue(100)

img\_queue = Queue(500)

for x in range(1,101):

url = "http://www.doutula.com/photo/list/?page=%d" % x

page\_queue.put(url)

for x in range(5):

t = Producer(page\_queue,img\_queue)

t.start()

for x in range(5):

t = Consumer(page\_queue,img\_queue)

t.start()

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

main()

### GIL全局解释器锁：

Python自带的解释器是CPython。CPython解释器的多线程实际上是一个假的多线程 (在多核CPU中, 只能利用一核, 不能利用多核) 。同一时刻只有一个线程在执行, 为了保证同一时刻只有一个线程在执行, 在CPython解释器中有一个东西叫做GIL (Global Intepreter Lock) , 叫做全局解释器锁。这个解释器锁是有必要的。因为CPython解释器的内存管理不是线程安全的。当然除了CPython解释器, 还有其他的解释器, 有些解释器是没有GIL锁的, 见下面：

1. Jython：用Java实现的Python解释器。不存在GIL锁。更多详情请见：<https://zh.wikipedia.org/wiki/Jython>
2. IronPython：用.net实现的Python解释器。不存在GIL锁。更多详情请见：<https://zh.wikipedia.org/wiki/IronPython>
3. PyPy：用Python实现的Python解释器。存在GIL锁。更多详情请见：<https://zh.wikipedia.org/wiki/PyPy>
4. GIL虽然是一个假的多线程。但是在处理一些IO操作 (比如文件读写和网络请求) 还是可以在很大程度上提高效率的。在IO操作上建议使用多线程提高效率。在一些CPU计算操作上不建议使用多线程, 如渲染, 复杂的大型计算时, 建议使用多进程。

### 多线程下载百思不得姐段子作业：

import requests

from lxml import etree

import threading

from queue import Queue

import csv

class BSSpider(threading.Thread):

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

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

super(BSSpider, self).\_\_init\_\_(\*args,\*\*kwargs)

self.base\_domain = 'http://www.budejie.com'

self.page\_queue = page\_queue

self.joke\_queue = joke\_queue

def run(self):

while True:

if self.page\_queue.empty():

break

url = self.page\_queue.get()

response = requests.get(url, headers=self.headers)

text = response.text

html = etree.HTML(text)

descs = html.xpath("//div[@class='j-r-list-c-desc']")

for desc in descs:

jokes = desc.xpath(".//text()")

joke = "\n".join(jokes).strip()

link = self.base\_domain+desc.xpath(".//a/@href")[0]

self.joke\_queue.put((joke,link))

print('='\*30+"第%s页下载完成！"%url.split('/')[-1]+"="\*30)

class BSWriter(threading.Thread):

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36'

}

def \_\_init\_\_(self, joke\_queue, writer,gLock, \*args, \*\*kwargs):

super(BSWriter, self).\_\_init\_\_(\*args, \*\*kwargs)

self.joke\_queue = joke\_queue

self.writer = writer

self.lock = gLock

def run(self):

while True:

try:

joke\_info = self.joke\_queue.get(timeout=40)

joke,link = joke\_info

self.lock.acquire()

self.writer.writerow((joke,link))

self.lock.release()

print('保存一条')

except:

break

def main():

page\_queue = Queue(10)

joke\_queue = Queue(500)

gLock = threading.Lock()

fp = open('bsbdj.csv', 'a',newline='', encoding='utf-8')

writer = csv.writer(fp)

writer.writerow(('content', 'link'))

for x in range(1,11):

url = 'http://www.budejie.com/text/%d' % x

page\_queue.put(url)

for x in range(5):

t = BSSpider(page\_queue,joke\_queue)

t.start()

for x in range(5):

t = BSWriter(joke\_queue,writer,gLock)

t.start()

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

main()

## [4.2. 动态网页爬虫](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\2-ajaxdong-tai-shu-ju-zhua-qu.html)

### 什么是AJAX：

AJAX (Asynchronouse JavaScript And XML) 异步JavaScript和XML。过在后台与服务器进行少量数据交换, Ajax 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下, 对网页的某部分进行更新。传统的网页 (不使用Ajax) 如果需要更新内容, 必须重载整个网页页面。因为传统的在传输数据格式方面, 使用的是XML语法。因此叫做AJAX, 其实现在数据交互基本上都是使用JSON。使用AJAX加载的数据, 即使使用了JS, 将数据渲染到了浏览器中, 在右键->查看网页源代码还是不能看到通过ajax加载的数据, 只能看到使用这个url加载的html代码。想要知道网站是否使用了ajax技术, 还是要Network中查看网络请求.

www.xfz.cn

### 获取ajax数据的方式：

1. 直接分析ajax调用的接口。然后通过代码请求这个接口。
2. 使用Selenium+chromedriver模拟浏览器行为获取数据。

|  |  |  |
| --- | --- | --- |
| **方式** | **优点** | **缺点** |
| 分析接口 | 直接可以请求到数据。不需要做一些解析工作。代码量少, 性能高。 | 分析接口比较复杂, 特别是一些通过js混淆的接口, 要有一定的js功底。容易被发现是爬虫。生产和传递的参数. |
| selenium | 直接模拟浏览器的行为。浏览器能请求到的, 使用selenium也能请求到。爬虫更稳定。 | 代码量多。性能低。 |

### Selenium+chromedriver获取动态数据：

Selenium相当于是一个机器人。可以模拟人类在浏览器上的一些行为, 自动处理浏览器上的一些行为, 比如点击, 填充数据, 删除cookie等。chromedriver是一个驱动Chrome浏览器的驱动程序, 使用他才可以驱动浏览器。当然针对不同的浏览器有不同的driver。以下列出了不同浏览器及其对应的driver：

1. Chrome：<https://sites.google.com/a/chromium.org/chromedriver/downloads>
2. Firefox：<https://github.com/mozilla/geckodriver/releases>
3. Edge：<https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/>
4. Safari：<https://webkit.org/blog/6900/webdriver-support-in-safari-10/>

### 安装Selenium和chromedriver：

1. 安装Selenium：Selenium有很多语言的版本, 有java、ruby、python等。我们下载python版本的就可以了。

pip install selenium

1. 安装chromedriver：下载完成后, 放到不需要权限的纯英文目录下就可以了。

#### 快速入门：

现在以一个简单的获取百度首页的例子来讲下Selenium和chromedriver如何快速入门：

from selenium import webdriver

# chromedriver的绝对路径

driver\_path = r'D:\ProgramApp\chromedriver\chromedriver.exe'

# 初始化一个driver, 并且指定chromedriver的路径

driver = webdriver.Chrome(executable\_path=driver\_path)

# 请求网页

driver.get("https://www.baidu.com/")

# 通过page\_source获取网页源代码

print(driver.page\_source)

#### selenium常用操作：

更多教程请参考：<http://selenium-python.readthedocs.io/installation.html#introduction>

##### 关闭页面：

1. driver.close()：关闭当前页面。
2. driver.quit()：退出整个浏览器。

##### 定位元素：

1. find\_element\_by\_id：根据id来查找某个元素。等价于：

submitTag = driver.find\_element\_by\_id('su')

submitTag1 = driver.find\_element(By.ID,'su')

1. find\_element\_by\_class\_name：根据类名查找元素。 等价于：

submitTag = driver.find\_element\_by\_class\_name('su')

submitTag1 = driver.find\_element(By.CLASS\_NAME,'su')

1. find\_element\_by\_name：根据name属性的值来查找元素。等价于：

submitTag = driver.find\_element\_by\_name('email')

submitTag1 = driver.find\_element(By.NAME,'email')

1. find\_element\_by\_tag\_name：根据标签名来查找元素。等价于：

submitTag = driver.find\_element\_by\_tag\_name('div')

submitTag1 = driver.find\_element(By.TAG\_NAME,'div')

1. find\_element\_by\_xpath：根据xpath语法来获取元素。等价于：

submitTag = driver.find\_element\_by\_xpath('//div')

submitTag1 = driver.find\_element(By.XPATH,'//div')

1. find\_element\_by\_css\_selector：根据css选择器选择元素。等价于：

submitTag = driver.find\_element\_by\_css\_selector('//div')

submitTag1 = driver.find\_element(By.CSS\_SELECTOR,'//div')

要注意, find\_element是获取第一个满足条件的元素。find\_elements是获取所有满足条件的元素。

##### 操作表单元素：

1. 操作输入框：分为两步。第一步：找到这个元素。第二步：使用send\_keys(value), 将数据填充进去。示例代码如下：

inputTag = driver.find\_element\_by\_id('kw')

inputTag.send\_keys('python')

使用clear方法可以清除输入框中的内容。示例代码如下：

inputTag.clear()

1. 操作checkbox：因为要选中checkbox标签, 在网页中是通过鼠标点击的。因此想要选中checkbox标签, 那么先选中这个标签, 然后执行click事件。示例代码如下：

rememberTag = driver.find\_element\_by\_name("rememberMe")

rememberTag.click()

1. 选择select：select元素不能直接点击。因为点击后还需要选中元素。这时候selenium就专门为select标签提供了一个类selenium.webdriver.support.ui.Select。将获取到的元素当成参数传到这个类中, 创建这个对象。以后就可以使用这个对象进行选择了。示例代码如下：

from selenium.webdriver.support.ui import Select

# 选中这个标签, 然后使用Select创建对象

selectTag = Select(driver.find\_element\_by\_name("jumpMenu"))

# 根据索引选择, 索引是从0开始的.

selectTag.select\_by\_index(1)

# 根据值选择

selectTag.select\_by\_value("http://www.95yueba.com")

# 根据可视的文本选择

selectTag.select\_by\_visible\_text("95秀客户端")

# 取消选中所有选项

selectTag.deselect\_all()

1. 操作按钮：操作按钮有很多种方式。比如单击、右击、双击等。这里讲一个最常用的。就是点击。直接调用click函数就可以了。示例代码如下：

inputTag = driver.find\_element\_by\_id('su')

inputTag.click()

##### 行为链：

有时候在页面中的操作可能要有很多步, 那么这时候可以使用鼠标行为链类ActionChains来完成。比如现在要将鼠标移动到某个元素上并执行点击事件。在爬虫中行为链使用的并不多, 在测试中使用的较多. 示例代码如下：

inputTag = driver.find\_element\_by\_id('kw')

submitTag = driver.find\_element\_by\_id('su')

actions = ActionChains(driver)

actions.move\_to\_element(inputTag)

actions.send\_keys\_to\_element(inputTag,'python')

actions.move\_to\_element(submitTag)

actions.click(submitTag)

actions.perform()

还有更多的鼠标相关的操作。

* click\_and\_hold(element)：点击但不松开鼠标。
* context\_click(element)：右键点击。
* double\_click(element)：双击。

更多方法请参考：<http://selenium-python.readthedocs.io/api.html>

##### Cookie操作：

1. 获取所有的cookie：

for cookie in driver.get\_cookies():

print(cookie)

1. 根据cookie的key获取value：

value = driver.get\_cookie(key)

1. 删除所有的cookie：

driver.delete\_all\_cookies()

1. 删除某个cookie：

driver.delete\_cookie(key)

##### 页面等待：

现在的网页越来越多采用了 Ajax 技术, 这样程序便不能确定何时某个元素完全加载出来了。如果实际页面等待时间过长导致某个dom元素还没出来, 但是你的代码直接使用了这个WebElement, 那么就会抛出NullPointer的异常。为了解决这个问题。所以 Selenium 提供了两种等待方式：一种是隐式等待、一种是显式等待。

1. 隐式等待：在获取元素之前等待一个确定的时间. 调用driver.implicitly\_wait。那么在获取~~不可用的~~元素之前, 会先等待设置的时间。示例代码如下：

driver = webdriver.Chrome(executable\_path=driver\_path)

driver.implicitly\_wait(10)

# 请求网页

driver.get("https://www.douban.com/")

1. 显示等待：显示等待是表明某个条件成立后才执行获取元素的操作。也可以在等待的时候指定一个最大的时间, 如果在等待期间要操作的元素被加载出来, 就不再进行等待, 如果超过这个时间要进行操作的元素还没有加载出来 , 就抛出一个异常。显示等待应该使用selenium.webdriver.support.excepted\_conditions期望的条件和selenium.webdriver.support.ui.WebDriverWait来配合完成。示例代码如下：

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

driver = webdriver.Firefox()

driver.get("http://somedomain/url\_that\_delays\_loading")

try:

element = WebDriverWait(driver, 10).until(

EC.presence\_of\_element\_located((By.ID, "myDynamicElement"))

)

finally:

driver.quit()

1. 一些其他的等待条件：
   * presence\_of\_element\_located：某个元素已经加载完毕了。
   * presence\_of\_all\_emement\_located：网页中所有满足条件的元素都加载完毕了。
   * element\_to\_be\_cliable：某个元素是可以点击了。

更多条件请参考：<http://selenium-python.readthedocs.io/waits.html>

##### 切换页面：

有时候窗口中有很多子tab页面。这时候肯定是需要进行切换的。selenium提供了一个叫做switch\_to\_window来进行切换, 具体切换到哪个页面, 可以从driver.window\_handles中找到。示例代码如下：

# 打开一个新的页面

self.driver.execute\_script("window.open('"+url+"')")

# 切换到这个新的页面中

self.driver.switch\_to\_window(self.driver.window\_handles[1])

##### 设置代理ip：

有时候频繁爬取一些网页。服务器发现你是爬虫后会封掉你的ip地址。这时候我们可以更改代理ip。更改代理ip, 不同的浏览器有不同的实现方式。这里以Chrome浏览器为例来讲解：

from selenium import webdriver

options = webdriver.ChromeOptions()

options.add\_argument("--proxy-server=http://110.73.2.248:8123")

driver\_path = r"D:\ProgramApp\chromedriver\chromedriver.exe"

driver = webdriver.Chrome(executable\_path=driver\_path,chrome\_options=options)

driver.get('http://httpbin.org/ip')

##### WebElement元素：

from selenium.webdriver.remote.webelement import WebElement类是每个获取出来的元素的所属类。

有一些常用的属性：

1. get\_attribute：这个标签的某个属性的值。
2. screentshot：获取当前页面的截图。这个方法只能在driver上使用。
3. driver的对象类, 也是继承自WebElement。
4. 更多请阅读相关源代码。

## [4.3. 图形验证码识别](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\3tu-xing-yan-zheng-ma-shi-bie.html)

阻碍我们爬虫的。有时候正是在登录或者请求一些数据时候的图形验证码。因此这里我们讲解一种能将图片翻译成文字的技术。将图片翻译成文字一般被成为光学文字识别 (Optical Character Recognition) , 简写为OCR。实现OCR的库不是很多, 特别是开源的。因为这块存在一定的技术壁垒 (需要大量的数据、算法、机器学习、深度学习知识等) , 并且如果做好了具有很高的商业价值。因此开源的比较少。这里介绍一个比较优秀的图像识别开源库：Tesseract。

### Tesseract：

Tesseract是一个OCR库, 目前由谷歌赞助。Tesseract是目前公认最优秀、最准确的开源OCR库。Tesseract具有很高的识别度, 也具有很高的灵活性, 他可以通过训练识别任何字体。

#### 安装：

##### Windows系统：

在以下链接下载可执行文件, 然后一顿点击下一步安装即可 (放在不需要权限的纯英文路径下) ：

<https://github.com/tesseract-ocr/>

##### Linux系统：

可以在以下链接下载源码自行编译。

<https://github.com/tesseract-ocr/tesseract/wiki/Compiling>

或者在ubuntu下通过以下命令进行安装：

sudo apt install tesseract-ocr

##### Mac系统：

用Homebrew即可方便安装：

brew install tesseract

##### 设置环境变量：

安装完成后, 如果想要在命令行中使用Tesseract, 那么应该设置环境变量。Mac和Linux在安装的时候就默认已经设置好了。在Windows下把tesseract.exe所在的路径添加到PATH环境变量中。

还有一个环境变量需要设置的是, 要把训练的数据文件路径也放到环境变量中。

在环境变量中, 添加一个TESSDATA\_PREFIX=C:\path\_to\_tesseractdata\teseractdata。

#### 在命令行中使用tesseract识别图像：

如果想要在cmd下能够使用tesseract命令, 那么需要把tesseract.exe所在的目录放到PATH环境变量中。然后使用命令：tesseract 图片路径 文件路径。

示例：

tesseract a.png a

那么就会识别出a.png中的图片, 并且把文字写入到a.txt中。如果不想写入文件直接想显示在终端, 那么不要加文件名就可以了。

# 使用tesseract识别中文

通过tesseract --list-langs查看所有已下载的语言包

使用-l chi\_sim来识别中文

tesseract c.png c -l chi\_sim

#### 在代码中使用tesseract识别图像：

在Python代码中操作tesseract。需要安装一个库, 叫做pytesseract。通过pip的方式即可安装：

pip install pytesseract

并且, 需要读取图片, 需要借助一个第三方库叫做PIL。通过pip list看下是否安装。如果没有安装, 通过pip的方式安装：

pip install PIL

使用pytesseract将图片上的文字转换为文本文字的示例代码如下：

# 导入pytesseract库

import pytesseract

# 导入Image库

from PIL import Image

# 指定tesseract.exe所在的路径

pytesseract.pytesseract.tesseract\_cmd = r'D:\ProgramApp\TesseractOCR\tesseract.exe'

# 打开图片

image = Image.open("a.png")

# 调用image\_to\_string将图片转换为文字

text = pytesseract.image\_to\_string(image)

print(text)

### 用pytesseract处理拉勾网图形验证码：

import pytesseract

from urllib import request

from PIL import Image

import time

pytesseract.pytesseract.tesseract\_cmd = r"D:\ProgramApp\TesseractOCR\tesseract.exe"

while True:

captchaUrl = "https://passport.lagou.com/vcode/create?from=register&refresh=1513081451891"

request.urlretrieve(captchaUrl,'captcha.png')

image = Image.open('captcha.png')

text = pytesseract.image\_to\_string(image,lang='eng')

print(text)

time.sleep(2)

# [5. 第五章：Scrapy框架](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia.html)

## [5.1. 框架架构](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\1-scrapykuang-jia-jia-gou.html)

### Scrapy框架介绍：

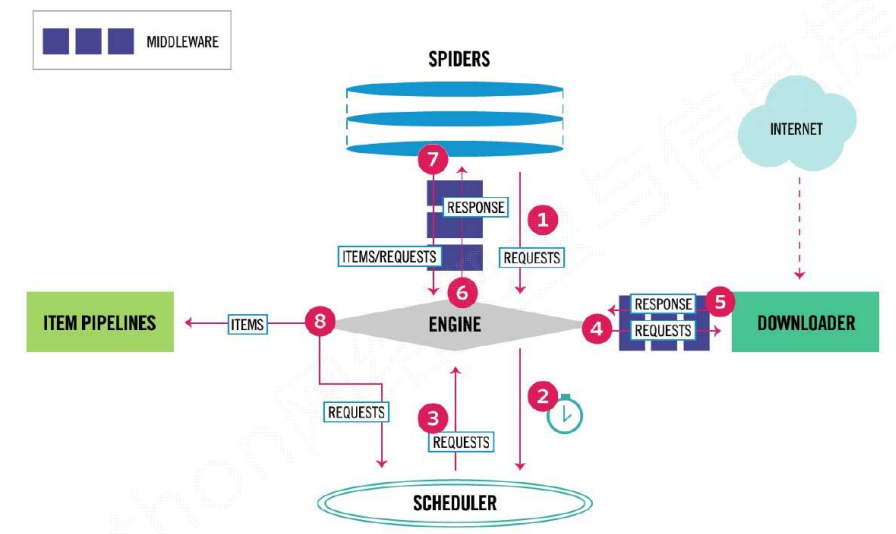
写一个爬虫, 需要做很多的事情。比如：发送网络请求、数据解析、数据存储、反反爬虫机制 (更换ip代理、设置请求头等) 、异步请求等。这些工作如果每次都要自己从零开始写的话, 比较浪费时间。因此Scrapy把一些基础的东西封装好了, 在他上面写爬虫可以变的更加的高效 (爬取效率和开发效率) 。因此真正在公司里, 一些上了量的爬虫, 都是使用Scrapy框架来解决。

### Scrapy架构图：

1. 流程图 (1) ：



1. 流程图 (2) ：



### Scrapy框架模块功能：

1. Scrapy Engine (引擎) ：Scrapy框架的核心部分。负责在Spider和ItemPipeline、Downloader、Scheduler中间通信、传递数据等。
2. Spider (爬虫) ：发送需要爬取的链接给引擎, 最后引擎把其他模块请求回来的数据再发送给爬虫, 爬虫就去解析想要的数据。这个部分是我们开发者自己写的, 因为要爬取哪些链接, 页面中的哪些数据是我们需要的, 都是由程序员自己决定。
3. Scheduler (调度器) ：负责接收引擎发送过来的请求, 并按照一定的方式进行排列和整理, 负责调度请求的顺序等。
4. Downloader (下载器) ：负责接收引擎传过来的下载请求, 然后去网络上下载对应的数据再交还给引擎。
5. Item Pipeline (管道) ：负责将Spider (爬虫) 传递过来的数据进行保存。具体保存在哪里, 应该看开发者自己的需求。
6. Downloader Middlewares (下载中间件) ：可以扩展下载器和引擎之间通信功能的中间件。
7. Spider Middlewares (Spider中间件) ：可以扩展引擎和爬虫之间通信功能的中间件。

## [5.2. 快速入门](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\2kuai-su-ru-men.html)

### 安装和文档：

1. 安装：通过pip install scrapy即可安装。
2. Scrapy官方文档：<http://doc.scrapy.org/en/latest>
3. Scrapy中文文档：<http://scrapy-chs.readthedocs.io/zh_CN/latest/index.html>

注意：

1. 在ubuntu上安装scrapy之前, 需要先安装以下依赖：

sudo apt-get install python3-dev build-essential python3-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev, 然后再通过pip install scrapy安装。

1. 如果在windows系统下, 提示这个错误ModuleNotFoundError: No module named 'win32api', 那么使用以下命令可以解决：pip install pypiwin32。

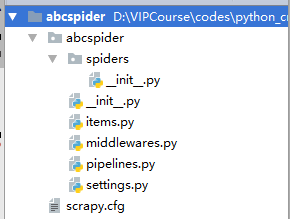
### 快速入门：

#### 创建项目：

要使用Scrapy框架创建项目, 需要通过命令来创建。首先进入到你想把这个项目存放的目录。然后使用以下命令创建：

scrapy startproject [项目名称]

#### 目录结构介绍：



以下介绍下主要文件的作用：

1. items.py：用来存放爬虫爬取下来数据的模型。
2. middlewares.py：用来存放各种中间件的文件。
3. pipelines.py：用来将items的模型存储到本地磁盘中。
4. settings.py：本爬虫的一些配置信息 (比如请求头、多久发送一次请求、ip代理池等) 。
5. scrapy.cfg：项目的配置文件。
6. spiders包：以后所有的爬虫, 都是存放到这个里面。

#### 使用Scrapy框架爬取糗事百科段子：

##### 使用命令创建一个爬虫：

进入项目根目录, 在项目根目录中创建爬虫

scrapy gensipder qsbk "qiushibaike.com"

创建了一个名字叫做qsbk的爬虫, 并且能爬取的网页只会限制在qiushibaike.com这个域名下。

##### 爬虫代码解析：

import scrapy

class QsbkSpider(scrapy.Spider):

name = 'qsbk'

allowed\_domains = ['qiushibaike.com']

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

def parse(self, response):

pass

其实这些代码我们完全可以自己手动去写, 而不用命令。只不过是不用命令, 自己写这些代码比较麻烦。

要创建一个Spider, 那么必须自定义一个类, 继承自scrapy.Spider, 然后在这个类中定义三个属性和一个方法。

1. name：这个爬虫的名字, 名字必须是唯一的。
2. allow\_domains：允许的域名。爬虫只会爬取这个域名下的网页, 其他不是这个域名下的网页会被自动忽略。
3. start\_urls：爬虫从这个变量中的url开始。
4. parse：引擎会把下载器下载回来的数据扔给爬虫解析, 爬虫再把数据传给这个parse方法。这个是个固定的写法。这个方法的作用有两个, 第一个是提取想要的数据。第二个是生成下一个请求的url。

##### 修改settings.py代码：

在做一个爬虫之前, 一定要记得修改setttings.py中的设置。两个地方是强烈建议设置的。

1. ROBOTSTXT\_OBEY设置为False。默认是True。即遵守机器协议, 那么在爬虫的时候, scrapy首先去找robots.txt文件, 如果没有找到。则直接停止爬取。
2. DEFAULT\_REQUEST\_HEADERS添加User-Agent。这个也是告诉服务器, 我这个请求是一个正常的请求, 不是一个爬虫。

##### 完成的爬虫代码：

1. 爬虫部分代码：

**import** **scrapy**

**from** **abcspider.items** **import** QsbkItem

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

name = 'qsbk'

allowed\_domains = ['qiushibaike.com']

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

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

outerbox = response.xpath("//div[@id='content-left']/div")

items = []

**for** box **in** outerbox:

author = box.xpath(".//div[contains(@class,'author')]//h2/text()").extract\_first().strip()

content = box.xpath(".//div[@class='content']/span/text()").extract\_first().strip()

item = QsbkItem()

item["author"] = author

item["content"] = content

items.append(item)

**return** items

1. items.py部分代码：

import scrapy

class QsbkItem(scrapy.Item):

author = scrapy.Field()

content = scrapy.Field()

1. pipeline部分代码：

import json

class AbcspiderPipeline(object):

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

self.items = []

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

self.items.append(dict(item))

print("="\*40)

return item

def close\_spider(self,spider):

with open('qsbk.json','w',encoding='utf-8') as fp:

json.dump(self.items,fp,ensure\_ascii=False)

##### 运行scrapy项目：

运行scrapy项目。需要在终端, 进入项目所在的路径, 然后scrapy crawl [爬虫名字]即可运行指定的爬虫。如果不想每次都在命令行中运行, 那么可以把这个命令写在一个文件中。以后就在pycharm中执行运行这个文件就可以了。比如现在新创建一个文件叫做start.py, 然后在这个文件中填入以下代码：

from scrapy import cmdline

cmdline.execute("scrapy crawl qsbk".split())

视频代码

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

**import** **scrapy**

**from** **qsbk.items** **import** QsbkItem

**from** **scrapy.http.response.html** **import** HtmlResponse

**from** **scrapy.selector.unified** **import** SelectorList

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

name = 'qsbk\_spider'

allowed\_domains = ['qiushibaike.com']

start\_urls = ['https://www.qiushibaike.com/text/page/1/']

base\_domain = "https://www.qiushibaike.com"

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

*# print(type(response))*

*# <class 'scrapy.http.response.html.HtmlResponse'>*

*# 查看HtmlResponse的所有方法和属性*

*# SelectorList*

duanzidivs = response.xpath("//div[@id='content-left']/div")

*# print(type(duanzidivs))*

*# <class 'scrapy.selector.unified.SelectorList'>*

# 查看HtmlResponse的源码

HtmlResponse是继承自TextResponse, TextResponse是继承自Response的.

'encoding' 编码方式

'text' 获取响应的文本

'body' 二进制方式的响应. 这里的text和body与requests中的response.text和response.content类似. text是经过解码之后的响应, body是没有经过解码的响应.

'xpath' 可以使用xpath提取其中的信息

'css' 可以对响应使用css选择器提取信息

scrapy中可以直接使用xpath提取信息, 但如果想要使用css选择器, 就要单独导入beautifulsoup, 使用bs把response.text解析为bs对象, 再从中提取信息.

from scrapy.http.response.html import HtmlResponse

from scrapy.http.response.text import TextResponse

from scrapy.http.response import Response

dir(HtmlResponse)

dir(TextResponse)

['\_DEFAULT\_ENCODING', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_auto\_detect\_fun', '\_body\_declared\_encoding', '\_body\_inferred\_encoding', '\_declared\_encoding', '\_get\_body', '\_get\_url', '\_headers\_encoding', '\_set\_body', '\_set\_url', 'body', 'body\_as\_unicode', 'copy', 'css', 'encoding', 'follow', 'meta', 'replace', 'selector', 'text', 'url', 'urljoin', 'xpath']

每次都要在命令行中执行scrapy crawl sqbk\_spider, 可以在项目根目录中新建一个文件, 如start.py

*#encoding: utf-8*

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

*# cmdline.execute("scrapy crawl qsbk\_spider".split())*

cmdline.execute(["scrapy",'crawl','qsbk\_spider'])

这样, 只需要执行start.py就可以运行爬虫了.

使用xpath提取出一个SlectorList对象.

查看SlectorList的源码

from scrapy.selector.unified import SelectorList

**class** SelectorList(\_ParselSelector.selectorlist\_cls, object\_ref):

@deprecated(use\_instead=**'.extract()'**)

**def** extract\_unquoted(self):

**return** [x.extract\_unquoted() **for** x **in** self]

@deprecated(use\_instead=**'.xpath()'**)

**def** x(self, xpath):

**return** self.select(xpath)

@deprecated(use\_instead=**'.xpath()'**)

**def** select(self, xpath):

**return** self.xpath(xpath)

其中的deprecated表示方法已经被抛弃, 使用后面定义的方法来代替.

其中的SelectorList中装的是所有的Selector, 查看Selector的方法

from parsel import Selector

dir(Selector)

['\_\_bool\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_nonzero\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_css2xpath', '\_csstranslator', '\_default\_namespaces', '\_default\_type', '\_expr', '\_get\_root', '\_lxml\_smart\_strings', '\_parser', '\_tostring\_method', 'css', 'extract', 'get', 'getall', 'namespaces', 're', 're\_first', 'register\_namespace', 'remove\_namespaces', 'root', 'selectorlist\_cls', 'text', 'type', 'xpath']

使用xpath获取到的元素类型是selector, 还可以使用xpath来进行进一步的选择. 也可以使用css, re来提取其中的信息.

extract把SelectorList转换为list, 其中的元素是Unicode格式的字符串.

extract\_first等价于get, 从selector中提取第1条符合规则的数据, 返回Unicode格式的字符串.

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

**import** **scrapy**

**from** **qsbk.items** **import** QsbkItem

**from** **scrapy.http.response.html** **import** HtmlResponse

**from** **scrapy.selector.unified** **import** SelectorList

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

name = 'qsbk\_spider'

allowed\_domains = ['qiushibaike.com']

start\_urls = ['https://www.qiushibaike.com/text/page/1/']

base\_domain = "https://www.qiushibaike.com"

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

*# print(type(response))*

*# <class 'scrapy.http.response.html.HtmlResponse'>*

*# 查看HtmlResponse的源码*

*# SelectorList*

duanzidivs = response.xpath("//div[@id='content-left']/div")

*# print(type(duanzidivs))*

*# <class 'scrapy.selector.unified.SelectorList'>*

**for** duanzidiv **in** duanzidivs:

*# 从SelectorList中遍历出来的每一个对象都是一个Selector, 可以再次使用xpath提取其中的内容, 同样得到SelectorList对象, 使用get()或者extract\_first()从SelectorList中提取出符合规则的第1个元素, 并转换为unicode字符串.*

author = duanzidiv.xpath(".//h2/text()").get().strip()

*# getall()方法等价于extract, 提取出符合规则的所有对象并以列表方式返回, 列表中的每一个元素都是一个unicdoe字符串.*

content = duanzidiv.xpath(".//div[@class='content']//text()").getall()

*# 把提取到的内容拼接为字符串*

content = "".join(content).strip()

# 把提取到的数据组装成字典, 传递给pipeline进行处理.

duanzi = {"author": author, "content": content}

**yield** duanzi

修改pipelines.py, 把spider传递过来的数据保存到文件中.

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

**import** **json**

**class** **QsbkPipeline**(object):

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

*# 要把数据写入到json文件中, 可以在open\_spider方法中打开文件, 也可以在构造函数中打开文件.*

self.fp = open("duanzi.json",'w',encoding='utf-8')

*# 爬虫开始运行时就会调用open\_spider*

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

**print**('爬虫开始了...')

*# 数据从spider中传递到pipelines中后会经过process\_item这个方法的处理.*

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

*# 把字典转换为json*

item\_json = json.dumps(item,ensure\_ascii=False)

*# 写入到文件中*

self.fp.write(item\_json+'**\n**')

**return** item

*# 爬虫关闭时会调用close\_spider这个函数*

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

*# 关闭文件*

self.fp.close()

**print**('爬虫结束了...')

在settigns.py中开启item\_pipelines, 同时进行必要的设置

ROBOTSTXT\_OBEY = False

DOWNLOAD\_DELAY = 1

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

ITEM\_PIPELINES = {

'qsbk.pipelines.QsbkPipeline': 300,

}

使用item模型对spider返回的数据进行规范化

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

**import** **scrapy**

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

author = scrapy.Field()

content = scrapy.Field()

修改spider

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

**import** **scrapy**

**from** **qsbk.items** **import** QsbkItem

**from** **scrapy.http.response.html** **import** HtmlResponse

**from** **scrapy.selector.unified** **import** SelectorList

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

name = 'qsbk\_spider'

allowed\_domains = ['qiushibaike.com']

start\_urls = ['https://www.qiushibaike.com/text/page/1/']

base\_domain = "https://www.qiushibaike.com"

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

*# print(type(response))*

*# <class 'scrapy.http.response.html.HtmlResponse'>*

*# 查看HtmlResponse的源码*

*# SelectorList*

duanzidivs = response.xpath("//div[@id='content-left']/div")

*# print(type(duanzidivs))*

*# <class 'scrapy.selector.unified.SelectorList'>*

**for** duanzidiv **in** duanzidivs:

*# 从SelectorList中遍历出来的每一个对象都是一个Selector, 可以再次使用xpath提取其中的内容, 同样得到SelectorList对象, 使用get()或者extract\_first()从SelectorList中提取出符合规则的第1个元素, 并转换为unicode字符串.*

author = duanzidiv.xpath(".//h2/text()").get().strip()

*# getall()方法等价于extract, 提取出符合规则的所有对象并以列表方式返回, 列表中的每一个元素都是一个unicdoe字符串.*

content = duanzidiv.xpath(".//div[@class='content']//text()").getall()

*# 把提取到的内容拼接为字符串*

content = "".join(content).strip()

item = QsbkItem(author=author, content=content)

**yield** item

此时传递到pipelines.py中的数据就不再是字典格式的了, 而是scrpy.item格式的数据了, 所以不能直接使用dumps转换为json了, 而是先要转换为字典, 再使用dumps转换为dson格式的数据.

**import** **json**

# 方法1 使用dumps把python对象转换为json再写入到文件中

**class** **QsbkJsonPipeline**(object):

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

*# 要把数据写入到json文件中, 可以在open\_spider方法中打开文件, 也可以在构造函数中打开文件.*

self.fp = open("duanzi.json",'w',encoding='utf-8')

*# 爬虫开始运行时就会调用open\_spider*

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

**print**('爬虫开始了...')

*# 数据从spider中传递到pipelines中后会经过process\_item这个方法的处理.*

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

*# item为scrpy.item格式的数据, 要先把item转换为字典, 再使用dumps转换为json*

item\_json = json.dumps(dict(item),ensure\_ascii=False)

*# 写入到文件中*

self.fp.write(item\_json+'**\n**')

**return** item

*# 爬虫关闭时会调用close\_spider这个函数*

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

*# 关闭文件*

self.fp.close()

**print**('爬虫结束了...')

使用JsonItemExporter和JsonLinesItemExporter来保存json数据

jsonItemExporter是等待所有数据都爬取完成后再写入到文件中, 在数量量比较大时对内存的需求比较高. 并且如果爬虫在运行过程中出错, 数据会全部丢失. JsonLinesItemExporter是逐行写入的, 对内存的要求比较低, 爬虫出错时原来已经提取的数据已经保存到硬盘中.

JsonItemExporter是整体以json格式写入的, 而JsonLinesItemExporter则是逐行写入的, 每一行都是一个字典, 整体上不是一个json格式. 可以通过重写在第一行开始时写入'[, 每一行结束时写入, 文件结尾处写入]'. 或者在文件写入完成后使用另一个函数对其进行处理, 转换为json格式.

**from** scrapy.exporters **import** JsonItemExporter

**class** QsbkJsonItemPipeline(object):

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

*# JsonItemExporter是以bytes方式写入数据的, 所以要使用二进制的方式打开文件. 注意使用二进制方式打开文件时不能指定编码*

self.fp = open(**"duanzi.json"**,**'wb'**)

*# 定义一个json exporter对象*

self.exporter = JsonItemExporter(self.fp,ensure\_ascii=**False**,encoding=**'utf-8'**)

*# 开始执行*

self.exporter.start\_exporting()

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

print(**'爬虫开始了...'**)

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

self.exporter.export\_item(item)

**return** item

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

*# 完成导入*

self.exporter.finish\_exporting()

self.fp.close()

print(**'爬虫结束了...'**)

**from** scrapy.exporters **import** JsonLinesItemExporter

**class** QsbkJsonLinesItemPipeline(object):

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

self.fp = open(**"duanzi.json"**,**'wb'**)

self.exporter = JsonLinesItemExporter(self.fp,ensure\_ascii=**False**,encoding=**'utf-8'**)

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

print(**'爬虫开始了...'**)

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

self.exporter.export\_item(item)

# 注意在process\_item处理完成之后必须要return item, 以供其它的pipline使用.

**return** item

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

self.fp.close()

print(**'爬虫结束了...'**)

查看scrapy中所有的可导出的数据类型

from scrapy import exporters

dir(exporters)

['BaseItem', 'BaseItemExporter', 'CsvItemExporter', 'JsonItemExporter', 'JsonLinesItemExporter', 'MarshalItemExporter', 'PickleItemExporter', 'PprintItemExporter', 'PythonItemExporter', 'ScrapyDeprecationWarning', 'ScrapyJSONEncoder', 'XMLGenerator', 'XmlItemExporter', '\_\_all\_\_', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'csv', 'io', 'is\_listlike', 'marshal', 'pickle', 'pprint', 'six', 'sys', 'to\_bytes', 'to\_native\_str', 'to\_unicode', 'warnings']

爬取多页的数据

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

**import** **scrapy**

**from** **qsbk.items** **import** QsbkItem

**from** **scrapy.http.response.html** **import** HtmlResponse

**from** **scrapy.selector.unified** **import** SelectorList

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

name = 'qsbk\_spider'

allowed\_domains = ['qiushibaike.com']

start\_urls = ['https://www.qiushibaike.com/text/page/1/']

base\_domain = "https://www.qiushibaike.com"

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

*# print(type(response))*

*# <class 'scrapy.http.response.html.HtmlResponse'>*

*# 查看HtmlResponse的源码*

*# SelectorList*

duanzidivs = response.xpath("//div[@id='content-left']/div")

*# print(type(duanzidivs))*

*# <class 'scrapy.selector.unified.SelectorList'>*

**for** duanzidiv **in** duanzidivs:

*# 从SelectorList中遍历出来的每一个对象都是一个Selector, 可以再次使用xpath提取其中的内容, 同样得到SelectorList对象, 使用get()或者extract\_first()从SelectorList中提取出符合规则的第1个元素, 并转换为unicode字符串.*

author = duanzidiv.xpath(".//h2/text()").get().strip()

*# getall()方法等价于extract, 提取出符合规则的所有对象并以列表方式返回, 列表中的每一个元素都是一个unicdoe字符串.*

content = duanzidiv.xpath(".//div[@class='content']//text()").getall()

*# 把提取到的内容拼接为字符串*

content = "".join(content).strip()

item = QsbkItem(author=author, content=content)

*# 如果这里使用return, 就不能提取出下一页的内容了, 所以这里要使用yield*

**yield** item

*# 提取出下一页的url地址*

next\_url = response.xpath("//ul[@class='pagination']/li[last()]/a/@href").get()

**if** next\_url:

*# 如果存在下一页的url地址, 就构建Request对象, 并指定响应的处理函数*

**yield** scrapy.Request(self.base\_domain + next\_url, callback=self.parse)

**else**:

**return**

## [5.3. CrawlSpider](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\3-crawlspider.html)

在上一个糗事百科的爬虫案例中。我们是自己在解析完整个页面后获取下一页的url, 然后重新发送一个请求。有时候我们想要这样做, 只要满足某个条件的url, 都给我进行爬取。那么这时候我们就可以通过CrawlSpider来帮我们完成了。CrawlSpider继承自Spider, 只不过是在之前的基础之上增加了新的功能, 可以定义爬取的url的规则, 以后scrapy碰到满足条件的url都进行爬取, 而不用手动的yield Request。

### CrawlSpider爬虫：

#### 创建CrawlSpider爬虫：

之前创建爬虫的方式是通过scrapy genspider [爬虫名字] [域名]的方式创建的。如果想要创建CrawlSpider爬虫, 那么应该通过以下命令创建：

scrapy genspider -c crawl [爬虫名字] [域名]

#### LinkExtractors链接提取器：

使用LinkExtractors可以不用程序员自己提取想要的url, 然后发送请求。这些工作都可以交给LinkExtractors, 他会在所有爬的页面中找到满足规则的url, 实现自动的爬取。以下对LinkExtractors类做一个简单的介绍：

class scrapy.linkextractors.LinkExtractor(

allow = (),

deny = (),

allow\_domains = (),

deny\_domains = (),

deny\_extensions = None,

restrict\_xpaths = (),

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

attrs = ('href'),

canonicalize = True,

unique = True,

process\_value = None

)

主要参数讲解：

* allow：允许的url。所有满足这个正则表达式的url都会被提取。
* deny：禁止的url。所有满足这个正则表达式的url都不会被提取。
* allow\_domains：允许的域名。只有在这个里面指定的域名的url才会被提取。
* deny\_domains：禁止的域名。所有在这个里面指定的域名的url都不会被提取。
* restrict\_xpaths：严格的xpath。和allow共同过滤链接。

#### Rule规则类：

定义爬虫的规则类。以下对这个类做一个简单的介绍：

class scrapy.spiders.Rule(

link\_extractor,

callback = None,

cb\_kwargs = None,

follow = None,

process\_links = None,

process\_request = None

)

主要参数讲解：

* link\_extractor：一个LinkExtractor对象, 用于定义爬取规则。
* callback：满足这个规则的url, 应该要执行哪个回调函数。因为CrawlSpider使用了parse作为回调函数, 因此不要覆盖parse作为回调函数自己的回调函数。
* follow：指定根据该规则从response中提取的链接是否需要跟进。
* process\_links：从link\_extractor中获取到链接后会传递给这个函数, 用来过滤不需要爬取的链接。

#### 微信小程序社区CrawlSpider案例

创建项目

scrapy startproject wxapp\_spider

创建爬虫

cd wxapp\_spider

scrapy genspider -t crawl wxapp "wxapp-union.com"

定义items.py

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

**import** **scrapy**

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

title = scrapy.Field()

author = scrapy.Field()

pub\_time = scrapy.Field()

content = scrapy.Field()

定义spider, wxapp.py

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

**import** **scrapy**

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

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

**from** **wxapp.items** **import** WxappItem

**from** **scrapy.http.response.html** **import** HtmlResponse

**class** **WxappSpiderSpider**(CrawlSpider):

name = 'wxapp\_spider'

allowed\_domains = ['wxapp-union.com']

start\_urls = ['http://www.wxapp-union.com/portal.php?mod=list&catid=2&page=1']

rules = (

*# 匹配列表页url, 不指定callback时默认是使用parse函数来解析响应. 需要从提取到的链接的响应中提取内容时才需要手动指定callback.*

Rule(LinkExtractor(allow=r'.+mod=list&catid=2&page=\d'), follow=True),

*# 匹配详情页url. 因为只需要提取列表页中的文章详情url地址, 所以在文章详情页中不需要提取详情页的url, 所以这里的follow=False*

Rule(LinkExtractor(allow=r".+article-.+\.html"),callback="parse\_detail",follow=False)

)

*# 解析详情页, 从中提取信息*

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

title = response.xpath("//h1[@class='ph']/text()").get()

author\_p = response.xpath("//p[@class='authors']")

author = author\_p.xpath(".//a/text()").get()

pub\_time = author\_p.xpath(".//span/text()").get()

article\_content = response.xpath("//td[@id='article\_content']//text()").getall()

content = "".join(article\_content).strip()

item = WxappItem(title=title,author=author,pub\_time=pub\_time,content=content)

**yield** item

修改pipelines.py, 把数据保存到json文件中

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

**from** **scrapy.exporters** **import** JsonLinesItemExporter

**class** **WxappPipeline**(object):

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

self.fp = open('wxjc.json','wb')

self.exporter = JsonLinesItemExporter(self.fp,ensure\_ascii=False,encoding='utf-8')

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

self.exporter.export\_item(item)

**return** item

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

self.fp.close()

修改settings.py

ROBOTSTXT\_OBEY = False

DOWNLOAD\_DELAY = 1

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

ITEM\_PIPELINES = {

'wxapp.pipelines.WxappPipeline': 300,

}

在项目根目录中新建start.py

*#encoding: utf-8*

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

cmdline.execute("scrapy crawl wxapp\_spider".split())

运行start.py, 进行测试

## [5.4. ScrapyShell](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\4-scrapyshell.html)

我们想要在爬虫中使用xpath、beautifulsoup、正则表达式、css选择器等来提取想要的数据。但是因为scrapy是一个比较重的框架。每次运行起来都要等待一段时间。因此要去验证我们写的提取规则是否正确, 是一个比较麻烦的事情。因此Scrapy提供了一个shell, 用来方便的测试规则。当然也不仅仅局限于这一个功能。

### 打开Scrapy Shell：

打开cmd终端, 进入到Scrapy项目所在的目录, 然后进入到scrapy框架所在的虚拟环境中, 输入命令scrapy shell [链接]。就会进入到scrapy的shell环境中。在这个环境中, 你可以跟在爬虫的parse方法中一样使用了。

scrapy shell可以在任意位置打开, 但是如果想要读取某个scrapy 项目的配置文件时, 必须要进入到项目的根目录中打开scrapy shell

scrapy shell http://www.wxapp-union.com/article-4180-1.html

[s] Available Scrapy objects:

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

[s] crawler <scrapy.crawler.Crawler object at 0x0000021FF1F50080>

[s] item {}

[s] request <GET http://www.wxapp-union.com/portal.php?mod=list>

[s] response <200 http://www.wxapp-union.com/portal.php?mod=list>

[s] settings <scrapy.settings.Settings object at 0x0000021FF462DC88>

[s] spider <DefaultSpider 'default' at 0x21ff48e43c8>

[s] Useful shortcuts:

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

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

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

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

response.xpath("//h1[@class='ph']/text()").get()

'小程序的十万个为什么丨框架 '

from bs4 import BeautifulSoup

soup = BeautifulSoup(response.text, 'lxml')

soup.find('h1', attrs=['class', 'ph'])

## [5.5. Request和Response对象](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\5-requesthe-response-dui-xiang.html)

### Request对象：

**from** scrapy.http.request **import** Request

dir(Request)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_get\_body', '\_get\_url', '\_set\_body', '\_set\_url', 'body', 'copy', 'encoding', 'meta', 'replace', 'url']

**class** Request(object\_ref):  
  
 **def** \_\_init\_\_(self, url, callback=**None**, method=**'GET'**, headers=**None**, body=**None**,  
 cookies=**None**, meta=**None**, encoding=**'utf-8'**, priority=0,  
 dont\_filter=**False**, errback=**None**, flags=**None**):  
  
 self.\_encoding = encoding *# this one has to be set first* self.method = str(method).upper()  
 self.\_set\_url(url)  
 self.\_set\_body(body)  
 **assert** isinstance(priority, int), **"Request priority not an integer: %r"** % priority  
 self.priority = priority  
  
 **if** callback **is not None and not** callable(callback):  
 **raise** TypeError(**'callback must be a callable, got %s'** % type(callback).\_\_name\_\_)  
 **if** errback **is not None and not** callable(errback):  
 **raise** TypeError(**'errback must be a callable, got %s'** % type(errback).\_\_name\_\_)  
 **assert** callback **or not** errback, **"Cannot use errback without a callback"** self.callback = callback  
 self.errback = errback  
  
 self.cookies = cookies **or** {}  
 self.headers = Headers(headers **or** {}, encoding=encoding)  
 self.dont\_filter = dont\_filter  
  
 self.\_meta = dict(meta) **if** meta **else None** self.flags = [] **if** flags **is None else** list(flags)

Request对象在我们写爬虫, 爬取一页的数据需要重新发送一个请求的时候调用。这个类需要传递一些参数, 其中比较常用的参数有：

1. url：这个request对象发送请求的url。
2. callback：在下载器下载完相应的数据后执行的回调函数。
3. method：请求的方法。默认为GET方法, 可以设置为其他方法。
4. headers：请求头, 对于一些固定的设置, 放在settings.py中指定就可以了。对于那些非固定的经常变化的请求头, 可以在发送请求的时候指定。
5. meta：比较常用。用于在不同的请求之间传递数据用的。
6. encoding：编码。默认的为utf-8, 使用默认的就可以了。
7. dont\_filter：默认值是True, 表示由调度器进行过滤。在执行多次重复的请求的时候要设置为False, 不经过调度器进行过滤. 如对于那些变化很快的页面, 如热闹的帖子页, 或者需要对某些字段进行更新的页面, 或者是验证码的生成页, 都需要多次向同一个url地址发送请求, 所以这些url地址的dont\_filter要设置为False.
8. errback：在发生错误的时候执行的函数。

### Response对象：

**from** scrapy.http.response **import** Response

dir(Response)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_get\_body', '\_get\_url', '\_set\_body', '\_set\_url', 'body', 'copy', 'css', 'follow', 'meta', 'replace', 'text', 'url', 'urljoin', 'xpath']

Response对象一般是由Scrapy给你自动构建的。因此开发者不需要关心如何创建Response对象, 而是如何使用他。Response对象有很多属性, 可以用来提取数据的。主要有以下属性：

1. meta：从其他请求传过来的meta属性, 可以用来保持多个请求之间的数据连接。
2. encoding：返回当前字符串编码和解码的格式。
3. text：将返回来的数据作为unicode字符串返回。text和body都是网页的源码, 区别就在于text为经过解码后的unicode字符串.
4. body：将返回来的数据作为bytes字符串返回。
5. xpath：xapth选择器。
6. css：css选择器。

### 发送POST请求：

有时候我们想要在请求数据的时候发送post请求, 那么这时候需要使用Request的子类FormRequest来实现。如果想要在爬虫一开始的时候就发送POST请求, 那么需要在爬虫类中重写start\_requests(self)方法, 并且不再调用start\_urls里的url。

**from** scrapy.http.request.form **import** FormRequest

dir(FormRequest)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_get\_body', '\_get\_url', '\_set\_body', '\_set\_url', 'body', 'copy', 'encoding', 'from\_response', 'meta', 'replace', 'url']

查看scrapy.Spider的源码

from scrapy import Spider

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_set\_crawler', 'close', 'custom\_settings', 'from\_crawler', 'handles\_request', 'log', 'logger', 'make\_requests\_from\_url', 'name', 'parse', 'set\_crawler', 'start\_requests', 'update\_settings']

查看start\_requests的源码

**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**)

从start\_urls中遍历所有的链接, 组装并发送Reuqest请求对象, 默认是使用的get请求.

from scrapy import FormRequest

dir(FormRequest)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_get\_body', '\_get\_url', '\_set\_body', '\_set\_url', 'body', 'copy', 'encoding', 'from\_response', 'meta', 'replace', 'url']

### 模拟登录

#### 案例一：模拟登录人人网

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

**import** **scrapy**

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

name = 'renren'

allowed\_domains = ['renren.com']

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

*# 如果想要在爬虫一开始的时候就发送POST请求, 那么需要在爬虫类中重写start\_requests(self)方法*

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

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

data = {"email": "970138074@qq.com", 'password': "pythonspider"}

request = scrapy.FormRequest(url,formdata=data,callback=self.parse\_page)

**yield** request

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

*# 把发送的post请求的响应数据写入到本地, 查看是否登陆成功*

*# with open('dp.html','w',encoding='utf-8') as fp:*

*# fp.write(response.text)*

*# 登陆成功后访问必须要登陆才能访问的页面, 并指定解析函数对响应进行处理*

request = scrapy.Request(url='http://www.renren.com/880151247/profile',callback=self.parse\_profile)

**yield** request

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

**with** open('dp.html','w',encoding='utf-8') **as** fp:

fp.write(response.text)

#### 案例二：模拟登录豆瓣网 (识别验证码)

图形验证码识别平台：<https://market.aliyun.com/products/57126001/cmapi014396.html#sku=yuncode839600006>

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

**import** **scrapy**

**from** **urllib** **import** request

**from** **PIL** **import** Image

**from** **urllib** **import** request

**from** **base64** **import** b64encode

**import** **requests**

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

name = 'douban'

allowed\_domains = ['douban.com']

start\_urls = ['https://accounts.douban.com/login']

login\_url = 'https://accounts.douban.com/login'

profile\_url = 'https://www.douban.com/people/97956064/'

editsignature\_url = 'https://www.douban.com/j/people/97956064/edit\_signature'

*# start\_urls中url的响应会使用parse函数进行解析.*

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

*# 构造登陆请求的post数据. 字段与登陆表单中的字段对应*

formdata = {

*# 从哪个页面跳转来的*

'source': 'None',

*# 登陆完成后要跳转到的页面*

'redir': 'https://www.douban.com/',

'form\_email': '970138074@qq.com',

'form\_password': 'pythonspider',

'remember': 'on',

'login': '登录'

}

*# 提取出验证码的地址*

captcha\_url = response.css('img#captcha\_image::attr(src)').get()

*# 如果登陆时有验证码, 才执行验证码识别的操作, 如果没有验证码, 就直接登陆.*

**if** captcha\_url:

*# 识别验证码并把结果添加到formdata中*

captcha = self.regonize\_captcha(captcha\_url)

formdata['captcha-solution'] = captcha

*# 提取出captcha\_id并添加到formdata中*

captcha\_id = response.xpath("//input[@name='captcha-id']/@value").get()

formdata['captcha-id'] = captcha\_id

*# 构造登陆的post请求, 并指定登陆响应的解析函数*

**yield** scrapy.FormRequest(url=self.login\_url, formdata=formdata, callback=self.parse\_after\_login)

**def** parse\_after\_login(self, response):

*# 登录后会自动跳转到豆瓣首页, 如果响应的url是豆瓣首页, 就表明登陆成功.*

**if** response.url == 'https://www.douban.com/':

*# 登陆成功后跳转到个人中心页*

**yield** scrapy.Request(self.profile\_url, callback=self.parse\_profile)

**print**('登录成功！')

**else**:

**print**('登录失败！')

*# 对个人中心页的响应进行处理*

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

**print**(response.url)

*# 如果响应的url是个人中心的url, 就修改个性签名.*

**if** response.url == self.profile\_url:

*# 构造修改个性命名的表单数据.*

ck = response.xpath("//input[@name='ck']/@value").get()

formdata = {

'ck': ck,

'signature': '我可以自动识别图形验证码啦~~'

}

*# 发送修改个性签名的post数据, 在scrapy爬虫中, 构造的Reuqest请求对象如果没有指定解析函数, 会自动调用parse方法. 所以如果这里不指定, 会再次进行登陆请求. 可以在parse方法中打上断点进行调试.*

**yield** scrapy.FormRequest(self.editsignature\_url, formdata=formdata, callback=self.parse\_none)

**else**:

**print**('没有进入到个人中心')

*# 修改个性签名的响应数据不需要进行进一步的处理.*

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

**pass**

*# 手动输入验证码*

**def** regonize\_captcha(self, image\_url):

request.urlretrieve(image\_url, 'captcha.png')

image = Image.open('captcha.png')

image.show()

captcha = input('请输入验证码：')

image.close()

**return** captcha

*# 自动识别验证码*

**def** regonize\_captcha(self, image\_url):

captcha\_url = image\_url

request.urlretrieve(captcha\_url, 'captcha.png')

*# 构建验证码识别的url*

recognize\_url = 'http://jisuyzmsb.market.alicloudapi.com/captcha/recognize?type=e'

formdata = {}

*# 对图片进行base64编码*

**with** open('captcha.png', 'rb') **as** fp:

data = fp.read()

pic = b64encode(data)

formdata['pic'] = pic

appcode = '831a890b2cfe4ea0a8e345078434ebfc'

headers = {

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

'Authorization': 'APPCODE ' + appcode

}

*# 发送识别验证码的post请求*

response = requests.post(recognize\_url, data=formdata, headers=headers)

*# 从返回的json数据中提取出验证码*

result = response.json()

code = result['result']['code']

**return** code

## 5.6. 下载文件和图片

Scrapy为下载item中包含的文件(比如在爬取到产品时, 同时也想保存对应的图片)提供了一个可重用的item pipelines。这些pipeline有些共同的方法和结构(我们称之为media pipeline)。一般来说你会使用Files Pipeline或者Images Pipeline。

images pipeline继承自files pipeline, 增加了图片的一些方法, 如生成缩略图.

### 为什么要使用scrapy内置的下载图片/文件的方法：

1. 避免重新下载最近已经下载过的文件。
2. 可以方便的指定文件存储的路径。
3. 可以将下载的图片转换成通用的格式。比如png或jpg。
4. 可以方便的生成缩略图。
5. 可以方便的检测图片的宽和高, 确保他们满足最小限制。
6. 异步下载, 效率非常高。

### 下载文件的Files Pipeline：

当使用Files Pipeline下载文件的时候, 按照以下步骤来完成：

1. 定义好一个Item, 然后在这个item中定义两个属性, 分别为file\_urls以及files。file\_urls是一个列表, 用来存储需要下载的文件的url链接. files pipeline就会从这个列表中读取所有的图片或文件的url地址进行下载。files不常用, 用来保存下载的文件的相应信息.
2. 当文件下载完成后, 会把文件下载的相关信息存储到item的files属性中。比如下载路径、下载的url和文件的校验码等。
3. 在配置文件settings.py中配置FILES\_STORE, 这个配置是用来设置文件下载下来的路径。
4. 启动pipeline：在ITEM\_PIPELINES中设置scrapy.pipelines.files.FilesPipeline:1。

### 下载图片的Images Pipeline：

当使用Images Pipeline下载文件的时候, 按照以下步骤来完成：

1. 定义好一个Item, 然后在这个item中定义两个属性, 分别为image\_urls以及images。image\_urls是一个列表, 用来存储需要下载的图片的url链接,
2. 当文件下载完成后, 会把文件下载的相关信息存储到item的images属性中。比如下载路径、下载的url和图片的校验码等。
3. 在配置文件settings.py中配置IMAGES\_STORE, 这个配置是用来设置图片下载下来的路径。
4. 启动pipeline：在ITEM\_PIPELINES中设置scrapy.pipelines.images.ImagesPipeline:1。

### 汽车之家宝马5系高清图片下载实战。

新建scrapy项目 scrapy startproject bmw\_spider

新建爬虫, scrapy genspider bmw5

修改items.py, 定义item类

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

**import** **scrapy**

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

category = scrapy.Field()

image\_urls = scrapy.Field()

images = scrapy.Field()

修改spider/bmw5.py, 定义爬虫

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

**import** **scrapy**

**from** **bmw.items** **import** BmwItem

**class** **Bmw5Spider**(scrapy.Spider):

name = 'bmw5'

allowed\_domains = ['car.autohome.com.cn']

start\_urls = ['https://car.autohome.com.cn/pic/series/65.html']

*# 提取图片的缩略图*

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

*# 每一个'uibox'代表一类的图片. 第1个uibox不需要获取.*

uiboxs = response.xpath("//div[@class='uibox']")[1:]

*# SelectorList -> list, 可以进行遍历*

**for** uibox **in** uiboxs:

*# 图片的分类*

category = uibox.xpath(".//div[@class='uibox-title']/a/text()").get()

*# 提取每一个图片分类下面的所有图片缩略图的url地址*

urls = uibox.xpath(".//ul/li/a/img/@src").getall()

*# 对提取到的url地址进行补全*

*# for url in urls:*

*# # url = "https" + url*

*# # 使用response.urljoin()对网址进行拼接*

*# url = response.urljoin(url)*

*# print(url)*

urls = list(map(**lambda** url:response.urljoin(url),urls))

urls = [response.urljoin(url) **for** url **in** urls]

item = BmwItem(category=category, urls=urls)

**yield** item

os路径的使用方法

*#encoding: utf-8*

**import** **os**

*# 获取本文件的路径*

file\_name = os.path.dirname(\_\_file\_\_)

**print**("file\_name:**%s**"%file\_name)

*# 获取本文件上一级的路径*

folder\_name = os.path.dirname(os.path.dirname(\_\_file\_\_))

**print**("folder\_name:**%s**"%folder\_name)

*# 拼接路径*

images\_path = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)),'images')

**print**("images\_path:**%s**"%images\_path)

*# 判断路径是否存在*

**if** **not** os.path.exists(images\_path):

os.mkdir(images\_path)

**else**:

**print**('images文件夹存在')

#### 使用传统方法来保存图片

缺点, 只能一张张的下载图片

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

**import** **os**

**from** **urllib** **import** request

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

**from** **bmw** **import** settings

**class** **BmwPipeline**(object):

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

*# 判断images文件夹是否存在, 如果不存在, 就新建一个images文件夹. 因为后期要不断的使用images这个路径, 所以要保存为一个类属性*

self.path = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'images')

**if** **not** os.path.exists(self.path):

os.mkdir(self.path)

*# 传统方式: 单线程自定义方法下载图片*

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

*# 取出图片的分类, 把图片保存在以分类命名的文件夹中*

category = item['category']

urls = item['urls']

*# 在images文件夹下新建一个以category命名的文件夹*

category\_path = os.path.join(self.path,category)

**if** **not** os.path.exists(category\_path):

os.mkdir(category\_path)

**for** url **in** urls:

*# 从url中取出来图片的名字*

image\_name = url.split('\_')[-1]

*# 把图片保存到分类目录下, images\_name命名的文件中*

request.urlretrieve(url,os.path.join(category\_path,image\_name))

**return** item

在settings.py中进行相应的设置, 运行爬虫

ROBOTSTXT\_OBEY = False

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

ITEM\_PIPELINES = {

'bmw.pipelines.BmwPipeline': 300

}

#### 使用Images Pipeline来下载图片

修改items.py文件, 定义image\_urls和images

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

**import** **scrapy**

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

category = scrapy.Field()

*# image\_urls保存图片的下载链接*

image\_urls = scrapy.Field()

*# images保存下载的图片的信息*

images = scrapy.Field()

修改bmw5.py爬虫文件, 把item中的urls修改为image\_urls

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

**import** **scrapy**

**from** **bmw.items** **import** BmwItem

**class** **Bmw5Spider**(scrapy.Spider):

name = 'bmw5'

allowed\_domains = ['car.autohome.com.cn']

start\_urls = ['https://car.autohome.com.cn/pic/series/65.html']

*# 提取图片的缩略图*

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

*# 每一个'uibox'代表一类的图片. 第1个uibox不需要获取.*

uiboxs = response.xpath("//div[@class='uibox']")[1:]

*# SelectorList -> list, 可以进行遍历*

**for** uibox **in** uiboxs:

*# 图片的分类*

category = uibox.xpath(".//div[@class='uibox-title']/a/text()").get()

*# 提取每一个图片分类下面的所有图片缩略图的url地址*

urls = uibox.xpath(".//ul/li/a/img/@src").getall()

*# 对提取到的url地址进行补全*

*# for url in urls:*

*# # url = "https" + url*

*# # 使用response.urljoin()对网址进行拼接*

*# url = response.urljoin(url)*

*# print(url)*

urls = list(map(**lambda** url:response.urljoin(url),urls))

urls = [response.urljoin(url) **for** url **in** urls]

item = BmwItem(category=category,image\_urls=urls)

**yield** item

修改settings.py, 启用imagespipeline并设置图片保存的路径.

**import** **os**

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = False

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

ITEM\_PIPELINES = {

*# 'bmw.pipelines.BmwPipeline': 300,*

'scrapy.pipelines.images.ImagesPipeline': 1

}

*# 图片下载的路径, 供images pipelines使用*

IMAGES\_STORE = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'images')

运行爬虫

#### 重写ImagesPipeline

使用默认的ImagesPipeline时所有的图片都被下载到同一个文件夹中, 想要把不同分类的图片保存到不同的文件夹中, 需要重写ImagesPipeline.

from scrapy.pipelines.images import ImagesPipeline

dir(ImagesPipeline)

['DEFAULT\_FILES\_RESULT\_FIELD', 'DEFAULT\_FILES\_URLS\_FIELD', 'DEFAULT\_IMAGES\_RESULT\_FIELD', 'DEFAULT\_IMAGES\_URLS\_FIELD', 'EXPIRES', 'LOG\_FAILED\_RESULTS', 'MEDIA\_NAME', 'MIN\_HEIGHT', 'MIN\_WIDTH', 'STORE\_SCHEMES', 'SpiderInfo', 'THUMBS', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_cache\_result\_and\_execute\_waiters', '\_check\_media\_to\_download', '\_get\_store', '\_handle\_statuses', '\_key\_for\_pipe', '\_modify\_media\_request', '\_process\_request', 'convert\_image', 'file\_downloaded', 'file\_key', 'file\_path', 'from\_crawler', 'from\_settings', 'get\_images', 'get\_media\_requests', 'image\_downloaded', 'image\_key', 'inc\_stats', 'item\_completed', 'media\_downloaded', 'media\_failed', 'media\_to\_download', 'open\_spider', 'process\_item', 'thumb\_key', 'thumb\_path']

查看ImagesPipeline的源码, 其中的file\_path就是图片保存路径的方法, 需要重写这个方法

**def** file\_path(self, request, response=**None**, info=**None**):  
 *## start of deprecation warning block (can be removed in the future)* **def** \_warn():  
 **from** scrapy.exceptions **import** ScrapyDeprecationWarning  
 **import** warnings  
 warnings.warn(**'ImagesPipeline.image\_key(url) and file\_key(url) methods are deprecated, '  
 'please use file\_path(request, response=None, info=None) instead'**,  
 category=ScrapyDeprecationWarning, stacklevel=1)  
  
 *# check if called from image\_key or file\_key with url as first argument* **if not** isinstance(request, Request):  
 \_warn()  
 url = request  
 **else**:  
 url = request.url  
  
 *# detect if file\_key() or image\_key() methods have been overridden* **if not** hasattr(self.file\_key, **'\_base'**):  
 \_warn()  
 **return** self.file\_key(url)  
 **elif not** hasattr(self.image\_key, **'\_base'**):  
 \_warn()  
 **return** self.image\_key(url)  
 *## end of deprecation warning block* image\_guid = hashlib.sha1(to\_bytes(url)).hexdigest() *# change to request.url after deprecation* **return 'full/%s.jpg'** % (image\_guid)

修改pipelines.py, 重写file\_path这个方法.

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

**import** **os**

**from** **urllib** **import** request

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

**from** **bmw** **import** settings

**class** **BmwPipeline**(object):

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

*# 判断images文件夹是否存在, 如果不存在, 就新建一个images文件夹. 因为后期要不断的使用images这个路径, 所以要保存为一个类属性*

self.path = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'images')

**if** **not** os.path.exists(self.path):

os.mkdir(self.path)

*# 传统方式: 单线程自定义方法下载图片*

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

*# 取出图片的分类, 把图片保存在以分类命名的文件夹中*

category = item['category']

urls = item['urls']

*# 在images文件夹下新建一个以category命名的文��夹*

category\_path = os.path.join(self.path, category)

**if** **not** os.path.exists(category\_path):

os.mkdir(category\_path)

**for** url **in** urls:

*# 从url中取出来图片的名字*

image\_name = url.split('\_')[-1]

*# 把图片保存到分类目录下, images\_name命名的文件中*

request.urlretrieve(url, os.path.join(category\_path, image\_name))

**return** item

**class** **BMWImagesPipeline**(ImagesPipeline):

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

*# get\_media\_requests这个方法是在发送下载请求之前调用。file\_path方法是在请求之后调用, 这个方法其实就是构造下载文件请求的*

*"""*

*# get\_media\_requests方法的源码*

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

*# images\_urls\_field最终还是等于image\_urls, 所以get\_media\_requests方法实际上就是从item中的image\_urls列表中读取每一个链接, 构造请求对象. 返回一个包含所有请求对象的列表*

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

*"""*

*# 因为get\_media\_requests就是从image\_urls中读取url, 构造Request请求对象, 得到的就是所有请求对象的列表.*

request\_objs = super(BMWImagesPipeline, self).get\_media\_requests(item, info)

**for** request\_obj **in** request\_objs:

*# 把item绑定到每一个Request请求对象上. 因为get\_media\_requests这个方法是在发送下载请求之前调用。file\_path方法是在请求之后调用, 所以就可以��file\_path这个方法中从Request请求对象中读取item*

request\_obj.item = item

*# 一定要把request\_objs返回, 与父类中的方法保持一致, 否则就不会去下载图片.*

**return** request\_objs

**def** file\_path(self, request, response=None, info=None):

*# 这个方法是在图片将要被存储的时���调用, 来获取这个图片存储的路径. 即图片将要被存储到硬盘中时就调用file\_path读取设置中的路径.*

*# 调用父类的file\_path方法.*

path = super(BMWImagesPipeline, self).file\_path(request, response, info)

*# 获取子分类, 判断是否存在子分类的目录, 如果存在, 就把图片保存到子分类的目录中, 如果不存在, 就创建对应的目录.*

*# 图片的目录信息保存在item中, 无法直接获取, 由于get\_media\_requests这个方法中有item, 所以可以重写get\_media\_requests这个方法. 把item绑定到一个file\_path和get\_media\_request共有的对象中, 再在file\_path中从这个对象中读取item即可.*

category = request.item.get('category')

images\_store = settings.IMAGES\_STORE

category\_path = os.path.join(images\_store, category)

**if** **not** os.path.exists(category\_path):

os.mkdir(category\_path)

*"""*

*# 查看file\_path的源码, 返回的是url地址的sha1值, 所以path中的内容就是图片url对应的sha1值. 把full/替换掉就得到了图片url地址的sha1值*

*def file\_path():*

*...*

*image\_guid = hashlib.sha1(to\_bytes(url)).hexdigest() # change to request.url after deprecation*

*return 'full/%s.jpg' % (image\_guid)*

*"""*

image\_name = path.replace("full/", "")

*# 构建包括文件名在内的路径并返回, 这样就与父类中返回的内容相同了.*

image\_path = os.path.join(category\_path, image\_name)

**return** image\_path

修改settings.py, 激活**BMWImagesPipeline**

**import** **os**

BOT\_NAME = 'bmw'

SPIDER\_MODULES = ['bmw.spiders']

NEWSPIDER\_MODULE = 'bmw.spiders'

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = False

*# Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

*# Configure item pipelines*

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

ITEM\_PIPELINES = {

*# 'bmw.pipelines.BmwPipeline': 300,*

*# 'scrapy.pipelines.images.ImagesPipeline': 1*

'bmw.pipelines.BMWImagesPipeline': 1

}

*# 图片下载的路径, 供images pipelines使用*

IMAGES\_STORE = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'images')

#### 把爬虫修改为crawl.Spider爬虫, 爬取所有高清图片

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

**import** **scrapy**

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

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

**from** **bmw.items** **import** BmwItem

*# 把爬虫修改为CrawlSpider类的爬虫*

**class** **Bmw5Spider**(CrawlSpider):

name = 'bmw5'

allowed\_domains = ['car.autohome.com.cn']

start\_urls = ['https://car.autohome.com.cn/pic/series/65.html']

rules = (

*# 提取出更多图片列表页的地址. 可能还存在着分页, 所以要对页面进行跟进. 这个规则既可以匹配到 "更多" 的地址, 又能匹配到 "下一页" 的地址*

Rule(LinkExtractor(allow=r"https://car.autohome.com.cn/pic/series/65.+"),callback="parse\_page",follow=True),

)

*# CrawlSpider爬虫中的parse方法是用来解析start\_urls中的地址的, 所以这里要定义一个新的解析函数*

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

*# 提取出图片的分类*

category = response.xpath("//div[@class='uibox']/div/text()").get()

*# 提取出图片的链接地址*

srcs = response.xpath('//div[contains(@class,"uibox-con")]/ul/li//img/@src').getall()

*# 构建高清图的url地址, 把缩略图中的t\_去掉就得到高清图的地址*

*# srcs = list(map(lambda x:response.urljoin(x),srcs))*

srcs = list(map(**lambda** x:response.urljoin(x.replace("t\_","")),srcs))

srcs = [response.urljoin(src.replace("t\_","")) **for** src **in** srcs]

**yield** BmwItem(category=category,image\_urls=srcs)

## [5.7. 下载中间件](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\6xia-zai-zhong-jian-jian.html)

Downloader Middlewares (下载器中间件)

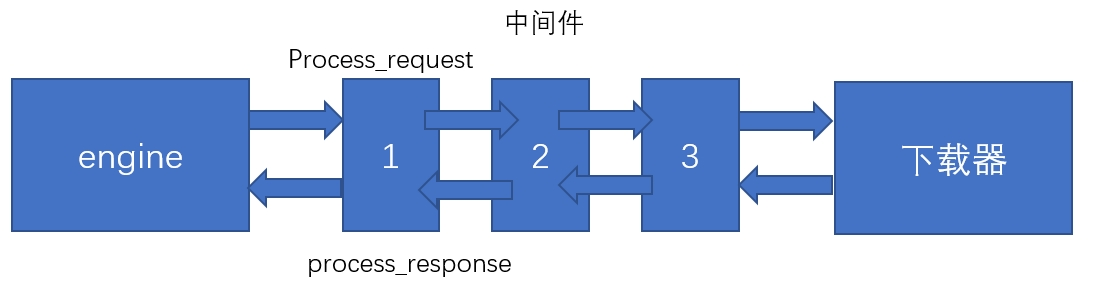
下载器中间件是引擎和下载器之间通信的中间件。在这个中间件中我们可以设置代理、更换请求头等来达到反反爬虫的目的。

Request对象由engine经过下载器中的process\_request方法处理后传递给下载器, Response对象则是由下载器经中间件中的process\_response方法处理后发送给引擎.

要写下载器中间件, 可以在下载器中实现两个方法。一个是process\_request(self,request,spider), 这个方法是在请求发送之前会执行, 是在请求发送给下载器之前对请求进行修改的方法, 如可以添加随机的请求头和随机的代理ip. 还有一个是process\_response(self,request,response,spider), 这个方法是数据下载到引擎之前执行。

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

这个方法是下载器在发送请求之前会执行的。一般可以在这个里面设置随机代理ip等。



1. 参数：
   * request：发送请求的request对象。
   * spider：发送请求的spider对象。
2. 返回值：
   * 返回None：如果返回None, Scrapy将继续处理该request, 把request对象交给其它中间件进行处理, 执行其他中间件中的相应方法, 直到合适的下载器处理函数被调用。
   * 返回Response对象：Scrapy将不会把这个Response对象发送给其他的下载器, 也就不会调用其它下载中间件中的process\_request方法, 而是把这个Response对象直接返回给引擎。已经激活的中间件的process\_response()方法则会在每个response返回时被调用。
   * 返回一个新的Request对象：不再使用之前的request对象去下载数据, 而是根据现在返回的request对象返回数据。会把这个新的Request对象交给下一个中间件进行处理.
   * 如果这个方法中抛出了异常, 则会调用process\_exception方法。

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

这个是下载器下载的数据到引擎中间会执行的方法。

1. 参数：
   * request：request对象。
   * response：被处理的response对象。
   * spider：spider对象。
2. 返回值：
   * 返回Response对象：会将这个新的response对象传给其他中间件, 最终传给爬虫。
   * 返回Request对象：下载器链被切断, 返回的request会重新被下载器调度下载。
   * 如果抛出一个异常, 那么调用request的errback方法, 如果没有指定这个方法, 那么会抛出一个异常。

### 爬虫中间件

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

**import** **random**

**import** **base64**

*# 爬虫中间件*

**class** **UseragentDemoSpiderMiddleware**(object):

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

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

*# passed objects.*

@classmethod

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

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

s = cls()

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

**return** s

*# 把数据交给爬虫处理之前会执行process\_spider\_input这个函数*

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

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

*# middleware and into the spider.*

*# Should return None or raise an exception.*

**return** None

*# 爬虫返回数据之前会执行process\_spider\_output这个函数*

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

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

*# it has processed the response.*

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

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

**yield** i

*# 爬虫发生意外时会执行这个函数*

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

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

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

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

*# or Item objects.*

**pass**

*# 爬虫开始执行之前会执行这个函数*

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

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

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

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

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

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

**yield** r

*# 爬虫开始运行时会执行这个函数*

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

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

"C:\Users\David\Envs\python3\_spider\Lib\site-packages\scrapy\downloadermiddlewares\useragent.py"

*"""Set User-Agent header per spider or use a default value from settings"""*

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

**class** **UserAgentMiddleware**(object):

*"""This middleware allows spiders to override the user\_agent"""*

**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)

下载中间件

**from** **scrapy.downloadermiddlewares.useragent** **import** UserAgentMiddleware

**print**(dir(UserAgentMiddleware))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'from\_crawler', 'process\_request', 'spider\_opened']

**print**(dir(UserAgentMiddleware()))

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'from\_crawler', 'process\_request', 'spider\_opened', 'user\_agent']

### 随机请求头中间件：

爬虫在频繁访问一个页面的时候, 这个请求头如果一直保持一致。那么很容易被服务器发现, 从而禁止掉这个请求头的访问。因此我们要在访问这个页面之前随机的更改请求头, 这样才可以避免爬虫被抓。随机更改请求头, 可以在下载中间件中实现。在请求发送给服务器之前, 随机的选择一个请求头。这样就可以避免总使用一个请求头了。示例代码如下：

class UserAgentDownloadMiddleware(object):

# user-agent随机请求头中间件

USER\_AGENTS = [

'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',

'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',

'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',

'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',

'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'

]

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

user\_agent = random.choice(self.USER\_AGENTS)

request.headers['User-Agent'] = user\_agent

user-agent列表：<http://www.useragentstring.com/pages/useragentstring.php?typ=Browser>

新建scrapy爬虫 httpbin.py

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

**import** **scrapy**

**import** **json**

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

name = 'httpbin'

allowed\_domains = ['httpbin.org']

start\_urls = ['http://httpbin.org/user-agent']

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

*# 返回的是json数据, 使用loads转换为字典, 再从中取出来'user-agent'的值*

user\_agent = json.loads(response.text)['user-agent']

**print**('='\*30)

**print**(user\_agent)

**print**('='\*30)

*# 为了使现象更加明显, 再次使用start\_urls中的url构建新的Request请求对象, 并且设置dont\_filter=True*

**yield** scrapy.Request(self.start\_urls[0], dont\_filter=True)

修改middlewares.py, 定义UserAgentDownloadMiddleware

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

**import** **random**

*# 定义下载中间件*

**class** **UserAgentDownloadMiddleware**(object):

USER\_AGENTS = [

'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',

'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',

'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',

'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',

'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'

]

*# 发送请求之前对Request请求对象进行处理, 设置请求头, 所以要重写process\_request方法*

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

user\_agent = random.choice(self.USER\_AGENTS)

request.headers['User-Agent'] = user\_agent

修改settings.py, 打开自定义的下载中间件, 并设置下载延时. 下载中间件中设置的User-Agent会覆盖掉settings.py中设置的User-Agent.

ROBOTSTXT\_OBEY = False

DOWNLOAD\_DELAY = 1

*# Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36'

}

*# Enable or disable downloader middlewares*

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

DOWNLOADER\_MIDDLEWARES = {

'useragent\_demo.middlewares.UserAgentDownloadMiddleware': 543,

}

### ip代理池中间件

#### 购买代理：

在以下代理商中购买代理：

芝麻代理：<http://www.zhimaruanjian.com/>

太阳代理：<http://www.taiyangruanjian.com/>

快代理：<http://www.kuaidaili.com/>

讯代理：<http://www.xdaili.cn/>

蚂蚁代理：<http://www.mayidaili.com/>

#### 使用ip代理池：

1. 开放代理池设置：

class IPProxyDownloadMiddleware(object):

PROXIES = [

"5.196.189.50:8080",

"134.17.141.44:8080",

"178.49.136.84:8080",

"45.55.132.29:82",

"178.44.185.15:8080"

]

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

# proxy = random.choice(self.PROXIES)

# print('被选中的代理：%s' % proxy)

# request.meta['proxy'] = "http://" + proxy

proxy = "121.199.6.124:16816"

user\_password = "970138074:rcdj35ur"

request.meta['proxy'] = proxy

request.headers['Proxy-Authorization'] = 'Basic ' + base64.b64encode(user\_password.encode('utf-8')).decode('utf-8')

1. 独享代理池设置：

class IPProxyDownloadMiddleware(object):

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

proxy = '121.199.6.124:16816'

user\_password = "970138074:rcdj35ur"

request.meta['proxy'] = proxy

# bytes

b64\_user\_password = base64.b64encode(user\_password.encode('utf-8'))

request.headers['Proxy-Authorization'] = 'Basic ' + b64\_user\_password.decode('utf-8')

视频演示

新建ipproxy.py爬虫文件

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

**import** **scrapy**

**import** **json**

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

name = 'ipproxy'

allowed\_domains = ['httpbin.org']

start\_urls = ['http://httpbin.org/ip']

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

origin = json.loads(response.text)['origin']

**print**('=' \* 30)

**print**(origin)

**print**('=' \* 30)

**yield** scrapy.Request(self.start\_urls[0], dont\_filter=True)

在middlewares.py中添加IPProxyDownloadMiddleware下载中间件

**import** **random**

*# 使用开放代理*

**class** **IPProxyDownloadMiddleware**(object):

PROXIES = ["178.44.170.152:8080", "110.44.113.182:8080", "209.126.124.73:8888", "84.42.79.243:8080", "209.126.124.73:9280", "209.126.124.73:8891", "177.97.31.180:8080", "42.104.84.106:8080", "117.64.234.7:808", "103.76.199.166:8080"]

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

proxy = random.choice(self.PROXIES)

*# 把代理设置到请求头中*

request.meta['proxy'] = proxy

**import** **base64**

*# 使用需要用户名和密码验证的独享代理*

**class** **IPProxyDownloadMiddleware**(object):

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

proxy = '121.199.6.124:16816'

request.meta['proxy'] = proxy

*# 用户名和密码, 也需要使用冒号进行分割*

user\_password = "970138074:rcdj35ur"

*# 需要对用户名和密码进行base64的编码转换. b64encode需要传递bytes的数据类型, 所以需要使用encode来进行编码*

b64\_user\_password = base64.b64encode(user\_password.encode('utf-8'))

*# 设置请求头*

request.headers['Proxy-Authorization'] = 'Basic ' + b64\_user\_password.decode('utf-8')

在settings.py中启动下载中间件

DOWNLOADER\_MIDDLEWARES = {

'useragent\_demo.middlewares.UserAgentDownloadMiddleware': 543,

'useragent\_demo.middlewares.IPProxyDownloadMiddleware': 100,

}

运行爬虫, 进行测试

### 下载中间件实战案例

Boss直聘爬虫, 爬取全国python的招聘信息

新建项目

scrapy startproject boss\_spider

新建爬虫

scrapy genspider -t crawl boss "zhipin.com"

修改items.py, 定义item类

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

**import** **scrapy**

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

name = scrapy.Field()

salary = scrapy.Field()

city = scrapy.Field()

work\_years = scrapy.Field()

education = scrapy.Field()

company = scrapy.Field()

修改boss.py爬虫文件

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

**import** **scrapy**

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

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

**from** **boss.items** **import** BossItem

**class** **ZhipingSpider**(CrawlSpider):

name = 'zhipin'

allowed\_domains = ['zhipin.com']

start\_urls = ['https://www.zhipin.com/c100010000/h\_100010000/?query=python&page=1']

rules = (

*# 匹配职位列表页的规则. 不定义callback, 会默认使用parse函数进行解析*

Rule(LinkExtractor(allow=r'.+\?query=python&page=\d'), follow=True),

*# 匹配职位详情页的规则. 因为符合要求的职位详情页都可以从列表页中提取到, 所以不需要对详情页中符合规则的职位进行跟进. 如果是一个全站的爬虫, 可以设置follow=True*

Rule(LinkExtractor(allow=r'.+job\_detail/\d+.html'),callback="parse\_job", follow=False)

)

*# 解析详情页, 从中提取职位信息. 可以使用scrapy shell来进行测试.*

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

name = response.xpath("//h1[@class='name']/text()").get().strip()

salary = response.xpath("//h1[@class='name']/span/text()").get().strip()

job\_info = response.xpath("//div[@class='job-primary']/div[@class='info-primary']/p//text()").getall()

city = job\_info[0]

work\_years = job\_info[1]

education = job\_info[2]

company = response.xpath("//div[@class='info-company']//a/text()").get()

item = BossItem(name=name,salary=salary,city=city,work\_years=work\_years,education=education,company=company)

**yield** item

修改pipelines.py, 把数据保存到json文件中.

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

**from** **scrapy.exporters** **import** JsonLinesItemExporter

**class** **BossPipeline**(object):

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

*# 以bytes格式写入数据, 要以wb方式打开文件*

self.fp = open('jobs.json','wb')

self.exporter = JsonLinesItemExporter(self.fp,ensure\_ascii=False)

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

self.exporter.export\_item(item)

**return** item

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

self.fp.close()

修改settings.py, 激活BossPipeline

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = False

DOWNLOAD\_DELAY = 1

*# Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

"User-Agent":"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/41.0.2228.0 Safari/537.36"

}

ITEM\_PIPELINES = {

'boss.pipelines.BossPipeline': 300,

}

修改middlewares.py, 添加随机请求头和随机ip地址

## [5.8. settings配置信息](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\7-settingspei-zhi-xin-xi.html)

### 常用的配置项：

1. BOT\_NAME：项目名称。
2. ROBOTSTXT\_OBEY：是否遵守爬虫协议。默认不遵守。
3. CONCURRENT\_ITEMS：代表pipeline同时处理的item数的最大值。默认是100
4. CONCURRENT\_REQUESTS：代表下载器并发请求的最大是, 默认是16。
5. DEFAULT\_REQUEST\_HEADERS：默认请求头。可以将一些不会经常变化的请求头放在这个里面。
6. DEPTH\_LIMIT：爬取网站最大允许的深度。默认为0, 如果为0, 则没有限制。
7. DOWNLOAD\_DELAY：下载器在下载某个页面前等待多长的时间。该选项用来限制爬虫的爬取速度, 减轻服务器压力。同时也支持小数。
8. DOWNLOAD\_TIMEOUT：下载器下载的超时时间。
9. ITEM\_PIPELINES：处理item的Pipeline, 是一个字典, 字典的key这个pipeline所在包的绝对路径, 值是一个整数, 优先级, 值越小, 优先级越高。
10. LOG\_ENABLED：是否启用logging。默认是True。
11. LOG\_ENCODING：log的编码。
12. LOG\_LEVEL：log的级别。默认为DEBUG。可选的级别有CRITICAL、ERROR、WARNING、INFO、DEBUG。
13. USER\_AGENT：请求头。默认为Scrapy/VERSION (+http://scrapy.org)。
14. PROXIES：代理设置。
15. COOKIES\_ENABLED：是否开启cookie。一般不要开启, 避免爬虫被追踪到。如果特殊情况也可以开启。

## [5.9. Scrapy爬虫实战](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-wu-zhang-ff1a-scrapy-kuang-jia\8-scrapypa-chong-shi-zhan.html)

### 糗事百科爬虫。

入门级爬虫。使用Spider。

### 小程序社区爬虫。

数据保存到json文件中。使用CrawlSpider。

### 模拟登录豆瓣网爬虫。

发送post请求模拟登录。

### 图片下载爬虫。

汽车之家宝马5系爬虫。

### BOSS直聘爬虫。

BOSS直聘有很高的反爬虫机制, 只要用同一个ip访问多个职位列表页, 就会被封掉ip。采用代理ip的方式可解决问题。

### 简书网站整站爬虫。

数据保存到mysql数据库中。

将selenium+chromedriver集成到scrapy。

简书整站爬虫

爬取所有文章的内容, 包括标题, 作者, 头像, 发表时间, 字数

阅读量, 评论数量, 喜欢的人数, 以及收录本文章的专题, 推荐阅读都是通过ajax的方式加载的, 所以可以把selinium和chrome driver整合在scrapy项目中

Elements中的内容是在内存中的, 是经过网页渲染后的内容, 整合了网页中所有的图片, js和html的内容. 而查看源码中看到的是服务器对本网页响应的内容, 只包含了本响应的内容, 不包含图片, js动态加载的内容等.

新建crawl spider爬虫

scrapy startproject jianshu\_spider

scrapy genspider -t crawl jianshu "jianshu.com"

修改settings.py, 不遵守robots协议, 修改默认的请求头, 修改下载延迟

修改items.py, 定义要爬取的字段

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

**import** **scrapy**

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

*# 标题*

title = scrapy.Field()

*# 内容*

content = scrapy.Field()

*# 文章id*

article\_id = scrapy.Field()

*# 文章url*

origin\_url = scrapy.Field()

*# 作者*

author = scrapy.Field()

*# 作者头像*

avatar = scrapy.Field()

*# 发表时间*

pub\_time = scrapy.Field()

*# 阅读数*

read\_count = scrapy.Field()

*# 喜欢数*

like\_count = scrapy.Field()

*# 字数*

word\_count = scrapy.Field()

*# 评论数*

comment\_count = scrapy.Field()

*# 所属专题*

subjects = scrapy.Field()

修改jianshu.py, 定义爬虫的内容

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

**import** **scrapy**

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

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

**from** **jianshu\_spider.items** **import** ArticleItem

**class** **JsSpider**(CrawlSpider):

name = 'jianshu'

allowed\_domains = ['jianshu.com']

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

*# 提取出所有文章详情页的url地址, 因为是整站爬虫, 所以需要在文章详情页中对链接进行跟进, 把文章详情页符合规则的url地址也提取出来.*

*# 注意元组中如果有一个元素, 后面必须要有逗号*

rules = (

Rule(LinkExtractor(allow=r'.\*/p/[0-9a-z]{12}.\*'), callback='parse\_detail', follow=True),

)

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

title = response.xpath("//h1[@class='title']/text()").get()

avatar = response.xpath("//a[@class='avatar']/img/@src").get()

author = response.xpath("//span[@class='name']/a/text()").get()

pub\_time = response.xpath("//span[@class='publish-time']/text()").get().replace("\*","")

*# 从url中提取出文章id*

*# https://www.jianshu.com/p/d30d0f91554a?utm\_campaign=maleskine&utm\_content=note&utm\_medium=pc\_all\_hots&utm\_source=recommendation*

*# https://www.jianshu.com/p/d30d0f91554a*

url = response.url

*# ['https://www.jianshu.com/p/d30d0f91554a','utm\_campaign=maleskine&utm\_content=note&utm\_medium=pc\_all\_hots&utm\_source=recommendation']*

*# ['https://www.jianshu.com/p/d30d0f91554a']*

*# 以问号对url进行分隔, 在正常的url中, 最多只有一个问号. 后面跟参数.*

url1 = url.split("?")[0]

article\_id = url1.split('/')[-1]

*# 爬取到content即网站的内容后, 一个需求可能是把爬取到的内容放到自己的网站中. 所以文章内容中的所有标签都要保留. 所以这里不要再去调用//text(), 否则会把所有的标签都删除, 只留下文字内容.*

content = response.xpath("//div[@class='show-content']").get()

item = ArticleItem(

title=title,

avatar=avatar,

author = author,

pub\_time = pub\_time,

origin\_url = response.url,

article\_id = article\_id,

content = content,

)

**yield** item

创建数据库 jianshu

创建数据表, article, 字段id int not null primary key auto increament, title varchar 255, content varchar longtext, author varchar 255, avatar varchar 255, pub\_time datetime, article\_id varchar 20, arigin\_url varchar 255

修改pipelines.py, 把爬取到的数据保存到数据库中

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

**import** **pymysql**

**from** **twisted.enterprise** **import** adbapi

**from** **pymysql** **import** cursors

**class** **JianshuSpiderPipeline**(object):

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

dbparams = {

'host': '127.0.0.1',

'port': 3306,

'user': 'root',

'password': 'root',

'database': 'jianshu',

*# 注意mysql中所有相关的编码都是utf8, 不是utf-8*

'charset': 'utf8'

}

*# 创建连接对象. 查看connect的源码, 确定需要传递的参数. \*\*dbparams会把字典中的key和value作为关键字参数传递过去.*

self.conn = pymysql.connect(\*\*dbparams)

*# 创建cursor游标对象*

self.cursor = self.conn.cursor()

self.\_sql = None

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

*# 把item中的内容通过sql语句保存到mysql中.*

self.cursor.execute(self.sql,(item['title'],item['content'],item['author'],item['avatar'],item['pub\_time'],item['origin\_url'],item['article\_id']))

self.conn.commit()

**return** item

@property

**def** sql(self):

*# 如果之前没有生成过sql语句, 就使用self.\_sql生成sql语句*

**if** **not** self.\_sql:

*# id是自动增加的, 所以这里可以写为null*

self.\_sql = """

insert into article(id,title,content,author,avatar,pub\_time,origin\_url,article\_id) values(null,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**)

"""

**return** self.\_sql

*# 如果已经生成了sql语句, 就直接使用self.\_sql*

**return** self.\_sql

修改settings.py, 启用JianshuSpiderPipeline'

ITEM\_PIPELINES = {

'jianshu\_spider.pipelines.JianshuSpiderPipeline': 300,

}

运行爬虫, 就可以看到数据会逐渐保存到数据库中.

在pipeline中, 把数据保存到mysql中的操作是同步的, io操作一般都比较费时, 效率会比较低, 可以把保存到mysql中的操作修改为异步的, 这样就能大大提高保存数据的效率.

修改pipelines.py, 使用Twisted中的方法, 把数据异步保存到数据库中

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

**import** **pymysql**

*# adbapi是专用用于做数据库处理的模块, 在其中创建了一个连接池, 使用连接池进行数据的保存*

**from** **twisted.enterprise** **import** adbapi

**from** **pymysql** **import** cursors

**class** **JianshuSpiderPipeline**(object):

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

dbparams = {

'host': '127.0.0.1',

'port': 3306,

'user': 'root',

'password': 'root',

'database': 'jianshu',

*# 注意mysql中所有相关的编码都是utf8, 不是utf-8*

'charset': 'utf8'

}

*# 创建连接对象. 查看connect的源码, 确定需要传递的参数. \*\*dbparams会把字典中的key和value作为关键字参数传递过去.*

self.conn = pymysql.connect(\*\*dbparams)

*# 创建cursor游标对象*

self.cursor = self.conn.cursor()

self.\_sql = None

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

*# 把item中的内容通过sql语句保存到mysql中.*

self.cursor.execute(self.sql,(item['title'],item['content'],item['author'],item['avatar'],item['pub\_time'],item['origin\_url'],item['article\_id']))

self.conn.commit()

**return** item

@property

**def** sql(self):

*# 如果之前没有生成过sql语句, 就使用self.\_sql生成sql语句*

**if** **not** self.\_sql:

*# id是自动增加的, 所以这里可以写为null*

self.\_sql = """

insert into article(id,title,content,author,avatar,pub\_time,origin\_url,article\_id) values(null,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**)

"""

**return** self.\_sql

*# 如果已经生成了sql语句, 就直接使用self.\_sql*

**return** self.\_sql

**class** **JianshuTwistedPipeline**(object):

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

dbparams = {

'host': '127.0.0.1',

'port': 3306,

'user': 'root',

'password': 'root',

'database': 'jianshu2',

'charset': 'utf8',

*# 还需要传入cursorclass这个参数, 即指定使用的cursor的类*

'cursorclass': cursors.DictCursor

}

*# 创建一个连接池对象 def \_\_init\_\_(self, dbapiName, \*connargs, \*\*connkw). 这里的dbapiName要传入数据库连接的模块, twisted框架会在底层加载pymysql这个模块.*

self.dbpool = adbapi.ConnectionPool('pymysql',\*\*dbparams)

self.\_sql = None

@property

**def** sql(self):

**if** **not** self.\_sql:

self.\_sql = """

insert into article(id,title,content,author,avatar,pub\_time,origin\_url,article\_id) values(null,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**,**%s**)

"""

**return** self.\_sql

**return** self.\_sql

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

*# 把数据插入到数据库中*

*# 这里如果直接调用cursor.execute来插入数据, 实现上与上面同步插入数据的方法是一样的, 使用dbpool.runInteraction调用插入数据的方法, 就会把插入数据库的操作变成是异步的.*

*# 所以这里就要使用dbpool中的方法来实现了. 在dbpool.runInteraction中调用self.insert\_item这个函数, 所以向数据库中插入数据实际上是在insert\_item这个方法中完成的.*

defer = self.dbpool.runInteraction(self.insert\_item,item)

*# 添加错误处理方法, 如果发生了错误, 就会调用handle\_error()方法进行错误处理.*

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

**def** insert\_item(self,cursor,item):

cursor.execute(self.sql,(item['title'],item['content'],item['author'],item['avatar'],item['pub\_time'],item['origin\_url'],item['article\_id']))

**def** handle\_error(self,error,item,spider):

**print**('='\*10+"error"+'='\*10)

**print**(error)

**print**('='\*10+"error"+'='\*10)

修改settings.py, 启动新的JianshuTwistedPipeline

ITEM\_PIPELINES = {

*# 'jianshu\_spider.pipelines.JianshuSpiderPipeline': 300,*

'jianshu\_spider.pipelines.JianshuTwistedPipeline': 300,

}

以上爬虫有一个问题, 如果首页中的文章爬完, 文章详情页中的推荐文章爬取完后, 就不会再爬取到更多的内容了. 所以要对ajax加载出来的内容进行爬取.

推荐阅读中的内容, 除了第1个之外, 其它的都是使用ajax方式加载出来的, 专题中的内容, 也是通过ajax加载出来的.

可以通过分析ajax请求找到对应的响应, 也可以使用selenium的方式来获取.

修改数据库, 添加几个字段

阅读量

read\_count int default 0,

喜欢量

like\_count int default 0,

字数

word\_count int default 0

专题

subjects text

在middlewares.py中添加下载中间件, 使用selenium提取数据.

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

*# Define here the models for your spider middleware*

*#*

*# See documentation in:*

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

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

**from** **selenium** **import** webdriver

**import** **time**

**from** **scrapy.http.response.html** **import** HtmlResponse

**class** **SeleniumDownloadMiddleware**(object):

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

self.driver = webdriver.Chrome(executable\_path=r"D:\ProgramApp\chromedriver\chromedriver.exe")

*# 如果process\_request中返回的是response对象, scrapy就不会把请求发送给下载器下载, 而是直接把response对象发送给爬虫进行解析. 可以在process\_request中捕获request, 使其不经过scrapy发送下载请求, 而是使selenium来发送请求, 然后使用return把对应响应的内容返回给spider进行解析. 这样就把selenium集成到了scrapy中, 使用scrapy对请求进行去重, 对数据进行解析和保存, 而使用selenium代替scrapy框架来发送请求**, 即把scrapy框架流程图中的downloadmiddleware替换为自定义的, 其它组件依旧使用scrapy自身的..*

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

self.driver.get(request.url)

*# 等待1s, 以免数据还未加载完成就返回网页源码*

time.sleep(1)

*# 点击获取帖子所属的专题, 循环进行点击, 直到没有找到show-more这个元素.*

**try**:

**while** True:

showMore = self.driver.find\_element\_by\_class\_name('show-more')

showMore.click()

time.sleep(0.3)

**if** **not** showMore:

**break**

**except**:

**pass**

*# 提取出网页源码, 构造Response响应对象并返回*

source = self.driver.page\_source

*# 查看HtmlResponse的源码确定需要传递的参数.*

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

response = HtmlResponse(url=self.driver.current\_url, body=source, request=request, encoding='utf-8')

**return** response

修改settings.py, 启动下载中间件

DOWNLOADER\_MIDDLEWARES = {

'jianshu\_spider.middlewares.SeleniumDownloadMiddleware': 543,

}

修改jianshu.py爬虫文件, 提取其它字段的数据.

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

**import** **scrapy**

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

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

**from** **jianshu\_spider.items** **import** ArticleItem

**class** **JsSpider**(CrawlSpider):

name = 'js'

allowed\_domains = ['jianshu.com']

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

*# 提取出所有文章详情页的url地址, 因为是整站爬虫, 所以需要在文章详情页中对链接进行跟进, 把文章详情页符合规则的url地址也提取出来.*

*# 注意元组中如果有一个元素, 后面必须要有逗号*

rules = (

Rule(LinkExtractor(allow=r'.\*/p/[0-9a-z]{12}.\*'), callback='parse\_detail', follow=True),

)

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

title = response.xpath("//h1[@class='title']/text()").get()

avatar = response.xpath("//a[@class='avatar']/img/@src").get()

author = response.xpath("//span[@class='name']/a/text()").get()

pub\_time = response.xpath("//span[@class='publish-time']/text()").get().replace("\*","")

*# 从url中提取出文章id*

*# https://www.jianshu.com/p/d30d0f91554a?utm\_campaign=maleskine&utm\_content=note&utm\_medium=pc\_all\_hots&utm\_source=recommendation*

*# https://www.jianshu.com/p/d30d0f91554a*

url = response.url

*# ['https://www.jianshu.com/p/d30d0f91554a','utm\_campaign=maleskine&utm\_content=note&utm\_medium=pc\_all\_hots&utm\_source=recommendation']*

*# ['https://www.jianshu.com/p/d30d0f91554a']*

*# 以问号对url进行分隔, 在正常的url中, 最多只有一个问号. 后面跟参数.*

url1 = url.split("?")[0]

article\_id = url1.split('/')[-1]

*# 爬取到content即网站的内容后, 一个需求可能是把爬取到的内容放到自己的网站中. 所以文章内容中的所有标签都要保留. 所以这里不要再去调用//text(), 否则会把所有的标签都删除, 只留下文字内容.*

content = response.xpath("//div[@class='show-content']").get()

word\_count = response.xpath("//span[@class='wordage']/text()").get()

comment\_count = response.xpath("//span[@class='comments-count']/text()").get()

read\_count = response.xpath("//span[@class='views-count']/text()").get()

like\_count = response.xpath("//span[@class='likes-count']/text()").get()

*# 提取到的专题是一个列表, 把列表传递给mysql时, mysql会把列表转换为字符串型的列表. 如[1,2,3]转换为'[1,2,3]', 所以这里需要手动把列表转换为字符串*

subjects = ",".join(response.xpath("//div[@class='include-collection']/a/div/text()").getall())

item = ArticleItem(

title=title,

avatar=avatar,

author = author,

pub\_time = pub\_time,

origin\_url = response.url,

article\_id = article\_id,

content = content,

subjects = subjects,

word\_count = word\_count,

comment\_count = comment\_count,

read\_count = read\_count,

like\_count = like\_count

)

**yield** item

清空mysql数据库中的内容, 再次运行爬虫, 就会看到打开了一个新的网页来执行.

# [6. 第六章：Scrapy-Redis分布式组件](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-liu-zhang-ff1a-scrapy-redis-fen-bu-shi-zu-jian.html)

## [6.1. redis数据库介绍](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-liu-zhang-ff1a-scrapy-redis-fen-bu-shi-zu-jian\1-redisshu-ju-ku-jie-shao.html)

### 概述

redis是一种支持分布式的nosql数据库,他的数据是保存在内存中, 同时redis可以定时把内存数据同步到磁盘, 即可以将数据持久化, 并且他比memcached支持更多的数据结构(string,list列表[队列和栈],set[集合],sorted set[有序集合],hash(hash表))。相关参考文档：<http://redisdoc.com/index.html>

### redis使用场景：

一般用到web服务器的后台中.

1. 登录会话存储：存储在redis中, 与memcached相比, 数据不会丢失。
2. 排行版/计数器：比如一些秀场类的项目, 经常会有一些前多少名的主播排名。还有一些文章阅读量的技术, 或者新浪微博的点赞数等。
3. 作为消息队列：比如celery就是使用redis作为中间人。
4. 当前在线人数：还是之前的秀场例子, 会显示当前系统有多少在线人数。
5. 一些常用的数据缓存：比如我们的BBS论坛, 板块不会经常变化的, 但是每次访问首页都要从mysql中获取, 可以在redis中缓存起来, 不用每次请求数据库。
6. 把前200篇文章缓存或者评论缓存：一般用户浏览网站, 只会浏览前面一部分文章或者评论, 那么可以把前面200篇文章和对应的评论缓存起来。用户访问超过的, 就访问数据库, 并且以后文章超过200篇, 则把之前的文章删除。
7. 好友关系：微博的好友关系使用redis实现。
8. 发布和订阅功能：可以用来做聊天软件。

### redis和memcached的比较：

|  |  |  |
| --- | --- | --- |
|  | memcached | redis |
| 类型 | 纯内存数据库 | 内存磁盘同步数据库 |
| 数据类型 | 在定义value时就要固定数据类型 | 不需要 |
| 虚拟内存 | 不支持 | 支持 |
| 过期策略 | 支持 | 支持 |
| 存储数据安全 | 不支持 | 可以将数据同步到dump.db中 |
| 灾难恢复 | 不支持 | 可以将磁盘中的数据恢复到内存中 |
| 分布式 | 支持 | 主从同步 |
| 订阅与发布 | 不支持 | 支持 |

### redis在ubuntu系统中的安装与启动

1. 安装：

sudo apt-get install redis-server

1. 卸载：

sudo apt-get purge --auto-remove redis-server

1. 启动：redis安装后, 默认会自动启动, 可以通过以下命令查看：

ps aux|grep redis

如果想自己手动启动, 可以通过以下命令进行启动：

sudo service redis-server start

1. 停止：

sudo service redis-server stop

### redis在windows系统中的安装与启动：

1. 下载：redis官方是不支持windows操作系统的。但是微软的开源部门将redis移植到了windows上。因此下载地址不是在redis官网上。而是在github上：<https://github.com/MicrosoftArchive/redis/releases。>
2. 安装：点击一顿下一步安装就可以了。
3. 运行：进入到redis安装所在的路径然后执行redis-server.exe redis.windows.conf就可以运行了。

cd "C:\Program Files\Redis"

redis-server.exe redis.windows.conf

1. 连接：redis和mysql以及mongo是一样的, 都提供了一个客户端进行连接。输入命令redis-cli (前提是redis安装路径已经加入到环境变量中了) 就可以连接到redis服务器了。

redis-cli -h 192.168.1.8 -p 6379

连接本机的redis时可以省略ip地址和端口

其中192.168.1.8是运行redis服务器的主机的ip地址.

get username

会出现一个错误, 表示没有真正连接上redis服务器

quit退出

### 其他机器访问本机redis服务器：

想要让其他机器访问本机的redis服务器。那么要修改redis.conf的配置文件, 将bind改成bind [自己的ip地址或者0.0.0.0], 其他机器才能访问。

**注意：bind绑定的是本机网卡的ip地址, 而不是想让其他机器连接的ip地址。如果有多块网卡, 那么可以绑定多个网卡的ip地址。如果绑定到ip地址是0.0.0.0, 那么意味着其他机器可以通过本机所有的ip地址进行访问。**

**绑定的是redis服务器的ip地址, 不是redis客户端的ip地址.**

先退出 redis-server服务器, 再修改redis.conf配置文件

在windows下, 使用的配置文件是redis.windows.conf, 如果运行redis服务器的电脑的ip地址是192.168.1.8, 要把bind修改为

bind 192.168.1.8

这样就能远程通过 redis-cli -h 192.168.1.8 -p 6379命令来连接redis服务器了.

重新启动redis服务器, 在远程redis-cli中连接redis服务器即可. 如果能够从reids中读取数据, 或者向redis中写入数据, 就表示连接成功了.

set username xxx

get username

### 对redis的操作

对redis的操作可以用两种方式, 第一种方式采用redis-cli, 第二种方式采用编程语言, 比如Python、PHP和JAVA等。

#### 使用redis-cli对redis进行字符串操作：

1. 启动redis：

sudo service redis-server start

2. 连接上redis-server：

redis-cli -h [ip] -p [端口]

3. 添加key value：

set key value

如：

set username xiaotuo

将字符串值value关联到key。如果key已经持有其他值, set命令就覆写旧值, 无视其类型。并且默认的过期时间是永久, 即永远不会过期。

4. 获取key对应的value

get key username

5. 删除：

del key

如：

del username

6. 设置过期时间：

过期后这个key就会自动从redis中删除

expire key timeout(单位为秒)

set username zhangsan

expire name 20

get name

也可以在设置值的时候, 一同指定过期时间：

set key value EX timeout

或：

setex key timeout value

如:

set age 18 EX 60

7. 查看过期时间：

ttl key

如：

ttl username

8. 查看当前redis中的所有key：

keys \*

#### 列表操作：

1. 向列表中添加元素

1.1 在列表左边添加元素：

lpush key value

将值value插入到列表key的表头。如果key不存在, 一个空列表会被创建并执行lpush操作。当key存在但不是列表类型时, 将返回一个错误。

如:

lpush websites baidu.com

1.2 在列表右边添加元素：

rpush key value

将值value插入到列表key的表尾。如果key不存在, 一个空列表会被创建并执行RPUSH操作。当key存在但不是列表类型时, 返回一个错误。

如:

rpush websites google.com

2. 查看列表中的元素：

lrange key start stop

返回列表key中指定区间内的元素, 区间以偏移量start和stop指定,如果要左边的第一个到最后的一个lrange key 0 -1。

这里lrange中的l代表的不是left, 而是list, 即列表的操作

如:

lrange websites 0 -1 # 获取第1个到最后1个元素

lpush websites 163.com

rpush websites qq.com

lrange websites 0 1

lrange websites 0 -1

3. 移除列表中的元素：

3.1 移除并返回列表key的头元素：

从左边移除第0个元素

lpop key

如:

lrange websites 0 -1

lpop websites

lrange websites 0 -1

3.2 移除并返回列表的尾元素：

rpop key

如:

lrange websites 0 -1

rpop websites

lrange websites 0 -1

3.3 移除并返回列表key的中间元素：

lrem key count value

将删除key这个列表中, count个值为value的元素。

如:

lpush websites baidu.com

rpush websites baidu.com

lrange websites 0 -1

lrem websites 2 baiduc.om # 删除websites这个列表中 2 个值为baidu.com的元素

lrange websites 0 -1

根据参数 count 的值, 移除列表中与参数 value 相等的元素。count的值可以是以下几种：

count > 0：从表头开始向表尾搜索, 移除与value相等的元素, 数量为count。如lrem websites 2 baidu.com, 从列表左边向右搜索, 移除2个值为baidu.com的元素.

count < 0：从表尾开始向表头搜索, 移除与 value相等的元素, 数量为count的绝对值。如lrem websites -2 baidu.com, 从列表右边向左搜索, 移除2个值为baidu.com的元素

count = 0：移除表中所有与value 相等的值。

4. 返回指定位置上的元素：

lindex key index

将返回key这个列表中, 索引为index的这个元素。

lindex websites 1 # 返回列表中的第1个元素

5. 获取列表中的元素个数：

llen key

如：

llen languages

llen websites

6. 删除指定的元素：

lrem key count value

如：

lrem languages 0 php

#### set集合的操作：

与python中的集合类似. redis中的集合与列表的不同之处, 一是集合是无序的, 列表是有序的. 二是集合中的元素是唯一的.

1. 添加元素：

sadd set value1 value2....

如：

sadd team xiaotuo datuo

2. 查看元素：

smembers set

如：

smembers team

3. 移除元素：

srem set member...

如：

srem team xiaotuo

smembers team

4. 查看集合中的元素个数：

scard set

如：

scard team

5. 获取多个集合的交集：

提取两个集合的交集, 并生成一个新的集合

sinter set1 set2

如：

sadd team1 yoaming kebi zhangsan

sadd team2 yaoming zhaosi wangwu

sinter team1 team2

6. 获取多个集合的并集：

所有元素放在一起, 并且删除重复的元素

sunion set1 set2

如：

sadd team1 yoaming kebi zhangsan

sadd team2 yaoming zhaosi wangwu

sunion team1 team2

7. 获取多个集合的差集：

sdiff set1 set2

如：

sdiff team1 team2

#### hash哈希操作, 即字典操作：

列表, 集合, 哈希操作都相似, 向一个列表/集合/哈希中添加值, 如果这个列表/集合/哈希不存在, 会自动创建.

1. hset添加一个新值：

hset key field value

如：

hset website baidu baidu.com

hset website google google.com

将哈希表key中的域field的值设为value。

如果key不存在, 一个新的哈希表被创建并进行 HSET操作。如果域 field已经存在于哈希表中, 旧值将被覆盖。

2. hget获取哈希中的field对应的值：

hget key field

如：

hget website baidu

3. hgetall获取某个哈希中所有的field和value：

hset key

如：

hgetall website

4. hkeys获取某个哈希中所有的field：

hkeys key

如：

hkeys website

5. hvals获取某个哈希中所有的值：

hvals key

如：

hvals website

6. hdel删除field中的某个field：

hdel key field

如：

hdel website baidu

7. hexists判断哈希中是否存在某个field：

返回0 表示不存在, 返回1 表示存在

hexists key field

如：

hexists website baidu

8. hlen获取哈希中总共的键值对：

hlen field

如：

hlen website

1. 事务操作：

Redis事务可以一次执行多个命令, 事务具有以下特征：

* + 隔离操作：事务中的所有命令都会序列化、按顺序地执行, 不会被其他命令打扰。
  + 原子操作：事务中的命令要么全部被执行, 要么全部都不执行。

1. 开启一个事务：

multi

以后执行的所有命令, 都在这个事务中执行的。

2. 执行事务：

exec

会将在multi和exec中的操作一并提交。

3. 取消事务：

discard

会将multi后的所有命令取消。

4. 监视一个或者多个key：

watch key...

监视一个(或多个)key, 如果在事务执行之前这个(或这些) key被其他命令所改动, 那么事务将被打断。

5. 取消所有key的监视：

unwatch

6. 发布/订阅操作：

给某个频道发布消息：

publish channel message

7. 订阅某个频道的消息：

subscribe channel

## [6.2. Scrapy-Redis组件介绍](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-liu-zhang-ff1a-scrapy-redis-fen-bu-shi-zu-jian\2-scrapy-rediszu-jian-jie-shao.html)

Scrapy是一个框架, 他本身是不支持分布式的。如果我们想要做分布式的爬虫, 就需要借助一个组件叫做Scrapy-Redis, 这个组件正是利用了Redis可以分布式的功能, 集成到Scrapy框架中, 使得爬虫可以进行分布式。可以充分的利用资源 (多个ip、更多带宽、同步爬取) 来提高爬虫的爬行效率。

### 分布式爬虫的优点：

1. 可以充分利用多台机器的带宽。
2. 可以充分利用多台机器的ip地址。
3. 多台机器做, 爬取效率更高。

### 分布式爬虫必须要解决的问题：

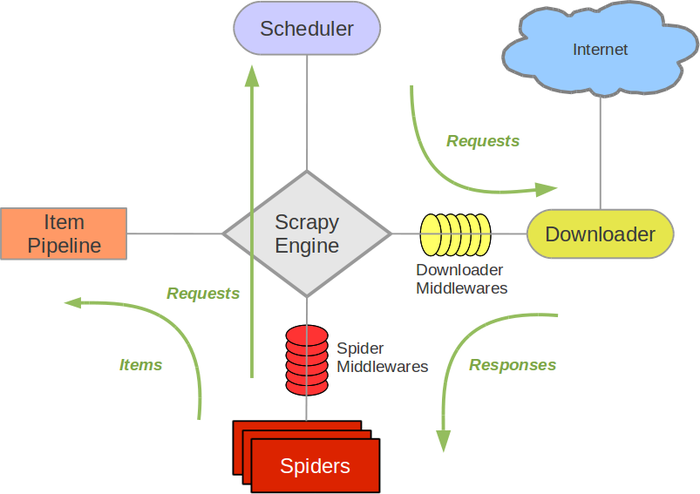
1. 分布式爬虫是好几台机器在同时运行, 如何保证不同的机器爬取页面的时候不会出现重复爬取的问题。使用redis去重.
2. 同样, 分布式爬虫在不同的机器上运行, 在把数据爬完后如何保证保存在同一个地方。把数据保存到redis中.

### 安装：

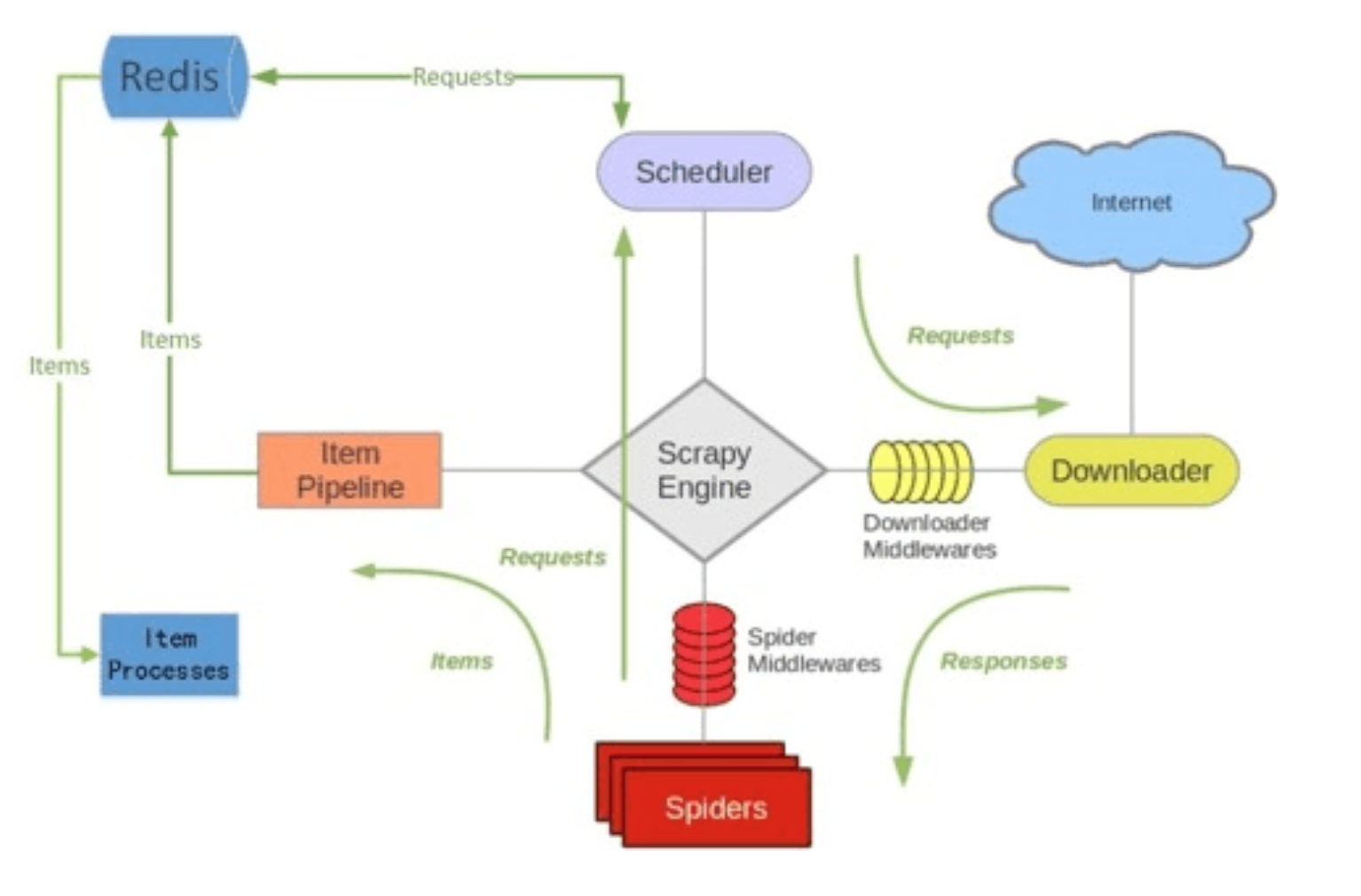
通过pip install scrapy-redis即可安装。

### Scrapy-Redis架构：

Scrapy架构图：

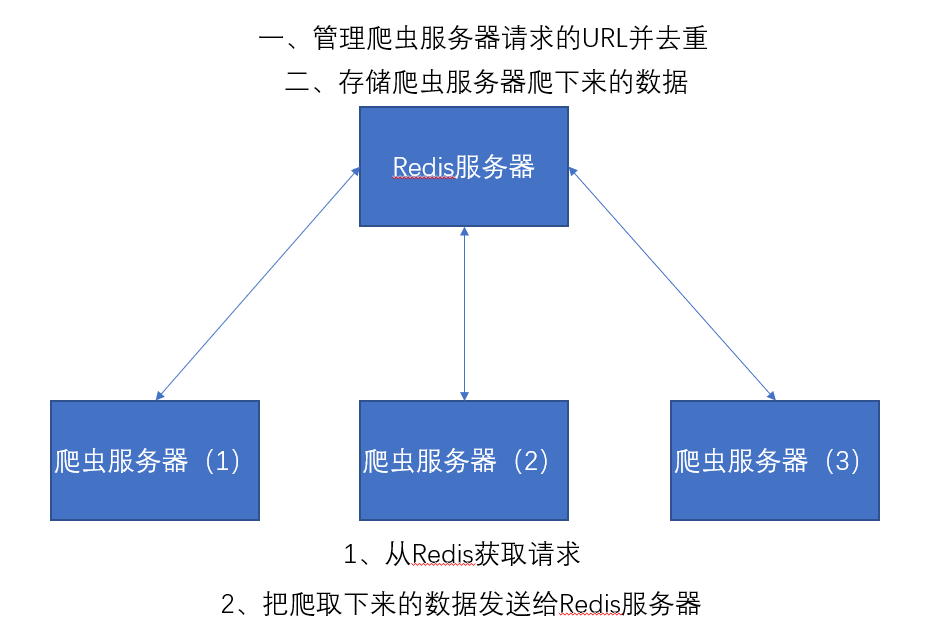


Scrapy-Redis架构图：



scheduler把request对象保存到redis中, 每次发送request请求前都去redis中进行查找, 如果之前没有爬取过, 就把该request对象返回给scheduler, 然后下行爬取, 爬取到的数据也经过item pipeline保存到redis数据库中.

分布式爬虫架构图：



以上两个图片对比我们可以发现。Item Pipeline在接收到数据后发送给了Redis、Scheduler调度器调度数据也是从Redis中来的、并且其实数据去重也是在Redis中做的。

### 编写Scrapy-Redis分布式爬虫：

要将一个Scrapy项目变成一个Scrapy-redis项目只需修改以下三点就可以了：

1. 将爬虫的类从scrapy.Spider变成scrapy\_redis.spiders.RedisSpider；或者是从scrapy.CrawlSpider变成scrapy\_redis.spiders.RedisCrawlSpider。
2. 将爬虫中的start\_urls删掉。增加一个redis\_key="xxx"。这个redis\_key是为了以后在redis中控制爬虫启动的。爬虫的第一个url, 就是在redis中通过这个发送出去的。
3. 在配置文件中增加如下配置：

# Scrapy-Redis相关配置

# 确保request存储到redis中

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# 确保所有爬虫共享相同的去重指纹

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

# 设置redis为item pipeline

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

# 在redis中保持scrapy-redis用到的队列, 不会清理redis中的队列, 从而可以实现暂停和恢复的功能。

SCHEDULER\_PERSIST = True

# 设置连接redis信息

REDIS\_HOST = '127.0.0.1'

REDIS\_PORT = 6379

1. 运行爬虫：
   1. 在爬虫服务器上。进入爬虫文件所在的路径, 然后输入命令：scrapy runspider [爬虫名字]。
   2. 在Redis服务器上, 推入一个开始的url链接：redis-cli> lpush [redis\_key] start\_url开始爬取。

## [6.3. 3-搜房网分布式爬虫](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\00%20%E7%88%AC%E8%99%AB%E4%B8%93%E9%A2%98\03_%E4%BA%BA%E5%B7%A5%E6%99%BA%E8%83%BD%E7%88%AC%E8%99%AB\python_crawler_web\di-liu-zhang-ff1a-scrapy-redis-fen-bu-shi-zu-jian\3sou-fang-wang-fen-bu-shi-pa-chong.html)

首先实现单机爬虫, 然后把单机爬虫修改为分布式爬虫

爬取全国各个城市中新房和二手房的信息

1. 获取所有的城市的url链接。

http://www.fang.com/SoufunFamily.htm

2. 获取所有城市的新房的url链接。

例：安庆：http://anqing.fang.com/

安庆新房：http://newhouse.anqing.fang.com/house/s/

3. 获取所有城市的二手房的url链接。

例：安庆：http://anqing.fang.com/

安庆二手房：http://esf.anqing.fang.com/

北京是个例外：

北京的新房链接：http://newhouse.fang.com/house/s/

北京的二手房链接：http://esf.fang.com/

获取所有城市的链接 > 获取所有城市新房/二手房的链接 >

按省份进行选择, 获取某个城市时保存对应的省的信息

每个tr是一行

tbody>tr>td[2]中是省份的信息,

tbody>tr>td[3]中是省份中城市的信息, 获取城市名称及链接

一个省的城市可以包含多行

某个省份第一行中td[2]是省份的信息, td[3]中是城市的信息

某个省份第二行开始td[2]中是空白内容 "&nbsp;", td[3]中是城市的信息

新建项目 scrapy startproject fang\_pro

新建爬虫 scrapy genspider fang

修改settings.py, 进行基本的配置

ROBOTSTXT\_OBEY = False

DOWNLOAD\_DELAY = 3

*# Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

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

'Accept-Language': 'en',

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.93 Safari/537.36'

}

修改items.py, 定义要爬取的字段

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

**import** **scrapy**

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

*# 省份*

province = scrapy.Field()

*# 城市*

city = scrapy.Field()

*# 小区的名字*

name = scrapy.Field()

*# 价格*

price = scrapy.Field()

*# 几居。这个是个列表*

rooms = scrapy.Field()

*# 面积*

area = scrapy.Field()

*# 地址*

address = scrapy.Field()

*# 行政区*

district = scrapy.Field()

*# 是否在售*

sale = scrapy.Field()

*# 房天下详情页面的url*

origin\_url = scrapy.Field()

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

*# 省份*

province = scrapy.Field()

*# 城市*

city = scrapy.Field()

*# 小区的名字*

name = scrapy.Field()

*# 几室几厅*

rooms = scrapy.Field()

*# 层*

floor = scrapy.Field()

*# 朝向*

toward = scrapy.Field()

*# 年代*

year = scrapy.Field()

*# 地址*

address = scrapy.Field()

*# 建筑面积*

area = scrapy.Field()

*# 总价*

price = scrapy.Field()

*# 单价*

unit = scrapy.Field()

*# 原始的url*

origin\_url = scrapy.Field()

修改fang.py爬虫文件

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

**import** **scrapy**

**import** **re**

**from** **fang.items** **import** NewHouseItem, ESFHouseItem

**from** **scrapy\_redis.spiders** **import** RedisSpider

**class** **SfwSpider**(RedisSpider):

name = 'fang'

allowed\_domains = ['fang.com']

start\_urls = ['http://www.fang.com/SoufunFamily.htm']

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

*# 提取出所有的tr标签, 每个tr标签是一行城市的信息*

trs = response.xpath("//div[@class='outCont']//tr")

province = None

**for** tr **in** trs:

*# 每个tr中都有3个td标签, 第1个td标签是不需要的, 它有class的属性, 后2个td标签是我们需要的, 它们都没有class属性. 使用td[not(@class)]来选择不包含class的属性*

tds = tr.xpath(".//td[not(@class)]")

*# 包含省份信息的td标签*

province\_td = tds[0]

province\_text = province\_td.xpath(".//text()").get()

*# 替换掉province\_text中的空格, 如果去除空格后得到的province\_text不为空值, 说明它是省份的信息*

province\_text = re.sub(r"\s", "", province\_text)

**if** province\_text:

*# 先在循环外定义一个province的变量, 如果在一个tr中提取到了省份的信息, 就把province的值修改为提取到的省份, 对于下一次循环的tr, 如果没有提取到省份的信息, 表明这一行tr中的城市是上一行tr中省份中的城市, 此时不对province重新进行赋值, province的值还是上一次tr循环中提取出来的省份, 就使用上一个tr中的省份*

province = province\_text

*# 不爬取海外的城市的房源*

**if** province == '其它':

**continue**

*# 提取出城市的信息*

city\_td = tds[1]

city\_links = city\_td.xpath(".//a")

**for** city\_link **in** city\_links:

city = city\_link.xpath(".//text()").get()

city\_url = city\_link.xpath(".//@href").get()

*# print("省份: ", province)*

*# print("城市: ", city)*

*# print("城市链接: ", city\_link)*

*# 从city\_url构建新房和二手房 url*

*# "http://anqing.fang.com/" 分割后第0个元素是域名, 第1个元素是网址*

url\_module = city\_url.split("//")

scheme = url\_module[0]

domain = url\_module[1]

**if** 'bj.' **in** domain:

newhouse\_url = 'http://newhouse.fang.com/house/s/'

esf\_url = 'http://esf.fang.com/'

**else**:

*# 构建新房的url链接, "http://newhouse.anqing.fang.com/house/s/"*

newhouse\_url = scheme + '//' + "newhouse." + domain + "house/s/"

*# 构建二手房的url链接, "http://esf.anqing.fang.com/"*

esf\_url = scheme + "//" + "esf." + domain

*# print('省份: %s\t城市: %s' % (province, city))*

*# print('新房url: %s' % newhouse\_url)*

*# print('二手房url: %s' % esf\_url)*

*# 构造新房的Request对象. 在写二手房的爬虫时可以把新房的Request注释掉*

**yield** scrapy.Request(url=newhouse\_url,callback=self.parse\_newhouse,meta={"info":(province,city)})

**yield** scrapy.Request(url=esf\_url, callback=self.parse\_esf, meta={"info": (province, city)})

*# 测试代码, 只爬取1个城市*

*# break*

*# 测试代码, 只爬取一行城市数据*

*# break*

*# 解析新房的响应*

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

province, city = response.meta.get('info')

*# 每一个li标签都是一个小区的数据*

lis = response.xpath("//div[contains(@class,'nl\_con')]/ul/li")

**for** li **in** lis:

*# 小区名称*

name = li.xpath(".//div[@class='nlcd\_name']/a/text()").get().strip()

*# 小区房型列表*

house\_type\_list = li.xpath(".//div[contains(@class,'house\_type')]/a/text()").getall()

house\_type\_list = list(map(**lambda** x: re.sub(r"\s", "", x), house\_type\_list))

house\_type\_list = [re.sub(r"\s", "", x) **for** x **in** house\_type\_list]

*# 几居室列表*

rooms = list(filter(**lambda** x: x.endswith("居"), house\_type\_list))

rooms = [x **for** x **in** house\_type\_list **if** x.endswith('居')]

*# 小区房屋的面积*

area = "".join(li.xpath(".//div[contains(@class,'house\_type')]/text()").getall())

area = re.sub(r"\s|－|/", "", area)

*# 地址*

address = li.xpath(".//div[@class='address']/a/@title").get()

*# 小区所在行政区, 注意不能用span标签进行提取, 因为不是所有的行政区信息都放在span标签中.*

district\_text = "".join(li.xpath(".//div[@class='address']/a//text()").getall())

district = re.search(r".\*\[(.+)\].\*", district\_text).group(1)

*# 销售状态是在售还是待售*

sale = li.xpath(".//div[contains(@class,'fangyuan')]/span/text()").get()

*# 价格, 也同时获取到价格的单位, 同时把列表转换为字符串*

price = "".join(li.xpath(".//div[@class='nhouse\_price']//text()").getall())

price = re.sub(r"\s|广告", "", price)

*# 详情页url*

origin\_url = li.xpath(".//div[@class='nlcd\_name']/a/@href").get()

*# 构建item并返回*

item = NewHouseItem(name=name, rooms=rooms, area=area, address=address, district=district, sale=sale, price=price, origin\_url=origin\_url, province=province, city=city)

**yield** item

*# 获取下一页的链接*

next\_url = response.xpath("//div[@class='page']//a[@class='next']/@href").get()

*# 构造下一页的请求对象, 并指定解析函数为自身, 同时把省份和城市信息通过meta传递过去*

**if** next\_url:

**yield** scrapy.Request(url=response.urljoin(next\_url), callback=self.parse\_newhouse, meta={"info": (province, city)})

*# 二手房信息, 新房是针对某个小区的, 二手房是具体到某个房子的.*

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

province, city = response.meta.get('info')

*# 以每个房子进行分组, 遍历分组提取每个房子的信息*

dls = response.xpath("//div[@class='houseList']/dl")

**for** dl **in** dls:

*# 对每一个房源, 都添加省市信息*

item = ESFHouseItem(province=province, city=city)

item['name'] = dl.xpath(".//p[@class='mt10']/a/span/text()").get()

infos = dl.xpath(".//p[@class='mt12']/text()").getall()

infos = list(map(**lambda** x: re.sub(r"\s", "", x), infos))

infos = [re.sub(r'\s', '', info) **for** info **in** infos]

*# 得到的infos是一个列表, 其中包含着房屋的信息, 对其元素进行过滤, 提取出房屋的各项信息*

**for** info **in** infos:

**if** "厅" **in** info:

item['rooms'] = info

**elif** '层' **in** info:

item['floor'] = info

**elif** '向' **in** info:

item['toward'] = info

**else**:

item['year'] = info.replace("建筑年代：", "")

item['address'] = dl.xpath(".//p[@class='mt10']/span/@title").get()

item['area'] = dl.xpath(".//div[contains(@class,'area')]/p/text()").get()

item['price'] = "".join(dl.xpath(".//div[@class='moreInfo']/p[1]//text()").getall())

item['unit'] = "".join(dl.xpath(".//div[@class='moreInfo']/p[2]//text()").getall())

detail\_url = dl.xpath(".//p[@class='title']/a/@href").get()

item['origin\_url'] = response.urljoin(detail\_url)

**yield** item

*# 提取下一页的url, 构造请求*

next\_url = response.xpath("//a[@id='PageControl1\_hlk\_next']/@href").get()

**yield** scrapy.Request(url=response.urljoin(next\_url), callback=self.parse\_esf, meta={"info": (province, city)})

修改pipelines.py, 把数据保存到json文件中

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

**from** **scrapy.exporters** **import** JsonLinesItemExporter

**class** **FangPipeline**(object):

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

self.newhouse\_fp = open('newhouse.json','wb')

self.esfhouse\_fp = open('esfhouse.json','web')

self.newhouse\_exporter = JsonLinesItemExporter(self.newhouse\_fp,ensure\_ascii=False)

self.esfhouse\_exporter = JsonLinesItemExporter(self.esfhouse\_fp,ensure\_ascii=False)

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

self.newhouse\_exporter.export\_item(item)

self.esfhouse\_exporter.export\_item(item)

**return** item

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

self.newhouse\_fp.close()

self.esfhouse\_fp.close()

修改middlewares.py, 添加随机请求头

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

**import** **random**

**class** **UserAgentDownloadMiddleware**(object):

*# user-agent随机请求头中间件*

USER\_AGENTS = [

'Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_8; en-us) AppleWebKit/534.50 (KHTML, like Gecko) Version/5.1 Safari/534.50',

'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Safari/537.36',

'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;',

'Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.0; Trident/4.0)',

'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv,2.0.1) Gecko/20100101 Firefox/4.0.1',

'Mozilla/5.0 (Windows NT 6.1; rv,2.0.1) Gecko/20100101 Firefox/4.0.1'

]

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

user\_agent = random.choice(self.USER\_AGENTS)

request.headers['User-Agent'] = user\_agent

修改settings.py, 启用pipeline及middleware

DOWNLOADER\_MIDDLEWARES = {

'fang.middlewares.UserAgentDownloadMiddleware': 543,

}

ITEM\_PIPELINES = {

'fang.pipelines.FangPipeline': 300,

}

以上已经实现了单机的scrapy爬虫, 下面要把单机的爬虫修改为分布式的爬虫

配置ubuntu linux版的Redis服务器和爬虫服务器

安装ubuntu支持软件

# 更新软件库

apt-get update

# 安装支持软件

sudo apt-get -y install python3-dev build-essential python3-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev

# 安装项目使用的python包

在实际的项目开发中, 一般也是先在windows中进行开发, 然后把项目部署到linux服务器中.

在windows中进入虚拟环境, 再进入到项目所在的根目录中, 使用以下命令把项目开发使用的python包保存到txt文件中

pip freeze > requirements.txt

如果是部署到linux服务器中, 因为其中的pypiwin32是windows上专用的包, 需要删除掉, 否则会在安装时报错.

在xshell中使用rz命令从本机向远程linux服务器中传递数据

在linux服务器中安装reqirements.txt中的python包

在linux中同样创建虚拟环境

# 安装包管理器

pip install virtualenvwrapper

# 查看python3命令所在的目录

which python3

/usr/bin/python3

# 创建虚拟环境, 并指定python3 的位置.

mkvirtualenv -p /usr/bin/python3 crawler-env

# 进入虚拟环境

workon crawler-env

# 删除

# 安装requirements.txt中的python包

pip install -r requirements.txt

# 在爬虫服务器中安装scrapy-redis组件

pip install scrapy-redis

把项目改造成分布式爬虫.

1. 要将一个Scrapy项目变成一个Scrapy-redis项目只需修改以下三点就可以了：

1.1 将爬虫的类从scrapy.Spider变成scrapy\_redis.spiders.RedisSpider；或者是从scrapy.CrawlSpider变成scrapy\_redis.spiders.RedisCrawlSpider。

1.2 将爬虫中的start\_urls删掉。增加一个redis\_key="xxx"。这个redis\_key是为了以后在redis中控制爬虫启动的。爬虫的第一个url, 就是在redis中通过这个发送出去的。redis\_key指定了之后爬虫会从redis服务器中的如个key中去读取start\_url. 使用fang:start\_urls对key进行区分.

**from** **scrapy\_redis.spiders** **import** RedisSpider

**class** **SfwSpider**(RedisSpider):

name = 'fang'

allowed\_domains = ['fang.com']

*# start\_urls = ['http://www.fang.com/SoufunFamily.htm']*

redis\_key = "fang:start\_urls"

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

1.3 在配置文件中增加如下配置：

*# Configure item pipelines*

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

# 注释掉单机的pipeline, 使用redis的pipeline把数据保存到redis中.

*# ITEM\_PIPELINES = {*

*# 'fang.pipelines.FangPipeline': 300,*

*# }*

*# Scrapy-Redis相关的配置：*

*# 使用redis的调度器, 确保request存储到redis中*

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

*# 使用scrapy\_redis进行去重, 确保所有爬虫共享相同的去重指纹*

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

*# 使用redis的item pipeline, 把数据都传输并保存到redis服务器中*

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

*# 在redis中保持scrapy-redis用到的队列, 不会清理redis中的队列, 从而可以实现暂停和恢复的功能。*

SCHEDULER\_PERSIST = True

*# 设置连接redis信息*

REDIS\_HOST = '192.168.1.6'

REDIS\_PORT = 6379

2. 把项目打包并上传到linux服务器中

zip fang\_pro fang\_pro.zip

unzip fang\_pro.zip

cd fang\_pro

workon crawl-env

3. 运行爬虫：

1. 启动redis服务器中的redis服务

redis-server redis.windows.conf

2. 在所有爬虫服务器上运行爬虫

进入爬虫文件所在的路径

cd fang\_pro/fang\_pro/spiders

然后输入命令运行爬虫

scrapy runspider fang.py

因为没有起始的url地址, 所以所有的爬虫都处于等待状态

3. 在redis服务器所在的主机上, 另外打开一个cmd窗口执行redis-cli连接到服务器中, 向Redis服务器中添加一个起始的url链接

lpush fang:start\_urls http://www.fang.com/SoufunFamily.com

只要redis中有了fang:start\_urls这个键, 所有的爬虫都开始运行起来了.