# 1. 爬虫原理与数据抓取

## 课程简介

### 为什么要做爬虫？

首先请问: 都说现在是"大数据时代", 那数据从何而来？

* 企业产生的用户数据: [百度指数](http://index.baidu.com)、[阿里指数](https://alizs.taobao.com)、[TBI腾讯浏览指数](http://tbi.tencent.com)、[新浪微博指数](http://data.weibo.com/index)
* 数据平台购买数据: [数据堂](http://www.datatang.com/about/about-us.html)、[国云数据市场](http://www.moojnn.com/data-market/)、[贵阳大数据交易所](http://trade.gbdex.com/trade.web/index.jsp)
* 政府/机构公开的数据: [中华人民共和国国家统计局数据](http://data.stats.gov.cn/index.htm)、[世界银行公开数据](http://data.worldbank.org.cn)、[联合国数据](http://data.un.org)、[纳斯达克](http://www.nasdaq.com/zh).
* 数据管理咨询公司: [麦肯锡](http://www.mckinsey.com.cn)、[埃森哲](https://www.accenture.com/cn-zh/)、[艾瑞咨询](http://www.iresearch.com.cn)
* 爬取网络数据: 如果需要的数据市场上没有, 或者不愿意购买, 那么可以选择招/做一名爬虫工程师, 自己动手丰衣足食. [拉勾网Python爬虫职位](https://www.lagou.com/jobs/list_python爬虫)

### 爬虫是什么？

[百度百科: 网络爬虫](http://baike.baidu.com/link?url=3SeA6RcStie6o9T5XGMTBoHK-BGiO_0IqtnI4IkVjDBsoKrJL0aotjA4cjqbx8wF)

### 关于Python爬虫, 我们需要学习的有:

1. Python基础语法学习(基础知识)

2. HTML页面的内容抓取(数据抓取)

3. HTML页面的数据提取(数据清洗)

4. Scrapy框架以及scrapy-redis分布式策略(第三方框架)

6. 爬虫(Spider)、反爬虫(Anti-Spider)、反反爬虫(Anti-Anti-Spider)之间的斗争....

## (了解)通用爬虫和聚焦爬虫

### 通用爬虫和聚焦爬虫

根据使用场景, 网络爬虫可分为 **通用爬虫** 和 **聚焦爬虫** 两种.

### 通用爬虫

通用网络爬虫 是 捜索引擎抓取系统(Baidu、Google、Yahoo等) 的重要组成部分. 主要目的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.

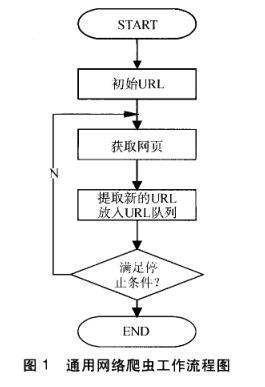
### 通用搜索引擎(Search Engine) 工作原理

**通用网络爬虫** 从互联网中搜集网页, 采集信息, 这些网页信息用于为搜索引擎建立索引从而提供支持, 它决定着整个引擎系统的内容是否丰富, 信息是否即时, 因此其性能的优劣直接影响着搜索引擎的效果.

#### 第一步: 抓取网页

搜索引擎网络爬虫的基本工作流程如下:

1. 首先选取一部分的种子URL, 将这些URL放入待抓取URL队列；
2. 取出待抓取URL, 解析DNS得到主机的IP, 并将URL对应的网页下载下来, 存储进已下载网页库中, 并且将这些URL放进已抓取URL队列.
3. 分析已抓取URL队列中的URL, 分析其中的其他URL, 并且将URL放入待抓取URL队列, 从而进入下一个循环....



**搜索引擎如何获取一个新网站的URL:**

1. 新网站向搜索引擎主动提交网址: 如百度

<http://zhanzhang.baidu.com/linksubmit/url>

2. 在其他网站上设置新网站外链(尽可能处于搜索引擎爬虫爬取范围)

3. 搜索引擎和DNS解析服务商(如DNSPod等) 合作, 新网站域名将被迅速抓取.

但是搜索引擎蜘蛛的爬行是被输入了一定的规则的, 它需要遵从一些命令或文件的内容, 如标注为nofollow的链接, 或者是Robots协议.

Robots协议(也叫爬虫协议、机器人协议等) , 全称是"网络爬虫排除标准"(Robots Exclusion Protocol) , 网站通过Robots协议告诉搜索引擎哪些页面可以抓取, 哪些页面不能抓取, 例如:

淘宝网: <https://www.taobao.com/robots.txt>

腾讯网:  <http://www.qq.com/robots.txt>

#### 第二步: 数据存储

搜索引擎通过爬虫爬取到的网页, 将数据存入原始页面数据库. 其中的页面数据与用户浏览器得到的HTML是完全一样的.

搜索引擎蜘蛛在抓取页面时, 也做一定的重复内容检测, 一旦遇到访问权重很低的网站上有大量抄袭、采集或者复制的内容, 很可能就不再爬行.

#### 第三步: 预处理

搜索引擎将爬虫抓取回来的页面, 进行各种步骤的预处理.

* 提取文字
* 中文分词
* 消除噪音(比如版权声明文字、导航条、广告等……)
* 索引处理
* 链接关系计算
* 特殊文件处理
* ....

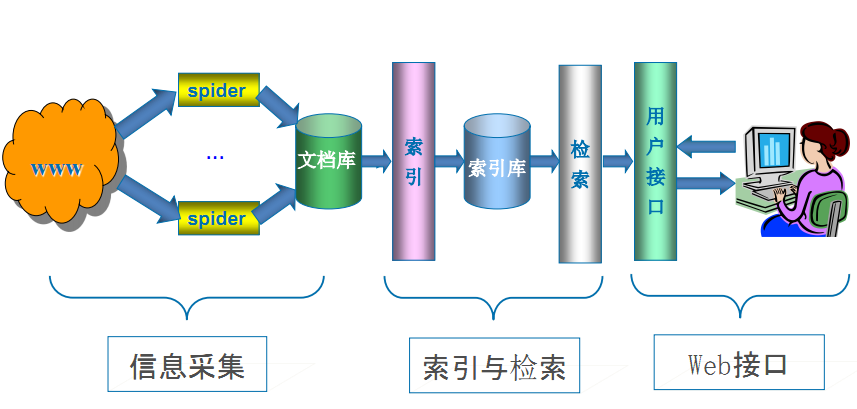
除了HTML文件外, 搜索引擎通常还能抓取和索引以文字为基础的多种文件类型, 如 PDF、Word、WPS、XLS、PPT、TXT 文件等. 我们在搜索结果中也经常会看到这些文件类型.

但搜索引擎还不能处理图片、视频、Flash 这类非文字内容, 也不能执行脚本和程序.

#### 第四步: 提供检索服务, 网站排名

搜索引擎在对信息进行组织和处理后, 为用户提供关键字检索服务, 将用户检索相关的信息展示给用户.

同时会根据页面的PageRank值(链接的访问量排名) 来进行网站排名, 这样Rank值高的网站在搜索结果中会排名较前, 当然也可以直接使用 Money 购买搜索引擎网站排名, 简单粗暴.



课外阅读: [Google搜索引擎的工作原理](http://article.yeeyan.org/view/99083/114884)

**但是, 这些通用性搜索引擎也存在着一定的局限性:**

1. 通用搜索引擎所返回的结果都是网页, 而大多情况下, 网页里90%的内容对用户来说都是无用的.
2. 不同领域、不同背景的用户往往具有不同的检索目的和需求, 搜索引擎无法提供针对具体某个用户的搜索结果.
3. 万维网数据形式的丰富和网络技术的不断发展, 图片、数据库、音频、视频多媒体等不同数据大量出现, 通用搜索引擎对这些文件无能为力, 不能很好地发现和获取.
4. 通用搜索引擎大多提供基于关键字的检索, 难以支持根据语义信息提出的查询, 无法准确理解用户的具体需求.

针对这些情况, 聚焦爬虫技术得以广泛使用.

### 聚焦爬虫

聚焦爬虫, 是"面向特定主题需求"的一种网络爬虫程序, 它与通用搜索引擎爬虫的区别在于:  **聚焦爬虫在实施网页抓取时会对内容进行处理筛选, 尽量保证只抓取与需求相关的网页信息.**

**而我们今后要学习的, 就是聚焦爬虫.**

## (复习)HTTP/HTTPS的请求与响应

### HTTP和HTTPS

HTTP协议(HyperText Transfer Protocol, 超文本传输协议) : 是一种发布和接收 HTML页面的方法.

HTTPS(Hypertext Transfer Protocol over Secure Socket Layer) 简单讲是HTTP的安全版, 在HTTP下加入SSL层.

SSL(Secure Sockets Layer 安全套接层) 主要用于Web的安全传输协议, 在传输层对网络连接进行加密, 保障在Internet上数据传输的安全.

* HTTP的端口号为80,
* HTTPS的端口号为443

#### HTTP工作原理

网络爬虫抓取过程可以理解为模拟浏览器操作的过程.

浏览器的主要功能是向服务器发出请求, 在浏览器窗口中展示您选择的网络资源, HTTP是一套计算机通过网络进行通信的规则.

#### HTTP的请求与响应

HTTP通信由两部分组成:  **客户端请求消息** 与 **服务器响应消息**

#### 浏览器发送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(Uniform / Universal Resource Locator的缩写) : 统一资源定位符, 是用于完整地描述Internet上网页和其他资源的地址的一种标识方法.

基本格式: scheme://host[:port#]/path/…/[?query-string][#anchor]

* scheme: 协议(例如: http, https, ftp)
* host: 服务器的IP地址或者域名
* port#: 服务器的端口(如果是走协议默认端口, 缺省端口80)
* path: 访问资源的路径
* query-string: 参数, 发送给http服务器的数据
* anchor: 锚(跳转到网页的指定锚点位置)

例如:

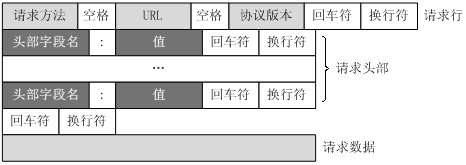
* ftp://192.168.0.116:8080/index
* http://www.baidu.com
* http://item.jd.com/11936238.html#product-detail

### 客户端HTTP请求

URL只是标识资源的位置, 而HTTP是用来提交和获取资源. 客户端发送一个HTTP请求到服务器的请求消息, 包括以下格式:

请求行、请求头部、空行、请求数据

四个部分组成, 下图给出了请求报文的一般格式.



**一个典型的HTTP请求示例**

GET https://www.baidu.com/ HTTP/1.1

Host: www.baidu.com

Connection: keep-alive

Upgrade-Insecure-Requests: 1

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

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

Referer: http://www.baidu.com/

Accept-Encoding: gzip, deflate, sdch, br

Accept-Language: zh-CN,zh;q=0.8,en;q=0.6

Cookie: BAIDUID=04E4001F34EA74AD4601512DD3C41A7B:FG=1; BIDUPSID=04E4001F34EA74AD4601512DD3C41A7B; PSTM=1470329258; MCITY=-343%3A340%3A; BDUSS=nF0MVFiMTVLcUh-Q2MxQ0M3STZGQUZ4N2hBa1FFRkIzUDI3QlBCZjg5cFdOd1pZQVFBQUFBJCQAAAAAAAAAAAEAAADpLvgG0KGyvLrcyfrG-AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFaq3ldWqt5XN; H\_PS\_PSSID=1447\_18240\_21105\_21386\_21454\_21409\_21554; BD\_UPN=12314753; sug=3; sugstore=0; ORIGIN=0; bdime=0; H\_PS\_645EC=7e2ad3QHl181NSPbFbd7PRUCE1LlufzxrcFmwYin0E6b%2BW8bbTMKHZbDP0g; BDSVRTM=0

#### 请求方法

GET https://www.baidu.com/ HTTP/1.1

根据HTTP标准, HTTP请求可以使用多种请求方法.

HTTP 0.9: 只有基本的文本 GET 功能.

HTTP 1.0: 完善的请求/响应模型, 并将协议补充完整, 定义了三种请求方法: GET, POST 和 HEAD方法.

HTTP 1.1: 在 1.0 基础上进行更新, 新增了五种请求方法: OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法.

HTTP 2.0(未普及) : 请求/响应首部的定义基本没有改变, 只是所有首部键必须全部小写, 而且请求行要独立为 :method、:scheme、:host、:path这些键值对.

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 描述 |
| 1 | GET | 请求指定的页面信息, 并返回实体主体. |
| 2 | HEAD | 类似于get请求, 只不过返回的响应中没有具体的内容, 用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求(例如提交表单或者上传文件) , 数据被包含在请求体中. POST请求可能会导致新的资源的建立和/或已有资源的修改. |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容. |
| 5 | DELETE | 请求服务器删除指定的页面. |
| 6 | CONNECT | HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器. |
| 7 | OPTIONS | 允许客户端查看服务器的性能. |
| 8 | TRACE | 回显服务器收到的请求, 主要用于测试或诊断. |

#### HTTP请求主要分为Get和Post两种方法

* GET是从服务器上获取数据, POST是向服务器传送数据
* GET请求参数显示, 都显示在浏览器网址上, HTTP服务器根据该请求所包含URL中的参数来产生响应内容, 即"Get"请求的参数是URL的一部分. 例如:  http://www.baidu.com/s?wd=Chinese
* POST请求参数在请求体当中, 消息长度没有限制而且以隐式的方式进行发送, 通常用来向HTTP服务器提交量比较大的数据(比如请求中包含许多参数或者文件上传操作等) , 请求的参数包含在"Content-Type"消息头里, 指明该消息体的媒体类型和编码,

**注意: 避免使用Get方式提交表单, 因为有可能会导致安全问题. 比如说在登陆表单中用Get方式, 用户输入的用户名和密码将在地址栏中暴露无遗.**

#### 常用的请求报头

##### 1. Host (主机和端口号)

Host: 对应网址URL中的Web名称和端口号, 用于指定被请求资源的Internet主机和端口号, 通常属于URL的一部分.

##### 2. Connection (链接类型)

Connection: 表示客户端与服务连接类型

1. Client 发起一个包含 Connection:keep-alive 的请求, HTTP/1.1使用 keep-alive 为默认值.
2. Server收到请求后:
   * 如果 Server 支持 keep-alive, 回复一个包含 Connection:keep-alive 的响应, 不关闭连接；
   * 如果 Server 不支持 keep-alive, 回复一个包含 Connection:close 的响应, 关闭连接.
3. 如果client收到包含 Connection:keep-alive 的响应, 向同一个连接发送下一个请求, 直到一方主动关闭连接.

**keep-alive在很多情况下能够重用连接, 减少资源消耗, 缩短响应时间, 比如当浏览器需要多个文件时(比如一个HTML文件和相关的图形文件), 不需要每次都去请求建立连接.**

##### 3. Upgrade-Insecure-Requests (升级为HTTPS请求)

Upgrade-Insecure-Requests: 升级不安全的请求, 意思是会在加载 http 资源时自动替换成 https 请求, 让浏览器不再显示https页面中的http请求警报.

**HTTPS 是以安全为目标的 HTTP 通道, 所以在 HTTPS 承载的页面上不允许出现 HTTP 请求, 一旦出现就是提示或报错.**

##### 4. User-Agent (浏览器名称)

User-Agent: 是客户浏览器的名称, 以后会详细讲.

##### 5. Accept (传输文件类型)

Accept: 指浏览器或其他客户端可以接受的MIME(Multipurpose Internet Mail Extensions(多用途互联网邮件扩展) ) 文件类型, 服务器可以根据它判断并返回适当的文件格式.

**举例:**

Accept: \*/\*: 表示什么都可以接收.

Accept: image/gif: 表明客户端希望接受GIF图像格式的资源；

Accept: text/html: 表明客户端希望接受html文本.

Accept: text/html, application/xhtml+xml;q=0.9, image/\*;q=0.8: 表示浏览器支持的 MIME 类型分别是 html文本、xhtml和xml文档、所有的图像格式资源.

**q是权重系数, 范围 0 =< q <= 1, q 值越大, 请求越倾向于获得其";"之前的类型表示的内容. 若没有指定q值, 则默认为1, 按从左到右排序顺序；若被赋值为0, 则用于表示浏览器不接受此内容类型.**

**Text: 用于标准化地表示的文本信息, 文本消息可以是多种字符集和或者多种格式的；Application: 用于传输应用程序数据或者二进制数据.** [***详细请点击***](http://blog.sina.com.cn/s/blog_866e403f010179f1.html)

##### 6. Referer (页面跳转处)

Referer: 表明产生请求的网页来自于哪个URL, 用户是从该 Referer页面访问到当前请求的页面. 这个属性可以用来跟踪Web请求来自哪个页面, 是从什么网站来的等.

有时候遇到下载某网站图片, 需要对应的referer, 否则无法下载图片, 那是因为人家做了防盗链, 原理就是根据referer去判断是否是本网站的地址, 如果不是, 则拒绝, 如果是, 就可以下载；

##### 7. Accept-Encoding(文件编解码格式)

Accept-Encoding: 指出浏览器可以接受的编码方式. 编码方式不同于文件格式, 它是为了压缩文件并加速文件传递速度. 浏览器在接收到Web响应之后先解码, 然后再检查文件格式, 许多情形下这可以减少大量的下载时间.

举例: Accept-Encoding:gzip;q=1.0, identity; q=0.5, \*;q=0

如果有多个Encoding同时匹配, 按照q值顺序排列, 本例中按顺序支持 gzip, identity压缩编码, 支持gzip的浏览器会返回经过gzip编码的HTML页面.

**如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受.**

##### 8. Accept-Language(语言种类)

Accept-Langeuage: 指出浏览器可以接受的语言种类, 如en或en-us指英语, zh或者zh-cn指中文, 当服务器能够提供一种以上的语言版本时要用到.

##### 9. Accept-Charset(字符编码)

Accept-Charset: 指出浏览器可以接受的字符编码.

举例: Accept-Charset:iso-8859-1,gb2312,utf-8

* ISO8859-1: 通常叫做Latin-1. Latin-1包括了书写所有西方欧洲语言不可缺少的附加字符, 英文浏览器的默认值是ISO-8859-1.
* gb2312: 标准简体中文字符集;
* utf-8: UNICODE 的一种变长字符编码, 可以解决多种语言文本显示问题, 从而实现应用国际化和本地化.

**如果在请求消息中没有设置这个域, 缺省是任何字符集都可以接受.**

##### 10. Cookie (Cookie)

Cookie: 浏览器用这个属性向服务器发送Cookie. Cookie是在浏览器中寄存的小型数据体, 它可以记载和服务器相关的用户信息, 也可以用来实现会话功能, 以后会详细讲.

##### 11. Content-Type (POST数据类型)

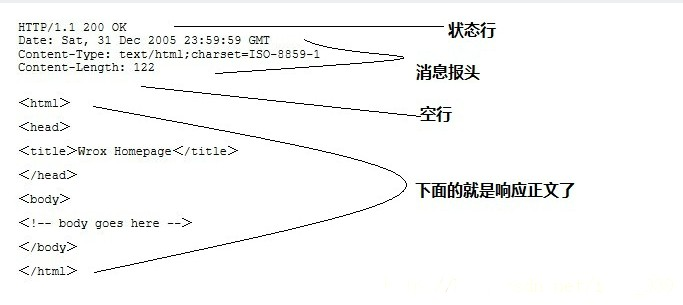
Content-Type: POST请求里用来表示的内容类型.

举例: Content-Type = Text/XML; charset=gb2312:

指明该请求的消息体中包含的是纯文本的XML类型的数据, 字符编码采用"gb2312".

### 服务端HTTP响应

HTTP响应也由四个部分组成, 分别是:  状态行、消息报头、空行、响应正文



HTTP/1.1 200 OK

Server: Tengine

Connection: keep-alive

Date: Wed, 30 Nov 2016 07:58:21 GMT

Cache-Control: no-cache

Content-Type: text/html;charset=UTF-8

Keep-Alive: timeout=20

Vary: Accept-Encoding

Pragma: no-cache

X-NWS-LOG-UUID: bd27210a-24e5-4740-8f6c-25dbafa9c395

Content-Length: 180945

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ....

#### 常用的响应报头(了解)

理论上所有的响应头信息都应该是回应请求头的. 但是服务端为了效率, 安全, 还有其他方面的考虑, 会添加相对应的响应头信息, 从上图可以看到:

##### 1. Cache-Control: must-revalidate, no-cache, private.

这个值告诉客户端, 服务端不希望客户端缓存资源, 在下次请求资源时, 必须要从新请求服务器, 不能从缓存副本中获取资源.

* Cache-Control是响应头中很重要的信息, 当客户端请求头中包含Cache-Control:max-age=0请求, 明确表示不会缓存服务器资源时, Cache-Control作为作为回应信息, 通常会返回no-cache, 意思就是说, "那就不缓存呗".
* 当客户端在请求头中没有包含Cache-Control时, 服务端往往会根据不同的资源采用不同的缓存策略, 比如说oschina在缓存图片资源的策略就是Cache-Control: max-age=86400,这个意思是, 从当前时间开始, 在86400秒的时间内, 客户端可以直接从缓存副本中读取资源, 而不需要向服务器请求.

##### 2. Connection: keep-alive

这个字段作为回应客户端的Connection: keep-alive, 告诉客户端服务器的tcp连接也是一个长连接, 客户端可以继续使用这个tcp连接发送http请求.

##### 3. Content-Encoding:gzip

告诉客户端, 服务端发送的资源是采用gzip编码的, 客户端看到这个信息后, 应该采用gzip对资源进行解码.

##### 4. Content-Type: text/html;charset=UTF-8

告诉客户端, 资源文件的类型, 还有字符编码, 客户端通过utf-8对资源进行解码, 然后对资源进行html解析. 通常我们会看到有些网站是乱码的, 往往就是服务器端没有返回正确的编码.

##### 5. Date: Sun, 21 Sep 2016 06:18:21 GMT

这个是服务端发送资源时的服务器时间, GMT是格林尼治所在地的标准时间. http协议中发送的时间都是GMT的, 这主要是解决在互联网上, 不同时区在相互请求资源的时候, 时间混乱问题.

##### 6. Expires:Sun, 1 Jan 2000 01:00:00 GMT

这个响应头也是跟缓存有关的, 告诉客户端在这个时间前, 可以直接访问缓存副本, 很显然这个值会存在问题, 因为客户端和服务器的时间不一定会都是相同的, 如果时间不同就会导致问题. 所以这个响应头是没有Cache-Control: max-age=\*这个响应头准确的, 因为max-age=date中的date是个相对时间, 不仅更好理解, 也更准确.

##### 7. Pragma:no-cache

这个含义与Cache-Control等同.

##### 8.Server: Tengine/1.4.6

这个是服务器和相对应的版本, 只是告诉客户端服务器的信息.

##### 9. Transfer-Encoding: chunked

这个响应头告诉客户端, 服务器发送的资源的方式是分块发送的. 一般分块发送的资源都是服务器动态生成的, 在发送时还不知道发送资源的大小, 所以采用分块发送, 每一块都是独立的, 独立的块都能标示自己的长度, 最后一块是0长度的, 当客户端读到这个0长度的块时, 就可以确定资源已经传输完了.

##### 10. Vary: Accept-Encoding

告诉缓存服务器, 缓存压缩文件和非压缩文件两个版本, 现在这个字段用处并不大, 因为现在的浏览器都是支持压缩的.

#### 响应状态码

响应状态代码有三位数字组成, 第一个数字定义了响应的类别, 且有五种可能取值.

##### 常见状态码:

* 100~199: 表示服务器成功接收部分请求, 要求客户端继续提交其余请求才能完成整个处理过程.
* 200~299: 表示服务器成功接收请求并已完成整个处理过程. 常用200(OK 请求成功) .
* 300~399: 为完成请求, 客户需进一步细化请求. 例如: 请求的资源已经移动一个新地址、常用302(所请求的页面已经临时转移至新的url) 、307和304(使用缓存资源) .
* 400~499: 客户端的请求有错误, 常用404(服务器无法找到被请求的页面) 、403(服务器拒绝访问, 权限不够) .
* 500~599: 服务器端出现错误, 常用500(请求未完成. 服务器遇到不可预知的情况) .

### Cookie 和 Session

服务器和客户端的交互仅限于请求/响应过程, 结束之后便断开, 在下一次请求时, 服务器会认为新的客户端.

为了维护他们之间的链接, 让服务器知道这是前一个用户发送的请求, 必须在一个地方保存客户端的信息.

**Cookie**: 通过在 客户端 记录的信息确定用户的身份.

**Session**: 通过在 服务器端 记录的信息确定用户的身份.

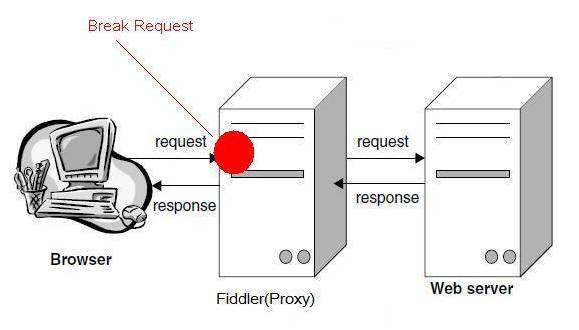
## HTTP/HTTPS抓包工具-Fiddler

HTTP代理神器Fiddler

Fiddler是一款强大Web调试工具, 它能记录所有客户端和服务器的HTTP请求. Fiddler启动的时候, 默认IE的代理设为了127.0.0.1:8888, 而其他浏览器是需要手动设置.

### 工作原理

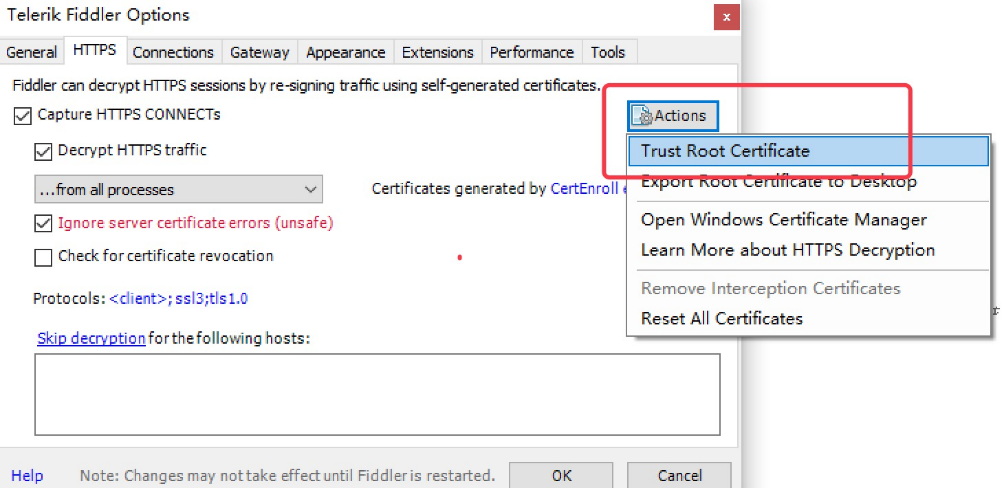
Fiddler 是以代理web服务器的形式工作的, 它使用代理地址: 127.0.0.1, 端口: 8888



### Fiddler抓取HTTPS设置

在浏览器中设置好Fiddler后就能抓取到http协议的信息, 但是https的信息由于需要证书, 还需要进行另外的设置.

1. 启动Fiddler, 打开菜单栏中的 Tools > Telerik Fiddler Options, 打开"Fiddler Options"对话框.
2. 对Fiddler进行设置:
   * 打开工具栏->Tools->Fiddler Options->HTTPS,
   * 选中Capture HTTPS CONNECTs (捕捉HTTPS连接),
   * 选中Decrypt HTTPS traffic(解密HTTPS通信)
   * 另外我们要用Fiddler获取本机所有进程的HTTPS请求, 所以中间的下拉菜单中选中...from all processes (从所有进程) 改为from browsers only
   * 选中下方Ignore server certificate errors(忽略服务器证书错误)
3. 为 Fiddler 配置Windows信任这个根证书解决安全警告: Trust Root Certificate(受信任的根证书) .



1. Fiddler 主菜单 Tools -> Fiddler Options…-> Connections
   * 选中Allow remote computers to connect(允许远程连接)
   * Act as system proxy on startup(作为系统启动代理)



1. 重启Fiddler, 使配置生效(这一步很重要, 必须做) .

### Fiddler 如何捕获Chrome的会话

1. 安装SwitchyOmega 代理管理 Chrome 浏览器插件
2. 如图所示, 设置代理服务器为127.0.0.1:8888
3. 通过浏览器插件切换为设置好的代理.

### Fiddler界面

设置好后, 本机HTTP通信都会经过127.0.0.1:8888代理, 也就会被Fiddler拦截到.

在以后写爬虫前最好先用抓包工具测试一下, 尽量模拟浏览器的形为来爬取数据.

#### 请求 (Request) 部分详解

1. Headers —— 显示客户端发送到服务器的 HTTP 请求的 header, 显示为一个分级视图, 包含了 Web 客户端信息、Cookie、传输状态等.
2. Textview —— 显示 POST 请求的 body 部分为文本.
3. WebForms —— 以表单形式显示请求的 GET 参数 和 POST body 内容.
4. HexView —— 用十六进制数据显示请求.
5. Auth —— 显示响应 header 中的 Proxy-Authorization(代理身份验证) 和 Authorization(授权) 信息.
6. Raw —— 将整个请求显示为纯文本.
7. JSON —— 显示JSON格式文件.
8. XML —— 如果请求的 body 是 XML 格式, 就是用分级的 XML 树来显示它.

#### 响应 (Response) 部分详解

1. Transformer —— 显示响应的编码信息.
2. Headers —— 用分级视图显示响应的 header.
3. TextView —— 使用文本显示响应的 body. 在TextView中看到的内容与在浏览器中查看源代码的代码是完全一样的.
4. ImageVies —— 如果请求是图片资源, 显示响应的图片.
5. HexView —— 用十六进制数据显示响应.
6. WebView —— 响应在 Web 浏览器中的预览效果.
7. Auth —— 显示响应 header 中的 Proxy-Authorization(代理身份验证) 和 Authorization(授权) 信息.
8. Caching —— 显示此请求的缓存信息.
9. Privacy —— 显示此请求的私密 (P3P) 信息.
10. Raw —— 将整个响应显示为纯文本.
11. JSON —— 显示JSON格式文件.
12. XML —— 如果响应的 body 是 XML 格式, 就是用分级的 XML 树来显示它 .

右下角响应部分的 Find为查找

左下角黑框QuickExec Alt+Q为执行命令. clear为清除

## urllib2模块的基本使用

所谓网页抓取, 就是把URL地址中指定的网络资源从网络流中读取出来, 保存到本地. 在Python中有很多库可以用来抓取网页, 我们先学习urllib2.

urllib2 是 Python2.7 自带的模块(不需要下载, 导入即可使用)

urllib2 官方文档: <https://docs.python.org/2/library/urllib2.html>

urllib2 源码: <https://hg.python.org/cpython/file/2.7/Lib/urllib2.py>

urllib2 在 python3.x 中被改为urllib.request, 只要把代码中所有的urllib2改为urllib.request就能在python3中正常使用了. 在python3中, python2.X 中的 urllib.urlopen()被废弃, urllib2.urlopen()相当于python3.X中的urllib.request.urlopen()

对于python3.X:

这里urllib成了一个包, 此包分成了几个模块,

urllib.request 用于打开和读取URL,

urllib.error 用于处理前面request引起的异常,

urllib.parse 用于解析URL,

urllib.robotparser用于解析robots.txt文件

### urlopen

我们先来段代码:

# urllib2\_urlopen.py

# 导入urllib2 库

import urllib2

# 向指定的url发送请求, 并返回服务器响应的类文件对象

response = urllib2.urlopen("http://www.baidu.com")

# 类文件对象支持 文件对象的操作方法, 如read()方法读取文件全部内容, 返回字符串

html = response.read()

# 打印字符串

print html

执行写的python代码, 将打印结果

Power@PowerMac ~$: python urllib2\_urlopen.py

把执行的结果重定向到baidu.html这个文件中. python3中会出错.

python 7.1.4\_urllib2.py > baidu.html

**python3代码**

*# urllib\_request\_urlopen.py*

# 导入urllib.request库

**import** urllib.request

# 向指定的url发送请求, 并返回服务器响应的类文件对象, 网址最后的/一定要写上, 表示根目录. 不然会抓取不到内容. urllib.request可以发送post请求, 也可以发送get请求, 如果发送的数据包含data信息, 就是post的请求.

response = urllib.request.urlopen(**'http://www.baidu.com/'**)

# response是一个类文件对象, 支持文件对象的操作方法, 如read()方法读取文件全部内容, 返回字符串

html = response.read()

# 文本在网络上是以二进制的方式传输的, 所以要以二进制的方式打开文件.

with open('baidu1.html', 'wb') as f:

f.write(html)

# 如果不以二进制的方式打开文件, 就要把网络中传输的二进制文本进行解码.

with open(' baidu2.html', 'w', encoding="utf-8") as f:

f.write(html.decode("utf-8"))

# 打印字符串

print(html)

**实际上, 如果我们在浏览器上打开百度主页, 右键选择"查看源代码", 你会发现, 跟我们刚才打印出来的是一模一样. 也就是说, 上面的4行代码就已经帮我们把百度的首页的全部代码爬了下来.**

**一个基本的url请求对应的python代码真的非常简单.**

### Request

在我们第一个例子里, urlopen()的参数就是一个url地址；

但是如果需要执行更复杂的操作, 比如增加HTTP报头, 必须创建一个 Request 实例来作为urlopen()的参数；而需要访问的url地址则作为 Request 实例的参数.

**python2中**

# urllib2\_request.py

import urllib2

# url 作为Request()方法的参数, 构造并返回一个Request对象

request = urllib2.Request("http://www.baidu.com")

# Request对象作为urlopen()方法的参数, 发送给服务器并接收响应

response = urllib2.urlopen(request)

html = response.read()

print html

**python3中**

# urllib\_request.Request.py

**import** urllib.request

#通过urllib.request.Request()方法构造一个请求对象, 网址最后的/一定要写上, 表示根目录. 不然会抓取不到内容.

request = urllib.request.Request(**'http://www.baidu.com/'**)

# 向指定的url发送请求, 并返回服务器响应的类文件对象

response = urllib.request.urlopen(request)

# 类文件对象支持 文件对象的操作方法, 如read()方法读取文件全部内容, 返回字符串

html = response.read()

with open('baidu1.html', 'wb') as f:

f.write(html)

with open('baidu2.html', 'w', encoding="utf-8") as f:

f.write(html.decode("utf-8"))

# 打印字符串

print(html)

**运行结果是完全一样的:**

新建Request实例, 除了必须要有 url 参数之外, 还可以设置另外两个参数:

1. data(默认空) : 是伴随 url 提交的数据(比如要post的数据) , 同时 HTTP 请求将从 "GET"方式 改为 "POST"方式.
2. headers(默认空) : 是一个字典, 包含了需要发送的HTTP报头的键值对.

这两个参数下面会说到.

### User-Agent

查看urllib2默认的user-agent

Python自带的模块: /usr/lib/python2.7/urllib2.py

找到其中的user-agent.

urllib2默认的User-Agent: python-urllib/2.7

python2的第三方模块: /usr/local/lib/python2.7/site-packages

python3中, 打开C:\Python35\Lib\urllib\request.py文件查看其中的urlopen函数. 找到其中的User-agent.

urllib.request.py默认是以url为user-agent发送信息的.

\_\_version\_\_ = '%d.%d' % sys.version\_info[:2]

class OpenerDirector:

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

client\_version = "Python-urllib/%s" % \_\_version\_\_

self.addheaders = [('User-agent', client\_version)]

ipython中进行操作:

import sys

sys.version\_info[:2]

3.6

urllib.request.py默认是以Python-urllib/3.x 的User-agent发送信息的.

想要伪装浏览器, 这个user-agent是一定要写的. 不写就算是一分钟爬取一次, 也会被浏览器认为是爬虫. User-Agent是爬虫和反爬虫关键的第一步, 养成好习惯, 发送请求都要带上

但是这样直接用urllib2给一个网站发送请求的话, 确实略有些唐突了, 就好比, 人家每家都有门, 你以一个路人的身份直接闯进去显然不是很礼貌. 而且有一些站点不喜欢被程序(非人为访问) 访问, 有可能会拒绝你的访问请求.

但是如果我们用一个合法的身份去请求别人网站, 显然人家就是欢迎的, 所以我们就应该给我们的这个代码加上一个身份, 就是所谓的User-Agent头.

* 浏览器 就是互联网世界上公认被允许的身份, 如果我们希望我们的爬虫程序更像一个真实用户, 那我们第一步, 就是需要伪装成一个被公认的浏览器. 用不同的浏览器在发送请求的时候, 会有不同的User-Agent头. urllib2默认的User-Agent头为: Python-urllib/x.y(x和y是Python主版本和次版本号,例如 Python-urllib/2.7)

#urllib2\_useragent.py

import urllib2

url = "http://www.itcast.cn"

#IE 9.0 的 User-Agent, 包含在 ua\_header里

ua\_header = {"User-Agent" : "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"}

# url 连同 headers, 一起构造Request请求, 这个请求将附带 IE9.0 浏览器的User-Agent

request = urllib2.Request(url, headers = ua\_header)

# 向服务器发送这个请求

response = urllib2.urlopen(request)

html = response.read()

print html

**python3代码**

添加User-Agent之后, 可以抓取到<https://www.baidu.com/>, 如果不加User-Agent, 不能抓取到https的百度网.

# 导入urllib.request库

**import** urllib.request

#User-Agent是爬虫和反爬虫关键的第一步

ua\_headers= {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'

}

url = 'http://www.baidu.com/'

#通过urllib.request.Request()方法构造一个请求对象

#向指定的地址发送请求, 并返回服务器响应的

request = urllib.request.Request(url, headers=ua\_headers)

# 向指定的url发送请求, 并返回服务器响应的类文件对象, 网址最后的/一定要写上, 表示根目录. 不然会抓取不到内容.

response = urllib.request.urlopen(request)

# 类文件对象支持 文件对象的操作方法, 如read()方法读取文件全部内容, 返回字符串

html = response.read()

# 打印字符串

print(html)

#返回HTTP的响应码, 成功返回200, 4表示服务器页面出错, 5表示服务器有问题

print(response.getcode())

#防止重定向, 查看实际返回数据的url.

print(response.geturl())

#返回服务器响应的HTTP报头信息

print(response.info())

### 添加更多的Header信息

在 HTTP Request 中加入特定的 Header, 来构造一个完整的HTTP请求消息.

可以通过调用Request.add\_header() 添加/修改一个特定的header 也可以通过调用Request.get\_header()来查看已有的header.

使用爬虫伪装浏览器时, 请求头中的哪些信息要加上.

1. User-agent是一定要写的.

2. Accept-Encoding是一定不能写的.

3. Connection: keep-alive有时候需要

4. cookie一般是用来模拟登陆的, 如果不需要登陆就能完成, 那就不用使用cookie

5. accept不用写, 是在爬取动态页面时用的.

* 添加一个特定的header

# urllib2\_headers.py

import urllib2

url = "http://www.itcast.cn"

#IE 9.0 的 User-Agent

header = {"User-Agent" : "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"}

request = urllib2.Request(url, headers = header)

#也可以通过调用Request.add\_header() 添加/修改一个特定的header

request.add\_header("Connection", "keep-alive")

# 也可以通过调用Request.get\_header()来查看header信息

# request.get\_header(header\_name="Connection")

response = urllib2.urlopen(req)

print response.code #可以查看响应状态码

html = response.read()

print html

* 随机添加/修改User-Agent

# urllib2\_add\_headers.py

import urllib2

import random

url = "http://www.itcast.cn"

ua\_list = [

"Mozilla/5.0 (Windows NT 6.1; ) Apple.... ",

"Mozilla/5.0 (X11; CrOS i686 2268.111.0)... ",

"Mozilla/5.0 (Macintosh; U; PPC Mac OS X.... ",

"Mozilla/5.0 (Macintosh; Intel Mac OS... "

]

user\_agent = random.choice(ua\_list)

request = urllib2.Request(url)

#也可以通过调用Request.add\_header() 添加/修改一个特定的header

request.add\_header("User-Agent", user\_agent)

# 第一个字母大写, 后面的全部小写

request.get\_header("User-agent")

response = urllib2.urlopen(req)

html = response.read()

print html

python3随机添加header

random\_user\_agent.py

**import** urllib.request, random

url = 'http://www.baidu.com/'

#可以是User-Agent列表, 也可以是代理列表

ua\_list = [

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv2.0.1) Gecko/20100101 Firefox/4.0.1",

"Mozilla/5.0 (Windows NT 6.1; rv2.0.1) Gecko/20100101 Firefox/4.0.1",

"Opera/9.80 (Macintosh; Intel Mac OS X 10.6.8; U; en) Presto/2.8.131 Version/11.11",

"Opera/9.80 (Windows NT 6.1; U; en) Presto/2.8.131 Version/11.11",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_0) AppleWebKit/535.11 (KHTML, like Gecko) Chrome/17.0.963.56 Safari/535.11"

]

#在User-Agent列表里随机选择一个User-Agent

user\_agent = random.choice(ua\_list)

#构造一个请求

request = urllib.request.Request(url)

#使用add\_header()方法, 向request对象中添加/修改一个HTTP报头

request.add\_header(**'User-Agent'**, user\_agent)

#get\_header()获取一个已有的HTTP报头的值, 注意只能是第一个字母大写, 其他的必须小写

print(request.get\_header(**'User-agent'**))

response = urllib.request.urlopen(request)

html = response.read()

with open("baidu.html", "wb") as f:

f.write(html.decode("utf-8"))

print(html)

## urllib2: GET请求和POST请求

**urllib2默认只支持HTTP/HTTPS的**GET**和**POST**方法**

### urllib.urlencode()

**urllib 和 urllib2 都是接受URL请求的相关模块, 但是提供了不同的功能. 两个最显著的不同如下:**

* urllib 仅可以接受URL, 不能创建设置了headers 的Request 类实例；
* 但是 urllib 提供 urlencode 方法用来GET查询字符串的产生, 而 urllib2 则没有. (这是 urllib 和 urllib2 经常一起使用的主要原因)
* 编码工作使用urllib的urlencode()函数, 帮我们将key:value这样的键值对转换成"key=value"这样的字符串, 解码工作可以使用urllib的unquote()函数. (注意, 不是urllib2.urlencode() )

# IPython2 中的测试结果

In [1]: import urllib

In [2]: word = {"wd" : "传智播客"}

# 通过urllib.urlencode()方法, 将字典键值对按URL编码转换, 从而能被web服务器接受.

In [3]: urllib.urlencode(word)

Out[3]: "wd=%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2"

# 通过urllib.unquote()方法, 把 URL编码字符串, 转换回原先字符串.

In [4]: print urllib.unquote("wd=%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2")

wd=传智播客

**一般HTTP请求提交数据, 需要编码成 URL编码格式, 然后做为url的一部分, 或者作为参数传到Request对象中.**

### Get方式

GET请求一般用于我们向服务器获取数据, 比如说, 我们用百度搜索传智播客: <https://www.baidu.com/s?wd=传智播客>

浏览器的url会跳转成如图所示:

<https://www.baidu.com/s?wd=%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2>

在其中我们可以看到在请求部分里, http://www.baidu.com/s? 之后出现一个长长的字符串, 其中就包含我们要查询的关键词传智播客, 于是我们可以尝试用默认的Get方式来发送请求.

http://tool.chinaz.com/tools/urlencode.aspx

站长之家工具的URL编码/解码

汉字以%作为分割, 每3个作为一段,

使用urllib中的工具进行编码和解码

在python3中, urllib中的unquote和urlencode被封装在了urllib.parse.py文件中, 只要把python2中的urllib.urlencode和urllib.unquote改为urllib.parse.urlencode和urllib.parse.unquote即可.

urllib.parse的urlencode工具接收的参数是一个字典

在ipython3中进行测试

**import** urllib.parse

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

wd1 = {**'wd'**: **'传智播客'**}

wd2 = {**'wd是'**: **'传智播客'**}

urllib.parse.urlencode(wd1)

urllib.parse.urlencode(wd2)

encoded1 = urllib.parse.urlencode(wd1)

encoded2 = urllib.parse.urlencode(wd2)

print(**"encoded1: \t %s"**%encoded1)

print(**"encoded2: \t %s"**%encoded2)

unquoted1 = urllib.parse.unquote(encoded1)

unquoted2 = urllib.parse.unquote(encoded2)

print(**"unquoted1: \t %s"**%unquoted1)

print(**"unquoted2: \t %s"**%unquoted2)

**urllib.parse.urlencode()函数会把字典格式转换为等号连接**

encoded1: wd=%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2

encoded2: wd%E6%98%AF=%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2

unquoted1: wd=传智播客

unquoted2: wd是=传智播客

# urllib2\_get.py

import urllib #负责url编码处理

import urllib2

url = "http://www.baidu.com/s"

word = {"wd":"传智播客"}

word = urllib.urlencode(word) #转换成url编码格式(字符串)

newurl = url + "?" + word # url首个分隔符就是 ?

headers={ "User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36"}

request = urllib2.Request(newurl, headers=headers)

response = urllib2.urlopen(request)

print response.read()

**使用GET请求写一个简单的百度搜索引擎接口**

import urllib.parse, urllib.request

*#根url*

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

headers= {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'

}

keyword = input('请输入需要查询的字符串: \n')

*#通过字典的形式保存输入的查询关键词*

wd = {'wd': keyword}

*#通过url.parse.urlencode()参数将一个字典类型的汉字转化为url编码*

encoded\_wd = urllib.parse.urlencode(wd)

*#组合一个完整的url地址出来.*

full\_url = url + encoded\_wd

request = urllib.request.Request(full\_url, headers=headers)

response = urllib.request.urlopen(request)

with open("baidu.html", "wb") as f:

f.write(response.read().decode("utf-8"))

print(response.read()

**批量爬取贴吧页面数据**

首先我们创建一个python文件, tiebaSpider.py, 我们要完成的是, 输入一个百度贴吧的地址, 比如:

百度贴吧LOL吧第一页: <http://tieba.baidu.com/f?kw=lol&ie=utf-8&pn=0>

第二页:  <http://tieba.baidu.com/f?kw=lol&ie=utf-8&pn=50>

第三页:  <http://tieba.baidu.com/f?kw=lol&ie=utf-8&pn=100>

发现规律了吧, 贴吧中每个页面不同之处, 就是url最后的pn的值, 其余的都是一样的, 我们可以抓住这个规律.

**简单写一个小爬虫程序, 来爬取百度LOL吧的所有网页.**

* 先写一个main, 提示用户输入要爬取的贴吧名, 并用urllib.urlencode()进行转码, 然后组合url, 假设是lol吧, 那么组合后的url就是: http://tieba.baidu.com/f?kw=lol

# 模拟 main 函数

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

kw = raw\_input("请输入需要爬取的贴吧:")

# 输入起始页和终止页, str转成int类型

beginPage = int(raw\_input("请输入起始页: "))

endPage = int(raw\_input("请输入终止页: "))

url = "http://tieba.baidu.com/f?"

key = urllib.urlencode({"kw" : kw})

# 组合后的url示例: http://tieba.baidu.com/f?kw=lol

url = url + key

tiebaSpider(url, beginPage, endPage)

* 接下来, 我们写一个百度贴吧爬虫接口, 我们需要传递3个参数给这个接口, 一个是main里组合的url地址, 以及起始页码和终止页码, 表示要爬取页码的范围.

def tiebaSpider(url, beginPage, endPage):

"""

作用: 负责处理url, 分配每个url去发送请求

url: 需要处理的第一个url

beginPage: 爬虫执行的起始页面

endPage: 爬虫执行的截止页面

"""

for page in range(beginPage, endPage + 1):

pn = (page - 1) \* 50

filename = "第" + str(page) + "页.html"

# 组合为完整的 url, 并且pn值每次增加50

fullurl = url + "&pn=" + str(pn)

#print fullurl

# 调用loadPage()发送请求获取HTML页面

html = loadPage(fullurl, filename)

# 将获取到的HTML页面写入本地磁盘文件

writeFile(html, filename)

* 我们已经之前写出一个爬取一个网页的代码. 现在, 我们可以将它封装成一个小函数loadPage, 供我们使用.

def loadPage(url, filename):

'''

作用: 根据url发送请求, 获取服务器响应文件

url: 需要爬取的url地址

filename: 文件名

'''

print "正在下载" + filename

headers = {"User-Agent": "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"}

request = urllib2.Request(url, headers = headers)

response = urllib2.urlopen(request)

return response.read()

* 最后如果我们希望将爬取到了每页的信息存储在本地磁盘上, 我们可以简单写一个存储文件的接口.

def writeFile(html, filename):

"""

作用: 保存服务器响应文件到本地磁盘文件里

html: 服务器响应文件

filename: 本地磁盘文件名

"""

print "正在存储" + filename

with open(filename, 'w') as f:

f.write(html)

print "-" \* 20

其实很多网站都是这样的, 同类网站下的html页面编号, 分别对应网址后的网页序号, 只要发现规律就可以批量爬取页面了.

python3 最终代码

tiebaSpider.py

**import** urllib.parse, urllib.request

#发送请求, 爬取网页

**def** loadPage(url, filename):

'''

作用: 根据url发送请求, 获取服务器响应文件

url: 需要爬取的url地址

filename: 文件名

'''

print(**"正在下载%s"**%filename)

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko"}

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

response = urllib.request.urlopen(request)

**return** response.read().decode("utf-8")

#把爬取到的信息写入到文件中

**def** writeFile(html, filename):

"""

作用: 保存服务器响应文件到本地磁盘文件里

html: 服务器响应文件

filename: 本地磁盘文件名

"""

print(**"正在存储%s"**%filename)

**with** open(filename, **'wb'**) **as** f:

f.write(html)

print(**"-"** \* 20)

#百度贴吧爬虫接口, 需要传递3个参数给这个接口, main里组合的url地址, 起始页码和终止页码

**def** tiebaSpider(url, beginPage, endPage):

"""

作用: 负责处理url, 分配每个url去发送请求

url: 需要处理的第一个url

beginPage: 爬虫执行的起始页面

endPage: 爬虫执行的截止页面

"""

# range(2,6)的范围为[2,5], 所以endPage要加上1才能代表要爬取的页面

**for** page **in** range(beginPage, endPage + 1):

pn = (page - 1) \* 50

filename = **"第"** + str(page) + **"页.html"**

# 组合为完整的 url, 并且pn值每次增加50

fullurl = url + **"&pn="** + str(pn)

#print fullurl

# 调用loadPage()发送请求获取HTML页面

html = loadPage(fullurl, filename)

# 将获取到的HTML页面写入本地磁盘文件

writeFile(html, filename)

#模拟main函数

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

kw = input(**"请输入需要爬取的贴吧:"**)

# 输入起始页和终止页, str转成int类型

beginPage = int(input(**"请输入起始页: "**))

endPage = int(input(**"请输入终止页: "**))

#组合后的url示例: http://tieba.baidu.com/f?kw=lol

url = "http://tieba.baidu.com/f?"

key = urllib.parse.urlencode({**"kw"** : kw})

# 组合后的url示例: http://tieba.baidu.com/f?kw=lol

url = url + key

tiebaSpider(url, beginPage, endPage)

### POST方式:

**GET和POST请求的区别**

GET: 请求的url会附带查询参数, 查询参数在QueryString里保存

POST: 请求的url不带参数, 查询参数在Form表单里保存

<http://fanyi.youdao.com>使用的是post的请求

需要找到POST请求的地址, POST请求的headers信息, POST请求的data信息.

**在Fiddler或Chrome中查看post请求的数据和post请求的地址**

Fiddler > Raw >

WebForms就显示出查询的Name和Value值.

在Fiddler > WebForm中的Body部分显示出来, 也会在Raw中的最后一行显示出来.

**chrome查看header信息**

POST <http://fanyi.youdao.com/translate_o?smartresult=dict&smartresult=rule&sessionFrom=null> HTTP/1.1

Host: fanyi.youdao.com

Connection: keep-alive

Content-Length: 231

# 表示返回的数据格式为json, 如果是以ajax方式加载的数据, 绝大部分都是json的格式

Accept: application/json, text/javascript, \*/\*; q=0.01

Origin: <http://fanyi.youdao.com>

# 表示使用ajax的方式进行加载的

X-Requested-With: XMLHttpRequest

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.81 Safari/537.36

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

DNT: 1

Referer: <http://fanyi.youdao.com/>

Accept-Encoding: gzip, deflate

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

**post请求的地址**

POST <http://fanyi.youdao.com/translate_o?smartresult=dict&smartresult=rule&sessionFrom=null> HTTP/1.1

**post请求的headers信息. 使用sublime的正则匹配来去掉冒号后面的空格, 并加上""**

find: ^(.\*):\s(.\*)$

replace: "\1":"\2",

"Host":"fanyi.youdao.com",

"Connection":"keep-alive",

"Content-Length":"231",

"Accept":"application/json, text/javascript, \*/\*; q=0.01",

"Origin":"http://fanyi.youdao.com",

"X-Requested-With":"XMLHttpRequest",

"User-Agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.81 Safari/537.36",

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

"DNT":"1",

"Referer":"http://fanyi.youdao.com/",

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

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

**POST请求的data信息, 在chrome中查看到的form data信息., 即向服务器提交的data信息, 使用urllib.request.Request构建的request对象中如果有data信息, 就是post的请求.**

i=%E4%BD%A0%E5%A5%BD%EF%BC%8Cpython&from=AUTO&to=AUTO&smartresult=dict&client=fanyideskweb&salt=1493768163104&sign=4d6f9cedeff67bea71cbcbef20459aff&doctype=json&version=2.1&keyfrom=fanyi.web&action=FY\_BY\_CLICKBUTTON&typoResult=true

**使用sublime正则替换转换格式把request的内容转化为字典格式**

在sublimetext的查找替换中ctrl+Enter输入回车进行替换

^(.\*):\s(.\*)$

\s表示匹配空格

"i":"%E4%BD%A0%E5%A5%BD%EF%BC%8Cpython",

"from":"AUTO",

"to":"AUTO",

"smartresult":"dict",

"client":"fanyideskweb",

"salt":"1493768163104",

"sign":"4d6f9cedeff67bea71cbcbef20459aff",

"doctype":"json",

"version":"2.1",

"keyfrom":"fanyi.web",

"action":"FY\_BY\_CLICKBUTTON",

"typoResult":"true",

在Raw的第一行中也会看到POST请求的网址

在response下面的JSON中也会看到返回的json格式的内容.

在Response内容中的RAW中也能看到返回的内容.

HTTP/1.1 200 OK

Server: nginx

Date: Wed, 26 Apr 2017 23:25:24 GMT

Content-Type: application/json; charset=utf-8

Connection: keep-alive

Vary: Accept-Encoding

Content-Length: 190

{"translateResult":[[{"tgt":"python","src":"python"}]],"errorCode":0,"type":"en2zh-CHS","smartResult":{"entries":["","n. 巨蟒；大蟒\r\n","n. （法）皮东（人名）\r\n"],"type":1}}

使用www.json.cn来转换json格式

{

"translateResult":[

[

{

"tgt":"python",

"src":"python"

}

]

],

"errorCode":0,

"type":"en2zh-CHS",

"smartResult":{

"entries":[

"",

"n. 巨蟒；大蟒 ",

"n. （法）皮东（人名） "

],

"type":1

}

}

上面我们说了Request请求对象的里有data参数, 它就是用在POST里的, 我们要传送的数据就是这个参数data, data是一个字典, 里面要匹配键值对.

**有道词典翻译网站:**

输入测试数据, 再通过使用Fiddler观察, 其中有一条是POST请求, 而向服务器发送的请求数据并不是在url里, 那么我们可以试着模拟这个POST请求.

python2 urllib版

import urllib

import urllib2

# POST请求的目标URL

url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=ugc&sessionFrom=null"

headers={"User-Agent": "Mozilla...."}

formdata = {

"type":"AUTO",

"i":"i love python",

"doctype":"json",

"xmlVersion":"1.8",

"keyfrom":"fanyi.web",

"ue":"UTF-8",

"action":"FY\_BY\_ENTER",

"typoResult":"true"

}

data = urllib.urlencode(formdata)

request = urllib2.Request(url, data = data, headers = headers)

response = urllib2.urlopen(request)

print response.read()

python3代码

**模拟网页的POST请求实现有道字典接口**

**import** urllib, urllib.request, urllib.parse

*# POST请求的目标URL, 通过抓包的方式获取的url地址, 并不是浏览器的地址栏中显示的url, 也可以在chrome中的network中看到post的地址.*

url = **"http://fanyi.youdao.com/translate\_o?smartresult=dict&smartresult=rule&sessionFrom=null"**

*#完整的headers*

headers = {

"Accept" : "application/json, text/javascript, \*/\*; q=0.01",

"X-Requested-With" : "XMLHttpRequest",

"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",

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

}

*#用户接口输入*

key = input(**"请输入要查询的内容: "**)

*#发送到web服务器的表单数据*

formdata = {

# **"i"**: **"我爱python"**,

**"i"**: **key**,

**"from"**: **"AUTO"**,

**"to"**: **"AUTO"**,

**"smartresult"**: **"dict"**,

**"client"**: **"fanyideskweb"**,

**"salt"**: **"1493768163104"**,

**"sign"**: **"4d6f9cedeff67bea71cbcbef20459aff"**,

**"doctype"**: **"json"**,

**"version"**: **"2.1"**,

**"keyfrom"**: **"fanyi.web"**,

**"action"**: **"FY\_BY\_CLICKBUTTON"**,

**"typoResult"**: **"true"**,

}

*#经过urlencode转码, 发送的时候是以keywork1=value1&keywork2=value2的方式发送的formdata, 所以要使用urlencode转码.*

data = urllib.parse.urlencode(formdata).encode(encoding=**'UTF8'**)

print(data.decode(**"utf-8"**))

*#如果Request()方法里的data参数有值, 那么这个请求就是POST*

request = urllib.request.Request(url, data = data, headers=headers)

response = urllib.request.urlopen(request)

print(response.read().decode(**"utf-8"**))

发送POST请求时, 需要特别注意headers的一些属性:

Content-Length: 144: 是指发送的表单数据长度为144, 也就是字符个数是144个.

X-Requested-With: XMLHttpRequest : 表示Ajax异步请求.

Content-Type: application/x-www-form-urlencoded : 表示浏览器提交 Web 表单时使用, 表单数据会按照 **name1=value1&name2=value2** 键值对形式进行编码.

### 获取AJAX加载的内容

有些网页内容使用AJAX加载, 只要记得, AJAX一般返回的是JSON,直接对AJAX地址进行post或get, 就返回JSON数据了.

"作为一名爬虫工程师, 你最需要关注的, 是数据的来源"

做爬虫最需要关注的不是页面信息, 而是页面信息的数据来源. 所以有的时候并不一定要获取网页上显示的内容.

AJAX方式加载的页面, 数据来源一定是JSON, 拿到了JSON, 就是拿到了网页的数据

访问

https://movie.douban.com/typerank?type\_name=科幻&type=17&interval\_id=100:90&action=

向下拖动加载更多的内容, 此时浏览器中显示的地址始终不发生变化, 这就是使用ajax异步请求的方式加载的数据.

在抓包软件中找到请求的地址:

GET https://movie.douban.com/j/chart/top\_list?type=17&interval\_id=100%3A90&action=&start=20&limit=20 HTTP/1.1

Host: movie.douban.com

Connection: keep-alive

Accept: \*/\*

X-Requested-With: XMLHttpRequest

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.81 Safari/537.36

DNT: 1

Referer: https://movie.douban.com/typerank?type\_name=%E7%A7%91%E5%B9%BB&type=17&interval\_id=100:90&action=

Accept-Encoding: gzip, deflate, sdch, br

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

Cookie:

构建post的form

type: 17

interval\_id: 100:90

action:

start: 0

limit: 20

data = {

"type":"17",

"interval\_id":"100:90",

"action":"",

"start":"0",

"limit":"20"

}

经过urllib.parse.urlencode编码之后的post data

type=17&interval\_id=100%3A90&action=&start=0&limit=20

# demo1

import urllib

import urllib2

url = "https://movie.douban.com/j/chart/top\_list?type=11&interval\_id=100%3A90&action"

headers={"User-Agent": "Mozilla...."}

# 变动的是这两个参数, 从start开始往后显示limit个

formdata = {

'start':'0',

'limit':'10'

}

data = urllib.urlencode(formdata)

request = urllib2.Request(url, data = data, headers = headers)

response = urllib2.urlopen(request)

print response.read()

# demo2

import urllib

import urllib2

url = "https://movie.douban.com/j/chart/top\_list?"

headers={"User-Agent": "Mozilla...."}

# 处理所有参数

formdata = {

'type':'11',

'interval\_id':'100:90',

'action':'',

'start':'0',

'limit':'10'

}

data = urllib.urlencode(formdata)

request = urllib2.Request(url, data = data, headers = headers)

response = urllib2.urlopen(request)

print response.read()

python3代码

**import** urllib.parse, urllib.request

*#https://movie.douban.com/j/chart/top\_list?type=17&interval\_id=100%3A90&action=&start=20&limit=20*

get\_url = **" https://movie.douban.com/j/chart/top\_list?type=17&interval\_id=100%3A90&action"**

headers={**"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

*# 变动的是这两个参数, 从start开始往后显示limit个*

formdata = {

**'start'**:**'0'**,

**'limit'**:**'20'**

}

data = urllib.parse.urlencode(formdata).encode(**"utf-8"**)

request = urllib.request.Request(get\_url, data = data, headers = headers)

response = urllib.request.urlopen(request)

html = response.read()

print(html.decode(**"utf-8"**))

**with** open(**'douban.html'**, **'wb'**) **as** f:

f.write(html)

如果无法写入文件, 把写入文件的代码改为如下

print(html.decode(**"utf-8"**))

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

f.write(html.decode(**"utf-8"**))

问题: 为什么有时候POST也能在URL内看到数据？

* GET方式是直接以链接形式访问, 链接中包含了所有的参数, 服务器端用Request.QueryString获取变量的值. 如果包含了密码的话是一种不安全的选择, 不过你可以直观地看到自己提交了什么内容.
* POST则不会在网址上显示所有的参数, 服务器端用Request.Form获取提交的数据, 在Form提交的时候. 但是HTML代码里如果不指定 method 属性, 则默认为GET请求, Form中提交的数据将会附加在url之后, 以?分开与url分开.
* 表单数据可以作为 URL 字段(method="get") 或者 HTTP POST (method="post") 的方式来发送. 比如在下面的HTML代码中, 表单数据将因为 (method="get") 而附加到 URL 上:

<form action="form\_action.asp" method="get">

<p>First name: <input type="text" name="fname" /></p>

<p>Last name: <input type="text" name="lname" /></p>

<input type="submit" value="Submit" />

</form>

### 处理HTTPS请求 SSL证书验证

现在随处可见 https 开头的网站, urllib2可以为 HTTPS 请求验证SSL证书, 就像web浏览器一样, 如果网站的SSL证书是经过CA认证的, 则能够正常访问, 如: https://www.baidu.com/等...

如果SSL证书验证不通过, 或者操作系统不信任服务器的安全证书, 比如浏览器在访问12306网站如: <https://www.12306.cn/mormhweb/>的时候, 会警告用户证书不受信任. (据说 12306 网站证书是自己做的, 没有通过CA认证)

urllib2在访问的时候则会报出SSLError:

import urllib2

url = "https://www.12306.cn/mormhweb/"

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"}

request = urllib2.Request(url, headers = headers)

response = urllib2.urlopen(request)

print response.read()

运行结果:

urllib2.URLError: <urlopen error [SSL: CERTIFICATE\_VERIFY\_FAILED] certificate verify failed (\_ssl.c:590)>

所以, 如果以后遇到这种网站, 我们需要单独处理SSL证书, 让程序忽略SSL证书验证错误, 即可正常访问.

import urllib

import urllib2

# 1. 导入Python SSL处理模块

import ssl

# 2. 表示忽略未经核实的SSL证书认证

context = ssl.\_create\_unverified\_context()

url = "https://www.12306.cn/mormhweb/"

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"}

request = urllib2.Request(url, headers = headers)

# 3. 在urlopen()方法里 指明添加 context 参数

response = urllib2.urlopen(request, context = context)

print response.read()

### 关于CA

CA(Certificate Authority)是数字证书认证中心的简称, 是指发放、管理、废除数字证书的受信任的第三方机构, 如[北京数字认证股份有限公司](http://www.bjca.org.cn)、[上海市数字证书认证中心有限公司](http://www.sheca.com)等...

CA的作用是检查证书持有者身份的合法性, 并签发证书, 以防证书被伪造或篡改, 以及对证书和密钥进行管理.

现实生活中可以用身份证来证明身份, 那么在网络世界里, 数字证书就是身份证. 和现实生活不同的是, 并不是每个上网的用户都有数字证书的, 往往只有当一个人需要证明自己的身份的时候才需要用到数字证书.

普通用户一般是不需要, 因为网站并不关心是谁访问了网站, 现在的网站只关心流量. 但是反过来, 网站就需要证明自己的身份了.

比如说现在钓鱼网站很多的, 比如你想访问的是[www.baidu.com](http://www.baidu.com), 但其实你访问的是[www.daibu.com](http://www.daibu.com)", 所以在提交自己的隐私信息之前需要验证一下网站的身份, 要求网站出示数字证书.

一般正常的网站都会主动出示自己的数字证书, 来确保客户端和网站服务器之间的通信数据是加密安全的.

## Handler处理器和自定义Opener

* opener是 urllib2.OpenerDirector 的实例, 我们之前一直都在使用的urlopen, 它是一个特殊的opener(也就是模块帮我们构建好的) .
* 但是基本的urlopen()方法不支持代理、cookie等其他的HTTP/HTTPS高级功能. 所以要支持这些功能:
  1. 使用相关的 Handler处理器 来创建特定功能的处理器对象；
  2. 然后通过 urllib2.build\_opener()方法使用这些处理器对象, 创建自定义opener对象；
  3. 使用自定义的opener对象, 调用open()方法发送请求.
* 如果程序里所有的请求都使用自定义的opener, 可以使用urllib2.install\_opener() 将自定义的 opener 对象 定义为 全局opener, 表示如果之后凡是调用urlopen, 都将使用这个opener(根据自己的需求来选择)

### 简单的自定义opener()

import urllib2

# 构建一个HTTPHandler 处理器对象, 支持处理HTTP请求

http\_handler = urllib2.HTTPHandler()

# 构建一个HTTPHandler 处理器对象, 支持处理HTTPS请求

# http\_handler = urllib2.HTTPSHandler()

# 调用urllib2.build\_opener()方法, 创建支持处理HTTP请求的opener对象

opener = urllib2.build\_opener(http\_handler)

# 构建 Request请求

request = urllib2.Request("http://www.baidu.com/")

# 调用自定义opener对象的open()方法, 发送request请求

response = opener.open(request)

# 获取服务器响应内容

print response.read()

python3代码

**import** urllib.request

*# 构建一个HTTPHandler 处理器对象, 支持处理HTTP请求, 同时开启Debug Log, debuglevel 值默认 0*

http\_handler = urllib.request.HTTPHandler(debuglevel=1)

*# 构建一个HTTPSHandler 处理器对象, 支持处理HTTPS请求, 同时开启Debug Log, debuglevel 值默认 0*

https\_handler = urllib.request.HTTPSHandler(debuglevel=1)

*# 调用urllib.request.build\_opener()方法, 创建支持处理HTTP请求的opener对象*

opener = urllib.request.build\_opener(http\_handler)

*# 构建 Request请求*

request = urllib.request.Request(**"http://www.baidu.com/"**)

*# 调用自定义opener对象的open()方法, 发送request请求*

response = opener.open(request)

*# 获取服务器响应内容*

print(response.read().decode("utf-8"))

这种方式发送请求得到的结果, 和使用urllib2.urlopen()发送HTTP/HTTPS请求得到的结果是一样的.

如果在 HTTPHandler()增加 debuglevel=1参数, 还会将 Debug Log 打开, 这样程序在执行的时候, 会把收包和发包的报头在屏幕上自动打印出来, 方便调试, 有时可以省去抓包的工作.

# 仅需要修改的代码部分:

# 构建一个HTTPHandler 处理器对象, 支持处理HTTP请求, 同时开启Debug Log, debuglevel 值默认 0

http\_handler = urllib2.HTTPHandler(debuglevel=1)

# 构建一个HTTPHSandler 处理器对象, 支持处理HTTPS请求, 同时开启Debug Log, debuglevel 值默认 0

https\_handler = urllib2.HTTPSHandler(debuglevel=1)

### 使用ProxyHandler向opener添加代理

使用代理IP, 这是爬虫/反爬虫的第二大招, 通常也是最好用的.

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.

urllib2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:

#urllib2\_proxy1.py

import urllib2

# 构建了两个代理Handler, 一个有代理IP, 一个没有代理IP

httpproxy\_handler = urllib2.ProxyHandler({"http" : "124.88.67.81:80"})

nullproxy\_handler = urllib2.ProxyHandler({})

proxySwitch = True #定义一个代理开关

# 通过 urllib2.build\_opener()方法使用这些代理Handler对象, 创建自定义opener对象

# 根据代理开关是否打开, 使用不同的代理模式

if proxySwitch:

opener = urllib2.build\_opener(httpproxy\_handler)

else:

opener = urllib2.build\_opener(nullproxy\_handler)

request = urllib2.Request("http://www.baidu.com/")

# 1. 如果这么写, 只有使用opener.open()方法发送请求才使用自定义的代理, 而urlopen()则不使用自定义代理.

response = opener.open(request)

# 2. 如果这么写, 就是将opener应用到全局, 之后所有的, 不管是opener.open()还是urlopen() 发送请求, 都将使用自定义代理.

# urllib2.install\_opener(opener)

# response = urlopen(request)

print response.read()

**使用ProxyHandler向opener添加代理**

python3代码

**import** urllib.request, urllib.parse

*# 构建了两个代理Handler, 一个有代理IP, 一个没有代理IP*

httpproxy\_handler = urllib.request.ProxyHandler({**"http"** : **"61.191.173.31:808"**})

nullproxy\_handler = urllib.request.ProxyHandler({})

*#定义一个代理开关*

proxySwitch = **True**

*# 通过 urllib.request.build\_opener()方法使用这些代理Handler对象, 创建自定义opener对象*

*# 根据代理开关是否打开, 使用不同的代理模式*

**if** proxySwitch:

opener = urllib.request.build\_opener(httpproxy\_handler)

**else**:

opener = urllib.request.build\_opener(nullproxy\_handler)

request = urllib.request.Request(**"http://www.baidu.com/"**)

*# 1. 如果这么写, 只有使用opener.open()方法发送请求才使用自定义的代理, 而urlopen()则不使用自定义代理.*

response = opener.open(request)

*# 2. 如果这么写, 就是将opener应用到全局, 之后所有的, 不管是opener.open()还是urlopen() 发送请求, 都将使用自定义代理.*

*# urllib.request.install\_opener(opener)*

*# response = urllib.parse.urlopen(request)*

print(response.read().decode(**"utf-8"**))

<html>

<head><meta http-equiv="Content-Type" content="text/html; charset=utf-8" /></head>

<body>

<h1>Unauthorized ...</h1>

<h2>

IP Address: 180.168.188.185<br>

MAC Address: <br>

Server Time: 2017-05-03 09:36:25<br>

Auth Result: 无效用户.

</h2>

</body>

</html>

免费的开放代理获取基本没有成本, 我们可以在一些代理网站上收集这些免费代理, 测试后如果可以用, 就把它收集起来用在爬虫上面.

免费短期代理网站举例:

* [西刺免费代理IP](http://www.xicidaili.com/)
* [快代理免费代理](http://www.kuaidaili.com/free/inha/)
* [Proxy360代理](http://www.proxy360.cn/default.aspx)
* [全网代理IP](http://www.goubanjia.com/free/index.shtml)

如果代理IP足够多, 就可以像随机获取User-Agent一样, 随机选择一个代理去访问网站.

import urllib2

import random

proxy\_list = [

{"http" : "124.88.67.81:80"},

{"http" : "124.88.67.81:80"},

{"http" : "124.88.67.81:80"},

{"http" : "124.88.67.81:80"},

{"http" : "124.88.67.81:80"}

]

# 随机选择一个代理

proxy = random.choice(proxy\_list)

# 使用选择的代理构建代理处理器对象

httpproxy\_handler = urllib2.ProxyHandler(proxy)

opener = urllib2.build\_opener(httpproxy\_handler)

request = urllib2.Request("http://www.baidu.com/")

response = opener.open(request)

print response.read()

**python3随机选择代理地址**

**import** urllib.request,random

proxy\_list = [

{**"http"** : **"124.88.67.81:80"**},

{**"http"** : **"124.88.67.81:80"**},

{**"http"** : **"124.88.67.81:80"**},

{**"http"** : **"124.88.67.81:80"**},

{**"http"** : **"124.88.67.81:80"**}

]

*# 随机选择一个代理*

proxy = random.choice(proxy\_list)

*# 使用选择的代理构建代理处理器对象*

httpproxy\_handler = urllib.request.ProxyHandler(proxy)

opener = urllib.request.build\_opener(httpproxy\_handler)

request = urllib.request.Request(**"http://www.baidu.com/"**)

response = opener.open(request)

print(response.read())

但是, 这些免费开放代理一般会有很多人都在使用, 而且代理有寿命短, 速度慢, 匿名度不高, HTTP/HTTPS支持不稳定等缺点(免费没好货) .

所以, 专业爬虫工程师或爬虫公司会使用高品质的私密代理, 这些代理通常需要找专门的代理供应商购买, 再通过用户名/密码授权使用(舍不得孩子套不到狼) .

### HTTPPasswordMgrWithDefaultRealm()

HTTPPasswordMgrWithDefaultRealm()类将创建一个密码管理对象, 用来保存 HTTP 请求相关的用户名和密码, 主要应用两个场景:

1. 验证代理授权的用户名和密码 (ProxyBasicAuthHandler())
2. 验证Web客户端的的用户名和密码 (HTTPBasicAuthHandler())

#### ProxyBasicAuthHandler(代理授权验证)

如果我们使用之前的代码来使用私密代理, 会报 HTTP 407 错误, 表示代理没有通过身份验证:

urllib2.HTTPError: HTTP Error 407: Proxy Authentication Required

所以我们需要改写代码, 通过:

* HTTPPasswordMgrWithDefaultRealm(): 来保存私密代理的用户密码
* ProxyBasicAuthHandler(): 来处理代理的身份验证.

#urllib2\_proxy2.py

#urllib2\_proxy2.py

import urllib2

import urllib

# 私密代理授权的账户

user = "mr\_mao\_hacker"

# 私密代理授权的密码

passwd = "sffqry9r"

# 私密代理 IP

proxyserver = "61.158.163.130:16816"

# 1. 构建一个密码管理对象, 用来保存需要处理的用户名和密码

passwdmgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# 2. 添加账户信息, 第一个参数realm是与远程服务器相关的域信息, 一般没人管它都是写None, 后面三个参数分别是 代理服务器、用户名、密码

passwdmgr.add\_password(None, proxyserver, user, passwd)

# 3. 构建一个代理基础用户名/密码验证的ProxyBasicAuthHandler处理器对象, 参数是创建的密码管理对象

# 注意, 这里不再使用普通ProxyHandler类了

proxyauth\_handler = urllib2.ProxyBasicAuthHandler(passwdmgr)

# 工作中常用这种方式直接创建一个附带私密代理验证的处理器

# 如果使用这种方式, 则上面3步不需要操作.

# authproxy\_handler = urllib2.ProxyHandler({"http" : "mr\_mao\_hacker:sffqry9r@114.215.104.49:16816"})

# 开放代理

# authproxy\_handler = urllib2.ProxyHandler({"http" : "114.215.104.49:16816"})

# 4. 通过 build\_opener()方法使用这些代理Handler对象, 创建自定义opener对象, 参数包括构建的 proxy\_handler 和 proxyauth\_handler

opener = urllib2.build\_opener(proxyauth\_handler)

#这是直接使用创建到的代理处理器的用法

#opener = urllib2.build\_opener(authproxy\_handler)

# 5. 构造Request 请求

request = urllib2.Request("http://www.baidu.com/")

# 6. 使用自定义opener发送请求

response = opener.open(request)

# 7. 打印响应内容

print response.read()

1. 1. 6工作中常用这种方式直接创建一个附带私密代理验证的处理器

如果使用这种方式, 则上面3步不需要操作.

#authproxy\_handler = urllib2.ProxyHandler({"http" : "mr\_mao\_hacker:sffqry9r@114.215.104.49:16816"})

#开放代理

#authproxy\_handler = urllib2.ProxyHandler({"http" : "114.215.104.49:16816"})

#### HTTPBasicAuthHandler处理器(Web客户端授权验证)

有些Web服务器(包括HTTP/FTP等) 访问时, 需要进行用户身份验证, 爬虫直接访问会报HTTP 401 错误, 表示访问身份未经授权:

urllib2.HTTPError: HTTP Error 401: Unauthorized

如果我们有客户端的用户名和密码, 我们可以通过下面的方法去访问爬取:

import urllib

import urllib2

# 用户名

user = "test"

# 密码

passwd = "123456"

# Web服务器 IP

webserver = "http://192.168.199.107"

# 1. 构建一个密码管理对象, 用来保存需要处理的用户名和密码

passwdmgr = urllib2.HTTPPasswordMgrWithDefaultRealm()

# 2. 添加账户信息, 第一个参数realm是与远程服务器相关的域信息, 一般没人管它都是写None, 后面三个参数分别是 Web服务器、用户名、密码

passwdmgr.add\_password(None, webserver, user, passwd)

# 3. 构建一个HTTP基础用户名/密码验证的HTTPBasicAuthHandler处理器对象, 参数是创建的密码管理对象

httpauth\_handler = urllib2.HTTPBasicAuthHandler(passwdmgr)

# 4. 通过 build\_opener()方法使用这些代理Handler对象, 创建自定义opener对象, 参数包括构建的 proxy\_handler

opener = urllib2.build\_opener(httpauth\_handler)

# 5. 可以选择通过install\_opener()方法定义opener为全局opener

urllib2.install\_opener(opener)

# 6. 构建 Request对象

request = urllib2.Request("http://192.168.199.107")

# 7. 定义opener为全局opener后, 可直接使用urlopen()发送请求

response = urllib2.urlopen(request)

# 8. 打印响应内容

print response.read()

**python3使用ProxyHandler向opener添加需要验证的代理**

import urllib.request

*# 使用需要用户名和密码验证的代理构建代理处理器对象*

auth\_proxy\_handler = urllib.request.ProxyHandler({"http":"mr\_mao\_hacker:sfqry9r@114.215.49:16815"})

*#构造一个自定义的opener*

opener = urllib.request.build\_opener(auth\_proxy\_handler)

*#构建请求*

request = urllib.request.Request("http://www.baidu.com/")

*#获取响应*

response = opener.open(request)

print(response.read())

一般要写一个文件, 把代理放进去, 然后引入这个文件就行了.

### Cookie

Cookie 是指某些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.

#### Cookie原理

HTTP是无状态的面向连接的协议, 为了保持连接状态, 引入了Cookie机制 Cookie是http消息头中的一种属性, 包括:

Cookie名字(Name)

Cookie的值(Value)

Cookie的过期时间(Expires/Max-Age)

Cookie作用路径(Path)

Cookie所在域名(Domain) ,

使用Cookie进行安全连接(Secure) .

前两个参数是Cookie应用的必要条件, 另外, 还包括Cookie大小(Size, 不同浏览器对Cookie个数及大小限制是有差异的) .

Cookie由变量名和值组成, 根据 Netscape公司的规定, Cookie格式如下:

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

#### Cookie应用

Cookies在爬虫方面最典型的应用是判定注册用户是否已经登录网站, 用户可能会得到提示, 是否在下一次进入此网站时保留用户信息以便简化登录手续.

# 获取一个有登录信息的Cookie模拟登陆

import urllib2

# 1. 构建一个已经登录过的用户的headers信息

headers = {

"Host":"www.renren.com",

"Connection":"keep-alive",

"Upgrade-Insecure-Requests":"1",

"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",

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

"Accept-Language":"zh-CN,zh;q=0.8,en;q=0.6",

# 便于终端阅读, 表示不支持压缩文件

# Accept-Encoding: gzip, deflate, sdch,

# 重点: 这个Cookie是保存了密码无需重复登录的用户的Cookie, 这个Cookie里记录了用户名, 密码(通常经过RAS加密)

"Cookie": "anonymid=ixrna3fysufnwv; depovince=GW; \_r01\_=1; JSESSIONID=abcmaDhEdqIlM7riy5iMv; jebe\_key=f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1484060607173; jebecookies=26fb58d1-cbe7-4fc3-a4ad-592233d1b42e|||||; ick\_login=1f2b895d-34c7-4a1d-afb7-d84666fad409; \_de=BF09EE3A28DED52E6B65F6A4705D973F1383380866D39FF5; p=99e54330ba9f910b02e6b08058f780479; ap=327550029; first\_login\_flag=1; ln\_uact=mr\_mao\_hacker@163.com; ln\_hurl=http://hdn.xnimg.cn/photos/hdn521/20140529/1055/h\_main\_9A3Z\_e0c300019f6a195a.jpg; t=214ca9a28f70ca6aa0801404dda4f6789; societyguester=214ca9a28f70ca6aa0801404dda4f6789; id=327550029; xnsid=745033c5; ver=7.0; loginfrom=syshome"

}

#一个好友的主页, 必须要在登陆的状态下才能查看

url = "http://www.renren.com/221510635/profile"

# 2. 通过headers里的报头信息(主要是Cookie信息) , 构建Request对象

request = urllib2.Request(url, headers = headers)

# 3. 直接访问renren主页, 服务器会根据headers报头信息(主要是Cookie信息) , 判断这是一个已经登录的用户, 并返回相应的页面

response = urllib2.urlopen(request)

# 4. 打印响应内容

print response.read()

python3代码

在fiddler中抓取登陆时的信息, 切换到[www.renren.com/home](http://www.renren.com/home)中查看更加详细的cookies信息. 必须使用这个详细的cookies信息才能正常登陆

**import** urllib.request

*# 获取一个有登录信息的Cookie模拟登陆*

*# 1. 构建一个已经登录过的用户的headers信息*

headers = {

**"Host"**:**"www.renren.com"**,

**"Connection"**:**"keep-alive"**,

**"Upgrade-Insecure-Requests"**:**"1"**,

**"User-Agent"**:**"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.81 Safari/537.36"**,

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

**"DNT"**:**"1"**,

**"Referer"**:**"http://www.renren.com/"**,

*# 便于终端阅读, 表示不支持压缩文件*

*# "Accept-Encoding":"gzip, deflate, sdch",*

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

*# 重点: 这个Cookie是保存了密码无需重复登录的用户的Cookie, 这个Cookie里记录了用户名, 密码(通常经过RAS加密)*

**"Cookie"**:**"anonymid=j28c82hf-60w88b; depovince=SH; \_r01\_=1; JSESSIONID=abcAXL6QbKWOBh4h4cmVv; ick=1b520c56-e8d1-4064-926f-6abc07948d24; jebe\_key=b9f5f520-f401-4faf-bd9d-079ba767677f%7C5107c503631c2f3b3a253f5eaf1353d4%7C1493777234621%7C1%7C1493777245464; XNESSESSIONID=18fb5878a74e; wp=0; wp\_fold=0; jebecookies=7a99e34b-0c54-4e27-ab45-c144a7fe8a3b|||||; ick\_login=9249fa41-f5f9-484b-90e9-7f2eb455c490; \_de=ADFFD137A72EE61740A09AB7395DD6151383380866D39FF5; p=282f16f30f2d66e6951558a5a0bdc8731; first\_login\_flag=1; ln\_uact=superwuzhiwei@163.com; ln\_hurl=http://hd10.xiaonei.com/photos/hd10/20070813/16/28/main\_1159a77.jpg; t=89cd4ff529950167ed187ce138ca344a1; societyguester=89cd4ff529950167ed187ce138ca344a1; id=221258551; xnsid=33e0df01; loginfrom=syshome"**,

}

*#一个好友的主页, 必须要在登陆的状态下才能查看*

url = **"http://www.renren.com/221510635/profile"**

*# 2. 通过headers里的报头信息（主要是Cookie信息）, 构建Request对象*

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

*# 3. 直接访问renren主页, 服务器会根据headers报头信息（主要是Cookie信息）, 判断这是一个已经登录的用户, 并返回相应的页面*

response = urllib.request.urlopen(request).read().decode(**"utf-8"**)

*# 4. 打印响应内容*

print(response)

但是这样做太过复杂, 我们先需要在浏览器登录账户, 并且设置保存密码, 并且通过抓包才能获取这个Cookie, 那有么有更简单方便的方法呢？最好是直接使用用户名和密码在程序中登陆, 然后程序自动保存cookies的值.

### cookielib库 和 HTTPCookieProcessor处理器

在Python处理Cookie, 一般是通过cookielib模块和 urllib2模块的HTTPCookieProcessor处理器类一起使用.

cookielib模块: 主要作用是提供用于存储cookie的对象

HTTPCookieProcessor处理器: 主要作用是处理这些cookie对象, 并构建handler对象.

模拟登陆的过程, 首先要有一个post过程, 使用post传递账号名和密码进行登陆. 登陆成功就会生成cookie值, 把生成的cookie值保存下来用于下一次登陆网站, 即get获取其它页面.

#### cookielib 库

该模块主要的对象有CookieJar、FileCookieJar、MozillaCookieJar、LWPCookieJar.

* 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.
* 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.
* 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.
* 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.

**其实大多数情况下, 我们只用CookieJar(), 如果需要和本地文件交互, 就用 MozillaCookjar() 或 LWPCookieJar()**

我们来做几个案例:

**1) 获取Cookie, 并保存到CookieJar()对象中**

# urllib2\_cookielibtest1.py

import urllib2

import cookielib

# 构建一个CookieJar对象实例来保存cookie

cookiejar = cookielib.CookieJar()

# 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象

handler=urllib2.HTTPCookieProcessor(cookiejar)

# 通过 build\_opener() 来构建opener

opener = urllib2.build\_opener(handler)

# 4. 以get方法访问页面, 访问之后会自动保存cookie到cookiejar中

opener.open("http://www.baidu.com")

## 可以按标准格式将保存的Cookie打印出来

cookieStr = ""

for item in cookiejar:

cookieStr = cookieStr + item.name + "=" + item.value + ";"

## 舍去最后一位的分号

print cookieStr[:-1]

我们使用以上方法将Cookie保存到cookiejar对象中, 然后打印出了cookie中的值, 也就是访问百度首页的Cookie值.

运行结果如下:

BAIDUID=4327A58E63A92B73FF7A297FB3B2B4D0:FG=1;BIDUPSID=4327A58E63A92B73FF7A297FB3B2B4D0;H\_PS\_PSSID=1429\_21115\_17001\_21454\_21409\_21554\_21398;PSTM=1480815736;BDSVRTM=0;BD\_HOME=0

**python3代码**

**获取Cookie并保存到CookieJar()对象中**

# urllib.request\_http.cookiejartest1.py

**import** urllib.request, http.cookiejar

# 通过CookieJar()类构建一个CookieJar对象实例, 用来保存cookie

cookiejar = http.cookiejar.CookieJar()

# 使用HTTPCookieProcessor()来创建cookie处理器对象, 用来处理cookie, 参数为CookieJar()对象

handler=urllib.request.HTTPCookieProcessor(cookiejar)

# 通过 build\_opener() 来构建opener

opener = urllib.request.build\_opener(handler)

#使用addheaders函数给opener添加headers信息

opener.addheaders = [(**"User-Agent"**, **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**)]

url = **"http://www.zhihu.com/"**

# 以get方法访问页面, 访问之后会自动保存cookie到cookiejar中

response = opener.open(url)

# print(cookiejar)

#可以按标准格式将保存的Cookie打印出来

cookieStr = **""**

**for** item **in** cookiejar:

cookieStr = cookieStr + item.name + **"="** + item.value + **";"**

#舍弃最后一位的分号

print(cookieStr[:-1])

**2. 访问网站获得cookie, 并把获得的cookie保存在cookie文件中**

# urllib2\_cookielibtest2.py

import cookielib

import urllib2

# 保存cookie的本地磁盘文件名

filename = 'cookie.txt'

# 声明一个MozillaCookieJar(有save实现)对象实例来保存cookie, 之后写入文件

cookiejar = cookielib.MozillaCookieJar(filename)

# 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象

handler = urllib2.HTTPCookieProcessor(cookiejar)

# 通过 build\_opener() 来构建opener

opener = urllib2.build\_opener(handler)

# 创建一个请求, 原理同urllib2的urlopen

response = opener.open("http://www.baidu.com")

# 保存cookie到本地文件

cookiejar.save()

python3代码, **获取Cookie并保存到本地文件中**

# urllib.request\_http.cookiejartest2.py

**import** http.cookiejar, urllib.request

*# 保存cookie的本地磁盘文件名*

filename = **'cookie.txt'**

*# 声明一个MozillaCookieJar(有save实现)对象实例来保存cookie, 之后写入文件*

cookiejar = http.cookiejar.MozillaCookieJar(filename)

*# 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象*

handler = urllib.request.HTTPCookieProcessor(cookiejar)

*# 通过 build\_opener() 来构建opener*

opener = urllib.request.build\_opener(handler)

*#使用addheaders函数给opener添加headers信息*

opener.addheaders = [(**"User-Agent"**, **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**)]

url = **"http://www.zhihu.com/"**

*# 创建一个请求, 原理同urllib.request的urlopen*

response = opener.open(url)

*# 保存cookie到本地文件*

cookiejar.save()

**3. 从文件中获取cookies, 做为请求的一部分去访问**

# urllib2\_cookielibtest2.py

import cookielib

import urllib2

# 创建MozillaCookieJar(有load实现)实例对象

cookiejar = cookielib.MozillaCookieJar()

# 从文件中读取cookie内容到变量

cookie.load('cookie.txt', ignore\_discard=True)

# 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象

handler = urllib2.HTTPCookieProcessor(cookiejar)

# 通过 build\_opener() 来构建opener

opener = urllib2.build\_opener(handler)

response = opener.open("http://www.baidu.com")

**利用cookielib和post登录人人网**

import urllib

import urllib2

import cookielib

# 1. 构建一个CookieJar对象实例来保存cookie

cookie = cookielib.CookieJar()

# 2. 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象

cookie\_handler = urllib2.HTTPCookieProcessor(cookie)

# 3. 通过 build\_opener() 来构建opener

opener = urllib2.build\_opener(cookie\_handler)

# 4. addheaders 接受一个列表, 里面每个元素都是一个headers信息的元祖, opener将附带headers信息

opener.addheaders = [("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")]

# 5. 需要登录的账户和密码

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

# 6. 通过urlencode()转码

postdata = urllib.urlencode(data)

# 7. 构建Request请求对象, 包含需要发送的用户名和密码

request = urllib2.Request("http://www.renren.com/PLogin.do", data = postdata)

# 8. 通过opener发送这个请求, 并获取登录后的Cookie值,

opener.open(request)

# 9. opener包含用户登录后的Cookie值, 可以直接访问那些登录后才可以访问的页面

response = opener.open("http://www.renren.com/410043129/profile")

# 10. 打印响应内容

print response.read()

python3代码

利用http.cookiejar和post登录人人网

# urllib.request\_http.cookiejartest2.py

**import** urllib.request, urllib.parse, http.cookiejar

*# 1. 构建一个CookieJar对象实例来保存cookie*

cookie = http.cookiejar.CookieJar()

*# 2. 使用HTTPCookieProcessor()来创建cookie处理器对象, 参数为CookieJar()对象*

cookie\_handler = urllib.request.HTTPCookieProcessor(cookie)

*# 3. 通过 build\_opener() 来构建opener*

opener = urllib.request.build\_opener(cookie\_handler)

*# 4. addheaders 接受一个列表, 里面每个元素都是一个headers信息的元祖, opener将附带headers信息*

opener.addheaders = [(**"User-Agent"**, **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**)]

*# 5. 需要登录的账户和密码*

data = {**"email"**:**"superwuzhiwei@163.com"**, **"password"**:**"w6u9y5a3n8zi"**}

*# 6. 通过urlencode()转码*

postdata = urllib.parse.urlencode(data).encode(**"utf-8"**)

*# bianary\_data=data.encode('utf-8')*

*# 7. 构建Request请求对象, 包含需要发送的用户名和密码*

request = urllib.request.Request(**"http://www.renren.com/PLogin.do"**, data = postdata, method = "POST")

*# 8. 通过opener发送这个请求, 并获取登录后的Cookie值, 因为request中有data的数值, 所以此请求为POST请求*

opener.open(request)

friend\_url = **"http://www.renren.com/221510635/profile"**

*# 9. opener包含用户登录后的Cookie值, 可以直接访问那些登录后才可以访问的页面. 此次请求是get请求*

response = opener.open(friend\_url).read().decode(**"utf-8"**)

*# 10. 打印响应内容*

print(response.getcode())

模拟登录要注意几点:

1. 登录一般都会先有一个HTTP GET, 用于拉取一些信息及获得Cookie, 然后再HTTP POST登录.
2. HTTP POST登录的链接有可能是动态的, 从GET返回的信息中获取.
3. password 有些是明文发送, 有些是加密后发送. 有些网站甚至采用动态加密的, 同时包括了很多其他数据的加密信息, 只能通过查看JS源码获得加密算法, 再去破解加密, 非常困难.
4. 大多数网站的登录整体流程是类似的, 可能有些细节不一样, 所以不能保证其他网站登录成功.

这个测试案例中, 为了想让大家快速理解知识点, 我们使用的人人网登录接口是人人网改版前的隐藏接口(嘘....), 登录比较方便.

当然, 我们也可以直接发送账号密码到登录界面模拟登录, 但是当网页采用JavaScript动态技术以后, 想封锁基于 HttpClient 的模拟登录就太容易了, 甚至可以根据你的鼠标活动的特征准确地判断出是不是真人在操作.

所以, 想做通用的模拟登录还得选别的技术, 比如用内置浏览器引擎的爬虫(关键词: Selenium , PhantomJS), 这个我们将在以后会学习到.

## urllib2: URLError与HTTPError

### urllib2 的异常错误处理

在我们用urlopen或opener.open方法发出一个请求时, 如果urlopen或opener.open不能处理这个response, 就产生错误.

这里主要说的是URLError和HTTPError, 以及对它们的错误处理.

#### URLError

URLError 产生的原因主要有:

1. 没有网络连接
2. 服务器连接失败
3. 找不到指定的服务器

我们可以用try except语句来捕获相应的异常. 下面的例子里我们访问了一个不存在的域名:

# urllib2\_urlerror.py

import urllib2

requset = urllib2.Request('http://www.ajkfhafwjqh.com')

try:

urllib2.urlopen(request, timeout=5)

except urllib2.URLError, err:

print err

运行结果如下:

<urlopen error [Errno 8] nodename nor servname provided, or not known>

urlopen error, 错误代码8, 错误原因是没有找到指定的服务器.

#### HTTPError

HTTPError是URLError的子类, 我们发出一个请求时, 服务器上都会对应一个response应答对象, 其中它包含一个数字"响应状态码".

如果urlopen或opener.open不能处理的, 会产生一个HTTPError, 对应相应的状态码, HTTP状态码表示HTTP协议所返回的响应的状态.

**注意, urllib2可以为我们处理重定向的页面(也就是3开头的响应码) , 100-299范围的号码表示成功, 所以我们只能看到400-599的错误号码.**

# urllib2\_httperror.py

import urllib2

requset = urllib2.Request('http://blog.baidu.com/itcast')

try:

urllib2.urlopen(requset)

except urllib2.HTTPError, err:

print err.code

print err

运行结果如下:

404

HTTP Error 404: Not Found

HTTP Error, 错误代号是404, 错误原因是Not Found, 说明服务器无法找到被请求的页面.

通常产生这种错误的, 要么url不对, 要么ip被封.

#### 改进版

由于HTTPError的父类是URLError, 所以父类的异常应当写到子类异常的后面, 所以上述的代码可以这么改写:

# urllib2\_botherror.py

import urllib2

requset = urllib2.Request('http://blog.baidu.com/itcast')

try:

urllib2.urlopen(requset)

except urllib2.HTTPError, err:

print err.code

except urllib2.URLError, err:

print err

else:

print "Good Job"

运行结果如下:

404

这样我们就可以做到, 首先捕获子类的异常, 如果子类捕获不到, 那么可以捕获父类的异常.

#### HTTP响应状态码参考:

1xx:信息

100 Continue

服务器仅接收到部分请求, 但是一旦服务器并没有拒绝该请求, 客户端应该继续发送其余的请求.

101 Switching Protocols

服务器转换协议: 服务器将遵从客户的请求转换到另外一种协议.

2xx:成功

200 OK

请求成功(其后是对GET和POST请求的应答文档)

201 Created

请求被创建完成, 同时新的资源被创建.

202 Accepted

供处理的请求已被接受, 但是处理未完成.

203 Non-authoritative Information

文档已经正常地返回, 但一些应答头可能不正确, 因为使用的是文档的拷贝.

204 No Content

没有新文档. 浏览器应该继续显示原来的文档. 如果用户定期地刷新页面, 而Servlet可以确定用户文档足够新, 这个状态代码是很有用的.

205 Reset Content

没有新文档. 但浏览器应该重置它所显示的内容. 用来强制浏览器清除表单输入内容.

206 Partial Content

客户发送了一个带有Range头的GET请求, 服务器完成了它.

3xx:重定向

300 Multiple Choices

多重选择. 链接列表. 用户可以选择某链接到达目的地. 最多允许五个地址.

301 Moved Permanently

所请求的页面已经转移至新的url.

302 Moved Temporarily

所请求的页面已经临时转移至新的url.

303 See Other

所请求的页面可在别的url下被找到.

304 Not Modified

未按预期修改文档. 客户端有缓冲的文档并发出了一个条件性的请求(一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档) . 服务器告诉客户, 原来缓冲的文档还可以继续使用.

305 Use Proxy

客户请求的文档应该通过Location头所指明的代理服务器提取.

306 Unused

此代码被用于前一版本. 目前已不再使用, 但是代码依然被保留.

307 Temporary Redirect

被请求的页面已经临时移至新的url.

4xx:客户端错误

400 Bad Request

服务器未能理解请求.

401 Unauthorized

被请求的页面需要用户名和密码.

401.1

登录失败.

401.2

服务器配置导致登录失败.

401.3

由于 ACL 对资源的限制而未获得授权.

401.4

筛选器授权失败.

401.5

ISAPI/CGI 应用程序授权失败.

401.7

访问被 Web 服务器上的 URL 授权策略拒绝. 这个错误代码为 IIS 6.0 所专用.

402 Payment Required

此代码尚无法使用.

403 Forbidden

对被请求页面的访问被禁止.

403.1

执行访问被禁止.

403.2

读访问被禁止.

403.3

写访问被禁止.

403.4

要求 SSL.

403.5

要求 SSL 128.

403.6

IP 地址被拒绝.

403.7

要求客户端证书.

403.8

站点访问被拒绝.

403.9

用户数过多.

403.10

配置无效.

403.11

密码更改.

403.12

拒绝访问映射表.

403.13

客户端证书被吊销.

403.14

拒绝目录列表.

403.15

超出客户端访问许可.

403.16

客户端证书不受信任或无效.

403.17

客户端证书已过期或尚未生效.

403.18

在当前的应用程序池中不能执行所请求的 URL. 这个错误代码为 IIS 6.0 所专用.

403.19

不能为这个应用程序池中的客户端执行 CGI. 这个错误代码为 IIS 6.0 所专用.

403.20

Passport 登录失败. 这个错误代码为 IIS 6.0 所专用.

404 Not Found

服务器无法找到被请求的页面.

404.0

没有找到文件或目录.

404.1

无法在所请求的端口上访问 Web 站点.

404.2

Web 服务扩展锁定策略阻止本请求.

404.3

MIME 映射策略阻止本请求.

405 Method Not Allowed

请求中指定的方法不被允许.

406 Not Acceptable

服务器生成的响应无法被客户端所接受.

407 Proxy Authentication Required

用户必须首先使用代理服务器进行验证, 这样请求才会被处理.

408 Request Timeout

请求超出了服务器的等待时间.

409 Conflict

由于冲突, 请求无法被完成.

410 Gone

被请求的页面不可用.

411 Length Required

"Content-Length" 未被定义. 如果无此内容, 服务器不会接受请求.

412 Precondition Failed

请求中的前提条件被服务器评估为失败.

413 Request Entity Too Large

由于所请求的实体的太大, 服务器不会接受请求.

414 Request-url Too Long

由于url太长, 服务器不会接受请求. 当post请求被转换为带有很长的查询信息的get请求时, 就会发生这种情况.

415 Unsupported Media Type

由于媒介类型不被支持, 服务器不会接受请求.

416 Requested Range Not Satisfiable

服务器不能满足客户在请求中指定的Range头.

417 Expectation Failed

执行失败.

423

锁定的错误.

5xx:服务器错误

500 Internal Server Error

请求未完成. 服务器遇到不可预知的情况.

500.12

应用程序正忙于在 Web 服务器上重新启动.

500.13

Web 服务器太忙.

500.15

不允许直接请求 Global.asa.

500.16

UNC 授权凭据不正确. 这个错误代码为 IIS 6.0 所专用.

500.18

URL 授权存储不能打开. 这个错误代码为 IIS 6.0 所专用.

500.100

内部 ASP 错误.

501 Not Implemented

请求未完成. 服务器不支持所请求的功能.

502 Bad Gateway

请求未完成. 服务器从上游服务器收到一个无效的响应.

502.1

CGI 应用程序超时. 　·

502.2

CGI 应用程序出错.

503 Service Unavailable

请求未完成. 服务器临时过载或当机.

504 Gateway Timeout

网关超时.

505 HTTP Version Not Supported

服务器不支持请求中指明的HTTP协议版本

## Requests模块

### Requests: 让 HTTP 服务人类

虽然Python的标准库中 urllib2 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 自称 "HTTP for Humans", 说明使用更简洁方便.

Requests 唯一的一个非转基因的 Python HTTP 库, 人类可以安全享用: )

Requests 继承了urllib2的所有特性. Requests支持HTTP连接保持和连接池, 支持使用cookie保持会话, 支持文件上传, 支持自动确定响应内容的编码, 支持国际化的 URL 和 POST 数据自动编码.

### requests 的底层实现其实就是 urllib3

Requests的文档非常完备, 中文文档也相当不错. Requests能完全满足当前网络的需求, 支持Python 2.6—3.5, 而且能在PyPy下完美运行.

开源地址: <https://github.com/kennethreitz/requests>

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

### 安装方式

利用 pip 安装 或者利用 easy\_install 都可以完成安装:

$ pip install requests

$ easy\_install requests

### 基本GET请求(headers参数 和 parmas参数)

#### 1. 最基本的GET请求可以直接用get方法

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

# 也可以这么写

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

#### 2. 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.

**import** requests

kw = {**'wd'**:**'长城'**}

headers = {**"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

*# 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(**"https://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)

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

f.write(response.text)

*# # 查看响应头部字符编码*

print(response.encoding)

*#*

*# # 查看响应码*

print(response.status\_code)

运行结果

......

......

'http://www.baidu.com/s?wd=%E9%95%BF%E5%9F%8E'

'utf-8'

200

* 使用response.text 时, Requests 会基于 HTTP 响应的文本编码自动解码响应内容, 大多数 Unicode 字符集都能被无缝地解码.
* 使用response.content 时, 返回的是服务器响应数据的原始二进制字节流, 可以用来保存图片等二进制文件.

python3代码

**import** requests

kw = {**'wd'**:**'长城'**}

headers = {**"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

*# 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(**"https://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)

with open("baidu1.html", "w", encoding="utf-8") as f:

f.write(response.text)

with open("baidu2.html", "wb") as f:

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

with open("baidu3.html", "wb") as f:

f.write(response.*content*)

*# # 查看响应头部字符编码*

print(response.encoding)

*#*

*# # 查看响应码*

print(response.status\_code)

### 通过requests获取新浪首页

#coding=utf-8

import requests

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

print(response.request.headers)

print(response.content.decode())

结果

{'User-Agent': 'python-requests/2.12.4', 'Accept-Encoding': 'gzip, deflate', 'Accept': '\*/\*', 'Connection': 'keep-alive'}

<!DOCTYPE html>

<!-- [ published at 2017-06-09 15:15:23 ] -->

<html>

<head>

<meta http-equiv="Content-type" content="text/html; charset=utf-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<title>新浪首页</title>

<meta name="keywords" content="新浪,新浪网,SINA,sina,sina.com.cn,新浪首页,门户,资讯" />

...

#coding=utf-8

import requests

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

print(response.request.headers)

print(response.text)

结果

{'User-Agent': 'python-requests/2.12.4', 'Accept-Encoding': 'gzip, deflate', 'Accept': '\*/\*', 'Connection': 'keep-alive'}

<!DOCTYPE html>

<!-- [ published at 2017-06-09 15:18:10 ] -->

<html>

<head>

<meta http-equiv="Content-type" content="text/html; charset=utf-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<title>æ–°æµªé¦–é¡µ</title>

<meta name="keywords" content="æ–°æµª,æ–°æµªç½',SINA,sina,sina.com.cn,æ–°æµªé¦–é¡µ,é—¨æˆ·,èµ„è®¯" />

<meta name="description" content="æ–°æµªç½'ä¸ºå…¨çƒç"¨æˆ·24å°æ—¶æä¾›å…¨é¢åŠæ—¶çš„ä¸­æ–‡èµ„è®¯ï¼Œå†…å®¹è¦†ç›–å›½å†…å¤–çªå'æ–°é—»äº‹ä»¶ã€ä½"å›èµ›äº‹ã€å¨±ä¹æ—¶å°šã€äº§ä¸šèµ„è®¯ã€å®žç"¨ä¿¡æ¯ç­‰ï¼Œè®¾æœ‰æ–°é—»ã€ä½"è‚²ã€å¨±ä¹ã€è´¢ç»ã€ç§'æŠ€ã€æˆ¿äº§ã€æ±½è½¦ç­‰30å¤šä¸ªå†…å®¹é¢'é"ï¼ŒåŒæ—¶å¼€è®¾åšå®¢ã€è§†é¢'ã€è®ºå›ç­‰è‡ªç"±äº'åŠ¨äº¤æµç©ºé—´ã€‚" />

<link rel="mask-icon" sizes="any" href="//www.sina.com.cn/favicon.svg" color="red">

`

#### 产生问题的原因分析

1. requests默认自带的Accept-Encoding导致或者新浪默认发送的就是压缩之后的网页
2. 但是为什么content.read()没有问题, 因为requests, 自带解压压缩网页的功能. python3中content.read()会出错.
3. 当收到一个响应时, Requests 会猜测响应的编码方式, 用于在你调用response.text 方法时对响应进行解码. Requests 首先在 HTTP 头部检测是否存在指定的编码方式, 如果不存在, 则会使用 chardet.detect来尝试猜测编码方式（存在误差）
4. **更推荐使用response.content.decode(). python3中只能使用response.content.decode()方法.**

### 通过requests获取网络上图片的大小

from io import BytesIO,StringIO

import requests

from PIL import Image

img\_url = "http://imglf1.ph.126.net/pWRxzh6FRrG2qVL3JBvrDg==/6630172763234505196.png"

response = requests.get(img\_url)

f = BytesIO(response.content)

img = Image.open(f)

print(img.size)

输出结果：

(500, 262)

理解一下 BytesIO 和StringIO

很多时候, 数据读写不一定是文件, 也可以在内存中读写.

StringIO顾名思义就是在内存中读写str.

BytesIO 就是在内存中读写bytes类型的二进制数据

例子中如果使用StringIO 即f = StringIO(response.text)会产生"cannot identify image file"的错误

当然上述例子也可以把图片存到本地之后再使用Image打开来获取图片大小

动手

使用requests爬取任意输入的百度贴吧的网页, 并保存到本地

import requests

import time

*# 发送请求, 爬取网页*

def load\_page(kw, pn):

*"""*

*作用: 根据url发送请求, 获取服务器响应文件*

*url: 需要爬取的url地址*

*"""*

url = "http://tieba.baidu.com/f?"

payload = {'kw': kw, 'pn': pn}

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko"}

*# https://tieba.baidu.com/f?kw=lol&pn=50*

response = requests.get(url, params=payload, headers=headers)

return response

*# 把爬取到的信息写入到文件中*

def write\_file(html, filename):

*"""*

作用: 保存服务器响应文件到本地磁盘文件里

*html: 服务器响应文件*

*filename: 本地磁盘文件名*

*"""*

print("正在存储%s"%filename)

with open(filename, 'wb') as f:

f.write(html.content)

print("-" \* 20)

*# 百度贴吧爬虫接口, 需要传递3个参数给这个接口, main里组合的url地址, 起始页码和终止页码*

def tieba\_spider(kw, begin\_page, end\_page):

*"""*

*作用: 负责处理url, 分配每个url去发送请求*

*url: 需要处理的第一个url*

*begin\_page: 爬虫执行的起始页面*

*end\_page: 爬虫执行的截止页面*

*"""*

*# range(2,6)的范围为[2,5], 所以end\_page要加上1才能代表要爬取的页面*

for page in range(begin\_page, end\_page + 1):

pn = (page - 1) \* 50

filename = "第" + str(page) + "页.html"

print("正在下载%s" % filename)

*# 调用load\_page()发送请求获取HTML页面*

html = load\_page(kw, pn)

*# 将获取到的HTML页面写入本地磁盘文件*

write\_file(html, filename)

time.sleep(2)

*# 模拟main函数*

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

kw = input("请输入需要爬取的贴吧:")

*# 输入起始页和终止页, str转成int类型*

begin\_page = int(input("请输入起始页: "))

end\_page = int(input("请输入终止页: "))

tieba\_spider(kw, begin\_page, end\_page)

### 基本POST请求(data参数)

#### 1. 最基本的post方法

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

#### 2. 传入data数据

对于 POST 请求来说, 我们一般需要为它增加一些参数. 那么最基本的传参方法可以利用 data 这个参数.

python2

import requests

formdata = {

"type":"AUTO",

"i":"i love python",

"doctype":"json",

"xmlVersion":"1.8",

"keyfrom":"fanyi.web",

"ue":"UTF-8",

"action":"FY\_BY\_ENTER",

"typoResult":"true"

}

url = "http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule&smartresult=ugc&sessionFrom=null"

headers={ "User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36"}

response = requests.post(url, data = formdata, headers = headers)

print response.text

# 如果是json文件可以直接显示

print response.json()

运行结果

{"type":"EN2ZH\_CN","errorCode":0,"elapsedTime":2,"translateResult":[[{"src":"i love python","tgt":"我喜欢python"}]],"smartResult":{"type":1,"entries":["","肆文","高德纳"]}}

{u'errorCode': 0, u'elapsedTime': 0, u'translateResult': [[{u'src': u'i love python', u'tgt': u'\u6211\u559c\u6b22python'}]], u'smartResult': {u'type': 1, u'entries': [u'', u'\u8086\u6587', u'\u9ad8\u5fb7\u7eb3']}, u'type': u'EN2ZH\_CN'}

有道翻译, 必须要有cookie才能正确返回内容, 所以要构建完整的headers信息

python3

import requests

formdata = {

"i": "python",

"from": "AUTO",

"to": "AUTO",

"smartresult": "dict",

"client": "fanyideskweb",

"salt": "1522802917710",

"sign": "ab797ccb9b533256c4f4aef9dc036074",

"doctype": "json",

"version": "2.1",

"keyfrom": "fanyi.web",

"action": "lan-select",

"typoResult": "false",

}

url = "http://fanyi.youdao.com/translate\_o?smartresult=dict&smartresult=rule"

headers = {

"Accept": "application/json, text/javascript, \*/\*; q=0.01",

"Accept-Language": "en-US,en;q=0.9,zh-CN;q=0.8,zh;q=0.7",

"Connection": "keep-alive",

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

"Cookie": "OUTFOX\_SEARCH\_USER\_ID=782686979@10.168.8.61; OUTFOX\_SEARCH\_USER\_ID\_NCOO=1027783784.6348062; JSESSIONID=aaa9Neo\_JY54hgeSaiokw; \_\_\_rl\_\_test\_\_cookies=1522802907313",

"DNT": "1",

"Host": "fanyi.youdao.com",

"Origin": "http://fanyi.youdao.com",

"Referer": "http://fanyi.youdao.com/",

"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36",

"X-Requested-With": "XMLHttpRequest",

}

response = requests.post(url, data=formdata, headers=headers)

print(response.text)

# 如果是json文件可以直接显示

print(response.json())

### 代理(proxies参数)

如果需要使用代理, 你可以通过为任意请求方法提供 proxies 参数来配置单个请求:

import requests

# 根据协议类型, 选择不同的代理

proxies = {

"http": "http://12.34.56.79:9527",

"https": "http://12.34.56.79:9527",

}

headers={ **"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

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

print(response.text)

也可以通过本地环境变量 HTTP\_PROXY 和 HTTPS\_PROXY 来配置代理:

export HTTP\_PROXY="http://12.34.56.79:9527"

export HTTPS\_PROXY=https://12.34.56.79:9527

python3用proxies参数为Requests添加代理

**import** requests

url = **"http://www.douban.com/"**

headers={ **"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

*# 根据协议类型, 选择不同的代理*

proxies = {

**"https"**: **"222.85.39.80:808"**,

**"https"**: **"115.203.85.95:808"**,

}

response = requests.post(url, headers = headers, proxies = proxies)

print(response.text)

### 私密代理验证(特定格式) 和 Web客户端验证(auth 参数)

urllib2 这里的做法比较复杂, requests只需要一步:

#### 私密代理

python2

import requests

# 如果代理需要使用HTTP Basic Auth, 可以使用下面这种格式:

proxy = { "http": "mr\_mao\_hacker:sffqry9r@61.158.163.130:16816" }

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

print response.text

python3

**import** requests

url = **"http://www.douban.com/"**

headers={ **"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

proxy = { **"http"**: **"mr\_mao\_hacker:sffqry9r@61.158.163.130:16816"** }

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

print(response.text)

#### web客户端验证

如果是Web客户端验证, 需要添加 auth = (账户名, 密码)

python2版

import requests

auth=('test', '123456')

response = requests.get('http://192.168.199.107', auth = auth)

print response.text

python3

**import** requests

url = **"http://192.168.199.107/"**

headers={ **"User-Agent"**: **"****Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

proxy = { **"http"**: **"mr\_mao\_hacker:sffqry9r@61.158.163.130:16816"** }

auth=(**'test'**, **'123456'**)

response = requests.get(url, headers = headers, proxies = proxy, auth = auth)

print(response.text)

urllib2 泪奔...

### Cookies 和 Sission

#### Cookies

如果一个响应中包含了cookie, 那么我们可以利用 cookies参数拿到:

import requests

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

# 7. 返回CookieJar对象:

cookiejar = response.cookies

# 8. 将CookieJar转为字典:

cookiedict = requests.utils.dict\_from\_cookiejar(cookiejar)

print cookiejar

print cookiedict

运行结果:

<RequestsCookieJar[<Cookie BDORZ=27315 for .baidu.com/>]>

{'BDORZ': '27315'}

**import** requests

url = **"http://www.douban.com/"**

headers={ **"User-Agent"**: **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

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

*# 7. 返回CookieJar对象:*

cookiejar = response.cookies

*# 8. 将CookieJar转为字典:*

cookiedict = requests.utils.dict\_from\_cookiejar(cookiejar)

print(response.text)

print(cookiejar)

print(cookiedict)

#### Sission模拟登陆

在 requests 里, session对象是一个非常常用的对象, 这个对象代表一次用户会话: 从客户端浏览器连接服务器开始, 到客户端浏览器与服务器断开.

会话能让我们在跨请求时候保持某些参数, 比如在同一个 Session 实例发出的所有请求之间保持 cookie .

实现人人网登录

import requests

# 1. 创建session对象, 可以保存Cookie值

ssion = requests.session()

# 2. 处理 headers

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"}

# 3. 需要登录的用户名和密码

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

# 4. 发送附带用户名和密码的请求, 并获取登录后的Cookie值, 保存在ssion里

ssion.post("http://www.renren.com/PLogin.do", data = data)

# 5. ssion包含用户登录后的Cookie值, 可以直接访问那些登录后才可以访问的页面

response = ssion.get("http://www.renren.com/410043129/profile")

# 6. 打印响应内容

print response.text

python3实现豆瓣登陆

import requests

*# 1. 创建session对象, 可以保存Cookie值*

ssion = requests.session()

url = "https://accounts.douban.com/login"

*# 2. 处理 headers*

headers={ "User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"}

*# 3. 需要登录的用户名和密码, 在request中的form\_data中查看向服务器发送的表单信息*

data = {"form\_email": "superwuzhiwei@163.com", "form\_password":"w6u9y5a3n8zi"}

*# 4. 发送附带用户名和密码的请求, 并获取登录后的Cookie值, 保存在ssion里*

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

*# 个人的订单页面, 如果没有登录, 会跳转到登录页面, 必须要在登陆的状态下才能查看*

order\_url = "https://www.douban.com/mine/orders/"

*# 5. ssion包含用户登录后的Cookie值, 可以直接访问那些登录后才可以访问的页面*

response = ssion.get(order\_url)

*# 6. 打印响应内容*

*# print(response.text)*

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

f.write(response.text)

### 处理HTTPS请求 SSL证书验证

Requests也可以为HTTPS请求验证SSL证书:

* 要想检查某个主机的SSL证书, 你可以使用 verify 参数(也可以不写)

import requests

response = requests.get("https://www.baidu.com/", verify=True)

# 也可以省略不写

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

print r.text

运行结果:

<!DOCTYPE html>

<!--STATUS OK--><html> <head><meta http-equiv=content-type content=text/html;charset=utf-8><meta http-equiv=X-UA-Compatible content=IE=Edge>百度一下, 你就知道 ....

* 如果SSL证书验证不通过, 或者不信任服务器的安全证书, 则会报出SSLError, 据说 12306 证书是自己做的:

来测试一下:

import requests

response = requests.get("https://www.12306.cn/mormhweb/")

print response.text

果然:

SSLError: ("bad handshake: Error([('SSL routines', 'ssl3\_get\_server\_certificate', 'certificate verify failed')],)",)

**如果我们想跳过 12306 的证书验证, 把 verify 设置为 False 就可以正常请求了.**

r = requests.get("https://www.12306.cn/mormhweb/", verify = False)

## 参考阅读：编码故事

很久很久以前, 有一群人, 他们决定用8个可以开合的晶体管来组合成不同的状态, 以表示世界上的万物. 他们看到8个开关状态是好的, 于是他们把这称为"字节".

再后来, 他们又做了一些可以处理这些字节的机器, 机器开动了, 可以用字节来组合出很多状态, 状态开始变来变去. 他们看到这样是好的, 于是它们就这机器称为"计算机".

开始计算机只在美国用. 八位的字节一共可以组合出256(2的8次方)种不同的状态.

他们把其中的编号从0开始的32种状态分别规定了特殊的用途, 一但终端、打印机遇上约定好的这些字节被传过来时, 就要做一些约定的动作. 遇上00x10, 终端就换行, 遇上0x07, 终端就向人们嘟嘟叫, 例好遇上0x1b, 打印机就打印反白的字, 或者终端就用彩色显示字母. 他们看到这样很好, 于是就把这些0x20以下的字节状态称为"控制码".

他们又把所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示, 一直编到了第127号, 这样计算机就可以用不同字节来存储英语的文字了. 大家看到这样, 都感觉很好, 于是大家都把这个方案叫做 ANSI 的"Ascii"编码（American Standard Code for Information Interchange, 美国信息互换标准代码）. 当时世界上所有的计算机都用同样的ASCII方案来保存英文文字.

后来, 就像建造巴比伦塔一样, 世界各地的都开始使用计算机, 但是很多国家用的不是英文, 他们的字母里有许多是ASCII里没有的, 为了可以在计算机保存他们的文字, 他们决定采用127号之后的空位来表示这些新的字母、符号, 还加入了很多画表格时需要用下到的横线、竖线、交叉等形状, 一直把序号编到了最后一个状态255. 从128到255这一页的字符集被称"扩展字符集". 从此之后, 贪婪的人类再没有新的状态可以用了, 美帝国主义可能没有想到还有第三世界国家的人们也希望可以用到计算机吧！

等中国人们得到计算机时, 已经没有可以利用的字节状态来表示汉字, 况且有6000多个常用汉字需要保存呢. 但是这难不倒智慧的中国人民, 我们不客气地把那些127号之后的奇异符号们直接取消掉, 规定：一个小于127的字符的意义与原来相同, 但两个大于127的字符连在一起时, 就表示一个汉字, 前面的一个字节（他称之为高字节）从0xA1用到0xF7, 后面一个字节（低字节）从0xA1到0xFE, 这样我们就可以组合出大约7000多个简体汉字了. 在这些编码里, 我们还把数学符号、罗马希腊的字母、日文的假名们都编进去了, 连在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码, 这就是常说的"全角"字符, 而原来在127号以下的那些就叫"半角"字符了.

中国人民看到这样很不错, 于是就把这种汉字方案叫做 "GB2312". GB2312 是对 ASCII 的中文扩展.

但是中国的汉字太多了, 我们很快就就发现有许多人的人名没有办法在这里打出来, 特别是某些很会麻烦别人的国家领导人. 于是我们不得不继续把 GB2312 没有用到的码位找出来老实不客气地用上.

后来还是不够用, 于是干脆不再要求低字节一定是127号之后的内码, 只要第一个字节是大于127就固定表示这是一个汉字的开始, 不管后面跟的是不是扩展字符集里的内容. 结果扩展之后的编码方案被称为 GBK 标准, GBK 包括了 GB2312 的所有内容, 同时又增加了近20000个新的汉字（包括繁体字）和符号.

后来少数民族也要用电脑了, 于是我们再扩展, 又加了几千个新的少数民族的字, GBK 扩成了 GB18030. 从此之后, 中华民族的文化就可以在计算机时代中传承了.

中国的程序员们看到这一系列汉字编码的标准是好的, 于是通称他们叫做 "DBCS"（Double Byte Charecter Set 双字节字符集）. 在DBCS系列标准里, 最大的特点是两字节长的汉字字符和一字节长的英文字符并存于同一套编码方案里, 因此他们写的程序为了支持中文处理, 必须要注意字串里的每一个字节的值, 如果这个值是大于127的, 那么就认为一个双字节字符集里的字符出现了. 那时候凡是受过加持, 会编程的计算机僧侣们都要每天念下面这个咒语数百遍：

"一个汉字算两个英文字符！一个汉字算两个英文字符……"

因为当时各个国家都像中国这样搞出一套自己的编码标准, 结果互相之间谁也不懂谁的编码, 谁也不支持别人的编码, 连大陆和台湾这样只相隔了150海里, 使用着同一种语言的兄弟地区, 也分别采用了不同的 DBCS 编码方案——当时的中国人想让电脑显示汉字, 就必须装上一个"汉字系统", 专门用来处理汉字的显示、输入的问题, 但是那个台湾的愚昧封建人士写的算命程序就必须加装另一套支持 BIG5 编码的什么"倚天汉字系统"才可以用, 装错了字符系统, 显示就会乱了套！这怎么办？而且世界民族之林中还有那些一时用不上电脑的穷苦人民, 他们的文字又怎么办？

真是计算机的巴比伦塔命题啊！

正在这时, 大天使加百列及时出现了——一个叫 ISO （国际标谁化组织）的国际组织决定着手解决这个问题. 他们采用的方法很简单：废了所有的地区性编码方案, 重新搞一个包括了地球上所有文化、所有字母和符号的编码！他们打算叫它"Universal Multiple-Octet Coded Character Set", 简称 UCS, 俗称 "UNICODE".

UNICODE 开始制订时, 计算机的存储器容量极大地发展了, 空间再也不成为问题了. 于是 ISO 就直接规定必须用两个字节, 也就是16位来统一表示所有的字符, 对于ascii里的那些"半角"字符, UNICODE 包持其原编码不变, 只是将其长度由原来的8位扩展为16位, 而其他文化和语言的字符则全部重新统一编码. 由于"半角"英文符号只需要用到低8位, 所以其高8位永远是0, 因此这种大气的方案在保存英文文本时会多浪费一倍的空间.

这时候, 从旧社会里走过来的程序员开始发现一个奇怪的现象：他们的strlen函数靠不住了, 一个汉字不再是相当于两个字符了, 而是一个！是的, 从 UNICODE 开始, 无论是半角的英文字母, 还是全角的汉字, 它们都是统一的"一个字符"！同时, 也都是统一的"两个字节", 请注意"字符"和"字节"两个术语的不同, "字节"是一个8位的物理存贮单元, 而"字符"则是一个文化相关的符号. 在UNICODE 中, 一个字符就是两个字节. 一个汉字算两个英文字符的时代已经快过去了.

从前多种字符集存在时, 那些做多语言软件的公司遇上过很大麻烦, 他们为了在不同的国家销售同一套软件, 就不得不在区域化软件时也加持那个双字节字符集咒语, 不仅要处处小心不要搞错, 还要把软件中的文字在不同的字符集中转来转去. UNICODE 对于他们来说是一个很好的一揽子解决方案, 于是从 Windows NT 开始, MS 趁机把它们的操作系统改了一遍, 把所有的核心代码都改成了用 UNICODE 方式工作的版本, 从这时开始, WINDOWS 系统终于无需要加装各种本土语言系统, 就可以显示全世界上所有文化的字符了.

但是, UNICODE 在制订时没有考虑与任何一种现有的编码方案保持兼容, 这使得 GBK 与UNICODE 在汉字的内码编排上完全是不一样的, 没有一种简单的算术方法可以把文本内容从UNICODE编码和另一种编码进行转换, 这种转换必须通过查表来进行.

如前所述, UNICODE 是用两个字节来表示为一个字符, 他总共可以组合出65535不同的字符, 这大概已经可以覆盖世界上所有文化的符号. 如果还不够也没有关系, ISO已经准备了UCS-4方案, 说简单了就是四个字节来表示一个字符, 这样我们就可以组合出21亿个不同的字符出来（最高位有其他用途）, 这大概可以用到银河联邦成立那一天吧！

UNICODE 来到时, 一起到来的还有计算机网络的兴起, UNICODE 如何在网络上传输也是一个必须考虑的问题, 于是面向传输的众多 UTF（UCS Transfer Format）标准出现了, 顾名思义, UTF8就是每次8个位传输数据, 而UTF16就是每次16个位, 只不过为了传输时的可靠性, 从UNICODE到UTF时并不是直接的对应, 而是要过一些算法和规则来转换.

**总结：**

* 字符(Character)是各种文字和符号的总称, 包括各国家文字、标点符号、图形符号、数字等. 字符集(Character set)是多个字符的集合
* 字符集包括：ASCII字符集、GB2312字符集、GB18030字符集、Unicode字符集等
* ASCII编码是1个字节, 而Unicode编码通常是2个字节.
* UTF-8是Unicode的实现方式之一, UTF-8是它是一种变长的编码方式, 可以是1, 2, 3个字节

# 2. 非结构化数据与结构化数据提取

## 页面解析和数据提取

一般来讲对我们而言, 需要抓取的是某个网站或者某个应用的内容, 提取有用的价值. 内容一般分为两部分, 非结构化的数据 和 结构化的数据.

* 非结构化数据：先有数据, 再有结构, （<http://www.baidu.com）>
* 结构化数据：先有结构、再有数据（<http://wangyi.butterfly.mopaasapp.com/news/api?type=war&page=1&limit=10）>
* 不同类型的数据, 我们需要采用不同的方式来处理.

### 非结构化的数据处理

#### 文本、电话号码、邮箱地址

* 正则表达式

#### HTML 文件

* 正则表达式
* XPath
* CSS选择器

### 结构化的数据处理

#### JSON 文件

* JSON Path
* 转化成Python类型进行操作(json类)

#### XML 文件

* 转化成Python类型(xmltodict)
* XPath
* CSS选择器
* 正则表达式

## 正则表达式re模块

### 为什么要学正则表达式

实际上爬虫一共就四个主要步骤:

1. 明确目标 (要知道你准备在哪个范围或者网站去搜索)
2. 爬 (将所有的网站的内容全部爬下来)
3. 取 (去掉对我们没用处的数据)
4. 处理数据(按照我们想要的方式存储和使用)

我们在昨天的案例里实际上省略了第3步, 也就是"取"的步骤. 因为我们down下了的数据是全部的网页, 这些数据很庞大并且很混乱, 大部分的东西使我们不关心的, 因此我们需要将之按我们的需要过滤和匹配出来.

那么对于文本的过滤或者规则的匹配, 最强大的就是正则表达式, 是Python爬虫世界里必不可少的神兵利器.

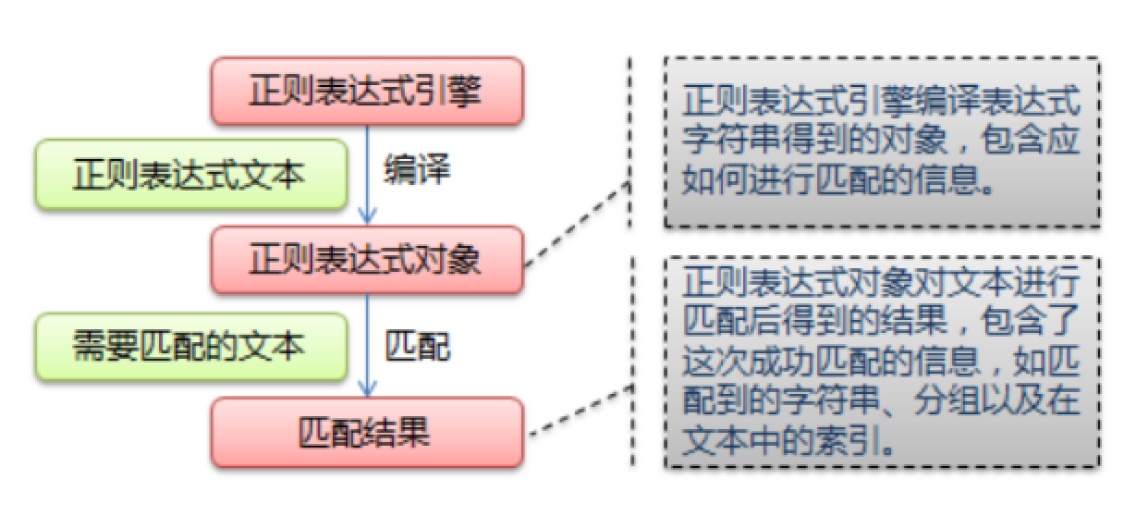
### 什么是正则表达式

正则表达式, 又称规则表达式, 通常被用来检索、替换那些符合某个模式(规则)的文本.

正则表达式是对字符串操作的一种逻辑公式, 就是用事先定义好的一些特定字符、及这些特定字符的组合, 组成一个"规则字符串", 这个"规则字符串"用来表达对字符串的一种过滤逻辑.

给定一个正则表达式和另一个字符串, 我们可以达到如下的目的:

* 给定的字符串是否符合正则表达式的过滤逻辑("匹配") ；
* 通过正则表达式, 从文本字符串中获取我们想要的特定部分("过滤") .



### 正则表达式匹配规则

|  |  |  |  |
| --- | --- | --- | --- |
| 语法 | 说明 | 表达式实例 | 完整匹配的字符串 |
| **字符** | | | |
| . | 匹配任意除换行符"\n"外的字符 | a.b | abc |
| \ | 转义字符 | a\.b a\\c | a.c a\c |
| [...] | 字符集, 对应的位置可以式字符集中任意字符 | a[bcd]e | abe ace ade |
| **预定义字符集（可以写在字符集[…]中）** | | | |
| \d | 数字：[0-9] | a\dc | a1c |
| \D | 非数字：[^\d] | a\Dc | abc |
| \s | 空白字符：[<空格>\t\r\n\f\v] | a\sc | a c |
| \S | 非空白字符：[^\s] | a\sc | abc |
| \w | 单词字符：[A-Za-z0-9\_] | a\wc | abc |
| \W | 非单词字符：[^\w] | a\wc | a c |
| **数量词（用在 字符或(…)之后）** | | | |
| \* | 匹配前一个字符0或无限次 | abc\* | ab abcccc |
| + | 匹配前一个字符1或无限次 | abc+ | abc abcccc |
| ? | 匹配前一个字符0次或1次 | abc? | ab abc |
| {m} | 匹配前一个字符m次 | av{2}c | abbc |
| {m,n} | 匹配前一个字符m至n次（注：m和n可以省略：若省略m, 则匹配0至n次；若省略n, 则匹配m至无限次） | ab{1,2}c | abc abbc |
| \*?+? ?? {m,n}? | 使\* + > {m,n}变成非贪婪模式 |  |  |
| **边界匹配（不消耗待匹配字符串中的字符）** | | | |
| ^ | 匹配字符串开头 在多行模式中匹配每一行的开头 | ^abc | abc |
| $ | 匹配字符串末尾 在多行模式中匹配每一行的末尾 | abc$ | abc |
| \A | 仅匹配字符串开头 | \Aabc | abc |
| \Z | 仅匹配字符串末尾 | abc\Z | abc |
| \b | 匹配\w和\W之间 | a\b!bc | a!bc |
| \B | [^\b] | a\Bbc | abc |
| **逻辑、分组** | | | |
| | | 代表左右表达式任意匹配一个 总是先尝试匹配左边的表达式, 一旦成功匹配则跳过匹配右边的表达式 | abc|def | abc def |
| (…) | 被括起来的表达式将作为分组, 从表达式左边开始每遇到一个分组的左括号'（', 编号+1 另外分组表达式作为整体, 可以后接数量词 | (abc){2} a(123|456)c | abcabc a456c |
| (?P<name>…) | 分组, 除了原有的编号外再指定一个额外的别名 | (?P<id>abc){2} | abcabc |
| \<number> | 引用编号为<number>的分组匹配到的字符串 | (\d)abc\1 | 1abc1 5abc5 |
| (?P=name) | 引用别名为<name>的分组匹配到的字符串 | (?P<id>\d)abc(?P=id) | 1abc1 5abc5 |
| **特殊构造（不作为分组）** | | | |
| (?:...) | (…)的不分组版本, 用于使用'|'或后接数量词 | (?:abc){2} | abcabc |
| (?iLmsux) | iLmsux的每个字符代表一个匹配模式, 只能用在正则表达式的开头, 可选多个 | (?i)abc | AbC |
| (?#...) | #后的内容将作为注释被忽略 | abc(?#comment)123 | abc123 |
| (?=…) | 之后的字符串内容需要匹配表达式才能成功匹配 | a(?=\d) | 后面是数字的a |
| (?!=…) | 之后的字符串内容需要不匹配表达式才能成功匹配 | a(?!=\d) | 后面不是数字的a |
| (?<=…) | 之前的字符串内容需要匹配表达式才能成功匹配 | (?<=\d)a | 前面是数字的a |
| (?<!...) | 之前的字符串内容需要不匹配表达式才能成功匹配 | (?<!\d)a | 前面不是数字的a |
| (?(id/name) yes-pattern |no-pattren) | 如果编号为id/别名为name的组匹配到字符, 则需要匹配yes-pattern, 否则需要匹配no-pattern | (\d)abc(?(1)\d|abc) | 1abc2 abcabc |

### Python 的 re 模块

<https://docs.python.org/3.5/library/re.html>

https://yiyibooks.cn/xx/python\_352/library/re.html

有一点需要特别注意的是, 正则表达式使用对特殊字符进行转义, 所以如果我们要使用原始字符串, 只需加一个r前缀, 示例：为了使字符不受转义字符的影响, 在字符前面加一个r, 所以正则表达式一般都需要在前面加一个r

r'chuanzhiboke\t\.\tpython'

str="nihao\tpython\naab"

print(str)

nihao python

aab

str2=r"nihao\tpython\naab"

print(str2)

加一个u表示是unicode字符串.

#### Re模块的两种使用模式

python里re模块有两种使用模式：

第一种模式: 将正则表达式编辑成一个pattern规则对象, 调用pattern对象的match, search, findall等方法.

1. pattern = re.compile("\d")
2. pattern.match(123)

指定位置或从起始位置开始往后查找, 返回第一个符合规则的, 只匹配一次

1. pattern.search()

从任何位置开始往后查找, 返回第一个符合规则的, 只匹配一次

1. pattern.findall()

匹配全部, 返回列表

1. pattern.finditer()

匹配全部, 返回一个迭代器

1. pattern.split()

分割字符串, 返回列表

1. pattern.sub()

匹配替换

第二种模式: 直接使用re中的search, match, findall等方法.

如re.complie(), re.search(), re.match(), re.findall()

#### Re模块的第一种使用模式

##### re 模块的一般使用步骤如下

1. 使用 compile() 函数将正则表达式的字符串形式编译为一个 Pattern 对象
2. 通过 Pattern 对象提供的一系列方法对文本进行匹配查找, 获得匹配结果, 返回一个 Match 对象.
3. 最后使用 Match 对象提供的属性和方法获得信息, 根据需要进行其他的操作

##### compile 方法

compile 函数用于编译正则表达式, 生成一个 Pattern 对象.

对于频繁使用的表达式, 编译这些表达式会更有效. compile()函数会把一个表达式字符串转化成为一个RegexObject.

模块级函数会维护已编译表达式的一个缓存. 不过, 这个缓存的大小是有限的, 直接使用已编译的表达式可以避免缓存查找开销.

使用已编译表达式的另一个好处是, 通过在加载模块时预编译所有表达式, 可以把编译工作转到应用开始时, 而不是当程序响应一个用户动作时才进行编译.

例如, 直接使用findall(rule, target)的方式来匹配字符串, 一次两次没什么, 如果是多次使用的话, 由于正则引擎每次都要把规则解释一遍, 而规则的解释又是相当费时间的, 所以这样的效率就很低了. 如果要多次使用同一规则来进行匹配的话, 可以使用re.compile函数来将规则预编译, 使用编译过返回的Regular Expression Object或叫做Pattern对象来进行查找.

**re.compile**(pattern, flags=0)

Compile a regular expression pattern, returning a pattern object.

将正则表达式模式编译成一个正则表达式对象, 匹配时可以调用它的 [match()](http://usyiyi.cn/documents/python_352/library/re.html#re.regex.match) 和 [search()](http://usyiyi.cn/documents/python_352/library/re.html#re.regex.search) 方法, 如下所述.

可以通过指定**flags** 值修改表达式的行为. 值可以是任何以下变量, 组合使用 OR ( | 运算符).

flags有很多可选值：

* re.I(IGNORECASE)忽略大小写, 括号内是完整的写法
* re.M(MULTILINE)多行模式, 改变^和$的行为
* re.S(DOTALL)点可以匹配任意字符, 包括换行符. re.DOTALL
* re.L(LOCALE)做本地化识别的匹配, 不推荐使用
* re.U(UNICODE) 使用\w \W \s \S \d \D使用取决于unicode定义的字符属性. 在python3中默认使用该flag
* re.X(VERBOSE)冗长模式, 该模式下pattern字符串可以是多行的, 忽略空白字符, 并可以添加注释

import re

# 将正则表达式编译成 Pattern 对象

pattern = re.compile(r'\d+')

在上面, 我们已将一个正则表达式编译成 Pattern 对象, 接下来, 我们就可以利用 pattern 的一系列方法对文本进行匹配查找了.

**Pattern 对象的一些常用方法主要有：**

match 方法：从起始位置开始查找, 一次匹配

search 方法：从任何位置开始查找, 一次匹配

findall 方法：全部匹配, 返回列表

finditer 方法：全部匹配, 返回迭代器

split 方法：分割字符串, 返回列表

sub 方法：替换

两种使用方法, 一种是在pattern中编译, 第二种是在re函数中使用

pattern = re.compile(r"([a-z]+)([a-z]+)", re.I)

str = "Hello world, Hello python!"

m=pattern.match(str)

pattern = re.compile(r"([a-z]+)([a-z]+)")

str = "Hello world, Hello python!"

m=pattern.match(str, re.I)

##### match 方法

**作用:**

match 方法用于查找字符串的头部(也可以指定起始位置) , 它是一次匹配, 只要找到了一个匹配的结果就返回, 而不是查找所有匹配的结果.

**使用格式:**

match(string[, pos[, endpos]])

**使用说明:**

其中, string 是待匹配的字符串, pos 和 endpos 是可选参数, 指定字符串的起始和终点位置, 默认值分别是 0 和 len (字符串长度). 因此, 当不指定 pos 和 endpos 时, match 方法默认匹配字符串的头部.

当匹配成功时, 返回一个 Match 对象, 如果没有匹配上, 则返回 None.

注意：匹配成功后：

1. group([group1, …]) 方法用于获得一个或多个分组匹配的字符串, 当要获得整个匹配的子串时, 可直接使用 group() 或 group(0)；
2. start([group]) 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引）, 参数默认值为 0, 即获得第0组匹配的起始位置；
3. end([group]) 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1）, 参数默认值为 0；
4. span([group]) 方法返回 (start(group), end(group)). 默认值为0. 可以省略.

例子:

import re

pattern = re.compile(r'\d+') # 用于匹配至少一个数字

m = pattern.match('one12twothree34four') # 查找头部, 没有匹配

print(m)

None

m = pattern.match('one12twothree34four', 2, 10) # 从'e'的位置开始匹配, 没有匹配

print(m)

None

m = pattern.match('one12twothree34four', 3, 10) # 从'1'的位置开始匹配, 正好匹配, 返回一个 Match 对象

print(m)

<\_sre.SRE\_Match object at 0x10a42aac0>

m.group(0) # 可省略 0

'12'

m.start(0) # 可省略 0

3

m.end(0) # 可省略 0

5

m.span(0) # 可省略 0

(3, 5)

import re

pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I) # re.I 表示忽略大小写

m = pattern.match('Hello World Wide Web')

print(m) # 匹配成功, 返回一个 Match 对象

<\_sre.SRE\_Match object at 0x10bea83e8>

m.group(0) # 返回匹配成功的整个子串

'Hello World'

m.span(0) # 返回匹配成功的整个子串的索引

(0, 11)

m.group(1) # 返回第一个分组匹配成功的子串

'Hello'

m.span(1) # 返回第一个分组匹配成功的子串的索引

(0, 5)

m.group(2) # 返回第二个分组匹配成功的子串

'World'

m.span(2) # 返回第二个分组匹配成功的子串

(6, 11)

m.groups() # 等价于 (m.group(1), m.group(2), ...)

('Hello', 'World')

m.group(3) # 不存在第三个分组

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: no such group

import re

*#re.I 忽略大小写*

pattern = re.compile(r"([a-z]+)([a-z]+)", re.I)

str = "Hello world, Hello python!"

# m=pattern.match(str)

m = re.match(pattern, str)

m = re.match(r"([a-z]+)([a-z]+)", str)

m = re.match(r"([a-z]+)([a-z]+)", "Hello world, Hello python!", re.I)

*#m.group(0)和m.goup()表示取出组中的所有元素, m.goup(1)表示第1个元素. 因为2个匹配规则都为([a-z]+), 即匹配1或多次, 所以会使用2个规则进行匹配, 而不是使用1个规则匹配整个Hello*

print(m.group(), m.group(0), m.group(1), m.group(2), sep="\n"+"\*"\*30+"\n")

print("="\*40)

*#符合第一个匹配规则子串的开始位置和结束位置*

print(m.span(), m.span(0), m.span(1), m.span(2), sep="\n"+"@"\*30+"\n")

Hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hell

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

o

========================================

(0, 5)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

(0, 5)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

(0, 4)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

(4, 5)

re表达式中间加一空格

import re

*#re.I 忽略大小写*

pattern = re.compile(r"([a-z]+) ([a-z]+)", re.I)

str = "Hello world, Hello python!"

m=pattern.match(str)

print(m.group(), m.group(0), m.group(1), m.group(2), sep="\n"+"\*"\*30+"\n")

print("="\*40)

*# 符合第一个匹配规则子串的下标范围*

print(m.span(), m.span(0), m.span(1), m.span(2), sep="\n"+"@"\*30+"\n")

import re

result = re.search(r"([a-z]+) ([a-z]+)", "Hello world, Hello python!", re.I)

print(result, result.group(), result.group(0), result.group(1), result.group(2), sep="\n"+"\*"\*30+"\n")

print("="\*40)

print(result, result.span(), result.span(0), result.span(1), result.span(2), sep="\n"+"\*"\*30+"\n")

<\_sre.SRE\_Match object; span=(0, 11), match='Hello world'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello world

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello world

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

world

========================================

<\_sre.SRE\_Match object; span=(0, 11), match='Hello world'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(0, 11)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(0, 11)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(0, 5)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(6, 11)

##### search 方法

**作用:**

扫描字符串查找正则表达式模式产生匹配的第一个位置, 并返回相应的match object. 如果字符串中没有位置与模式匹配, 则返回None；请注意, 这不同于在字符串中的某个点找到零长度匹配.

search 方法用于查找字符串的任何位置, 它也是一次匹配, 只要找到了一个匹配的结果就返回, 而不是查找所有匹配的结果,

**使用格式:**

search(string[, pos[, endpos]])

**说明:**

其中, string 是待匹配的字符串, pos 和 endpos 是可选参数, 指定字符串的起始和终点位置, 默认值分别是 0 和 len (字符串长度).

当匹配成功时, 返回一个 Match 对象, 如果没有匹配上, 则返回 None.

import re

>>> pattern = re.compile('\d+')

>>> m = pattern.search('one12twothree34four') # 这里如果使用 match 方法则不匹配

>>> m

<\_sre.SRE\_Match object at 0x10cc03ac0>

>>> m.group()

'12'

>>> m = pattern.search('one12twothree34four', 10, 30) # 指定字符串区间

>>> m

<\_sre.SRE\_Match object at 0x10cc03b28>

>>> m.group()

'34'

>>> m.span()

(13, 15)

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

import re

# 将正则表达式编译成 Pattern 对象

pattern = re.compile(r'\d+')

# 使用 search() 查找匹配的子串, 不存在匹配的子串时将返回 None

# 这里使用 match() 无法成功匹配

m = pattern.search('hello 123456 789')

if m:

# 使用 Match 获得分组信息

print 'matching string:',m.group()

# 起始位置和结束位置

print 'position:',m.span()

执行结果:

matching string: 123456

position: (6, 12)

视频演示

import re

pattern = re.compile("\d+")

str1="aaa123bbb456"

str2 = "abc 123 def 789"

s1 = pattern.search(str1)

s2 = pattern.search(str2)

print(s1, s1.group(), s1.group(0), sep="\n" + "\*" \* 30 + "\n")

print("="\*40)

print(s1, s1.span(), s1.span(0), sep="\n" + "\*" \* 30 + "\n")

print(s2, s2.group(), s2.group(0), sep="\n" + "\*" \* 30 + "\n")

print("="\*40)

print(s2, s2.span(), s2.span(0), sep="\n" + "\*" \* 30 + "\n")

<\_sre.SRE\_Match object; span=(3, 6), match='123'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123

========================================

<\_sre.SRE\_Match object; span=(3, 6), match='123'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(3, 6)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(3, 6)

<\_sre.SRE\_Match object; span=(4, 7), match='123'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123

========================================

<\_sre.SRE\_Match object; span=(4, 7), match='123'>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(4, 7)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

(4, 7)

##### findall 方法

上面的 match 和 search 方法都是一次匹配, 只要找到了一个匹配的结果就返回. 然而, 在大多数时候, 我们需要搜索整个字符串, 获得所有匹配的结果.

findall 方法的使用形式如下:

findall(string[, pos[, endpos]])

其中, string 是待匹配的字符串, pos 和 endpos 是可选参数, 指定字符串的起始和终点位置, 默认值分别是 0 和 len (字符串长度).

findall 以列表形式返回全部能匹配的子串, 如果没有匹配, 则返回一个空列表.

看看例子:

import re

pattern = re.compile(r'\d+') # 查找数字

result1 = pattern.findall('hello 123456 789')

result2 = pattern.findall('one1two2three3four4', 0, 10)

print result1

print result2

执行结果:

['123456', '789']

['1', '2']

再先看一个栗子:

# re\_test.py

import re

#re模块提供一个方法叫compile模块, 提供我们输入一个匹配的规则

#然后返回一个pattern实例, 我们根据这个规则去匹配字符串

pattern = re.compile(r'\d+\.\d\*')

#通过partten.findall()方法就能够全部匹配到我们得到的字符串

result = pattern.findall("123.141593, 'bigcat', 232312, 3.15")

#findall 以列表形式 返回全部能匹配的子串给result

for item in result:

print item

运行结果:

123.141593

3.15

import re

pattern1 = re.compile(r"\d+")

pattern2 = re.compile(r"\d?")

str1="hello 1234 world"

str2="hello 1234 world 789"

r1 = pattern1.findall(str1)

r2 = pattern1.findall(str2)

*# ?表示匹配前一个字符0或1次, 因为前面的hello都不匹配, 所以都返回为空的字符串. 所以尽量不要使用这种可能会返回空的正则匹配方式, 而是使用d+这种方式.*

r3 = pattern2.findall(str1)

r4 = pattern2.findall(str2)

print(r1,r2,r3,r4, sep="\n"+"\*"\*30+"\n")

['1234']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['1234', '789']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['', '', '', '', '', '', '1', '2', '3', '4', '', '', '', '', '', '', '']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['', '', '', '', '', '', '1', '2', '3', '4', '', '', '', '', '', '', '', '7', '8', '9', '']

##### finditer 方法

finditer 方法的行为跟 findall 的行为类似, 也是搜索整个字符串, 获得所有匹配的结果. 但它返回一个顺序访问每一个匹配结果(Match 对象) 的迭代器.

看看例子:

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

import re

pattern = re.compile(r'\d+')

result\_iter1 = pattern.finditer('hello 123456 789')

result\_iter2 = pattern.finditer('one1two2three3four4', 0, 10)

print type(result\_iter1)

print type(result\_iter2)

print 'result1...'

for m1 in result\_iter1: # m1 是 Match 对象

print 'matching string: {}, position: {}'.format(m1.group(), m1.span())

print 'result2...'

for m2 in result\_iter2:

print 'matching string: {}, position: {}'.format(m2.group(), m2.span())

执行结果:

<type 'callable-iterator'>

<type 'callable-iterator'>

result1...

matching string: 123456, position: (6, 12)

matching string: 789, position: (13, 16)

result2...

matching string: 1, position: (3, 4)

matching string: 2, position: (7, 8)

import re

pattern1 = re.compile(r"\d+")

pattern2 = re.compile(r"\d?")

str1="hello 1234 world"

str2="hello 1234 world 789"

r1 = pattern1.finditer(str1)

r2 = pattern2.finditer(str1)

r3 = pattern1.finditer(str2)

r4 = pattern2.finditer(str2)

for i in r1:

print(i.group(), i.span(), sep="\n"+"\*"\*30+"\n")

print("="\*40)

for i in r2:

print(i.group(), i.span(), sep="\n"+"\*"\*30+"\n")

print("="\*40)

for i in r3:

print(i.group(), i.span(), sep="\n"+"\*"\*30+"\n")

print("="\*40)

for i in r4:

print(i.group(), i.span(), sep="\n"+"\*"\*30+"\n")

*# 但是如果再次迭代取值, 就为空了. 因为上面已经使用完了迭代器.*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['', '', '', '', '', '', '1', '2', '3', '4', '', '', '', '', '', '', '']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['', '', '', '', '', '', '1', '2', '3', '4', '', '', '', '', '', '', '', '7', '8', '9', '']

##### split 方法

split 方法按照能够匹配的子串将字符串分割后返回列表, 它的使用形式如下:

split(string[, maxsplit])

其中, maxsplit 用于指定最大分割次数, 不指定将全部分割.

看看例子:

import re

p = re.compile(r'[\s\,\;]+')

print p.split('a,b;; c d')

['a', 'b', 'c', 'd']

**import** re  
p = re.compile(**r'[\s\,\;]'**)  
print(p.split(**'a,b;; c d'**))

['a', 'b', '', '', 'c', '', '', 'd']

import re

pattern = re.compile("[\s\d\\\;]+")

str1 = "a bb\cc;dd; e"

str2 = r"a bb\cc;dd; e"

str3 = "a bb\aa;dd; e"

str4 = r"a bb\aa;dd; e"

r1 = pattern.split(str1)

r2 = pattern.split(str2)

r3 = pattern.split(str3)

r4 = pattern.split(str4)

print(r1, r2, r3, r4, sep="\n"+"\*"\*30+"\n")

['a', 'bb', 'cc', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb', 'cc', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb\x07a', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb', 'aa', 'dd', 'e']

##### sub 方法

sub 方法用于替换. 它的使用形式如下:

sub(repl, string[, count])

其中, repl 可以是字符串也可以是一个函数:

* 如果 repl 是字符串, 则会使用 repl 去替换字符串每一个匹配的子串, 并返回替换后的字符串, 另外, repl 还可以使用 id 的形式来引用分组, 但不能使用编号 0；
* 如果 repl 是函数, 这个方法应当只接受一个参数(Match 对象) , 并返回一个字符串用于替换(返回的字符串中不能再引用分组) .
* count 用于指定最多替换次数, 不指定时全部替换.

看看例子:

**import** re  
p = re.compile(**r'(\w+) (\w+)'**) *# \w = [A-Za-z0-9]*s = **'itcast 123, itcast 456'***# 先使用规则p对字符串s进行匹配, 得到匹配的结果, 再使用sub中的第一个参数中的字符串对匹配的结果进行替换*print(p.sub(**r'hello world'**, s)) *# 使用 'hello world' 替换 'hello 123' 和 'hello 456'*print(p.sub(**r'\2 \1'**, s)) *# 引用分组*

# 把匹配的对象作为参数传递给func

def func(m):

return 'hi' + ' ' + m.group(2)

print(p.sub(func, s))

print(p.sub(func, s, 1)) # 最多替换一次

执行结果:

hello world, hello world

123 itcast, 456 itcast

hi 123, hi 456

hi 123, itcast 456

import re

*# \w表示字母数字和下划线*

pattern = re.compile("(\w+) (\w+)")

str = "hello 123, hello python, hello 456"

s1 = pattern.sub("hello world", str)

*#\1 \2表示分组, 分别表示匹配的str字符中的第1组, 第2组.*

s2 = pattern.sub(r'\1 \2', str)

s3 = pattern.sub('\1 \2', str)

s4 = pattern.sub(r'\2 \1', str)

s5 = pattern.sub('\2 \1', str)

print(s1,s2,s3,s4,s5, sep="\n" + "\*" \* 30 + "\n")

hello world, hello world, hello world

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello 123, hello python, hello 456

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

, ,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123 hello, python hello, 456 hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

, ,

import re

pattern1 = re.compile("\d+")

str = "hello 123, hello python, hello 456"

s1 = pattern1.sub("hello world", str)

s2 = pattern1.sub(r'xxx', str)

s3 = pattern1.sub('xxx', str)

print(s1,s2,s3, sep="\n" + "\*" \* 30 + "\n")

import re

*# \w表示字母数字和下划线*

pattern = re.compile("(\w+) (\w+)")

str = "hello 123, hello python, hello 456"

s1 = pattern.sub("hello world", str)

*#\1 \2表示分组, 分别表示匹配的str字符中的第1组, 第2组.*

s2 = pattern.sub(r'\1 \2', str)

s3 = pattern.sub('\1 \2', str)

s4 = pattern.sub(r'\2 \1', str)

s5 = pattern.sub('\2 \1', str)

print(s1,s2,s3,s4,s5, sep="\n" + "\*" \* 30 + "\n")

hello world, hello world, hello world

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello 123, hello python, hello 456

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

, ,

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

123 hello, python hello, 456 hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

, ,

import re

pattern1 = re.compile("\d+")

str = "hello 123, hello python, hello 456"

s1 = pattern1.sub("hello world", str)

s2 = pattern1.sub(r'xxx', str)

s3 = pattern1.sub('xxx', str)

print(s1,s2,s3, sep="\n" + "\*" \* 30 + "\n")

hello hello world, hello python, hello hello world

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello xxx, hello python, hello xxx

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

hello xxx, hello python, hello xxx

##### 匹配中文

在某些情况下, 我们想匹配文本中的汉字, 有一点需要注意的是, 中文的 unicode 编码范围 主要在 [u4e00-u9fa5], 这里说主要是因为这个范围并不完整, 比如没有包括全角(中文) 标点, 不过, 在大部分情况下, 应该是够用的.

假设现在想把字符串 title = u'你好, hello, 世界' 中的中文提取出来, 可以这么做:

import re

title = u'你好, hello, 世界'

pattern = re.compile(ur'[\u4e00-\u9fa5]+')

result = pattern.findall(title)

print result

注意到, 我们在正则表达式前面加上了两个前缀 ur, 其中 r 表示使用原始字符串, u 表示是 unicode 字符串.

执行结果:

[u'\u4f60\u597d', u'\u4e16\u754c']

#### Re模块的第二种使用模式

##### re.compile(pattern, flags=0)

将正则表达式模式编译成一个正则表达式对象, 匹配时可以调用它的 [match()](http://usyiyi.cn/documents/python_352/library/re.html#re.regex.match) 和 [search()](http://usyiyi.cn/documents/python_352/library/re.html#re.regex.search) 方法, 如下所述.

可以通过指定**flags** 值修改表达式的行为. 值可以是任何以下变量, 组合使用 OR ( | 运算符).

import re

# 将正则表达式编译成 Pattern 对象

pattern = re.compile(r'\d+')

##### re.search(pattern, string, flags=0)

扫描**字符串**查找正则表达式**模式**产生匹配的第一个位置, 并返回相应的match object. 如果字符串中没有位置与模式匹配, 则返回None；请注意, 这不同于在字符串中的某个点找到零长度匹配.

##### re.match(pattern, string, flags=0)

如果**字符串**开头的零个或多个字符与正则表达式**模式**相匹配, 则返回相应的[match object](http://usyiyi.cn/documents/python_352/library/re.html#match-objects). 如果字符串与模式不匹配, 则返回None；注意, 这不同于零长度匹配.

请注意, 即使在[MULTILINE](http://usyiyi.cn/documents/python_352/library/re.html#re.MULTILINE)模式下, [re.match()](http://usyiyi.cn/documents/python_352/library/re.html#re.match)仅在字符串开头匹配, 而不在每行开头匹配.

如果您要在**string**中找到匹配项, 请改用[search()](http://usyiyi.cn/documents/python_352/library/re.html#re.search)（另请参阅[search() vs. match()](http://usyiyi.cn/documents/python_352/library/re.html#search-vs-match)）.

##### re.split(pattern, string, maxsplit=0, flags=0)

Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

用模式对字符串进行拆分, 如果在**模式**中使用捕获括号, 则然后也作为结果列表的一部分返回的文本模式中的所有组. 如果**maxsplit**不为零, 顶多**maxsplit**分裂发生, 并且该字符串的其余部分将作为列表的最后一个元素返回.

**>>>** re.split('\W+', 'Words, words, words.')

['Words', 'words', 'words', '']

**>>>** re.split('(\W+)', 'Words, words, words.')

['Words', ', ', 'words', ', ', 'words', '.', '']

**>>>** re.split('\W+', 'Words, words, words.', 1)

['Words', 'words, words.']

**>>>** re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)

['0', '3', '9']

If there are capturing groups in the separator and it matches at the start of the string, the result will start with an empty string. The same holds for the end of the string. 如果在分离器中有捕获组, 它匹配字符串的开头, 结果将以空字符串开头. 这一规则同样也对于字符串的末尾的匹配适用:

**>>>** re.split('(\W+)', '...words, words...')

['', '...', 'words', ', ', 'words', '...', '']

In [50]: re.split('\W+', '...words, words...')

Out[50]: ['', 'words', 'words', '']

In [51]: re.split('(\W+)', '...words, words...')

Out[51]: ['', '...', 'words', ', ', 'words', '...', '']

That way, separator components are always found at the same relative indices within the result list.

这样的话, 分割的组分也会在结果列表中存在.

**注**

[split()](http://usyiyi.cn/documents/python_352/library/re.html#re.split)当前没有在空模式匹配时分割字符串. 举个例子：

**>>>** re.split('x\*', 'axbc')

['a', 'bc']

即使'x\*'也匹配'a'之前的0'x', 'b'和'c'之间和'c'之后, 当前这些匹配被忽略. 正确的行为（即分割为空匹配并返回[",  'a',  'b',  'c' '']）将会在未来的Python版本中实现, 但由于这是一个向后兼容的更改, 因此会出现[FutureWarning](http://usyiyi.cn/documents/python_352/library/exceptions.html#FutureWarning)同时.

只能匹配空字符串的模式当前从不拆分字符串. 由于这与预期的行为不匹配, 因此将从Python 3.5开始提升[ValueError](http://usyiyi.cn/documents/python_352/library/exceptions.html#ValueError)：

>>>

**>>>** re.split("^$", "foo**\n\n**bar**\n**", flags=re.M)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

*...*

ValueError: split() requires a non-empty pattern match.

*在版本3.1中已更改：*添加了可选标志参数.

*在版本3.5中更改：*在可能匹配空字符串的模式上拆分现在引发警告. 只能匹配空字符串的模式现在被拒绝.

匹配规则前不加r, 字符串前面加r. \a不使用转义字符为ascii符, 表示让主板响铃.

**import** re

pattern = re.compile(**"[\s\d\\\;]+"**)

str1 = **"a bb\cc;dd; e"**

str2 = **r"a bb\cc;dd; e"**

str3 = **"a bb\aa;dd; e"**

str4 = **r"a bb\aa;dd; e"**

r1 = pattern.split(str1)

r2 = pattern.split(str2)

r3 = pattern.split(str3)

r4 = pattern.split(str4)

print(r1, r2, r3, r4, sep=**"\n"**+**"\*"**\*30+**"\n"**)

['a', 'bb', 'cc', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb', 'cc', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb\x07a', 'dd', 'e']

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

['a', 'bb', 'aa', 'dd', 'e']

##### re.findall(pattern, string, flags=0)

作为一个字符串列表, 在**字符串**中, 返回所有非重叠匹配的**模式**. The **string**是从左到右扫描的, 所以匹配的内容是按照该顺序来的如果模式中存在一个或多个组, 请返回组列表；如果模式具有多个组, 这将是元组的列表. Return all non-overlapping matches of pattern in string, as a list of strings. The string是从左到右扫描的, 所以匹配的内容是按照该顺序来的If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result unless they touch the beginning of another match.

##### re.finditer(pattern, string, flags=0)

在**string**中为RE **模式**的所有非重叠匹配返回[iterator](http://usyiyi.cn/documents/python_352/glossary.html#term-iterator)产生[match objects](http://usyiyi.cn/documents/python_352/library/re.html#match-objects). 该**字符串**是从左到右扫描的, 匹配按照发现的顺序返回. 空匹配包含在结果中, 除非他们接触到另一场匹配的开头.

##### re.sub(pattern, repl, string, count=0, flags=0)

pattern可以是一个字符串或一个RE对象.

将**string**中最左侧非重叠出现的**pattern**替换为**repl**, 返回所获得的字符串. If the pattern isn't found, **string** is returned unchanged. **repl** 可以是一个字符串或一个函数；如果是一个字符串, 则会处理每个反斜杠转义. 即, \n被转换为单个换行符, \r被转换为回车符, 依此类推. Unknown escapes such as \& are left alone. Backreferences, such as \6, are replaced with the substring matched by group 6 in the pattern. For example:

**>>>** re.sub(r'def\s+([a-zA-Z\_][a-zA-Z\_0-9]\*)\s\*\(\s\*\):',

**...**  r'static PyObject\*\npy\_\1(void)\n{',

**...**  'def myfunc():')

'static PyObject\*\npy\_myfunc(void)\n{'

如果**repl**是一个函数, 它被呼吁每个非重叠**模式**发生. 该函数采用单个匹配对象作为参数, 并返回替换字符串. 举个例子:

>>>

**>>>** def dashrepl(matchobj):

**...**  if matchobj.group(0) == '-': return ' '

**...**  else: return '-'

**>>>** re.sub('-{1,2}', dashrepl, 'pro----gram-files')

'pro--gram files'

**>>>** re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)

'Baked Beans & Spam'

模式可以是一个字符串或Re对象.

可选参数**count**是要替换的模式出现的最大数量； **计数**必须是非负整数. 如果省略或为零, 所有事件将被替换. 只有当不与上一个匹配相邻时, 才替换模式的空匹配, 因此sub（'x \*',  ' - ',  'abc '）返回'-a-b-c-'.

In string-type **repl** arguments, in addition to the character escapes and backreferences described above, \g<name> will use the substring matched by the group named name, as defined by the (?P<name>...) syntax. \g<number>使用相应的组号；因此, \g<2>等效于\2, 但在替换例如\g<2>0中不含糊. \20将被解释为对组20的引用, 而不是对后面跟着字面值字符'0'的组2的引用. backreference \g<0>替换由RE匹配的整个子字符串.

^(.\*):(.\*)$

"\1":"\2",

#### 2.数量词的贪婪模式和非贪婪模式：

1. 贪婪模式: 在整个表达式匹配成功的前提下, 尽可能多的匹配 ( \* )；
2. 非贪婪模式: 在整个表达式匹配成功的前提下, 尽可能少的匹配 ( ? )；
3. **Python里数量词默认是贪婪的.**

**示例一 :**

源字符串: abbbc

* 使用贪婪的数量词的正则表达式 ab\* , 匹配结果: abbb.

\* 决定了尽可能多匹配 b, 所以a后面所有的 b 都出现了.

* 使用非贪婪的数量词的正则表达式ab\*?, 匹配结果: a.

即使前面有 \*, 但是 ? 决定了尽可能少匹配 b, 所以没有 b.

#### 示例二 :

源字符串: aa<div>test1</div>bb<div>test2</div>cc

* 使用贪婪的数量词的正则表达式: <div>.\*</div>
* 匹配结果: <div>test1</div>bb<div>test2</div>

这里采用的是贪婪模式. 在匹配到第一个"</div>"时已经可以使整个表达式匹配成功, 但是由于采用的是贪婪模式, 所以仍然要向右尝试匹配, 查看是否还有更长的可以成功匹配的子串. 匹配到第二个"</div>"后, 向右再没有可以成功匹配的子串, 匹配结束, 匹配结果为"<div>test1</div>bb<div>test2</div>"

* 使用非贪婪的数量词的正则表达式: <div>.\*?</div>
* 匹配结果: <div>test1</div>

正则表达式二采用的是非贪婪模式, 在匹配到第一个"</div>"时使整个表达式匹配成功, 由于采用的是非贪婪模式, 所以结束匹配, 不再向右尝试, 匹配结果为"<div>test1</div>".

[正则表达式测试网址](http://tool.oschina.net/regex/)

## 案例: 使用正则表达式的爬虫

现在拥有了正则表达式这把神兵利器, 我们就可以进行对爬取到的全部网页源代码进行筛选了.

下面我们一起尝试一下爬取内涵段子网站:  [http://www.neihan8.com/article/list\_5\_1.html](http://www.neihan8.com/article/list_5_1.html" \t "_blank)

打开之后, 不难看到里面一个一个灰常有内涵的段子, 当你进行翻页的时候, 注意url地址的变化:

* 第一页url: http: //www.neihan8.com/article/list\_5\_1 .html
* 第二页url: http: //www.neihan8.com/article/list\_5\_2 .html
* 第三页url: http: //www.neihan8.com/article/list\_5\_3 .html
* 第四页url: http: //www.neihan8.com/article/list\_5\_4 .html

这样我们的url规律找到了, 要想爬取所有的段子, 只需要修改一个参数即可. 下面我们就开始一步一步将所有的段子爬取下来吧.

### 第一步: 获取数据

**1. 按照我们之前的用法, 我们需要写一个加载页面的方法.**

这里我们统一定义一个类, 将url请求作为一个成员方法处理.

我们创建一个文件, 叫duanzi\_spider.py

然后定义一个Spider类, 并且添加一个加载页面的成员方法

import urllib2

class Spider:

"""

内涵段子爬虫类

"""

def loadPage(self, page):

"""

@brief 定义一个url请求网页的方法

@param page 需要请求的第几页

@returns 返回的页面html

"""

url = "http://www.neihan8.com/article/list\_5\_" + str(page)

+ ".html"

#User-Agent头

user\_agent = 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT

6.1; Trident/5.0'

headers = {'User-Agent': user\_agent}

req = urllib2.Request(url, headers = headers)

response = urllib2.urlopen(req)

html = response.read()

print html

#return html

**以上的loadPage的实现体想必大家应该很熟悉了, 需要注意定义python类的成员方法需要额外添加一个参数self.**

* 那么loadPage(self, page) 中的page是我们指定去请求第几页.
* 最后通过 print html打印到屏幕上.
* 然后我们写一个main函数见到测试一个loadPage方法

**2. 写main函数测试一个loadPage方法**

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

"""

======================

内涵段子小爬虫

======================

"""

print '请按下回车开始'

raw\_input()

#定义一个Spider对象

mySpider = Spider()

mySpider.loadpage(1)

* 程序正常执行的话, 我们会在屏幕上打印了内涵段子第一页的全部html代码. 但是我们发现, html中的中文部分显示的可能是乱码 .

**那么我们需要简单的将得到的网页源代码处理一下:**

def loadPage(self, page):

"""

@brief 定义一个url请求网页的方法

@param page 需要请求的第几页

@returns 返回的页面html

"""

url = "http://www.neihan8.com/article/list\_5\_" + str(page)+ ".html"

#User-Agent头

user\_agent = 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT6.1; Trident/5.0'

headers = {'User-Agent': user\_agent}

req = urllib2.Request(url, headers = headers)

response = urllib2.urlopen(req)

html = response.read()

gbk\_html = html.decode('gbk').encode('utf-8')

# print gbk\_html

return gbk\_html

注意 : 对于每个网站对中文的编码各自不同, 所以html.decode('gbk')的写法并不是通用写法, 根据网站的编码而异

* 这样我们再次执行以下duanzi\_spider.py , 会发现之前的中文乱码可以正常显示了.

### 第二步: 筛选数据

接下来我们已经得到了整个页面的数据. 但是, 很多内容我们并不关心, 所以下一步我们需要进行筛选. 如何筛选, 就用到了上一节讲述的正则表达式.

* 首先

import re

* 然后, 在我们得到的gbk\_html中进行筛选匹配.

**我们需要一个匹配规则:**

我们可以打开内涵段子的网页, 鼠标点击右键 " 查看源代码 " 你会惊奇的发现, 我们需要的每个段子的内容都是在一个 <div>标签中, 而且每个div都有一个属性class = "f18 mb20"

<a href="/article/44959.html"><b>回家奔丧</b></a></h4>

<div class="f18 mb20">

一老太太跋山涉水来到部队, 看望她的孙子, <br />

　　警卫问："她找谁？"老太说："找xx, "警卫打完电话说：<br />

　　"xx三天前说她他奶奶过世, 回家奔丧去了, 奔丧去了, 去了. . "

</div>

所以, 我们只需要匹配到网页中所有<div class="f18 mb20"> 到 </div> 的数据就可以了.

**根据正则表达式, 我们可以推算出一个公式是:**

<div.\*?class="f18 mb20">(.\*?)</div>

* 这个表达式实际上就是匹配到所有div中class="f18 mb20 里面的内容(具体可以看前面正则介绍)
* 然后将这个正则应用到代码中, 我们会得到以下代码:

def loadPage(self, page):

"""

@brief 定义一个url请求网页的方法

@param page 需要请求的第几页

@returns 返回的页面html

"""

url = "http://www.neihan8.com/article/list\_5\_" + str(page)+ ".html"

#User-Agent头

user\_agent = 'Mozilla/5.0 (compatible; MSIE 9.0; Windows NT6.1; Trident/5.0'

headers = {'User-Agent': user\_agent}

req = urllib2.Request(url, headers = headers)

response = urllib2.urlopen(req)

html = response.read()

gbk\_html = html.decode('gbk').encode('utf-8')

#找到所有的段子内容<div class = "f18 mb20"></div>

#re.S 如果没有re.S 则是只匹配一行有没有符合规则的字符串, 如果没有则下一行重新匹配

# 如果加上re.S 则是将所有的字符串将一个整体进行匹配

pattern = re.compile(r'<div.\*?class="f18 mb20">(.\*?)</div>', re.S)

item\_list = pattern.findall(gbk\_html)

return item\_list

def printOnePage(self, item\_list, page):

"""

@brief 处理得到的段子列表

@param item\_list 得到的段子列表

@param page 处理第几页

"""

print "\*\*\*\*\*\*\* 第 %d 页 爬取完毕...\*\*\*\*\*\*\*" %page

for item in item\_list:

print "================"

print ite

re.**S**

re.**DOTALL**

Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything except a newline. Corresponds to the inline flag (?s).

* 这里需要注意一个是re.S是正则表达式中匹配的一个参数.
* 如果 没有re.S 则是 只匹配一行 有没有符合规则的字符串, 如果没有则下一行重新匹配.
* 如果 加上re.S 则是将 所有的字符串 将一个整体进行匹配, findall 将所有匹配到的结果封装到一个list中.
* 然后我们写了一个遍历item\_list的一个方法 printOnePage() . ok程序写到这, 我们再一次执行一下.

Power@PowerMac ~$ python duanzi\_spider.py

**我们第一页的全部段子, 不包含其他信息全部的打印了出来.**

* 你会发现段子中有很多 <p> , </p> 很是不舒服, 实际上这个是html的一种段落的标签.
* 在浏览器上看不出来, 但是如果按照文本打印会有<p>出现, 那么我们只需要把我们不希望的内容去掉即可了.
* 我们可以如下简单修改一下 printOnePage().

def printOnePage(self, item\_list, page):

"""

@brief 处理得到的段子列表

@param item\_list 得到的段子列表

@param page 处理第几页

"""

print "\*\*\*\*\*\*\* 第 %d 页 爬取完毕...\*\*\*\*\*\*\*" %page

for item in item\_list:

print "================"

item = item.replace("<p>", "").replace("</p>", "").replace("<br />", "")

print item

### 第三步: 保存数据

* 我们可以将所有的段子存放在文件中. 比如, 我们可以将得到的每个item不是打印出来, 而是存放在一个叫 duanzi.txt 的文件中也可以.

def writeToFile(self, text):

'''

@brief 将数据追加写进文件中

@param text 文件内容

'''

myFile = open("./duanzi.txt", 'a') #追加形式打开文件

myFile.write(text)

myFile.write("---------------------------------------------

--------")

myFile.close()

* 我们可以如下简单修改一下 get\_content().

j = re.sub(r"[<b>|</b>|<br />|<br />|<p>|</p>|\\u3000|\\r\\n|\s]","",j)

j = j.replace("&ldqo;",'"').replace("&helli;","...").replace("&dqo;",'"').strip()

* 然后我们将print的语句 改成writeToFile() , 当前页面的所有段子就存在了本地的MyStory.txt文件中.

def printOnePage(self, item\_list, page):

'''

@brief 处理得到的段子列表

@param item\_list 得到的段子列表

@param page 处理第几页

'''

print "\*\*\*\*\*\*\* 第 %d 页 爬取完毕...\*\*\*\*\*\*\*" %page

for item in item\_list:

# print "================"

item = item.replace("<p>", "").replace("</p>", "").replace("<br />", "")

# print item

self.writeToFile(item)

### 第四步: 显示数据

* 接下来我们就通过参数的传递对page进行叠加来遍历 内涵段子吧的全部段子内容.
* 只需要在外层加一些逻辑处理即可.

def doWork(self):

'''

让爬虫开始工作

'''

while self.enable:

try:

item\_list = self.loadPage(self.page)

except urllib2.URLError, e:

print e.reason

continue

#对得到的段子item\_list处理

self.printOnePage(item\_list, self.page)

self.page += 1 #此页处理完毕, 处理下一页

print "按回车继续..."

print "输入 quit 退出"

command = raw\_input()

if (command == "quit"):

self.enable = False

break

* 最后, 我们执行我们的代码, 完成后查看当前路径下的duanzi.txt文件, 里面已经有了我们要的内涵段子.

**以上便是一个非常精简使用的小爬虫程序, 使用起来很是方便, 如果想要爬取其他网站的信息, 只需要修改其中某些参数和一些细节就行了.**

## 案例：使用正则表达式的爬虫-新版

现在拥有了正则表达式这把神兵利器, 我们就可以进行对爬取到的全部网页源代码进行筛选了.

下面我们一起尝试一下爬取内涵段子网站： <http://www.neihan8.com/article/list_5_1.html>

打开之后, 不难看到里面一个一个灰常有内涵的段子, 当你进行翻页的时候, 注意url地址的变化：

* 第一页url: http: //www.neihan8.com/article/list\_5\_1 .html
* 第二页url: http: //www.neihan8.com/article/list\_5\_2 .html
* 第三页url: http: //www.neihan8.com/article/list\_5\_3 .html
* 第四页url: http: //www.neihan8.com/article/list\_5\_4 .html

这样我们的url规律找到了, 要想爬取所有的段子, 只需要修改一个参数即可. 下面我们就开始一步一步将所有的段子爬取下来吧.

### 第一步：获取数据

1. 按照我们之前的用法, 我们需要写一个加载页面的方法.

这里我们统一定义一个类, 将url请求作为一个成员方法处理.

我们创建一个文件, 叫duanzi\_spider.py

然后定义一个Spider类, 并且添加一个加载页面的成员方法

class Duanzi\_spider():

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

self.url = "http://www.neihan8.com/article/list\_5\_%s.html"

self.headers = {

"User\_Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_5) AppleW\

ebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36",

"Accept-Encoding":None,

"Accept-Language": "zh-CN,zh;q=0.8"

}

def load\_page(self,url):

'''可以复用的页面请求方法

'''

response = requests.get(url,timeout=10,headers=self.headers)

if response.status\_code==200:

print(response.request.headers)

return response.content.decode("gbk")

else:

raise ValueError("status\_code is:",response.status\_code)

* 程序正常执行的话, 我们会在屏幕上打印了内涵段子第一页的全部html代码. 但是我们发现, html中的中文部分显示的可能是乱码 .

注意 ：对于每个网站对中文的编码各自不同, 所以html.decode('gbk')的写法并不是通用写法, 根据网站的编码而异

### 第二步：筛选数据

接下来我们已经得到了整个页面的数据. 但是, 很多内容我们并不关心, 所以下一步我们需要进行筛选. 如何筛选, 就用到了上一节讲述的正则表达式.

* 首先

import re

* 然后, 在我们得到的response中进行筛选匹配.

我们需要一个匹配规则:

我们可以打开内涵段子的网页, 鼠标点击右键 " 查看源代码 " 你会发现每条段子的内容大致如下

<a href="/article/44959.html"><b>回家奔丧</b></a></h4>

<div class="f18 mb20">

　　一老太太跋山涉水来到部队, 看望她的孙子, <br />

　　警卫问："她找谁？"老太说："找xx, "警卫打完电话说：<br />

　　"xx三天前说她他奶奶过世, 回家奔丧去了, 奔丧去了, 去了. . "

</div>

def get\_content(self,html):

''' 根据网页内容, 同时匹配标题和段子内容

'''

pattern = re.compile(r'<a\shref="/article/\d+\.html">(.\*?)</a>.\*?<div\sclass="f18 mb20">(.\*?)</div>', re.S)

t = pattern.findall(html)

result = []

for i in t:

temp = []

for j in i:

j = re.sub(r"[<b>|</b>|<br />|<br>|<p>|</p>|\\u3000|\\r\\n|\s]","",j)

j = j.replace("&ldqo;",'"').replace("&helli;","...").replace("&dqo;",'"').strip()

# j = re.sub(r"[&ldqo;|&dqo;]","\"",j)?

# j = re.sub(r"…","...",j)

temp.append(j)

print(temp)

result.append(temp)

return result

* 这里需要注意一个是re.S是正则表达式中匹配的一个参数.
* 如果 没有re.S 则是 只匹配一行 有没有符合规则的字符串, 如果没有则下一行重新匹配.
* 如果 加上re.S 则是将 所有的字符串 将一个整体进行匹配, findall 将所有匹配到的结果封装到一个list中.
* ok程序写到这, 我们再一次执行一下.

Power@PowerMac ~$ python duanzi\_spider.py

我们第一页的全部段子, 不包含其他信息全部的打印了出来.

* 你会发现段子中有很多 <p> , </p> 很是不舒服, 实际上这个是html的一种段落的标签.
* 在浏览器上看不出来, 但是如果按照文本打印会有<p>出现, 那么我们只需要把我们不希望的内容去掉即可了.
* 我们可以如下简单修改一下 get\_content().

j = re.sub(r"[<b>|</b>|<br />|<br>|<p>|</p>|\\u3000|\\r\\n|\s]","",j)

j = j.replace("&ldqo;",'"').replace("&helli;","...").replace("&dqo;",'"').strip()

### 第三步：保存数据

* 我们可以将所有的段子存放在文件中. 比如, 我们可以将得到的每个item不是打印出来, 而是存放在一个叫 duanzi.txt 的文件中也可以.

def save\_content(self,content):

myFile = open("./duanzi.txt", 'a')

for temp in content:

myFile.write("\n"+temp[0]+"\n"+temp[1]+"\n")

myFile.write("-----------------------------------------------------")

myFile.close()

* 然后我们实现保存的方法 , 当前页面的所有段子就存在了本地的duanzi.txt文件中.

### 第四步：实现循环抓取

* 接下来我们就通过参数的传递对page进行叠加来遍历 内涵段子吧的全部段子内容.
* 同时也通过这个run方法实现整个程序的主要逻辑

def run(self):

i = 1

while True:

html = self.load\_page(self.url%i)

result = self.get\_content(html)

print ("按回车继续...")

print ("输入 quit 退出")

command = input()

if (command == "quit"):

break

i+=1

最后, 我们执行我们的代码, 完成后查看当前路径下的duanzi.txt文件, 里面已经有了我们要的内涵段子.

**以上便是一个非常精简使用的小爬虫程序, 使用起来很是方便, 如果想要爬取其他网站的信息, 只需要修改其中某些参数和一些细节就行了.**

继续完善, 使用多线程和多进程来完成.

添加功能, 同时爬取标题和正文

添加功能, 判断是否到最后一页, 如果到最后一页, 就停止

添加功能, 使用中文进行匹配.

添加功能, 每爬取5页就检测一下, 如果用户输入了q, quit, exit等, 就结束 爬取. 可以使用非阻塞的多进程来完成.

### 动手

* 获取<http://36kr.com/网站首页的所有新闻>

## XPath与lxml类库

**使用XPath解析时如果取不到内容, 就把user-agent改为IE的, 这样一般就能取出来了.**

**W3School官方文档：http://www.w3school.com.cn/xpath/index.asp**

有同学说, 我正则用的不好, 处理HTML文档很累, 有没有其他的方法？

有！那就是XPath, 我们可以先将 HTML文件 转换成 XML文档, 然后用 XPath 查找 HTML 节点或元素.

### 什么是XML

* XML 指可扩展标记语言(EXtensible Markup Language)
* XML 是一种标记语言, 很类似 HTML
* XML 的设计宗旨是传输数据, 而非显示数据
* XML 的标签需要我们自行定义.
* XML 被设计为具有自我描述性.
* XML 是 W3C 的推荐标准

W3School官方文档: <http://www.w3school.com.cn/xml/index.asp>

#### XML 和 HTML 的区别

|  |  |  |
| --- | --- | --- |
| 数据格式 | 描述 | 设计目标 |
| XML | Extensible Markup Language (可扩展标记语言) | 被设计为传输和存储数据, 其焦点是数据的内容. |
| HTML | HyperText Markup Language (超文本标记语言) | 显示数据以及如何更好显示数据. |
| HTML DOM | Document Object Model for HTML (文档对象模型) | 通过 HTML DOM, 可以访问所有的 HTML 元素, 连同它们所包含的文本和属性. 可以对其中的内容进行修改和删除, 同时也可以创建新的元素. |

##### XML文档示例

<?xml version="1.0" encoding="utf-8"?>

<bookstore>

<book category="cooking">

<title lang="en">Everyday Italian</title>

<author>Giada De Laurentiis</author>

<year>2005</year>

<price>30.00</price>

</book>

<book category="children">

<title lang="en">Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

<book category="web">

<title lang="en">XQuery Kick Start</title>

<author>James McGovern</author>

<author>Per Bothner</author>

<author>Kurt Cagle</author>

<author>James Linn</author>

<author>Vaidyanathan Nagarajan</author>

<year>2003</year>

<price>49.99</price>

</book>

<book category="web" cover="paperback">

<title lang="en">Learning XML</title>

<author>Erik T. Ray</author>

<year>2003</year>

<price>39.95</price>

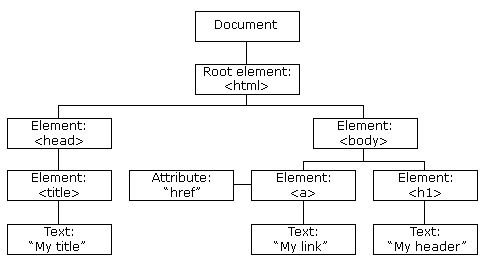
</book>

</bookstore>

把文档复制到XMLQuire中, 然后查找练习

##### HTML DOM 模型示例

HTML DOM 定义了访问和操作 HTML 文档的标准方法, 以树结构方式表达 HTML 文档.



#### XML的节点关系

##### 1. 父(Parent)

每个元素以及属性都有一个父.

下面是一个简单的XML例子中, book 元素是 title、author、year 以及 price 元素的父:

<?xml version="1.0" encoding="utf-8"?>

<book>

<title>Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

##### 2. 子(Children)

元素节点可有零个、一个或多个子.

在下面的例子中, title、author、year 以及 price 元素都是 book 元素的子:

<?xml version="1.0" encoding="utf-8"?>

<book>

<title>Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

##### 3. 同胞(Sibling)

拥有相同的父的节点

在下面的例子中, title、author、year 以及 price 元素都是同胞:

<?xml version="1.0" encoding="utf-8"?>

<book>

<title>Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

##### 4. 先辈(Ancestor)

某节点的父、父的父, 等等.

在下面的例子中, title 元素的先辈是 book 元素和 bookstore 元素:

<?xml version="1.0" encoding="utf-8"?>

<bookstore>

<book>

<title>Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

</bookstore>

##### 5. 后代(Descendant)

某个节点的子, 子的子, 等等.

在下面的例子中, bookstore 的后代是 book、title、author、year 以及 price 元素:

<?xml version="1.0" encoding="utf-8"?>

<bookstore>

<book>

<title>Harry Potter</title>

<author>J K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

</bookstore>

### 什么是XPath？

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

W3School官方文档: <http://www.w3school.com.cn/xpath/index.asp>

#### XPath 开发工具

1. 开源的XPath表达式编辑工具:XMLQuire(XML格式文件可用)
2. Chrome插件 XPath Helper
3. Firefox插件 XPath Checker

#### 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.

下面列出了最常用的路径表达式:

|  |  |
| --- | --- |
| 表达式 | 描述 |
| nodename | 选取此节点的所有子节点. |
| / | 从根节点选取. |
| // | 从匹配选择的当前节点选择文档中的节点, 而不考虑它们的位置. |
| . | 选取当前节点. |
| .. | 选取当前节点的父节点. |
| @ | 选取属性. |

在下面的表格中, 我们已列出了一些路径表达式以及表达式的结果:

|  |  |  |
| --- | --- | --- |
|  | 路径表达式 | 结果 |
| bookstore | 选取 bookstore 元素的所有子节点. |  |
| /bookstore | 选取根元素 bookstore. 注释: 假如路径起始于正斜杠( / ), 则此路径始终代表到某元素的绝对路径！ |  |
| bookstore/book | 选取属于 bookstore 的子元素的所有 book 元素. |  |
| //book | 选取所有 book 子元素, 而不管它们在文档中的位置. |  |
| bookstore//book | 选择属于 bookstore 元素的后代的所有 book 元素, 而不管它们位于 bookstore 之下的什么位置. |  |
| //@lang | 选取名为 lang 的所有属性. |  |

#### 谓语(Predicates)

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 | 结果 |
| /bookstore/book[1] | 选取属于 bookstore 子元素的第一个 book 元素. |
| /bookstore/book[last()] | 选取属于 bookstore 子元素的最后一个 book 元素. |
| /bookstore/book[last()-1] | 选取属于 bookstore 子元素的倒数第二个 book 元素. |
| /bookstore/book[position()<3] | 选取最前面的两个属于 bookstore 元素的子元素的 book 元素. |
| //title[@lang] | 选取所有拥有名为 lang 的属性的 title 元素. |
| //title[@lang='eng'] | 选取所有 title 元素, 且这些元素拥有值为 eng 的 lang 属性. |
| /bookstore/book[price>35.00] | 选取 bookstore 元素的所有 book 元素, 且其中的 price 元素的值须大于 35.00. |
| /bookstore/book[price>35.00]/title | 选取 bookstore 元素中的 book 元素的所有 title 元素, 且其中的 price 元素的值须大于 35.00. |

#### 选取未知节点

XPath 通配符可用来选取未知的 XML 元素.

|  |  |
| --- | --- |
| 通配符 | 描述 |
| \* | 匹配任何元素节点. |
| @\* | 匹配任何属性节点. |
| node() | 匹配任何类型的节点. |

在下面的表格中, 我们列出了一些路径表达式, 以及这些表达式的结果:

|  |  |
| --- | --- |
| 路径表达式 | 结果 |
| /bookstore/\* | 选取 bookstore 元素的所有子元素. |
| //\* | 选取文档中的所有元素. |
| //title[@\*] | 选取所有带有属性的 title 元素. |

#### 选取若干路径

通过在路径表达式中使用"|"运算符, 您可以选取若干个路径.

实例

在下面的表格中, 我们列出了一些路径表达式, 以及这些表达式的结果:

|  |  |
| --- | --- |
| 路径表达式 | 结果 |
| //book/title | //book/price | 选取所有book元素以及book元素下所有的title元素 |
| //title | //price | 选取文档中的所有 title 和 price 元素. |
| /bookstore/book/title | //price | 选取属于 bookstore 元素的 book 元素的所有 title 元素, 以及文档中所有的 price 元素. |

#### XPath的运算符

下面列出了可用在 XPath 表达式中的运算符:



这些就是XPath的语法内容, 在运用到Python抓取时要先转换为xml.

#### Xpath模糊查询

*contains()模糊查询方法, 第一个参数是要匹配的标签或属性名, 如下面查找的是id, 第二个参数是要匹配的字符串, 是标签或属性的部分内容*

node\_list = text.xpath(**'//div[contains(@id, "qiushi\_tag")]'**)

### 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 (或通过wheel方式安装)

#### 初步使用

我们利用它来解析 HTML 代码, 简单示例:

# lxml\_test.py

# 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)

# python3

result = etree.tostring(html).decode("utf-8")

print(result)

输出结果:

<**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**>

lxml 可以自动修正 html 代码, 例子里不仅补全了 li 标签, 还添加了 body, html 标签.

#### lxml从文件中读取内容:

除了直接读取字符串, lxml还支持从文件里读取内容. 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>

再利用 etree.parse() 方法来读取文件.

# lxml\_parse.py

from lxml import etree

# 读取外部文件 hello.html

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

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

# python3

result = etree.tostring(html, pretty\_print=**True**).decode(**"utf-8"**)

print(result)

输出结果与之前相同:

<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>

### XPath实例测试

#### 1. 获取所有的 <li> 标签

# xpath\_li.py

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

print len(result)

print type(result)

print type(result[0])

输出结果:

<type 'lxml.etree.\_ElementTree'>

[<Element li at 0x1014e0e18>, <Element li at 0x1014e0ef0>, <Element li at 0x1014e0f38>, <Element li at 0x1014e0f80>, <Element li at 0x1014e0fc8>]

5

<type 'list'>

<type 'lxml.etree.\_Element'>

#### 2. 获取<li> 标签的所有 class属性

# xpath\_li.py

from lxml import etree

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

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

print result

运行结果

['item-0', 'item-1', 'item-inactive', 'item-1', 'item-0']

#### 3. 获取<li>标签下href 为 link1.html 的 <a> 标签

# xpath\_li.py

from lxml import etree

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

result = html.xpath('//li/a[@href="link1.html"]')

print result

运行结果

[<Element a at 0x10ffaae18>]

#### 4. 获取<li> 标签下的所有 <span> 标签

# xpath\_li.py

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

运行结果

[<Element span at 0x10d698e18>]

#### 5. 获取 <li> 标签下的<a>标签里的所有 class

# xpath\_li.py

from lxml import etree

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

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

print result

运行结果

['blod']

#### 6. 获取最后一个 <li> 的 <a> 的 href

# xpath\_li.py

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

运行结果

['link5.html']

#### 7. 获取倒数第二个元素的内容

# xpath\_li.py

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

运行结果

fourth item

#### 8. 获取 class 值为 bold 的标签名

# xpath\_li.py

from lxml import etree

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

result = html.xpath('//\*[@class="bold"]')

# tag方法可以获取标签名

print result[0].tag

运行结果

span

## 案例: 使用XPath的爬虫

现在我们用XPath来做一个简单的爬虫, 我们尝试爬取某个贴吧里的所有帖子, 并且将该这个帖子里每个楼层发布的图片下载到本地.

# tieba\_xpath.py

#!/usr/bin/env python

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

import os

import urllib

import urllib2

from lxml import etree

class Spider:

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

self.tiebaName = raw\_input("请需要访问的贴吧: ")

self.beginPage = int(raw\_input("请输入起始页: "))

self.endPage = int(raw\_input("请输入终止页: "))

self.url = 'http://tieba.baidu.com/f'

self.ua\_header = {"User-Agent" : "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1 Trident/5.0;"}

# 图片编号

self.userName = 1

def tiebaSpider(self):

for page in range(self.beginPage, self.endPage + 1):

pn = (page - 1) \* 50 # page number

word = {'pn' : pn, 'kw': self.tiebaName}

word = urllib.urlencode(word) #转换成url编码格式(字符串)

myUrl = self.url + "?" + word

# 示例: http://tieba.baidu.com/f? kw=%E7%BE%8E%E5%A5%B3 & pn=50

# 调用 页面处理函数 load\_Page

# 并且获取页面所有帖子链接,

links = self.loadPage(myUrl) # urllib2\_test3.py

# 读取页面内容

def loadPage(self, url):

req = urllib2.Request(url, headers = self.ua\_header)

html = urllib2.urlopen(req).read()

# 解析html 为 HTML 文档

selector=etree.HTML(html)

#抓取当前页面的所有帖子的url的后半部分, 也就是帖子编号

# http://tieba.baidu.com/p/4884069807里的 "p/4884069807"

links = selector.xpath('//div[@class="threadlist\_lz clearfix"]/div/a/@href')

# links 类型为 etreeElementString 列表

# 遍历列表, 并且合并成一个帖子地址, 调用 图片处理函数 loadImage

for link in links:

link = "http://tieba.baidu.com" + link

self.loadImages(link)

# 获取图片

def loadImages(self, link):

req = urllib2.Request(link, headers = self.ua\_header)

html = urllib2.urlopen(req).read()

selector = etree.HTML(html)

# 获取这个帖子里所有图片的src路径

imagesLinks = selector.xpath('//img[@class="BDE\_Image"]/@src')

# 依次取出图片路径, 下载保存

for imagesLink in imagesLinks:

self.writeImages(imagesLink)

# 保存页面内容

def writeImages(self, imagesLink):

'''

将 images 里的二进制内容存入到 userNname 文件中

'''

print imagesLink

print "正在存储文件 %d ..." % self.userName

# 1. 打开文件, 返回一个文件对象

file = open('./images/' + str(self.userName) + '.png', 'wb')

# 2. 获取图片里的内容

images = urllib2.urlopen(imagesLink).read()

# 3. 调用文件对象write() 方法, 将page\_html的内容写入到文件里

file.write(images)

# 4. 最后关闭文件

file.close()

# 计数器自增1

self.userName += 1

# 模拟 main 函数

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

# 首先创建爬虫对象

mySpider = Spider()

# 调用爬虫对象的方法, 开始工作

mySpider.tiebaSpider()

**视频演示:**

表示任意目录下查找class="threadlist\_lz clearfix"的标签

//div[@class="threadlist\_lz clearfix"]

//div[@class="threadlist\_lz clearfix"]//a[@class="j\_th\_tit"]/@href

取href的值

https://tieba.baidu.com/p/5092200667

//img[@class="BDE\_Image"]

在任意目录下查找class='BDE\_Image'的img标签

//img[@class="BDE\_Image"]/@src

就能找到所有的image的src标签

tieba\_xpath.py

Version1不定义类

**import** urllib.parse, urllib.request

**from** lxml **import** etree

**def** loadPage(url):

print(**"正在下载"**)

*#网站会根据不同的浏览器对内容进行一定的修改, 所以用xpath时最好写成IE的user-agent*

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko'**

headers = {**'User-Agent'**: user\_agent}

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

html = urllib.request.urlopen(request).read()

*#解析HTML文档为XMl DOM模型*

page = etree.HTML(html)

*#返回所有匹配成功的列表*

link\_list = page.xpath(**'//\*[@id="thread\_list"]/li/div/div/div[1]/div[1]/a/@href'**)

print(link\_list)

**for** link **in** link\_list:

*#full\_link是帖子的链接*

full\_link = **"http://tieba.baidu.com"** + link

print(full\_link)

loadImage(full\_link)

*#取出每个帖子里的图片链接*

**def** loadImage(link):

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'**

headers = {**'User-Agent'**: user\_agent}

request = urllib.request.Request(link, headers=headers)

html = urllib.request.urlopen(request).read()

page = etree.HTML(html)

*#返回帖子里的所有图片地址的列表*

image\_list = page.xpath(**'//img[@class="BDE\_Image"]/@src'**)

**for** link **in** image\_list:

writeImage(link)

*#把图片写入到本地*

**def** writeImage(link):

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'**

headers = {**'User-Agent'**: user\_agent}

request = urllib.request.Request(link, headers=headers)

image = urllib.request.urlopen(request).read()

file\_name = link[-10:]

**with** open(file\_name, **'wb'**) **as** f:

f.write(image)

**def** tiebaSpider(url, startPage, endPage):

*'''*

*作用：贴吧爬虫调度器, 负责组合处理每个页面的url*

**:param** *url: 贴吧url的前部分*

**:param** *startPage: 开始页*

**:param** *endPage: 结束页*

*'''*

**for** page **in** range(startPage,endPage+1):

pn = (page - 1)\*50

full\_url = url + **"&pn="** +str(pn)

print(full\_url)

loadPage(full\_url)

*# loadPage(html)*

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

kw = input(**"请输入需要爬取的贴吧名："**)

startPage = int(input(**"请输入起始页："**))

endPage = int(input(**"请输入结束页："**))

url = **"http://tieba.baidu.com/f?"**

key = urllib.parse.urlencode({**'kw'**:kw})

full\_url = url + key

tiebaSpider(full\_url, startPage, endPage)

使用request改写上面的程序

**import** requests

**from** lxml **import** etree

**def** loadPage(url):

print(**"正在下载"**)

*#网站会根据不同的浏览器对内容进行一定的修改, 所以用xpath时最好写成IE的user-agent*

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko'**

headers = {**'User-Agent'**: user\_agent}

request = requests.get(url, headers=headers)

html = request.text

*#解析HTML文档为XMl DOM模型*

page = etree.HTML(html)

*#返回所有匹配成功的列表*

link\_list = page.xpath(**'//\*[@id="thread\_list"]/li/div/div/div[1]/div[1]/a/@href'**)

print(link\_list)

**for** link **in** link\_list:

*#full\_link是帖子的链接*

full\_link = **"http://tieba.baidu.com"** + link

print(full\_link)

loadImage(full\_link)

*#取出每个帖子里的图片链接*

**def** loadImage(link):

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'**

headers = {**'User-Agent'**: user\_agent}

request = requests.get(link, headers=headers)

html = request.text

page = etree.HTML(html)

*#返回帖子里的所有图片地址的列表*

image\_list = page.xpath(**'//img[@class="BDE\_Image"]/@src'**)

**for** link **in** image\_list:

writeImage(link)

*#把图片写入到本地*

**def** writeImage(link):

user\_agent = **'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'**

headers = {**'User-Agent'**: user\_agent}

request = requests.get(link, headers=headers)

print(link)

*#获取图片里的内容*

images = requests.get(link)

file\_name = link[-15:]

print(**"正在存储文件 %s ..."** %file\_name)

*# 3. 调用文件对象write() 方法, 将page\_html的内容写入到文件里*

**with** open(**"./images/"** + file\_name, **'wb'**) **as** f:

**for** chunk **in** images.iter\_content(100):

f.write(chunk)

**def** tiebaSpider(url, startPage, endPage):

*'''*

*作用：贴吧爬虫调度器, 负责组合处理每个页面的url*

**:param** *url: 贴吧url的前部分*

**:param** *startPage: 开始页*

**:param** *endPage: 结束页*

*'''*

**for** page **in** range(startPage,endPage+1):

pn = (page - 1)\*50

full\_url = url + **"&pn="** +str(pn)

print(full\_url)

loadPage(full\_url)

*# loadPage(html)*

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

kw = input(**"请输入需要爬取的贴吧名："**)

startPage = int(input(**"请输入起始页："**))

endPage = int(input(**"请输入结束页："**))

*# 示例：http://tieba.baidu.com/f?kw=%E7%BE%8E%E5%A5%B3 & pn=50*

url = **"http://tieba.baidu.com/f?kw="**

full\_url = url + kw

tiebaSpider(full\_url, startPage, endPage)

Version2 定义类

tieba\_xpath.py

**import** urllib.parse, urllib.request

**from** lxml **import** etree

**class** Spider:

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

self.tiebaName = input(**"请需要访问的贴吧："**)

self.beginPage = int(input(**"请输入起始页："**))

self.endPage = int(input(**"请输入终止页："**))

self.url = **'http://tieba.baidu.com/f'**

self.ua\_header = {**"User-Agent"** : **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

**def** tiebaSpider(self):

**for** page **in** range(self.beginPage, self.endPage + 1):

pn = (page - 1) \* 50 *# page number*

word = {**'pn'** : pn, **'kw'**: self.tiebaName}

word = urllib.parse.urlencode(word) *#转换成url编码格式（字符串）*

myUrl = self.url + **"?"** + word

*# 示例：http://tieba.baidu.com/f? kw=%E7%BE%8E%E5%A5%B3 & pn=50*

*# 调用 页面处理函数 load\_Page, 并且获取页面所有帖子链接,*

links = self.loadPage(myUrl)

*# 读取页面内容*

**def** loadPage(self, url):

req = urllib.request.Request(url, headers = self.ua\_header)

html = urllib.request.urlopen(req).read()

*# 解析html 为 HTML 文档*

selector=etree.HTML(html)

*#抓取当前页面的所有帖子的url的后半部分, 也就是帖子编号*

*# http://tieba.baidu.com/p/4884069807里的 "p/4884069807"*

links = selector.xpath(**'//div[@class="threadlist\_lz clearfix"]/div/a/@href'**)

*# links 类型为 etreeElementString 列表*

*# 遍历列表, 并且合并成一个帖子地址, 调用 图片处理函数 loadImage*

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

link = **"http://tieba.baidu.com"** + link

self.loadImages(link)

*# 获取图片*

**def** loadImages(self, link):

req = urllib.request.Request(link, headers = self.ua\_header)

html = urllib.request.urlopen(req).read()

selector = etree.HTML(html)

*# 获取这个帖子里所有图片的src路径*

imagesLinks = selector.xpath(**'//img[@class="BDE\_Image"]/@src'**)

*# 依次取出图片路径, 下载保存*

**for** imagesLink **in** imagesLinks:

self.writeImages(imagesLink)

*# 保存页面内容*

**def** writeImages(self, imagesLink):

*'''*

*将 images 里的二进制内容存入到 userNname 文件中*

*'''*

print(imagesLink)

file\_name = imagesLink[-10:]

print(**"正在存储文件 %s ..."** %file\_name)

*# 2. 获取图片里的内容*

images = urllib.request.urlopen(imagesLink).read()

*# 3. 调用文件对象write() 方法, 将page\_html的内容写入到文件里*

**with** open(**"./images/"**+file\_name, **'wb'**) **as** f:

f.write(images)

*# 模拟 main 函数*

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

*# 首先创建爬虫对象*

mySpider = Spider()

*# 调用爬虫对象的方法, 开始工作*

mySpider.tiebaSpider()

**使用requests改写上面的程序**

tieba\_xpath.py

**import** requests

**from** lxml **import** etree

**class** Spider:

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

self.tiebaName = input(**"请需要访问的贴吧："**)

self.beginPage = int(input(**"请输入起始页："**))

self.endPage = int(input(**"请输入终止页："**))

self.url = **'http://tieba.baidu.com/f?kw='**

self.ua\_header = {**"User-Agent"** : **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko"**}

**def** tiebaSpider(self):

**for** page **in** range(self.beginPage, self.endPage + 1):

pn = (page - 1) \* 50 *# page number*

*# 示例：http://tieba.baidu.com/f? kw=%E7%BE%8E%E5%A5%B3 & pn=50*

myUrl = self.url + self.tiebaName +**"&pn="** + str(pn)

*# 调用 页面处理函数 load\_Page, 并且获取页面所有帖子链接,*

links = self.loadPage(myUrl)

*# 读取页面内容*

**def** loadPage(self, url):

req = requests.get(url, headers = self.ua\_header)

html = req.text

*# 解析html 为 HTML 文档*

selector=etree.HTML(html)

*#抓取当前页面的所有帖子的url的后半部分, 也就是帖子编号*

*# http://tieba.baidu.com/p/4884069807里的 "p/4884069807"*

links = selector.xpath(**'//div[@class="threadlist\_lz clearfix"]/div/a/@href'**)

*# links 类型为 etreeElementString 列表*

*# 遍历列表, 并且合并成一个帖子地址, 调用 图片处理函数 loadImage*

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

link = **"http://tieba.baidu.com"** + link

self.loadImages(link)

*# 获取图片*

**def** loadImages(self, link):

req = requests.get(link, headers = self.ua\_header)

html = req.text

selector = etree.HTML(html)

*# 获取这个帖子里所有图片的src路径*

imagesLinks = selector.xpath(**'//img[@class="BDE\_Image"]/@src'**)

*# 依次取出图片路径, 下载保存*

**for** imagesLink **in** imagesLinks:

self.writeImages(imagesLink)

*# 保存页面内容*

**def** writeImages(self, imagesLink):

*'''*

*将 images 里的二进制内容存入到 userNname 文件中*

*'''*

print(imagesLink)

file\_name = imagesLink[-15:]

print(**"正在存储文件 %s ..."** %file\_name)

*# 2. 获取图片里的内容*

images = requests.get(imagesLink)

*# 3. 调用文件对象write() 方法, 将page\_html的内容写入到文件里*

**with** open(**"./images/"** + file\_name, **'wb'**) **as** f:

**for** chunk **in** images.iter\_content(100):

f.write(chunk)

*# 模拟 main 函数*

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

*# 首先创建爬虫对象*

mySpider = Spider()

*# 调用爬虫对象的方法, 开始工作*

mySpider.tiebaSpider()

**新版: 使用xpath的爬虫**

现在我们用XPath来做一个简单的爬虫, 我们尝试爬取某个贴吧里的所有帖子, 并且将该这个帖子里每个楼层发布的图片下载到本地.

#coding=utf-8

import requests

from lxml import etree

import json

class Tieba:

def \_\_init\_\_(self,tieba\_name):

self.tieba\_name = tieba\_name #接收贴吧名

#设置为手机端的UA

self.headers = {"User-Agent": "Mozilla/5.0 (iPhone; CPU iPhone OS 9\_1 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13B143 Safari/601.1"}

def get\_total\_url\_list(self):

'''获取所有的urllist'''

url = "https://tieba.baidu.com/f?kw="+self.tieba\_name+"&ie=utf-8&pn={}&"

url\_list = []

for i in range(100): #通过循环拼接100个url

url\_list.append(url.format(i\*50))

return url\_list #返回100个url的urllist

def parse\_url(self,url):

'''一个发送请求, 获取响应, 同时etree处理html'''

print("parsing url:",url)

response = requests.get(url,headers=self.headers,timeout=10) #发送请求

html = response.content.decode() #获取html字符串

html = etree.HTML(html) #获取element 类型的html

return html

def get\_title\_href(self,url):

'''获取一个页面的title和href'''

html = self.parse\_url(url)

li\_temp\_list = html.xpath("//li[@class='tl\_shadow']") #分组, 按照li标签分组

total\_items = []

for i in li\_temp\_list: #遍历分组

href = "https:"+i.xpath("./a/@href")[0] if len(i.xpath("./a/@href"))>0 else None

text = i.xpath("./a/div[1]/span[1]/text()")

text = text[0] if len(text)>0 else None

item = dict( #放入字典

href = href,

text = text

)

total\_items.append(item)

return total\_items #返回一个页面所有的item

def get\_img(self,url):

'''获取一个帖子里面的所有图片'''

html = self.parse\_url(url) #返回elemet累心的html, 具有xpath方法

img\_list = html.xpath('//div[@data-class="BDE\_Image"]/@data-url')

img\_list = [i.split("src=")[-1] for i in img\_list] #提取图片的url

img\_list = [requests.utils.unquote(i) for i in img\_list]

return img\_list

def save\_item(self,item):

'''保存一个item'''

with open("teibatupian.txt","a") as f:

f.write(json.dumps(item,ensure\_ascii=False,indent=2))

f.write("\n")

def run(self):

#1、找到了url规律, url list

url\_list = self.get\_total\_url\_list()

for url in url\_list:

#2、遍历urllist 发送请求, 获得响应, etree处理html

# 3、提取title, href

total\_item = self.get\_title\_href(url)

for item in total\_item:

href = item["href"]

img\_list = self.get\_img(href) #获取到了帖子的图片列表

item["img"] = img\_list

# 4、保存到本地

print(item)

self.save\_item(item)

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

tieba = Tieba("猫")

tieba.run()

## BeautifulSoup4 解析器

### CSS 选择器: BeautifulSoup4

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](http://www.w3school.com.cn/cssref/css_selectors.asp)、Python标准库中的HTML解析器, 也支持 lxml 的 XML解析器. <http://www.w3school.com.cn/cssref/css_selectors.asp>

Beautiful Soup 3 目前已经停止开发, 推荐现在的项目使用Beautiful Soup 4. 使用 pip 安装即可: pip install beautifulsoup4

官方文档: <http://beautifulsoup.readthedocs.io/zh_CN/v4.4.0>

|  |  |  |  |
| --- | --- | --- | --- |
| 抓取工具 | 速度 | 使用难度 | 安装难度 |
| 正则 | 最快 | 困难 | 无(内置) |
| lxml | 快 | 简单 | 一般 |
| BeautifulSoup | 慢 | 最简单 | 简单 |

**示例:**

首先必须要导入 bs4 库

# beautifulsoup4\_test.py

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)

#打开本地 HTML 文件的方式来创建对象

#soup = BeautifulSoup(open('index.html'))

#格式化输出 soup 对象的内容

print soup.prettify()

运行结果:

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

<!-- Elsie -->

</a>

,

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

Lacie

</a>

and

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

Tillie

</a>

;

and they lived at the bottom of a well.

</p>

<p class="story">

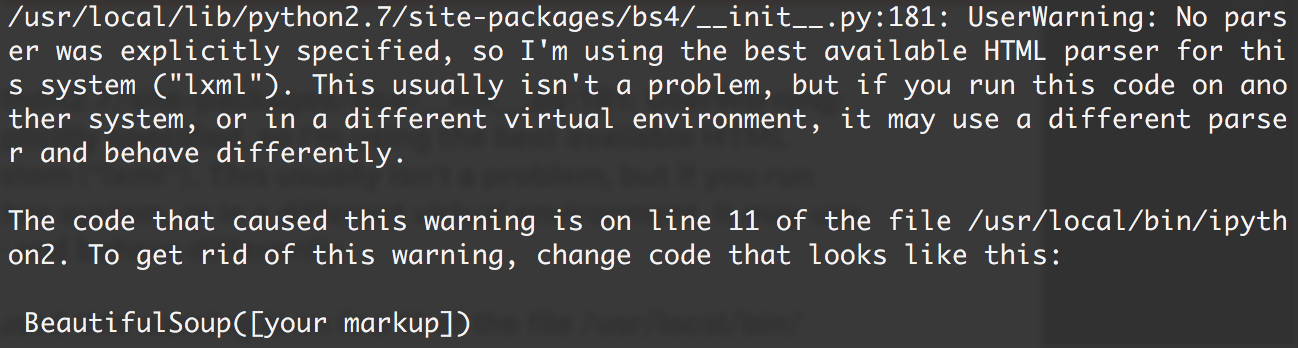
...

</p>

</body>

</html>

* 如果我们在 IPython2 下执行, 会看到这样一段警告:



* 意思是, 如果我们没有显式地指定解析器, 所以默认使用这个系统的最佳可用HTML解析器("lxml"). 如果你在另一个系统中运行这段代码, 或者在不同的虚拟环境中, 使用不同的解析器造成行为不同.
* 但是我们可以通过soup = BeautifulSoup(html,"lxml")方式指定lxml解析器.

### 四大对象种类

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:

* Tag
* NavigableString
* BeautifulSoup
* Comment

#### 1. Tag

Tag 通俗点讲就是 HTML 中的一个个标签, 例如:

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

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

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

上面的 title head a p等等 HTML 标签加上里面包括的内容就是 Tag, 那么试着使用 Beautiful Soup 来获取 Tags:

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)

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, 是 name 和 attrs**

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>

del soup.p['class'] # 还可以对这个属性进行删除

print soup.p

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

#### 2. NavigableString

既然我们已经得到了标签的内容, 那么问题来了, 我们要想获取标签内部的文字怎么办呢？很简单, 用 .string 即可, 例如

print soup.p.string

# The Dormouse's story

print type(soup.p.string)

# In [13]: <class 'bs4.element.NavigableString'>

#### 3. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的内容. 大部分时候,可以把它当作 Tag 对象, 是一个特殊的 Tag, 我们可以分别获取它的类型, 名称, 以及属性来感受一下

print type(soup.name)

# <type 'unicode'>

print soup.name

# [document]

print soup.attrs # 文档本身的属性为空

# {}

#### 4. Comment

Comment 对象是一个特殊类型的 NavigableString 对象, 其输出的内容不包括注释符号.

print soup.a

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

print soup.a.string

# Elsie

print type(soup.a.string)

# <class 'bs4.element.Comment'>

a 标签里的内容实际上是注释, 但是如果我们利用 .string 来输出它的内容时, 注释符号已经去掉了.

### 遍历文档树

#### 1. 直接子节点 : .contents .children 属性

##### .content

tag 的 .content 属性可以将tag的子节点以列表的方式输出

print soup.head.contents

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

输出方式为列表, 我们可以用列表索引来获取它的某一个元素

print soup.head.contents[0]

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

##### .children

它返回的不是一个 list, 不过我们可以通过遍历获取所有子节点.

我们打印输出 .children 看一下, 可以发现它是一个 list 生成器对象

print soup.head.children

#<listiterator object at 0x7f71457f5710>

for child in soup.body.children:

print child

结果:

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

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

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

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

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

#### 2. 所有子孙节点: .descendants 属性

.contents 和 .children 属性仅包含tag的直接子节点, .descendants 属性可以对所有tag的子孙节点进行递归循环, 和 children类似, 我们也需要遍历获取其中的内容.

for child in soup.descendants:

print child

运行结果:

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

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

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

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

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

</body></html>

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

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

The Dormouse's story

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

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

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

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

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

</body>

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

<b>The Dormouse's story</b>

The Dormouse's story

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

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

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

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

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

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

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

Elsie

,

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

Lacie

and

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

Tillie

;

and they lived at the bottom of a well.

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

...

#### 3. 节点内容: .string 属性

如果tag只有一个 NavigableString 类型子节点, 那么这个tag可以使用 .string 得到子节点. 如果一个tag仅有一个子节点,那么这个tag也可以使用 .string 方法, 输出结果与当前唯一子节点的 .string 结果相同.

通俗点说就是: 如果一个标签里面没有标签了, 那么 .string 就会返回标签里面的内容. 如果标签里面只有唯一的一个标签了, 那么 .string 也会返回最里面的内容. 例如:

print soup.head.string

#The Dormouse's story

print soup.title.string

#The Dormouse's story

### 搜索文档树

#### 1.find\_all(name, attrs, recursive, text, \*\*kwargs)

##### 1) name 参数

name 参数可以查找所有名字为 name 的tag,字符串对象会被自动忽略掉

###### A.传字符串

最简单的过滤器是字符串.在搜索方法中传入一个字符串参数,Beautiful Soup会查找与字符串完整匹配的内容,下面的例子用于查找文档中所有的<b>标签:

soup.find\_all('b')

# [<b>The Dormouse's story</b>]

print soup.find\_all('a')

#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

###### B.传正则表达式

如果传入正则表达式作为参数,Beautiful Soup会通过正则表达式的 match() 来匹配内容.下面例子中找出所有以b开头的标签,这表示<body>和<b>标签都应该被找到

import re

for tag in soup.find\_all(re.compile("^b")):

print(tag.name)

# body

# b

###### C.传列表

如果传入列表参数,Beautiful Soup会将与列表中任一元素匹配的内容返回.下面代码找到文档中所有<a>标签和<b>标签:

soup.find\_all(["a", "b"])

# [<b>The Dormouse's story</b>,

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

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

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

##### 2) keyword 参数

soup.find\_all(id='link2')

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

##### 3) text 参数

通过 text 参数可以搜搜文档中的字符串内容, 与 name 参数的可选值一样, text 参数接受 字符串 , 正则表达式 , 列表

soup.find\_all(text="Elsie")

# [u'Elsie']

soup.find\_all(text=["Tillie", "Elsie", "Lacie"])

# [u'Elsie', u'Lacie', u'Tillie']

soup.find\_all(text=re.compile("Dormouse"))

[u"The Dormouse's story", u"The Dormouse's story"]

### CSS选择器

这就是另一种与 find\_all 方法有异曲同工之妙的查找方法.

* 写 CSS 时, 标签名不加任何修饰, 类名前加., id名前加#
* 在这里我们也可以利用类似的方法来筛选元素, 用到的方法是 soup.select(), 返回类型是 list

#### (1) 通过标签名查找

print soup.select('title')

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

print soup.select('a')

#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

print soup.select('b')

#[<b>The Dormouse's story</b>]

#### (2) 通过类名查找

print soup.select('.sister')

#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

#### (3) 通过 id 名查找

print soup.select('#link1')

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

#### (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')

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

直接子标签查找, 则使用 > 分隔

print soup.select("head > title")

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

#### (5) 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.

print soup.select('a[class="sister"]')

#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]

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

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

同样, 属性仍然可以与上述查找方式组合, 不在同一节点的空格隔开, 同一节点的不加空格

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

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

#### (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()

## 案例: 使用bs4的爬虫

我们以腾讯社招页面来做演示: <http://hr.tencent.com/position.php?&start=10#a>

使用BeautifuSoup4解析器, 将招聘网页上的职位名称、职位类别、招聘人数、工作地点、发布时间, 以及每个职位详情的点击链接存储出来.

python2版

# bs4\_tencent.py

from bs4 import BeautifulSoup

import urllib2

import urllib

import json # 使用了json格式存储

def tencent():

url = 'http://hr.tencent.com/'

request = urllib2.Request(url + 'position.php?&start=10#a')

response =urllib2.urlopen(request)

resHtml = response.read()

output =open('tencent.json','w')

html = BeautifulSoup(resHtml,'lxml')

# 创建CSS选择器

result = html.select('tr[class="even"]')

result2 = html.select('tr[class="odd"]')

result += result2

items = []

for site in result:

item = {}

name = site.select('td a')[0].get\_text()

detailLink = site.select('td a')[0].attrs['href']

catalog = site.select('td')[1].get\_text()

recruitNumber = site.select('td')[2].get\_text()

workLocation = site.select('td')[3].get\_text()

publishTime = site.select('td')[4].get\_text()

item['name'] = name

item['detailLink'] = url + detailLink

item['catalog'] = catalog

item['recruitNumber'] = recruitNumber

item['publishTime'] = publishTime

items.append(item)

# 禁用ascii编码, 按utf-8编码

line = json.dumps(items,ensure\_ascii=False)

output.write(line.encode('utf-8'))

output.close()

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

tencent()

python版

# bs4\_tencent.py

**from** bs4 **import** BeautifulSoup

**import** urllib.request, urllib.parse, json *# 使用了json格式存储*

**def** tencent():

url = **'http://hr.tencent.com/'**

request = urllib.request.Request(url + **'position.php?&start=10#a'**)

response =urllib.request.urlopen(request)

resHtml = response.read()

html = BeautifulSoup(resHtml,**'lxml'**)

*# 创建CSS选择器*

result = html.select(**'tr[class="even"]'**)

result2 = html.select(**'tr[class="odd"]'**)

result += result2

items = []

**for** site **in** result:

item = {}

name = site.select(**'td a'**)[0].get\_text()

detailLink = site.select(**'td a'**)[0].attrs[**'href'**]

catalog = site.select(**'td'**)[1].get\_text()

recruitNumber = site.select(**'td'**)[2].get\_text()

workLocation = site.select(**'td'**)[3].get\_text()

publishTime = site.select(**'td'**)[4].get\_text()

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

item[**'detailLink'**] = url + detailLink

item[**'catalog'**] = catalog

item[**'recruitNumber'**] = recruitNumber

item[**'publishTime'**] = publishTime

item[**'workLocation'**] = workLocation

items.append(item)

*# 禁用ascii编码, 按utf-8编码*

line = json.dumps(items,ensure\_ascii=**False**)

*# output =open('tencent.json','wb')*

*# output.write(line.encode('utf-8'))*

*# output.close()*

**with** open(**"tencent.json"**, **"wb"**) **as** f:

f.write(line.encode(**'utf-8'**))

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

tencent()

新版代码

# bs4\_tencent.py

from bs4 import BeautifulSoup

import urllib

import json # 使用了json格式存储

def tencent():

url = 'http://hr.tencent.com/'

request = urllib.request.Request(url + 'position.php?&start=10#a')

response =urllib.request.urlopen(request)

resHtml = response.read()

output =open('tencent.json','w')

html = BeautifulSoup(resHtml,'lxml')

# 创建CSS选择器

result = html.select('tr[class="even"]')

result2 = html.select('tr[class="odd"]')

result += result2

items = []

for site in result:

item = {}

name = site.select('td a')[0].get\_text()

detailLink = site.select('td a')[0].attrs['href']

catalog = site.select('td')[1].get\_text()

recruitNumber = site.select('td')[2].get\_text()

workLocation = site.select('td')[3].get\_text()

publishTime = site.select('td')[4].get\_text()

item['name'] = name

item['detailLink'] = url + detailLink

item['catalog'] = catalog

item['recruitNumber'] = recruitNumber

item['publishTime'] = publishTime

items.append(item)

# 禁用ascii编码, 按utf-8编码

line = json.dumps(items,ensure\_ascii=False)

output.write(line.encode('utf-8'))

output.close()

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

tencent()

**模拟登陆知乎**

模拟登陆网站之前首先要抓包, 看在登陆网站时向网站发送了什么内容. 先输入错误的内容, 如果输入正确的内容, 可能页面刷新, 就找不到想要的数据了.

在Raw和webForm中就能看到向服务器发送的内容.

**from** bs4 **import** BeautifulSoup

**import** requests, time

**def** captcha(captcah\_data):

**with** open(**"captcha.jpg"**, **"wb"**) **as** f:

f.write(captcah\_data)

time.sleep(2)

captcah\_text = input(**"请输入验证码："**)

**return** captcah\_text

**def** zhihuLogin():

*#构建一个Session对象, 可以保存Cookie*

sess = requests.Session()

url = **"https://www.zhihu.com/#signin"**

user\_agent = **"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36"**

headers = {**"User-Agent"**: user\_agent}

*#首先获取登陆页面, 找到需要POST的数据(\_xsrf), 同时会记录当前网页的Cookie值.*

*#也可以直接使用request.get来直接发送请求, 但那样就无法保存Cookie值了.*

html = sess.get(url, headers = headers).text

*#使用BeautifulSoup调用lxml解析库*

soup = BeautifulSoup(html, **"lxml"**)

*#获取之前get的页面里的\_xsrf值*

*#\_xsrf作用是防止CSRF攻击, 跨站请求攻击, 是一种利用网站对用户的一种信任机制来进行攻击. 通常通过伪装成网站信任的用户的请求.*

xsrf = soup.find(**"input"**, attrs={**"name"**: **"\_xsrf"**}).get(**"value"**)

print(xsrf)

*#通过Unix时间戳获取知乎验证码的url*

captcha\_url = **"https://www.zhihu.com/captcha.gif?r=%d&type=login"**%(time.time()\*1000)

print(captcha\_url)

*#发送图片的请求, 获取图片二进制数据流*

captcha\_data = sess.get(captcha\_url, headers=headers).content

*#获取验证码里的文字, 需要手动输入*

captcha\_text = captcha(captcha\_data)

data = {

**"\_xsrf"**: xsrf,

**"phone\_num"**: **"15201933129"**,

**"password"**: **"s6u9p5e3r8"**,

**"captcha"**: captcha\_text,

}

response = sess.post(**"https://www.zhihu.com/login/phone\_num"**, data=data, headers = headers)

*# print(response.text)*

response = sess.get(**"https://www.zhihu.com/people/dreamingtech/activities"**, headers = headers)

**with** open(**"my\_zhihu.html"**, **"wb"**) **as** f:

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

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

zhihuLogin()

*#个人主页, 必须登陆后才能看.*

*# https://www.zhihu.com/people/dreamingtech/activities*

**'''**

**请输入验证码：VHF9**

**{"r":0,**

**"msg": "\u767b\u5f55\u6210\u529f"**

**}**

**print("\u767b\u5f55\u6210\u529f")**

**登录成功**

**'''**

## JSON模块与JsonPath

### 数据提取之JSON与JsonPATH

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式, 它使得人们很容易的进行阅读和编写. 同时也方便了机器进行解析和生成. 适用于进行数据交互的场景, 比如网站前台与后台之间的数据交互.

JSON和XML的比较可谓不相上下.

Python 2.7中自带了JSON模块, 直接import json就可以使用了.

官方文档: <http://docs.python.org/library/json.html>

Json在线解析网站: [http://www.json.cn/#](http://www.json.cn/)

### JSON

json简单说就是javascript中的对象和数组, 所以这两种结构就是对象和数组两种结构, 通过这两种结构可以表示各种复杂的结构

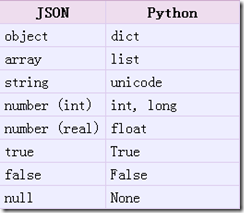
1. 对象: 对象在js中表示为{ }括起来的内容, 数据结构为 { key: value, key: value, ... }的键值对的结构, 在面向对象的语言中, key为对象的属性, value为对应的属性值, 所以很容易理解, 取值方法为 对象.key 获取属性值, 这个属性值的类型可以是数字、字符串、数组、对象这几种.
2. 数组: 数组在js中是中括号[ ]括起来的内容, 数据结构为 ["Python", "javascript", "C++", ...], 取值方式和所有语言中一样, 使用索引获取, 字段值的类型可以是 数字、字符串、数组、对象几种.

### import json

json模块提供了四个功能: dumps、dump、loads、load, 用于字符串 和 python数据类型间进行转换.

#### 1. json.loads()

把Json格式字符串解码转换成Python对象 从json到python的类型转化对照如下:



# json\_loads.py

import json

strList = '[1, 2, 3, 4]'

strDict = '{"city": "北京", "name": "大猫"}'

json.loads(strList)

# [1, 2, 3, 4]

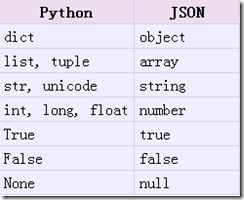
json.loads(strDict) # json数据自动按Unicode存储

# {u'city': u'\u5317\u4eac', u'name': u'\u5927\u732b'}

#### 2. json.dumps()

实现python类型转化为json字符串, 返回一个str对象 把一个Python对象编码转换成Json字符串

从python原始类型向json类型的转化对照如下:



# json\_dumps.py

import json

import chardet

listStr = [1, 2, 3, 4]

tupleStr = (1, 2, 3, 4)

dictStr = {"city": "北京", "name": "大猫"}

json.dumps(listStr)

# '[1, 2, 3, 4]'

json.dumps(tupleStr)

# '[1, 2, 3, 4]'

# 注意: json.dumps() 序列化时默认使用的ascii编码

# 添加参数 ensure\_ascii=False 禁用ascii编码, 按utf-8编码

# chardet.detect()返回字典, 其中confidence是检测精确度

json.dumps(dictStr)

# '{"city": "\\u5317\\u4eac", "name": "\\u5927\\u5218"}'

chardet.detect(json.dumps(dictStr))

# {'confidence': 1.0, 'encoding': 'ascii'}

print json.dumps(dictStr, ensure\_ascii=False)

# {"city": "北京", "name": "大刘"}

chardet.detect(json.dumps(dictStr, ensure\_ascii=False))

# {'confidence': 0.99, 'encoding': 'utf-8'}

**chardet是一个非常优秀的编码识别模块, 可通过pip安装**

#### 3. json.dump()

将Python内置类型序列化为json对象后写入文件

# json\_dump.py

import json

listStr = [{"city": "北京"}, {"name": "大刘"}]

json.dump(listStr, open("listStr.json","w"), ensure\_ascii=False)

dictStr = {"city": "北京", "name": "大刘"}

json.dump(dictStr, open("dictStr.json","w"), ensure\_ascii=False)

#### 4. json.load()

读取文件中json形式的字符串元素 转化成python类型

# json\_load.py

import json

strList = json.load(open("listStr.json"))

print strList

# [{u'city': u'\u5317\u4eac'}, {u'name': u'\u5927\u5218'}]

strDict = json.load(open("dictStr.json"))

print strDict

# {u'city': u'\u5317\u4eac', u'name': u'\u5927\u5218'}

### JsonPath

JsonPath 是一种信息抽取类库, 是从JSON文档中抽取指定信息的工具, 提供多种语言实现版本, 包括: Javascript, Python, PHP 和 Java.

JsonPath 对于 JSON 来说, 相当于 XPATH 对于 XML.

下载地址: [https://pypi.python.org/pypi/jsonpath](https://pypi.python.org/pypi/jsonpath/)

安装方法: 点击Download URL链接下载jsonpath, 解压之后执行python setup.py install

官方文档: [http://goessner.net/articles/JsonPath](http://goessner.net/articles/JsonPath/)

#### JsonPath与XPath语法对比:

Json结构清晰, 可读性高, 复杂度低, 非常容易匹配, 下表中对应了XPath的用法.

|  |  |  |
| --- | --- | --- |
| XPath | JSONPath | 描述 |
| / | $ | 根节点 |
| . | @ | 现行节点 |
| / | .or[] | 取子节点 |
| .. | n/a | 取父节点, Jsonpath未支持 |
| // | .. | 就是不管位置, 选择所有符合条件的条件 |
| \* | \* | 匹配所有元素节点 |
| @ | n/a | 根据属性访问, Json不支持, 因为Json是个Key-value递归结构, 不需要. |
| [] | [] | 迭代器标示（可以在里边做简单的迭代操作, 如数组下标, 根据内容选值等） |
| | | [,] | 支持迭代器中做多选. |
| [] | ?() | 支持过滤操作. |
| n/a | () | 支持表达式计算 |
| () | n/a | 分组, JsonPath不支持 |

#### 示例:

我们以拉勾网城市JSON文件 <http://www.lagou.com/lbs/getAllCitySearchLabels.json> 为例, 获取所有城市.

# jsonpath\_lagou.py

import urllib2

import jsonpath

import json

import chardet

url = 'http://www.lagou.com/lbs/getAllCitySearchLabels.json'

request =urllib2.Request(url)

response = urllib2.urlopen(request)

html = response.read()

# 把json格式字符串转换成python对象

jsonobj = json.loads(html)

# 从根节点开始, 匹配name节点

citylist = jsonpath.jsonpath(jsonobj,'$..name')

print citylist

print type(citylist)

fp = open('city.json','w')

content = json.dumps(citylist, ensure\_ascii=False)

print content

fp.write(content.encode('utf-8'))

fp.close()

python3

# jsonpath\_lagou.py

**import** urllib.request, json, jsonpath, chardet

*#json：json解析库, 对应于lxml, jsonpath：json的解析语法, 对应于xpath*

url = **'http://www.lagou.com/lbs/getAllCitySearchLabels.json'**

user\_agent = **"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36"**

headers = {**"User-Agent"**: user\_agent}

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

response = urllib.request.urlopen(request)

*#取出json文件里的内容, 返回的格式是字符串*

html = response.read().decode(**"utf-8"**)

*# print(html)*

*# print((html))*

*#把json形式的字条串转换成python形式的Unicode字符串*

unicode\_str = json.loads(html)

*#python形式的列表*

city\_list = jsonpath.jsonpath(unicode\_str, **"$..name"**)

print(type(city\_list))

*#返回ASCII格式, dumps()默认转换中文为ascII编码格式, ensure\_ascii默认为True*

*#以后格式尽量使用unicode格式, 各个平台都通用.*

array = json.dumps(city\_list)

*#返回Unicode格式*

array = json.dumps(city\_list, ensure\_ascii=**False**)

**with** open(**"lagou\_city.json"**, **"wb"**) **as** f:

*#再转换为unicode格式的*

f.write(array.encode(**"utf-8"**))

新版requests版

# jsonpath\_lagou.py

import requests

import jsonpath

import json

import chardet

url = 'http://www.lagou.com/lbs/getAllCitySearchLabels.json'

response = equests.get(url)

html = response.text

# 把json格式字符串转换成python对象

jsonobj = json.loads(html)

# 从根节点开始, 匹配name节点

citylist = jsonpath.jsonpath(jsonobj,'$..name')

print citylist

print type(citylist)

fp = open('city.json','w')

content = json.dumps(citylist, ensure\_ascii=False)

print content

fp.write(content.encode('utf-8'))

fp.close()

#### 注意事项:

json.loads() 是把 Json格式字符串解码转换成Python对象, 如果在json.loads的时候出错, 要注意被解码的Json字符的编码.

如果传入的字符串的编码不是UTF-8的话, 需要指定字符编码的参数 encoding

dataDict = json.loads(jsonStrGBK);

* dataJsonStr是JSON字符串, 假设其编码本身是非UTF-8的话而是GBK 的, 那么上述代码会导致出错, 改为对应的:
* dataDict = json.loads(jsonStrGBK, encoding="GBK");
* 如果 dataJsonStr通过encoding指定了合适的编码, 但是其中又包含了其他编码的字符, 则需要先去将dataJsonStr转换为Unicode, 然后再指定编码格式调用json.loads()

``` python

dataJsonStrUni = dataJsonStr.decode("GB2312"); dataDict = json.loads(dataJsonStrUni, encoding="GB2312");

##字符串编码转换

这是中国程序员最苦逼的地方, 什么乱码之类的几乎都是由汉字引起的.

其实编码问题很好搞定, 只要记住一点:

####任何平台的任何编码 都能和 Unicode 互相转换

UTF-8 与 GBK 互相转换, 那就先把UTF-8转换成Unicode, 再从Unicode转换成GBK, 反之同理.

``` python

# 这是一个 UTF-8 编码的字符串

utf8Str = "你好地球"

# 1. 将 UTF-8 编码的字符串 转换成 Unicode 编码

unicodeStr = utf8Str.decode("UTF-8")

# 2. 再将 Unicode 编码格式字符串 转换成 GBK 编码

gbkData = unicodeStr.encode("GBK")

# 1. 再将 GBK 编码格式字符串 转化成 Unicode

unicodeStr = gbkData.decode("gbk")

# 2. 再将 Unicode 编码格式字符串转换成 UTF-8

utf8Str = unicodeStr.encode("UTF-8")

decode的作用是将其他编码的字符串转换成 Unicode 编码

encode的作用是将 Unicode 编码转换成其他编码的字符串

一句话: UTF-8是对Unicode字符集进行编码的一种编码方式

## 糗事百科案例

爬取糗事百科段子, 假设页面的URL是 <http://www.qiushibaike.com/8hr/page/1>

**要求:**

1. 使用requests获取页面信息, 用XPath / re 做数据提取
2. 获取每个帖子里的用户头像链接、用户姓名、段子内容、点赞次数和评论次数
3. 保存到 json 文件内

参考代码

python2 requests版

#qiushibaike.py

#import urllib

#import re

#import chardet

import requests

from lxml import etree

page = 1

url = 'http://www.qiushibaike.com/8hr/page/' + str(page)

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36',

'Accept-Language': 'zh-CN,zh;q=0.8'}

try:

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

resHtml = response.text

html = etree.HTML(resHtml)

result = html.xpath('//div[contains(@id,"qiushi\_tag")]')

for site in result:

item = {}

imgUrl = site.xpath('./div/a/img/@src')[0].encode('utf-8')

username = site.xpath('./div/a/@title')[0].encode('utf-8')

#username = site.xpath('.//h2')[0].text

content = site.xpath('.//div[@class="content"]/span')[0].text.strip().encode('utf-8')

# 投票次数

vote = site.xpath('.//i')[0].text

#print site.xpath('.//\*[@class="number"]')[0].text

# 评论信息

comments = site.xpath('.//i')[1].text

print imgUrl, username, content, vote, comments

except Exception, e:

print e

python3版本

qiushibaike.py

**import** urllib.request, json

**from** lxml **import** etree

url=**"http://www.qiushibaike.com/8hr/page/1/"**

user\_agent = **"Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko "**

headers = {**"User-Agent"**: user\_agent}

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

html = urllib.request.urlopen(request).read()

*#响应返回的是字符串, 解析为HTML DOM模式*

text = etree.HTML(html)

*#返回所有段子的结点位置, contains()模糊查询方法, 每一个参数是要匹配的标签, 第二个参数是标签名部分内容*

node\_list = text.xpath(**'//div[contains(@id, "qiushi\_tag")]'**)

item = {}

**for** node **in** node\_list:

*#用户名, xpath返回列表, 这个列表中只有一个元素, 用索引方式取出来*

username = node.xpath(**'./div/a/@title'**)[0]

*#图片链接*

image = node.xpath(**'.//div[@class="thumb"]//@src'**)*#[0]*

*#text方法取出标签下的内容*

content = node.xpath(**'.//div[@class="content"]/span'**)[0].text

*#点赞数量*

zan = node.xpath(**".//i"**)[0].text

*#评论*

comments = node.xpath(**'.//i'**)[1].text

items = {

**"username"**: username,

**"image"**: image,

**"content"**: content,

**"zan"**: zan,

**"comments"**: comments,

}

*# print(items)*

wen = (json.dumps(items, ensure\_ascii=**False**)+**"\n"**).encode(**"utf-8"**)

*# print(chardet.detect(wen))*

**with** open(**'qiushi.json'**, **'ab'**) **as** f:

f.write(wen)

Version2

**import** requests, json

**from** lxml **import** etree

page = 1

url = **'http://www.qiushibaike.com/8hr/page/'** + str(page)

headers = {

**'User-Agent'**: **'Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko'**,

**'Accept-Language'**: **'zh-CN,zh;q=0.8'**}

**try**:

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

resHtml = response.text

html = etree.HTML(resHtml)

result = html.xpath(**'//div[contains(@id,"qiushi\_tag")]'**)

item = {}

**for** site **in** result:

*#用户头像*

*# imgUrl = site.xpath('./div/a/img/@src')[0]*

*#对于匿名的用户, 头像不在a/img标签下, 而在span/img标签下, 这样使用上面的方法就取不到.*

imgUrl = site.xpath(**'./div//img/@src'**)[0]

*# print(imgUrl)*

username = site.xpath(**'./div/a/@title'**)[0]

*#username = site.xpath('.//h2')[0].text*

content = site.xpath(**'.//div[@class="content"]/span'**)[0].text.strip()

*# 投票次数*

vote = site.xpath(**'.//i'**)[0].text

*#print site.xpath('.//\*[@class="number"]')[0].text*

*# 评论信息*

comments = site.xpath(**'.//i'**)[1].text

items = {

**"username"**: username,

**"image"**: **"http:"** + imgUrl,

**"content"**: content,

**"vote"**: vote,

**"comments"**: comments,

}

wen = (json.dumps(items, ensure\_ascii=**False**) + **"\n"**).encode(**"utf-8"**)

**with** open(**'qiushi.json'**, **'ab'**) **as** f:

f.write(wen)

print(**"done"**)

**except** Exception **as** e:

print(e)

新版

#coding=utf-8

import requests

from retrying import retry

from lxml import etree

class Qiubai\_spider():

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

self.url = "http://www.qiushibaike.com/8hr/page/{}/"

self.headers = {

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

}

@retry(stop\_max\_attempt\_number=5) #调用retry, 当assert出错时候, 重复请求5次

def parse\_url(self,url):

response = requests.get(url,timeout=10,headers=self.headers) #请求url

assert response.status\_code==200 #当响应码不是200时候, 做断言报错处理

print(url)

return etree.HTML(response.text) #返回etree之后的html

def parse\_content(self,html):

item\_temp = html.xpath("//div[@class='article block untagged mb15']")

print(len(item\_temp))

for item in item\_temp:

#获取用户头像地址

avatar = item.xpath("./div[1]/a[1]/img/@src")[0] if len(item.xpath("./div[1]/a[1]/img/@src"))>0 else None

#为头像地址添加前缀

if avatar is not None and not avatar.startswith("http:"):

avatar = "http:"+avatar

print(avatar)

name = item.xpath("./div[1]/a[2]/h2/text()")[0] #获取用户名

print(name)

content = item.xpath("./a[@class='contentHerf']/div/span/text()")[0] #获取内容

print(content)

star\_number = item.xpath("./div[@class='stats']/span[1]/i/text()")[0] #获取点赞数

print(star\_number)

comment\_number = item.xpath("./div[@class='stats']/span[2]/a/i/text()")[0] #获取评论数

print(comment\_number)

print("\*"\*100)

def run(self):

'''函数的主要逻辑实现

'''

url = self.url.format(1) #获取到url

html = self.parse\_url(url) #请求url

self.parse\_content(html) #解析页面内容并把内容存入内容队列

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

qiubai = Qiubai\_spider()

qiubai.run()

## 多线程爬虫案例

### Queue（队列对象）

**Python3.5 queue模块详解**

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

#### queue介绍

* queue是python中的标准库, 俗称队列, 可以直接import 引用, 在python2.x中, 模块名为Queue
* 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

#### 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)

#### 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, timeout等待时间
* queue.get\_nowait() 相当queue.get(False)
* queue.put(item) 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

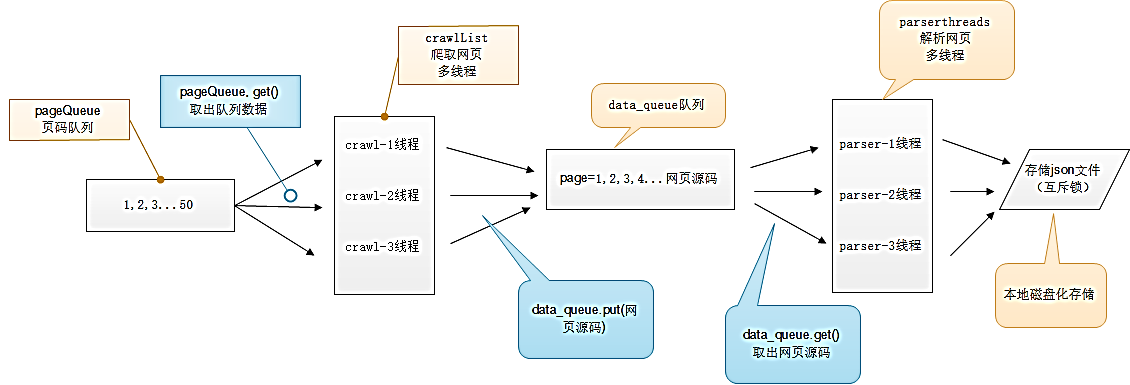
Queue是python中的标准库, 可以直接import Queue引用; 队列是线程间最常用的交换数据的形式

python下多线程的思考

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

1. 初始化: class Queue.Queue(maxsize) FIFO 先进先出
2. 包中的常用方法:
   * 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, timeout等待时间
3. 创建一个"队列"对象
   * import Queue
   * myqueue = Queue.Queue(maxsize = 10)
4. 将一个值放入队列中
   * myqueue.put(10)
5. 将一个值从队列中取出
   * myqueue.get()

### 多线程示意图



### 多线程糗事百科案例

案例要求参考上一个糗事百科单进程案例

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

import requests

from lxml import etree

from Queue import Queue

import threading

import time

import json

class thread\_crawl(threading.Thread):

'''

抓取线程类

'''

def \_\_init\_\_(self, threadID, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.q = q

def run(self):

print "Starting " + self.threadID

self.qiushi\_spider()

print "Exiting ", self.threadID

def qiushi\_spider(self):

# page = 1

while True:

if self.q.empty():

break

else:

page = self.q.get()

print 'qiushi\_spider=', self.threadID, ',page=', str(page)

url = 'http://www.qiushibaike.com/8hr/page/' + str(page) + '/'

headers = {

'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36',

'Accept-Language': 'zh-CN,zh;q=0.8'}

# 多次尝试失败结束、防止死循环

timeout = 4

while timeout > 0:

timeout -= 1

try:

content = requests.get(url, headers=headers)

data\_queue.put(content.text)

break

except Exception, e:

print 'qiushi\_spider', e

if timeout < 0:

print 'timeout', url

class Thread\_Parser(threading.Thread):

'''

页面解析类；

'''

def \_\_init\_\_(self, threadID, queue, lock, f):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.queue = queue

self.lock = lock

self.f = f

def run(self):

print 'starting ', self.threadID

global total, exitFlag\_Parser

while not exitFlag\_Parser:

try:

'''

调用队列对象的get()方法从队头删除并返回一个项目. 可选参数为block, 默认为True.

如果队列为空且block为True, get()就使调用线程暂停, 直至有项目可用.

如果队列为空且block为False, 队列将引发Empty异常.

'''

item = self.queue.get(False)

if not item:

pass

self.parse\_data(item)

self.queue.task\_done()

print 'Thread\_Parser=', self.threadID, ',total=', total

except:

pass

print 'Exiting ', self.threadID

def parse\_data(self, item):

'''

解析网页函数

:param item: 网页内容

:return:

'''

global total

try:

html = etree.HTML(item)

result = html.xpath('//div[contains(@id,"qiushi\_tag")]')

for site in result:

try:

imgUrl = site.xpath('.//img/@src')[0]

title = site.xpath('.//h2')[0].text

content = site.xpath('.//div[@class="content"]/span')[0].text.strip()

vote = None

comments = None

try:

vote = site.xpath('.//i')[0].text

comments = site.xpath('.//i')[1].text

except:

pass

result = {

'imgUrl': imgUrl,

'title': title,

'content': content,

'vote': vote,

'comments': comments,

}

with self.lock:

# print 'write %s' % json.dumps(result)

self.f.write(json.dumps(result, ensure\_ascii=False).encode('utf-8') + "\n")

except Exception, e:

print 'site in result', e

except Exception, e:

print 'parse\_data', e

with self.lock:

total += 1

data\_queue = Queue()

exitFlag\_Parser = False

lock = threading.Lock()

total = 0

def main():

output = open('qiushibaike.json', 'a')

#初始化网页页码page从1-10个页面

pageQueue = Queue(50)

for page in range(1, 11):

pageQueue.put(page)

#初始化采集线程

crawlthreads = []

crawlList = ["crawl-1", "crawl-2", "crawl-3"]

for threadID in crawlList:

thread = thread\_crawl(threadID, pageQueue)

thread.start()

crawlthreads.append(thread)

#初始化解析线程parserList

parserthreads = []

parserList = ["parser-1", "parser-2", "parser-3"]

#分别启动parserList

for threadID in parserList:

thread = Thread\_Parser(threadID, data\_queue, lock, output)

thread.start()

parserthreads.append(thread)

# 等待队列清空

while not pageQueue.empty():

pass

# 等待所有线程完成

for t in crawlthreads:

t.join()

while not data\_queue.empty():

pass

# 通知线程是时候退出

global exitFlag\_Parser

exitFlag\_Parser = True

for t in parserthreads:

t.join()

print "Exiting Main Thread"

with lock:

output.close()

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

main()

新版代码

# coding=utf-8

import requests

from lxml import etree

import json

from queue import Queue

import threading

class Qiubai:

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

self.headers = {

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_12\_5) AppleWeb\

Kit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36"}

self.url\_queue = Queue() #实例化三个队列, 用来存放内容

self.html\_queue =Queue()

self.content\_queue = Queue()

def get\_total\_url(self):

'''

获取了所有的页面url, 并且返回urllist

return ：list

'''

url\_temp = 'https://www.qiushibaike.com/8hr/page/{}/'

url\_list = []

for i in range(1,36):

# url\_list.append(url\_temp.format(i))

self.url\_queue.put(url\_temp.format(i))

def parse\_url(self):

'''

一个发送请求, 获取响应, 同时etree处理html

'''

while self.url\_queue.not\_empty:

url = self.url\_queue.get()

print("parsing url:",url)

response = requests.get(url,headers=self.headers,timeout=10) #发送请求

html = response.content.decode() #获取html字符串

html = etree.HTML(html) #获取element 类型的html

self.html\_queue.put(html)

self.url\_queue.task\_done()

def get\_content(self):

'''

:param url:

:return: 一个list, 包含一个url对应页面的所有段子的所有内容的列表

'''

while self.html\_queue.not\_empty:

html = self.html\_queue.get()

total\_div = html.xpath('//div[@class="article block untagged mb15"]') #返回divelememtn的一个列表

items = []

for i in total\_div: #遍历div标枪, 获取糗事百科每条的内容的全部信息

author\_img = i.xpath('./div[@class="author clearfix"]/a[1]/img/@src')

author\_img = "https:" + author\_img[0] if len(author\_img) > 0 else None

author\_name = i.xpath('./div[@class="author clearfix"]/a[2]/h2/text()')

author\_name = author\_name[0] if len(author\_name) > 0 else None

author\_href = i.xpath('./div[@class="author clearfix"]/a[1]/@href')

author\_href = "https://www.qiushibaike.com" + author\_href[0] if len(author\_href) > 0 else None

author\_gender = i.xpath('./div[@class="author clearfix"]//div/@class')

author\_gender = author\_gender[0].split(" ")[-1].replace("Icon", "") if len(author\_gender) > 0 else None

author\_age = i.xpath('./div[@class="author clearfix"]//div/text()')

author\_age = author\_age[0] if len(author\_age) > 0 else None

content = i.xpath('./a[@class="contentHerf"]/div/span/text()')

content\_vote = i.xpath('./div[@class="stats"]/span[1]/i/text()')

content\_vote = content\_vote[0] if len(content\_vote) > 0 else None

content\_comment\_numbers = i.xpath('./div[@class="stats"]/span[2]/a/i/text()')

content\_comment\_numbers = content\_comment\_numbers[0] if len(content\_comment\_numbers) > 0 else None

hot\_comment\_author = i.xpath('./a[@class="indexGodCmt"]/div/span[last()]/text()')

hot\_comment\_author = hot\_comment\_author[0] if len(hot\_comment\_author) > 0 else None

hot\_comment = i.xpath('./a[@class="indexGodCmt"]/div/div/text()')

hot\_comment = hot\_comment[0].replace("\n：", "").replace("\n", "") if len(hot\_comment) > 0 else None

hot\_comment\_like\_num = i.xpath('./a[@class="indexGodCmt"]/div/div/div/text()')

hot\_comment\_like\_num = hot\_comment\_like\_num[-1].replace("\n", "") if len(hot\_comment\_like\_num) > 0 else None

item = dict(

author\_name=author\_name,

author\_img=author\_img,

author\_href=author\_href,

author\_gender=author\_gender,

author\_age=author\_age,

content=content,

content\_vote=content\_vote,

content\_comment\_numbers=content\_comment\_numbers,

hot\_comment=hot\_comment,

hot\_comment\_author=hot\_comment\_author,

hot\_comment\_like\_num=hot\_comment\_like\_num

)

items.append(item)

self.content\_queue.put(items)

self.html\_queue.task\_done() #task\_done的时候, 队列计数减一

def save\_items(self):

'''

保存items

:param items:列表

'''

while self.content\_queue.not\_empty:

items = self.content\_queue.get()

f = open("qiubai.txt","a")

for i in items:

json.dump(i,f,ensure\_ascii=False,indent=2)

# f.write(json.dumps(i))

f.close()

self.content\_queue.task\_done()

def run(self):

# 1.获取url list

# url\_list = self.get\_total\_url()

thread\_list = []

thread\_url = threading.Thread(target=self.get\_total\_url)

thread\_list.append(thread\_url)

#发送网络请求

for i in range(10):

thread\_parse = threading.Thread(target=self.parse\_url)

thread\_list.append(thread\_parse)

#提取数据

thread\_get\_content = threading.Thread(target=self.get\_content)

thread\_list.append(thread\_get\_content)

#保存

thread\_save = threading.Thread(target=self.save\_items)

thread\_list.append(thread\_save)

for t in thread\_list:

t.setDaemon(True) #为每个进程设置为后台进程, 效果是主进程退出子进程也会退出

t.start() #为了解决程序结束无法退出的问题

#

# for t in thread\_list:

# t.join()

self.url\_queue.join() #让主线程等待, 所有的队列为空的时候才能退出

self.html\_queue.join()

self.content\_queue.join()

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

qiubai = Qiubai()

qiubai.run()

# 3. 动态HTML处理和机器图像识别

## 动态HTML介绍

### JavaScript

JavaScript 是网络上最常用也是支持者最多的客户端脚本语言. 它可以收集 用户的跟踪数据,不需要重载页面直接提交表单, 在页面嵌入多媒体文件, 甚至运行网页游戏.

我们可以在网页源代码的<script>标签里看到, 比如:

<script type="text/javascript" src="https://statics.huxiu.com/w/mini/static\_2015/js/sea.js?v=201601150944"></script>

### jQuery

jQuery 是一个十分常见的库,70% 最流行的网站(约 200 万)和约 30% 的其他网站(约 2 亿)都在使用. 一个网站使用 jQuery 的特征,就是源代码里包含了 jQuery 入口,比如:

<script type="text/javascript" src="https://statics.huxiu.com/w/mini/static\_2015/js/jquery-1.11.1.min.js?v=201512181512"></script>

如果你在一个网站上看到了 jQuery, 那么采集这个网站数据的时候要格外小心. jQuery 可以动态地创建 HTML 内容,只有在 JavaScript 代码执行之后才会显示. 如果你用传统的方法采集页面内容,就只能获得JavaScript代码执行之前页面上的内容.

### Ajax

我们与网站服务器通信的唯一方式, 就是发出 HTTP 请求获取新页面. 如果提交表单之后, 或从服务器获取信息之后, 网站的页面不需要重新刷新, 那么你访问的网站就在用Ajax 技术.

Ajax 其实并不是一门语言,而是用来完成网络任务(可以认为 它与网络数据采集差不多)的一系列技术. Ajax 全称是 Asynchronous JavaScript and XML(异步 JavaScript 和 XML), 网站不需要使用单独的页面请求就可以和网络服务器进行交互 (收发信息).

### DHTML

Ajax 一样, 动态 HTML(Dynamic HTML, DHTML)也是一系列用于解决网络问题的 技术集合. DHTML 是用客户端语言改变页面的 HTML 元素(HTML、CSS, 或者二者皆 被改变). 比如页面上的按钮只有当用户移动鼠标之后才出现,背景色可能每次点击都会改变, 或者用一个 Ajax 请求触发页面加载一段新内容, 网页是否属于DHTML, 关键要看有没有用 JavaScript 控制 HTML 和 CSS 元素.

### 那么, 如何搞定？

那些使用了 Ajax 或 DHTML 技术改变 / 加载内容的页面, 可能有一些采集手段. 但是用 Python 解决这个问题只有两种途径:

1. 直接从 JavaScript 代码里采集内容（费时费力）
2. 用 Python 的 第三方库运行 JavaScript, 直接采集你在浏览器里看到的页面（这个可以有）.

## Selenium与PhantomJS

## Selenium

[Selenium](http://www.seleniumhq.org/)是一个Web的自动化测试工具, 最初是为网站自动化测试而开发的, 类型像我们玩游戏用的按键精灵, 可以按指定的命令自动操作, 不同是Selenium 可以直接运行在浏览器上, 它支持所有主流的浏览器（包括PhantomJS这些无界面的浏览器）.

Selenium 可以根据我们的指令, 让浏览器自动加载页面, 获取需要的数据, 甚至页面截屏, 或者判断网站上某些动作是否发生.

Selenium 自己不带浏览器, 不支持浏览器的功能, 它需要与第三方浏览器结合在一起才能使用. 但是我们有时候需要让它内嵌在代码中运行, 所以我们可以用一个叫 PhantomJS 的工具代替真实的浏览器.

可以从 PyPI 网站下载 Selenium库[https://pypi.python.org/simple/selenium](https://pypi.python.org/simple/selenium/) , 也可以用 第三方管理器 pip用命令安装: pip install selenium

Selenium 官方参考文档: <http://selenium-python.readthedocs.io/index.html>

## PhantomJS

[PhantomJS](http://phantomjs.org" \t "_blank) 是一个基于Webkit的"无界面"(headless)浏览器, 它会把网站加载到内存并执行页面上的 JavaScript, 因为不会展示图形界面, 所以运行起来比完整的浏览器要高效.

如果我们把 Selenium 和 PhantomJS 结合在一起, 就可以运行一个非常强大的网络爬虫了, 这个爬虫可以处理 JavaScrip、Cookie、headers, 以及任何我们真实用户需要做的事情.

注意: PhantomJS 只能从它的官方网站[http://phantomjs.org/download.html)](http://phantomjs.org/download.html) 下载.

因为 PhantomJS 是一个功能完善(虽然无界面)的浏览器而非一个 Python 库, 所以它不需要像 Python 的其他库一样安装, 但我们可以通过Selenium调用PhantomJS来直接使用.

在Ubuntu16.04中可以使用命令安装：sudo apt-get install phantomjs

如果其他系统无法安装, 可以从它的官方网站[http://phantomjs.org/download.html)](http://phantomjs.org/download.html) 下载.

PhantomJS 官方参考文档: [http://phantomjs.org/documentation](http://phantomjs.org/documentation/)

## 快速入门

Selenium 库里有个叫 WebDriver 的 API. WebDriver 有点儿像可以加载网站的浏览器, 但是它也可以像 BeautifulSoup 或者其他 Selector 对象一样用来查找页面元素, 与页面上的元素进行交互 (发送文本、点击等), 以及执行其他动作来运行网络爬虫.

# IPython2 测试代码

# 导入 webdriver

from selenium import webdriver

# 要想调用键盘按键操作需要引入keys包

from selenium.webdriver.common.keys import Keys

# 调用环境变量指定的PhantomJS浏览器创建浏览器对象

driver = webdriver.PhantomJS()

# 如果没有在环境变量指定PhantomJS位置

# driver = webdriver.PhantomJS(executable\_path="./phantomjs"))

# get方法会一直等到页面被完全加载, 然后才会继续程序, 通常测试会在这里选择 time.sleep(2)

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

# 获取页面名为 wrapper的id标签的文本内容

data = driver.find\_element\_by\_id("wrapper").text

# 打印数据内容

print data

# 打印页面标题 "百度一下, 你就知道"

print driver.title

# 生成当前页面快照并保存

driver.save\_screenshot("baidu.png")

# id="kw"是百度搜索输入框, 输入字符串"长城"

driver.find\_element\_by\_id("kw").send\_keys(u"长城")

# id="su"是百度搜索按钮, click() 是模拟点击

driver.find\_element\_by\_id("su").click()

# 获取新的页面快照

driver.save\_screenshot("长城.png")

# 打印网页渲染后的源代码

print driver.page\_source

# 获取当前页面Cookie

print driver.get\_cookies()

# ctrl+a 全选输入框内容

driver.find\_element\_by\_id("kw").send\_keys(Keys.CONTROL,'a')

# ctrl+x 剪切输入框内容

driver.find\_element\_by\_id("kw").send\_keys(Keys.CONTROL,'x')

# 输入框重新输入内容

driver.find\_element\_by\_id("kw").send\_keys("itcast")

# 模拟Enter回车键

driver.find\_element\_by\_id("su").send\_keys(Keys.RETURN)

# 清除输入框内容

driver.find\_element\_by\_id("kw").clear()

# 生成新的页面快照

driver.save\_screenshot("itcast.png")

# 获取当前url

print driver.current\_url

# 关闭当前页面, 如果只有一个页面, 会关闭浏览器

# driver.close()

# 关闭浏览器

driver.quit()

## 页面操作

Selenium 的 WebDriver提供了各种方法来寻找元素, 假设下面有一个表单输入框:

<input type="text" name="user-name" id="passwd-id" />

那么:

# 获取id标签值

element = driver.find\_element\_by\_id("passwd-id")

# 获取name标签值

element = driver.find\_element\_by\_name("user-name")

# 获取标签名值

element = driver.find\_elements\_by\_tag\_name("input")

# 也可以通过XPath来匹配

element = driver.find\_element\_by\_xpath("//input[@id='passwd-id']")

### 定位UI元素 (WebElements)

关于元素的选取, 有如下的API 单个元素选取

find\_element\_by\_id

find\_elements\_by\_name

find\_elements\_by\_xpath

find\_elements\_by\_link\_text

find\_elements\_by\_partial\_link\_text

find\_elements\_by\_tag\_name

find\_elements\_by\_class\_name

find\_elements\_by\_css\_selector

1. By ID

<div id="coolestWidgetEvah">...</div>

**实现**

element = driver.find\_element\_by\_id("coolestWidgetEvah")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

element = driver.find\_element(by=By.ID, value="coolestWidgetEvah")

1. By Class Name

<div class="cheese"><span>Cheddar</span></div><div class="cheese"><span>Gouda</span></div>

**实现**

cheeses = driver.find\_elements\_by\_class\_name("cheese")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

cheeses = driver.find\_elements(By.CLASS\_NAME, "cheese")

1. By Tag Name

<iframe src="..."></iframe>

**实现**

frame = driver.find\_element\_by\_tag\_name("iframe")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

frame = driver.find\_element(By.TAG\_NAME, "iframe")

1. By Name

<input name="cheese" type="text"/>

**实现**

cheese = driver.find\_element\_by\_name("cheese")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

cheese = driver.find\_element(By.NAME, "cheese")

1. By Link Text

<a href="http://www.google.com/search?q=cheese">cheese</a>

**实现**

cheese = driver.find\_element\_by\_link\_text("cheese")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

cheese = driver.find\_element(By.LINK\_TEXT, "cheese")

1. By Partial Link Text

<a href="http://www.google.com/search?q=cheese">search for cheese</a>>

**实现**

cheese = driver.find\_element\_by\_partial\_link\_text("cheese")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

cheese = driver.find\_element(By.PARTIAL\_LINK\_TEXT, "cheese")

1. By CSS

<div id="food"><span class="dairy">milk</span><span class="dairy aged">cheese</span></div>

**实现**

cheese = driver.find\_element\_by\_css\_selector("#food span.dairy.aged")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

cheese = driver.find\_element(By.CSS\_SELECTOR, "#food span.dairy.aged")

1. By XPath

<input type="text" name="example" />

<INPUT type="text" name="other" />

**实现**

inputs = driver.find\_elements\_by\_xpath("//input")

------------------------ or -------------------------

from selenium.webdriver.common.by import By

inputs = driver.find\_elements(By.XPATH, "//input")

## 鼠标动作链

有些时候, 我们需要再页面上模拟一些鼠标操作, 比如双击、右击、拖拽甚至按住不动等, 我们可以通过导入 ActionChains 类来做到:

示例:

#导入 ActionChains 类

from selenium.webdriver import ActionChains

# 鼠标移动到 ac 位置

ac = driver.find\_element\_by\_xpath('element')

ActionChains(driver).move\_to\_element(btn).perform()

# 在 ac 位置单击

ac = driver.find\_element\_by\_xpath("elementA")

ActionChains(driver).move\_to\_element(ac).click(ac).perform()

# 在 ac 位置双击

ac = driver.find\_element\_by\_xpath("elementB")

ActionChains(driver).move\_to\_element(ac).double\_click(ac).perform()

# 在 ac 位置右击

ac = driver.find\_element\_by\_xpath("elementC")

ActionChains(driver).move\_to\_element(ac).context\_click(ac).perform()

# 在 ac 位置左键单击hold住

ac = driver.find\_element\_by\_xpath('elementF')

ActionChains(driver).move\_to\_element(ac).click\_and\_hold(ac).perform()

# 将 ac1 拖拽到 ac2 位置

ac1 = driver.find\_element\_by\_xpath('elementD')

ac2 = driver.find\_element\_by\_xpath('elementE')

ActionChains(driver).drag\_and\_drop(ac1, ac2).perform()

## 填充表单

我们已经知道了怎样向文本框中输入文字, 但是有时候我们会碰到<select> </select>标签的下拉框. 直接点击下拉框中的选项不一定可行.

<select id="status" class="form-control valid" onchange="" name="status">

<option value=""></option>

<option value="0">未审核</option>

<option value="1">初审通过</option>

<option value="2">复审通过</option>

<option value="3">审核不通过</option>

</select>

Selenium专门提供了Select类来处理下拉框. 其实 WebDriver 中提供了一个叫 Select 的方法, 可以帮助我们完成这些事情:

# 导入 Select 类

from selenium.webdriver.support.ui import Select

# 找到 name 的选项卡

select = Select(driver.find\_element\_by\_name('status'))

#

select.select\_by\_index(1)

select.select\_by\_value("0")

select.select\_by\_visible\_text(u"未审核")

以上是三种选择下拉框的方式, 它可以根据索引来选择, 可以根据值来选择, 可以根据文字来选择. 注意:

* index 索引从 0 开始
* value是option标签的一个属性值, 并不是显示在下拉框中的值
* visible\_text是在option标签文本的值, 是显示在下拉框的值

全部取消选择怎么办呢？很简单:

select.deselect\_all()

## 弹窗处理

当你触发了某个事件之后, 页面出现了弹窗提示, 处理这个提示或者获取提示信息方法如下:

alert = driver.switch\_to\_alert()

## 页面切换

一个浏览器肯定会有很多窗口, 所以我们肯定要有方法来实现窗口的切换. 切换窗口的方法如下:

driver.switch\_to.window("this is window name")

也可以使用 window\_handles 方法来获取每个窗口的操作对象. 例如:

for handle in driver.window\_handles:

driver.switch\_to\_window(handle)

## 页面前进和后退

页面前进

driver.forward()

页面后退

driver.back()

# encoding: utf-8

**from** **selenium** **import** webdriver

**import** **time**

driver = webdriver.Chrome()

# 因为对于在第一个窗口中打开的页面, 都是等所有元素完全加载完成后才执行下一步的操作, 所以这里不需要使用time.sleep进行等待.

driver.get("https://www.baidu.com/s?wd=python")

**print**("当前的url地址是**%s**"%driver.current\_url)

driver.get("https://www.baidu.com/s?wd=java")

**print**("当前的url地址是**%s**"%driver.current\_url)

driver.get("https://www.baidu.com/s?wd=c++")

**print**("当前的url地址是**%s**"%driver.current\_url)

driver.back()

**print**("第1次执行后退操作**%s**"%driver.current\_url)

driver.back()

**print**("第2次执行后退操作**%s**"%driver.current\_url)

driver.forward()

**print**("第1次执行前进操作**%s**"%driver.current\_url)

driver.forward()

**print**("第2次执行前进操作**%s**"%driver.current\_url)

time.sleep(3)

driver.quit()

## Cookies

获取页面每个Cookies值, 用法如下

for cookie in driver.get\_cookies():

print "%s -> %s" % (cookie['name'], cookie['value'])

删除Cookies, 用法如下

# By name

driver.delete\_cookie("CookieName")

# all

driver.delete\_all\_cookies()

## 页面等待

注意: 这是非常重要的一部分！！

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.

为了避免这种元素定位困难而且会提高产生 ElementNotVisibleException 的概率. 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.

隐式等待是等待特定的时间, 显式等待是指定某一条件直到这个条件成立时继续执行.

#### 显式等待

显式等待指定某个条件, 然后设置最长等待时间. 如果在这个时间还没有找到元素, 那么便会抛出异常了.

from selenium import webdriver

from selenium.webdriver.common.by import By

# WebDriverWait 库, 负责循环等待

from selenium.webdriver.support.ui import WebDriverWait

# expected\_conditions 类, 负责条件出发

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

driver = webdriver.Chrome()

driver.get("http://www.xxxxx.com/loading")

try:

# 页面一直循环, 直到 id="myDynamicElement" 出现

element = WebDriverWait(driver, 10).until(

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

)

finally:

driver.quit()

如果不写参数, 程序默认会 0.5s 调用一次来查看元素是否已经生成, 如果本来元素就是存在的, 那么会立即返回.

下面是一些内置的等待条件, 你可以直接调用这些条件, 而不用自己写某些等待条件了.

title\_is

title\_contains

presence\_of\_element\_located

visibility\_of\_element\_located

visibility\_of

presence\_of\_all\_elements\_located

text\_to\_be\_present\_in\_element

text\_to\_be\_present\_in\_element\_value

frame\_to\_be\_available\_and\_switch\_to\_it

invisibility\_of\_element\_located

element\_to\_be\_clickable – it is Displayed and Enabled.

staleness\_of

element\_to\_be\_selected

element\_located\_to\_be\_selected

element\_selection\_state\_to\_be

element\_located\_selection\_state\_to\_be

alert\_is\_present

#### 隐式等待

隐式等待比较简单, 就是简单地设置一个等待时间, 单位为秒.

from selenium import webdriver

driver = webdriver.Chrome()

driver.implicitly\_wait(10) # seconds

driver.get("http://www.xxxxx.com/loading")

myDynamicElement = driver.find\_element\_by\_id("myDynamicElement")

当然如果不设置, 默认等待时间为0.

## 案例一: 模拟登录豆瓣

douban\_login\_selenium.py

# douban.py

#导入webdriver API对象, 可以调用浏览器和操作页面

**from** selenium **import** webdriver

#导入Keys, 可以使用操作键盘、标签、鼠标

**from** selenium.webdriver.common.keys **import** Keys

**import** time

#创建PhantomJS浏览器对象

driver = webdriver.PhantomJS()

driver.get(**"http://www.douban.com"**)

driver.save\_screenshot(**"douban.jpg"**)

# 输入账号密码

driver.find\_element\_by\_name(**"form\_email"**).send\_keys(**"xxxxx@xxxx.com"**)

driver.find\_element\_by\_name(**"form\_password"**).send\_keys(**"xxxxxxxx"**)

# 模拟点击登录

driver.find\_element\_by\_xpath(**"//input[@class='bn-submit']"**).click()

#或者

driver.find\_element\_by\_class\_name(**"bn-submit"**).click()

# 等待3秒

time.sleep(3)

# 生成登陆后快照

driver.save\_screenshot(**"douban.png"**)

**with** open(**"douban.html"**, **"wb"**) **as** file:

file.write(driver.page\_source.encode(**"utf-8"**))

**with** open(**"douban2.html"**, **"w"**, encoding=**"utf-8"**) **as** file:

file.write(driver.page\_source)

driver.quit()

新版代码

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

# douban.py

#coding=utf-8

import time

from selenium import webdriver

from selenium.webdriver.common.keys import Keys

class Douban():

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

self.url = "https://www.douban.com/"

self.driver = webdriver.PhantomJS()

def log\_in(self):

self.driver.get(self.url)

time.sleep(3)#睡3分钟, 等待页面加载

self.driver.save\_screenshot("0.jpg")

#输入账号

self.driver.find\_element\_by\_xpath('//\*[@id="form\_email"]').send\_keys("xxxxx@qq.com")

#输入密码

self.driver.find\_element\_by\_xpath('//\*[@id="form\_password"]').send\_keys("xxxx")

#点击登陆

self.driver.find\_element\_by\_class\_name("bn-submit").click()

time.sleep(2)

self.driver.save\_screenshot("douban.jpg")

#输出登陆之后的cookies

print(self.driver.get\_cookies())

def \_\_del\_\_(self):

'''调用内建的稀构方法, 在程序退出的时候自动调用

类似的还可以在文件打开的时候调用close, 数据库链接的断开

'''

self.driver.quit()

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

douban = Douban() #实例化

douban.log\_in() #之后调用登陆方法

## 案例二: 动态页面模拟点击

爬取斗鱼直播平台的所有房间信息：

# 准备工作

from selenium import web driver

from selenium.webdriver.common.keys import Keys

#修改phantomjs的user-agent

from selenium.webdriver.common.desired\_capabilities import DesiredCapabilities

dcap = dict(DesiredCapabilities.PHANTOMJS)

dcap["phantomjs.page.settings.userAgent"] = ("Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko")

driver = webdriver.PhantomJS(desired\_capabilities=dcap)

driver.get("https://www.douyu.com/directory/all")

driver.save\_screenshot("douyu.jpg")

driver.current\_url

driver.page\_source

cap\_dict = driver.desired\_capabilities

for key in cap\_dict:

print('%s: %s'%(key,cap\_dict[key]))

driver.page\_source.find("shark-pager-disable-next")

Out[41]: -1

driver.page\_source.find("shark-pager-disable-prev")

Out[42]: 256204

python2版

#!/usr/bin/env python

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

# unittest 为python的测试模块

import unittest

from selenium import webdriver

from bs4 import BeautifulSoup

class douyuSelenium(unittest.TestCase):

# 初始化方法

def setUp(self):

self.driver = webdriver.PhantomJS()

#具体的测试用例方法, 一定要以test开头

def testDouyu(self):

self.driver.get('http://www.douyu.com/directory/all')

while True:

# 指定xml解析

soup = BeautifulSoup(driver.page\_source, 'xml')

# 返回当前页面所有房间标题列表 和 观众人数列表

titles = soup.find\_all('h3', {'class': 'ellipsis'})

nums = soup.find\_all('span', {'class': 'dy-num fr'})

# 使用zip()函数来可以把列表合并, 并创建一个元组对的列表[(1,2), (3,4)]

for title, num in zip(nums, titles):

print u"观众人数:" + num.get\_text().strip(), u"\t房间标题: " + title.get\_text().strip()

# page\_source.find()未找到内容则返回-1

if driver.page\_source.find('shark-pager-disable-next') != -1:

break

# 模拟下一页点击

self.driver.find\_element\_by\_class\_name('shark-pager-next').click()

# 退出时的清理方法

def tearDown(self):

print '加载完成...'

self.driver.quit()

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

unittest.main()

案例二：动态页面模拟点击

*# unittest为python的测试模块*

**import** unittest, time

**from** selenium **import** webdriver

**from** bs4 **import** BeautifulSoup

**from** selenium.webdriver.common.desired\_capabilities **import** DesiredCapabilities

*#是一个测试模块, 它有三个方法. 第一个方法setUP, 第二个test, 第三个tearDown*

**class** douyuSelenium(unittest.TestCase):

*# 初始化方法, 必须是setUp()*

**def** setUp(self):

*#创建一个浏览器对象*

dcap = dict(DesiredCapabilities.PHANTOMJS)

dcap[**"phantomjs.page.settings.userAgent"**] = (**"Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.81 Safari/537.36"**)

self.driver = webdriver.PhantomJS(desired\_capabilities=dcap)

*# self.driver = webdriver.PhantomJS()*

*#统计一下douyu现在共有多少人在看直播*

self.num = 0

*#统计一下多少人在看直播*

self.watchNum = 0

*#具体的测试用例方法, 一定要以test开头*

**def** testDouyu(self):

self.driver.get(**'http://www.douyu.com/directory/all'**)

*# self.driver.save\_screenshot("douyu.jpg")*

**while True**:

*# 指定lxml解析*

soup = BeautifulSoup(self.driver.page\_source, **'lxml'**)

*# 返回当前页面所有房间标题列表 和 观众人数列表*

titles = soup.find\_all(**'h3'**, {**'class'**: **'ellipsis'**})

nums = soup.find\_all(**'span'**, {**'class'**: **'dy-num fr'**})

*# 使用zip()函数来可以把列表合并, 并创建一个元组对的列表[(1,2), (3,4)]*

**for** title, num **in** zip(titles, nums):

print(**u"观众人数:"** + num.get\_text().strip(), **u"\t房间标题: "** + title.get\_text().strip())

*#统计正在进行直播的人数*

self.num += 1

*#统计观看直播的人数*

*# watch\_num = num.get\_text().strip().replace("观众人数:")*

*# if "万" in watch\_num:*

*# watch\_num.replace("万","")*

*# self.watchNum += num.get\_text().strip().replace("观众人数:")*

*# page\_source.find()未找到内容则返回-1*

*#如果在页面源码里找到下一页为隐藏的标签, 就退出循环*

time.sleep(3)

**if** self.driver.page\_source.find(**'shark-pager-disable-next'**) != -1:

**break**

*# 模拟下一页点击*

*#如果上一步中不退出, 即下一页还存在并可点击时, 就一直不停的点击下一页*

self.driver.find\_element\_by\_class\_name(**'shark-pager-next'**).click()

*# 退出时的清理方法, 测试结束时执行的方法*

**def** tearDown(self):

print(**'加载完成...'**)

print(self.num)

*#退出PhantomJS浏览器*

self.driver.quit()

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

*#启动测试模块*

unittest.main()

新版代码

#coding=utf-8

from selenium import webdriver

import json

import time

class Douyu:

# 1.发送首页的请求

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

self.driver = webdriver.PhantomJS()

self.driver.get("https://www.douyu.com/directory/all") #请求首页

#获取没页面内容

def get\_content(self):

time.sleep(3) #每次发送完请求等待三秒, 等待页面加载完成

li\_list = self.driver.find\_elements\_by\_xpath('//ul[@id="live-list-contentbox"]/li')

contents = []

for i in li\_list: #遍历房间列表

item = {}

item["img"] = i.find\_element\_by\_xpath("./a//img").get\_attribute("src") #获取房间图片

item["title"] = i.find\_element\_by\_xpath("./a").get\_attribute("title") #获取房间名字

item["category"] = i.find\_element\_by\_xpath("./a/div[@class='mes']/div/span").text #获取房间分类

item["name"] = i.find\_element\_by\_xpath("./a/div[@class='mes']/p/span[1]").text #获取主播名字

item["watch\_num"] = i.find\_element\_by\_xpath("./a/div[@class='mes']/p/span[2]").text #获取观看人数

print(item)

contents.append(item)

return contents

#保存本地

def save\_content(self,contents):

f = open("douyu.txt","a")

for content in contents:

json.dump(content,f,ensure\_ascii=False,indent=2)

f.write("\n")

f.close()

def run(self):

#1.发送首页的请求

#2.获取第一页的信息

contents = self.get\_content()

#保存内容

self.save\_content(contents)

#3.循环 点击下一页按钮, 知道下一页对应的class名字不再是"shark-pager-next"

while self.driver.find\_element\_by\_class\_name("shark-pager-next"): #判断有没有下一页

#点击下一页的按钮

self.driver.find\_element\_by\_class\_name("shark-pager-next").click() #

# 4.继续获取下一页的内容

contents = self.get\_content()

#4.1.保存内容

self.save\_content(contents)

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

douyu = Douyu()

douyu.run()

## 案例三: 执行JavaScript语句

### 隐藏百度图片

from selenium import webdriver

driver = webdriver.PhantomJS()

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

# 给搜索输入框标红的javascript脚本

js = "var q=document.getElementById(\"kw\");q.style.border=\"2px solid red\";"

# 调用给搜索输入框标红js脚本

driver.execute\_script(js)

#查看页面快照

driver.save\_screenshot("redbaidu.png")

#js隐藏元素, 将获取的图片元素隐藏

img = driver.find\_element\_by\_xpath("//\*[@id='lg']/img")

driver.execute\_script('$(arguments[0]).fadeOut()',img)

# 向下滚动到页面底部

driver.execute\_script("$('.scroll\_top').click(function(){$('html,body').animate({scrollTop: '0px'}, 800);});")

#查看页面快照

driver.save\_screenshot("nullbaidu.png")

driver.quit()

### 模拟滚动条滚动到底部

from selenium import webdriver

import time

driver = webdriver.PhantomJS()

driver.get("https://movie.douban.com/typerank?type\_name=剧情&type=11&interval\_id=100:90&action=")

# 向下滚动10000像素

js = "document.body.scrollTop=10000"

#js="var q=document.documentElement.scrollTop=10000"

time.sleep(3)

#查看页面快照

driver.save\_screenshot("douban.png")

# 执行JS语句

driver.execute\_script(js)

time.sleep(10)

#查看页面快照

driver.save\_screenshot("newdouban.png")

driver.quit()

## 机器视觉与Tesseract介绍

### 机器视觉

从 Google 的无人驾驶汽车到可以识别假钞的自动售卖机, 机器视觉一直都是一个应用广 泛且具有深远的影响和雄伟的愿景的领域.

我们将重点介绍机器视觉的一个分支: 文字识别, 介绍如何用一些 Python库来识别和使用在线图片中的文字.

我们可以很轻松的阅读图片里的文字, 但是机器阅读这些图片就会非常困难, 利用这种人类用户可以正常读取但是大多数机器人都没法读取的图片, 验证码 (CAPTCHA)就出现了. 验证码读取的难易程度也大不相同, 有些验证码比其他的更加难读.

将图像翻译成文字一般被称为光学文字识别(Optical Character Recognition, OCR). 可以实现OCR的底层库并不多,目前很多库都是使用共同的几个底层 OCR 库,或者是在上面 进行定制.

### ORC库概述

在读取和处理图像、图像相关的机器学习以及创建图像等任务中, Python 一直都是非常出色的语言. 虽然有很多库可以进行图像处理, 但在这里我们只重点介绍: [Tesseract](https://pypi.python.org/pypi/pytesseract)

#### Tesseract

Tesseract 是一个 OCR 库,目前由 Google 赞助(Google 也是一家以 OCR 和机器学习技术闻名于世的公司). Tesseract 是目前公认最优秀、最精确的开源 OCR 系统. 除了极高的精确度,Tesseract 也具有很高的灵活性. 它可以通过训练识别出任何字体, 也可以识别出任何 Unicode 字符.

### 安装Tesseract

#### Windows 系统

下载可执行安装文件<https://code.google.com/p/tesseract-ocr/downloads/list>安装.

#### Linux 系统

可以通过 apt-get 安装: $sudo apt-get tesseract-ocr

#### Mac OS X系统

用 Homebrew(<http://brew.sh/)等第三方库可以很方便地安装> brew install tesseract

要使用 Tesseract 的功能, 比如后面的示例中训练程序识别字母, 要先在系统中设置一 个新的环境变量 $TESSDATA\_PREFIX, 让 Tesseract 知道训练的数据文件存储在哪里, 然后搞一份tessdata数据文件, 放到Tesseract目录下.

* 在大多数 Linux 系统和 Mac OS X 系统上,你可以这么设置: $export TESSDATA\_PREFIX=/usr/local/share/Tesseract
* 在 Windows 系统上也类似,你可以通过下面这行命令设置环境变量: #setx TESSDATA\_PREFIX C:\Program Files\Tesseract OCR\Tesseract

### 安装pytesseract

Tesseract 是一个 Python 的命令行工具, 不是通过 import 语句导入的库. 安装之后,要用 tesseract 命令在 Python 的外面运行, 但我们可以通过 pip 安装支持Python 版本的 Tesseract库:

pip install pytesseract

## 处理一些格式规范的文字

你要处理的大多数文字都是比较干净、格式规范的. 格式规范的文字通常可以满足一些需求,不过究竟什么是"格式混乱",什么算"格式规范",确实因人而异. 通常,格式规范的文字具有以下特点:

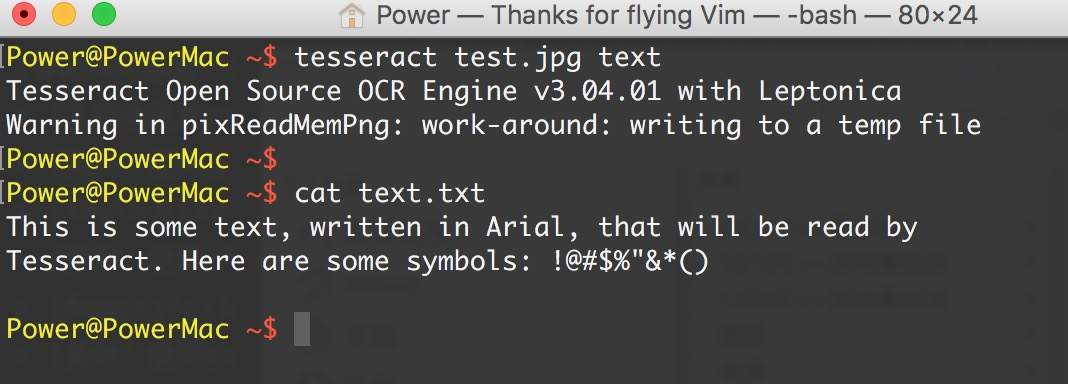
* 使用一个标准字体(不包含手写体、草书,或者十分"花哨的"字体) • 虽然被复印或拍照,字体还是很清晰,没有多余的痕迹或污点
* 排列整齐,没有歪歪斜斜的字
* 没有超出图片范围,也没有残缺不全,或紧紧贴在图片的边缘

文字的一些格式问题在图片预处理时可以进行解决. 例如,可以把图片转换成灰度图,调 整亮度和对比度,还可以根据需要进行裁剪和旋转（详情请关注图像与信号处理）, 但是,这些做法在进行更具扩展性的 训练时会遇到一些限制.

### 格式规范文字的理想示例



通过下面的命令运行 Tesseract, 读取文件并把结果写到一个文本文件中: `tesseract test.jpg text



cat text.txt 即可显示结果.

识别结果很准确,不过符号^和\*分别被表示成了双引号和单引号. 大体上可以让你很舒服地阅读.

### 通过Python代码实现

import pytesseract

from PIL import Image

image = Image.open('test.jpg')

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

print text

运行结果:

This is some text, written in Arial, that will be read by

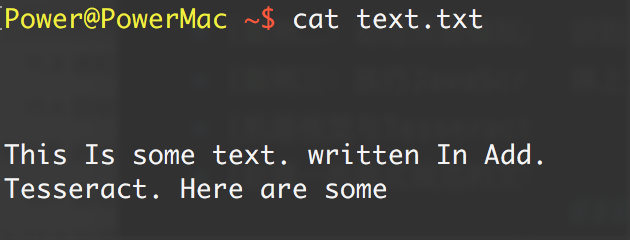
Tesseract. Here are some symbols: !@#$%"&\*()

### 对图片进行阈值过滤和降噪处理（了解即可）

很多时候我们在网上会看到这样的图片:



Tesseract 不能完整处理这个图片,主要是因为图片背景色是渐变的,最终结果是这样:



随着背景色从左到右不断加深,文字变得越来越难以识别,Tesseract 识别出的 每一行的最后几个字符都是错的.

遇到这类问题,可以先用 Python 脚本对图片进行清理. 利用 Pillow 库,我们可以创建一个 阈值过滤器来去掉渐变的背景色,只把文字留下来,从而让图片更加清晰,便于 Tesseract 读取:

from PIL import Image

import subprocess

def cleanFile(filePath, newFilePath):

image = Image.open(filePath)

# 对图片进行阈值过滤,然后保存

image = image.point(lambda x: 0 if x<143 else 255)

image.save(newFilePath)

# 调用系统的tesseract命令对图片进行OCR识别

subprocess.call(["tesseract", newFilePath, "output"])

# 打开文件读取结果

file = open("output.txt", 'r')

print(file.read())

file.close()

cleanFile("text2.jpg", "text2clean.png")

通过一个阈值对前面的"模糊"图片进行过滤的结果

除了一些标点符号不太清晰或丢失了,大部分文字都被读出来了. Tesseract 给出了最好的 结果:

### 从网站图片中抓取文字

用 Tesseract 读取硬盘里图片上的文字,可能不怎么令人兴奋,但当我们把它和网络爬虫组合使用时,就能成为一个强大的工具.

网站上的图片可能并不是故意把文字做得很花哨 (就像餐馆菜单的 JPG 图片上的艺术字),但它们上面的文字对网络爬虫来说就是隐藏起来 了, 举个例子:

* 虽然亚马逊的 robots.txt 文件允许抓取网站的产品页面,但是图书的预览页通常不让网络机 器人采集.
* 图书的预览页是通过用户触发 Ajax 脚本进行加载的,预览图片隐藏在 div 节点 下面;其实,普通的访问者会觉得它们看起来更像是一个 Flash 动画,而不是一个图片文 件. 当然,即使我们能获得图片,要把它们读成文字也没那么简单.
* 下面的程序就解决了这个问题:首先导航到托尔斯泰的《战争与和平》的大字号印刷版 1, 打开阅读器,收集图片的 URL 链接,然后下载图片,识别图片,最后打印每个图片的文 字. 因为这个程序很复杂,利用了前面几章的多个程序片段,所以我增加了一些注释以让 每段代码的目的更加清晰:

import time

from urllib.request import urlretrieve

import subprocess

from selenium import webdriver

#创建新的Selenium driver

driver = webdriver.PhantomJS()

# 用Selenium试试Firefox浏览器:

# driver = webdriver.Firefox()

driver.get("http://www.amazon.com/War-Peace-Leo-Nikolayevich-Tolstoy/dp/1427030200")

# 单击图书预览按钮 driver.find\_element\_by\_id("sitbLogoImg").click() imageList = set()

# 等待页面加载完成

time.sleep(5)

# 当向右箭头可以点击时,开始翻页

while "pointer" in driver.find\_element\_by\_id("sitbReaderRightPageTurner").get\_attribute("style"):

driver.find\_element\_by\_id("sitbReaderRightPageTurner").click()

time.sleep(2)

# 获取已加载的新页面(一次可以加载多个页面,但是重复的页面不能加载到集合中)

pages = driver.find\_elements\_by\_xpath("//div[@class='pageImage']/div/img")

for page in pages:

image = page.get\_attribute("src")

imageList.add(image)

driver.quit()

# 用Tesseract处理我们收集的图片URL链接

for image in sorted(imageList):

# 保存图片

urlretrieve(image, "page.jpg")

p = subprocess.Popen(["tesseract", "page.jpg", "page"], stdout=subprocess.PIPE,stderr=subprocess.PIPE)

f = open("page.txt", "r")

p.wait() print(f.read())

和我们前面使用 Tesseract 读取的效果一样,这个程序也会完美地打印书中很多长长的段 落,第六页的预览如下所示:

6

"A word of friendly advice, mon

cher. Be off as soon as you can,

that's all I have to tell you. Happy

he who has ears to hear. Good-by,

my dear fellow. Oh, by the by!" he

shouted through the doorway after

Pierre, "is it true that the countess

has fallen into the clutches of the

holy fathers of the Society of je-

sus?"

Pierre did not answer and left Ros-

topchin's room more sullen and an-

gry than he had ever before shown

himself.

但是,当文字出现在彩色封面上时,结果就不那么完美了:

WEI' nrrd Peace

Len Nlkelayevldu Iolfluy

Readmg shmdd be ax

wlnvame asnossxble Wenfler

an mm m our cram: Llhvary

- Leo Tmsloy was a Russian rwovelwst

I and moval phflmopher med lur

A ms Ideas 01 nonviolenx reswslance m 5 We range 0, "and"

如果想把文字加工成普通人可以看懂的 效果,还需要花很多时间去处理.

下一节将介绍另一种方法来解决文字混乱的问题,尤其是当你愿意花一点儿时间训练 Tesseract 的时候.

通过给 Tesseract 提供大量已知的文字与图片映射集,经过训练 Tesseract 就可以"学会"识别同一种字体,而且可以达到极高的精确率和准确率,甚至可以忽略图 片中文字的背景色和相对位置等问题.

## 案例: 尝试对知乎验证码进行机器识别处理

许多流行的内容管理系统即使加了验证码模块, 其众所周知的注册页面也经常会遭到网络 机器人的垃圾注册.

那么, 这些网络机器人究, 竟是怎么做的呢?既然我们已经, 可以成功地识别出保存在电脑上 的验证码了, 那么如何才能实现一个全能的网络机器人呢?

大多数网站生成的验证码图片都具有以下属性.

* 它们是服务器端的程序动态生成的图片. 验证码图片的 src 属性可能和普通图片不太一 样, 比如 <img src="WebForm.aspx?id=8AP85CQKE9TJ">, 但是可以和其他图片一样进行 下载和处理.
* 图片的答案存储在服务器端的数据库里.
* 很多验证码都有时间限制, 如果你太长时间没解决就会失效.
* 常用的处理方法就是, 首先把验证码图片下载到硬盘里, 清理干净, 然后用 Tesseract 处理 图片, 最后返回符合网站要求的识别结果.

#!/usr/bin/env python

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

import requests

import time

import pytesseract

from PIL import Image

from bs4 import BeautifulSoup

def captcha(data):

with open('captcha.jpg','wb') as fp:

fp.write(data)

time.sleep(1)

image = Image.open("captcha.jpg")

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

print "机器识别后的验证码为: " + text

command = raw\_input("请输入Y表示同意使用, 按其他键自行重新输入: ")

if (command == "Y" or command == "y"):

return text

else:

return raw\_input('输入验证码: ')

def zhihuLogin(username,password):

# 构建一个保存Cookie值的session对象

sessiona = requests.Session()

headers = {'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:47.0) Gecko/20100101 Firefox/47.0'}

# 先获取页面信息, 找到需要POST的数据（并且已记录当前页面的Cookie）

html = sessiona.get('https://www.zhihu.com/#signin', headers=headers).content

# 找到 name 属性值为 \_xsrf 的input标签, 取出value里的值

\_xsrf = BeautifulSoup(html ,'lxml').find('input', attrs={'name':'\_xsrf'}).get('value')

# 取出验证码, r后面的值是Unix时间戳,time.time()

captcha\_url = 'https://www.zhihu.com/captcha.gif?r=%d&type=login' % (time.time() \* 1000)

response = sessiona.get(captcha\_url, headers = headers)

data = {

"\_xsrf":\_xsrf,

"email":username,

"password":password,

"remember\_me":True,

"captcha": captcha(response.content)

}

response = sessiona.post('https://www.zhihu.com/login/email', data = data, headers=headers)

print response.text

response = sessiona.get('https://www.zhihu.com/people/maozhaojun/activities', headers=headers)

print response.text

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

#username = raw\_input("username")

#password = raw\_input("password")

zhihuLogin('xxxx@qq.com','ALAxxxxIME')

值得注意的是, 有两种异常情况会导致这个程序运行失败. 第一种情况是, 如果 Tesseract 从验证码图片中识别的结果不是四个字符(因为训练样本中验证码的所有有效答案都必须 是四个字符), 结果不会被提交, 程序失败. 第二种情况是虽然识别的结果是四个字符, 被提交到了表单, 但是服务器对结果不认可, 程序仍然失败.

在实际运行过程中, 第一种 情况发生的可能性大约为 50%, 发生时程序不会向表单提交, 程序直接结束并提示验证码 识别错误. 第二种异常情况发生的概率约为 20%, 四个字符都对的概率约是 30%(每个字 母的识别正确率大约是 80%, 如果是五个字符都识别, 正确的总概率是 32.8%).

### 视频演示知乎验证码自动识别

import time

time.time()就得到了时间戳

time.ctime()得到标准时间

知乎的验证码地址

<https://www.zhihu.com/captcha.gif?r=1493340106170&type=login&lang=cn>

通过站长工具进行转换

<http://tool.chinaz.com/Tools/unixtime.aspx>

[https://www.zhihu.com/captcha.gif?r=%d&type=login](https://www.zhihu.com/captcha.gif?r=%25d&type=login)

构造类似如下的POST data

"\_xsrf":"38cf46ab1f5feef14dfe49f6fbdca4b2",

"password":"adsfsdfsdf",

"captcha":"r5eh",

"email":"2sdfasfdasf",

Windows NT 6.3 Windows 8.1

Windows NT 6.2 Windows 8

Windows NT 6.1 Windows 7

Windows NT 6.0 Windows Vista

Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko

xpath的模糊查询

根节点, 然后再在根节点下面去取其中的元素

//div[contains(@id, "qiushi\_tag")]

//a[contains(@class, "contentHerf")]/@href

//span[contains(@class, "stats-vote")]/i

//span[contains(@class, "stats-comment")]/a/i

//div[contains(@id, "qiushi\_tag")]//h2

**from** bs4 **import** BeautifulSoup

**import** requests, time

**from** pytesseract **import** \*

**from** PIL **import** Image

**def** captcha(captcah\_data):

**with** open(**"captcha.jpg"**, **"wb"**) **as** f:

f.write(captcah\_data)

time.sleep(2)

*# captcah\_text = input("请输入验证码：")*

*# return captcah\_text*

*#以下为使用机器识别验证码的代码部分*

image = Image.open(**"captcha.jpg"**)

text = image\_to\_string(image)

print(**"机器识别后的验证码为：%s"**%text)

command = input(**"输入Y表示验证码识别正确, 继续使用；输入其他表示手动输入验证码"**)

**if** (command == **"y" or** command == **"Y"**):

**return** text

**else**:

**return** input(**"请输入验证码："**)

**def** zhihuLogin():

*#构建一个Session对象, 可以保存Cookie*

sess = requests.Session()

url = **"https://www.zhihu.com/#signin"**

user\_agent = **"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36"**

headers = {**"User-Agent"**: user\_agent}

*#首先获取登陆页面, 找到需要POST的数据(\_xsrf), 同时会记录当前网页的Cookie值.*

*#也可以直接使用request.get来直接发送请求, 但那样就无法保存Cookie值了.*

html = sess.get(url, headers = headers).text

*#使用BeautifulSoup调用lxml解析库*

soup = BeautifulSoup(html, **"lxml"**)

*#获取之前get的页面里的\_xsrf值*

*#\_xsrf作用是防止CSRF攻击, 跨站请求攻击, 是一种利用网站对用户的一种信任机制来进行攻击. 通常通过伪装成网站信任的用户的请求.*

xsrf = soup.find(**"input"**, attrs={**"name"**: **"\_xsrf"**}).get(**"value"**)

print(xsrf)

*#通过Unix时间戳获取知乎验证码的url*

captcha\_url = **"https://www.zhihu.com/captcha.gif?r=%d&type=login"**%(time.time()\*1000)

print(captcha\_url)

*#发送图片的请求, 获取图片二进制数据流*

captcha\_data = sess.get(captcha\_url, headers=headers).content

*#获取验证码里的文字, 需要手动输入*

captcha\_text = captcha(captcha\_data)

data = {

**"\_xsrf"**: xsrf,

**"phone\_num"**: **"15201933129"**,

**"password"**: **"s6u9p5e3r8"**,

**"captcha"**: captcha\_text,

}

response = sess.post(**"https://www.zhihu.com/login/phone\_num"**, data=data, headers = headers)

*# print(response.text)*

response = sess.get(**"https://www.zhihu.com/people/dreamingtech/activities"**, headers = headers)

**with** open(**"my\_zhihu.html"**, **"wb"**) **as** f:

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

*# print(response.text)*

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

zhihuLogin()

*#个人主页, 必须登陆后才能看.*

*# https://www.zhihu.com/people/dreamingtech/activities*

**'''**

**请输入验证码：VHF9**

**{"r":0,**

**"msg": "\u767b\u5f55\u6210\u529f"**

**}**

**print("\u767b\u5f55\u6210\u529f")**

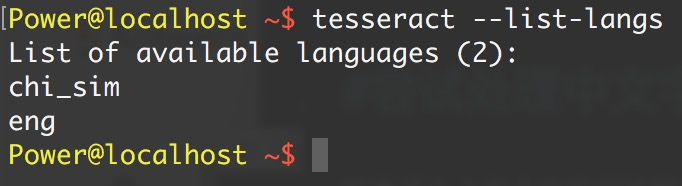
**登录成功**

**'''**

### 尝试处理中文字符

如果手头上有中文的训练数据, 也可以尝试对中文进行识别.

命令：tesseract --list-langs可以查看当前支持的语言, chi\_sim表示支持简体中文.



那么在使用时候, 可以指定某个语言来进行识别, 如：

tesseract -l chi\_sim paixu.png paixu



表现在程序里, 则可以这么写：

#!/usr/bin/env python

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

from PIL import Image

import subprocess

def cleanFile(filePath)

image = Image.open(filePath)

# 调用系统的tesseract命令, 对图片进行OCR中文识别

subprocess.call(["tesseract", "-l", "chi\_sim", filePath, "paixu"])

# 打开文件读取结果

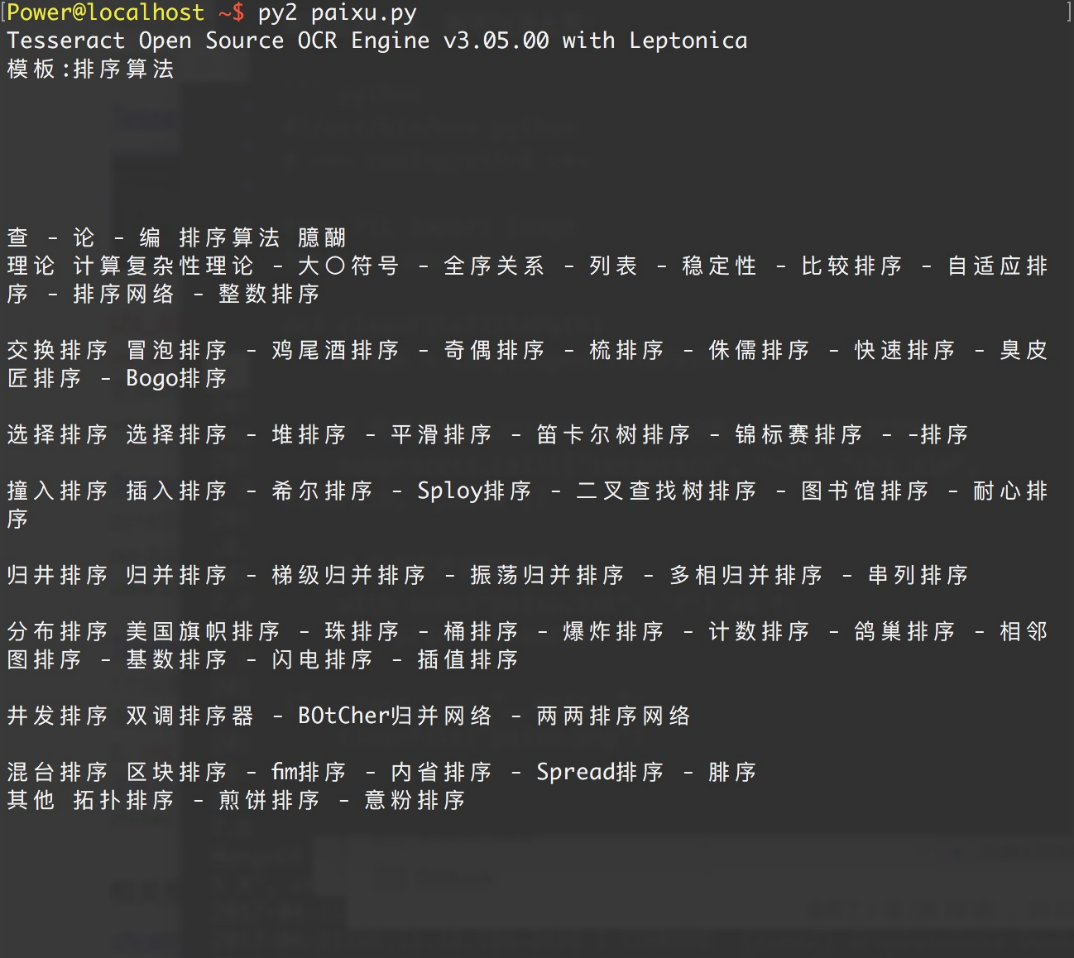
with open("paixu.txt", 'r') as f:

print(f.read())

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

cleanFile("paixu.png")

结果如下：



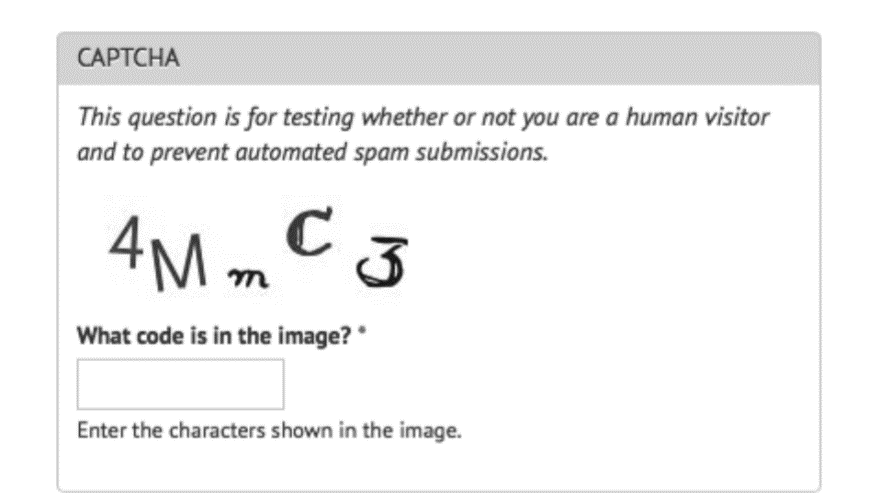
## 机器学习: 训练Tesseract

### 训练Tesseract

要使用 Tesseract 的功能, 比如后面的示例中训练程序识别字母, 要先在系统中设置一 个新的环境变量 $TESSDATA\_PREFIX, 让 Tesseract 知道训练的数据文件存储在哪里, 然后搞一份tessdata数据文件, 放到Tesseract目录下.

* 在大多数 Linux 系统和 Mac OS X 系统上,你可以这么设置: $export TESSDATA\_PREFIX=/usr/local/share/Tesseract
* 在 Windows 系统上也类似,你可以通过下面这行命令设置环境变量: #setx TESSDATA\_PREFIX C:\Program Files\Tesseract OCR\Tesseract

大多数其他的验证码都是比较简单的. 例如, 流行的 PHP 内容管理系统 Drupal 有一个著 名的验证码模块([https://www.drupal.org/project/captcha), 可以生成不同难度的验证码.](https://www.drupal.org/project/captcha)，可以生成不同难度的验证码。)



那么与其他验证码相比, 究竟是什么让这个验证码更容易被人类和机器读懂呢?

* 字母没有相互叠加在一起, 在水平方向上也没有彼此交叉. 也就是说, 可以在每一个字 母外面画一个方框, 而不会重叠在一起.
* 图片没有背景色、线条或其他对 OCR 程序产生干扰的噪点.
* 虽然不能因一个图片下定论, 但是这个验证码用的字体种类很少, 而且用的是 sans-serif 字体(像"4"和"M")和一种手写形式的字体(像"m""C"和"3").
* 白色背景色与深色字母之间的对比度很高.

这个验证码只做了一点点改变, 就让 OCR 程序很难识别.

* 字母和数据都使用了, 这会增加待搜索字符的数量.
* 字母随机的倾斜程度会迷惑 OCR 软件, 但是人类还是很容易识别的.
* 那个比较陌生的手写字体很有挑战性, 在"C"和"3"里面还有额外的线条. 另外这 个非常小的小写"m", 计算机需要进行额外的训练才能识别. 用下面的代码运行 Tesseract 识别图片:

tesseract captchaExample.png output

我们得到的结果 output.txt 是: 4N\, , , C<3

### 创建样本库训练Tesseract

要训练 Tesseract 识别一种文字, 无论是晦涩难懂的字体还是验证码, 你都需要向 Tesseract 提供每个字符不同形式的样本.

做这个枯燥的工作可能要花好几个小时的时间, 你可能更想用这点儿时间找个好看的视频 或电影看看.

首先要收集大量的验证码样本, 样本的数量和复杂程度, 会决定训练的效果.

下载的样本数量由验证码的复杂程度决定; 我在训练集里一共放了 100 个样本(一共 500 个字符, 平均每个字符 8 个样本;a~z 大小写字母加 0~9 数字, 一共 62 个字符), 应该足够训练的了.

提示:建议使用验证码的真实结果给每个样本文件命名(即4MmC3.jpg). 这样可以帮你 一次性对大量的文件进行快速检查——你可以先把图片调成缩略图模式, 然后通过文件名 对比不同的图片. 这样在后面的步骤中进行训练效果的检查也会很方便.

第二步是准确地告诉 Tesseract 一张图片中的每个字符是什么, 以及每个字符的具体位置. 这里需要创建一些矩形定位文件(box file), 一个验证码图片生成一个矩形定位文件. 一 个图片的矩形定位文件如下所示:

4 15 26 33 55 0

M 38 13 67 45 0

m 79 15 101 26 0

C 111 33 136 60 0

3 147 17 176 45 0

第一列符号是图片中的每个字符, 后面的 4 个数字分别是包围这个字符的最小矩形的坐标 (图片左下角是原点 (0, 0), 4 个数字分别对应每个字符的左下角 x 坐标、左下角 y 坐标、右上角 x 坐标和右上角 y 坐标), 最后一个数字"0"表示图片样本的编号.

显然, 手工创建这些图片矩形定位文件很无聊, 不过有一些工具可以帮你完成. 我很喜欢 在线工具 Tesseract OCR Chopper(http://pp19dd.com/tesseract-ocr-chopper/), 因为它不需要 安装, 也没有其他依赖, 只要有浏览器就可以运行, 而且用法很简单:上传图片, 如果要 增加新矩形就单击"add"按钮, 还可以根据需要调整矩形的尺寸, 最后把新生成的矩形 定位文件复制到一个新文件里就可以了.

矩形定位文件必须保存在一个 .box 后缀的文本文件中. 和图片文件一样, 文本文件也是用 验证码的实际结果命名(例如, 4MmC3.box). 另外, 这样便于检查 .box 文件的内容和文件的名称, 而且按文件名对目录中的文件排序之后, 就可以让 .box 文件与对应的图片文件 的实际结果进行对比.

你还需要创建大约 100 个 .box 文件来保证你有足够的训练数据. 因为 Tesseract 会忽略那 些不能读取的文件, 所以建议你尽量多做一些矩形定位文件, 以保证训练足够充分. 如果 你觉得训练的 OCR 结果没有达到你的目标, 或者 Tesseract 识别某些字符时总是出错, 多 创建一些训练数据然后重新训练将是一个不错的改进方法.

创建完满载 .box 文件和图片文件的数据文件夹之后, 在做进一步分析之前最好备份一下这 个文件夹. 虽然在数据上运行训练程序不太可能删除任何数据, 但是创建 .box 文件用了你 好几个小时的时间, 来之不易, 稳妥一点儿总没错. 此外, 能够抓取一个满是编译数据的 混乱目录, 然后再尝试一次, 总是好的.

前面的内容只是对 Tesseract 库强大的字体训练和识别能力的一个简略概述. 如果你对 Tesseract 的其他训练方法感兴趣, 甚至打算建立自己的验证码训练文件库, 或者想和全世 界的 Tesseract 爱好者分享自己对一种新字体的识别成果, 推荐阅读 Tesseract 的文档: <https://github.com/tesseract-ocr/tesseract/wiki>, 加油！

博客园的一篇不错的训练教程：

<http://www.cnblogs.com/mjorcen/p/3800739.html?utm_source=tuicool&utm_medium=referral>

## 爬虫与的爬虫之间的斗争

### 爬虫(Spider), 反爬虫(Anti-Spider), 反反爬虫(Anti-Anti-Spider) 之间恢宏壮阔的斗争...

#### Day 1

* 小莫想要某站上所有的电影, 写了标准的爬虫(基于HttpClient库), 不断地遍历某站的电影列表页面, 根据 Html 分析电影名字存进自己的数据库.
* 这个站点的运维小黎发现某个时间段请求量陡增, 分析日志发现都是 IP(xxx.xxx.xxx.xxx)这个用户, 并且 user-agent 还是 Python-urllib/2.7 , 基于这两点判断非人类后直接在服务器上封杀.

#### Day 2

* 小莫电影只爬了一半, 于是也针对性的变换了下策略: 1. user-agent 模仿百度("Baiduspider..."), 2. IP每爬半个小时就换一个IP代理.
* 小黎也发现了对应的变化, 于是在服务器上设置了一个频率限制, 每分钟超过120次请求的再屏蔽IP. 同时考虑到百度家的爬虫有可能会被误伤, 想想市场部门每月几十万的投放, 于是写了个脚本, 通过 hostname 检查下这个 ip 是不是真的百度家的, 对这些 ip 设置一个白名单.

#### Day 3

* 小莫发现了新的限制后, 想着我也不急着要这些数据, 留给服务器慢慢爬吧, 于是修改了代码, 随机1-3秒爬一次, 爬10次休息10秒, 每天只在8-12, 18-20点爬, 隔几天还休息一下.
* 小黎看着新的日志头都大了, 再设定规则不小心会误伤真实用户, 于是准备换了一个思路, 当3个小时的总请求超过50次的时候弹出一个验证码弹框, 没有准确正确输入的话就把 IP 记录进黑名单.

#### Day 4

* 小莫看到验证码有些傻脸了, 不过也不是没有办法, 先去学习了图像识别（关键词 PIL, tesseract）, 再对验证码进行了二值化, 分词, 模式训练之后, 总之最后识别了小黎的验证码（关于验证码, 验证码的识别, 验证码的反识别也是一个恢弘壮丽的斗争史...）, 之后爬虫又跑了起来.
* 小黎是个不折不挠的好同学, 看到验证码被攻破后, 和开发同学商量了变化下开发模式, 数据并不再直接渲染, 而是由前端同学异步获取, 并且通过 JavaScript 的加密库生成动态的 token, 同时加密库再进行混淆（比较重要的步骤的确有网站这样做, 参见淘宝和微博的登陆流程）.

#### Day 5

* 混淆过的加密库就没有办法了么？当然不是, 可以慢慢调试, 找到加密原理, 不过小莫不准备用这么耗时耗力的方法, 他放弃了基于 HttpClient的爬虫, 选择了内置浏览器引擎的爬虫(关键词: PhantomJS, Selenium), 在浏览器引擎运行页面, 直接获取了正确的结果, 又一次拿到了对方的数据.
* 小黎: .....

### 爬虫与发爬虫的斗争还在继续...

通常情况下, 在爬虫与反爬虫的对弈中, 爬虫一定会胜利.

换言之, 只要人类能够正常访问的网页, 爬虫在具备同等资源的情况下就一定可以抓取到.

### 关于爬虫部分一些建议:

1. 尽量减少请求次数, 能抓列表页就不抓详情页, 减轻服务器压力, 程序员都是混口饭吃不容易.
2. 不要只看 Web 网站, 还有手机 App 和 H5, 这样的反爬虫措施一般比较少.
3. 实际应用时候, 一般防守方做到根据 IP 限制频次就结束了, 除非很核心的数据, 不会再进行更多的验证, 毕竟成本的问题会考虑到.
4. 如果真的对性能要求很高, 可以考虑多线程(一些成熟的框架如 Scrapy都已支持), 甚至分布式...

### 关于反爬虫部分的一些建议:

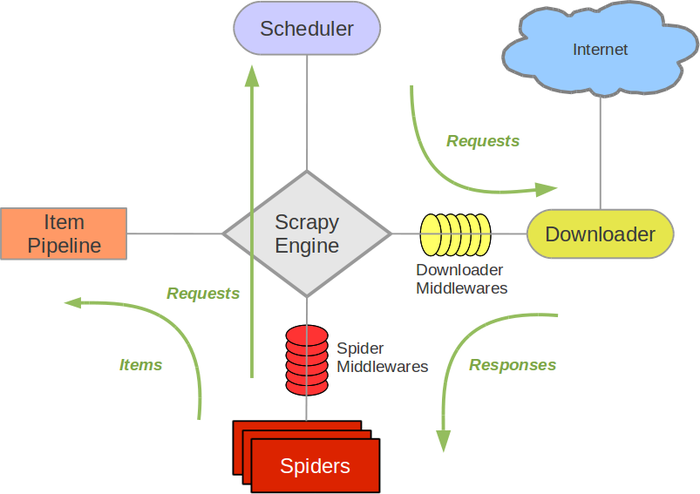
* 这篇文章就够了: [携程技术中心 - 携程酒店研发部研发经理崔广宇 <爬虫与反爬虫> 技术分享](https://segmentfault.com/a/1190000005840672)

# 4. Scrapy框架

## Scrapy 框架

* Scrapy是用纯Python实现一个为了爬取网站数据、提取结构性数据而编写的应用框架, 用途非常广泛.
* 框架的力量, 用户只需要定制开发几个模块就可以轻松的实现一个爬虫, 用来抓取网页内容以及各种图片, 非常之方便.
* Scrapy 使用了 Twisted['twɪstɪd](其主要对手是Tornado)异步网络框架来处理网络通讯, 可以加快我们的下载速度, 不用自己去实现异步框架, 并且包含了各种中间件接口, 可以灵活的完成各种需求.

### Scrapy架构图(绿线是数据流向)：



* Scrapy Engine(引擎): 负责Spider、ItemPipeline、Downloader、Scheduler中间的通讯, 信号、数据传递等.
* Scheduler(调度器): 它负责接受引擎发送过来的Request请求, 并按照一定的方式进行整理排列, 入队, 当引擎需要时, 交还给引擎.
* Downloader（下载器）：负责下载Scrapy Engine(引擎)发送的所有Requests请求, 并将其获取到的Responses交还给Scrapy Engine(引擎), 由引擎交给Spider来处理,
* Spider（爬虫）：它负责处理所有Responses,从中分析提取数据, 获取Item字段需要的数据, 并将需要跟进的URL提交给引擎, 再次进入Scheduler(调度器),
* Item Pipeline(管道)：它负责处理Spider中获取到的Item, 并进行进行后期处理（详细分析、过滤、存储等）的地方.
* Downloader Middlewares（下载中间件）：你可以当作是一个可以自定义扩展下载功能的组件.
* Spider Middlewares（Spider中间件）：你可以理解为是一个可以自定扩展和操作引擎和Spider中间通信的功能组件（比如进入Spider的Responses;和从Spider出去的Requests）

### Scrapy的运作流程

代码写好, 程序开始运行...

1. 引擎：Hi！Spider, 你要处理哪一个网站？
2. Spider：老大要我处理xxxx.com.
3. 引擎：你把第一个需要处理的URL给我吧.
4. Spider：给你, 第一个URL是xxxxxxx.com.
5. 引擎：Hi！调度器, 我这有request请求你帮我排序入队一下.
6. 调度器：好的, 正在处理你等一下.
7. 引擎：Hi！调度器, 把你处理好的request请求给我.
8. 调度器：给你, 这是我处理好的request
9. 引擎：Hi！下载器, 你按照老大的下载中间件的设置帮我下载一下这个request请求
10. 下载器：好的！给你, 这是下载好的东西. （如果失败：sorry, 这个request下载失败了. 然后引擎告诉调度器, 这个request下载失败了, 你记录一下, 我们待会儿再下载）
11. 引擎：Hi！Spider, 这是下载好的东西, 并且已经按照老大的下载中间件处理过了, 你自己处理一下（注意！这儿responses默认是交给def parse()这个函数处理的）
12. Spider：（处理完毕数据之后对于需要跟进的URL）, Hi！引擎, 我这里有两个结果, 这个是我需要跟进的URL, 还有这个是我获取到的Item数据.
13. 引擎：Hi ！管道 我这儿有个item你帮我处理一下！调度器！这是需要跟进URL你帮我处理下. 然后从第四步开始循环, 直到获取完老大需要全部信息.
14. 管道``调度器：好的, 现在就做！

**注意！只有当调度器中不存在任何request了, 整个程序才会停止, （也就是说, 对于下载失败的URL, Scrapy也会重新下载. ）**

制作 Scrapy 爬虫 一共需要4步：

* 新建项目 (scrapy startproject xxx)：新建一个新的爬虫项目
* 明确目标 （编写items.py）：明确你想要抓取的目标
* 制作爬虫 （spiders/xxspider.py）：制作爬虫开始爬取网页
* 存储内容 （pipelines.py）：设计管道存储爬取内容

## 配置安装

### Scrapy的安装介绍

Scrapy框架官方网址：[http://doc.scrapy.org/en/latest](http://doc.scrapy.org/en/latest/)

Scrapy中文维护站点：<http://scrapy-chs.readthedocs.io/zh_CN/latest/index.html>

pip install pillow

from PIL import Image

新建一个单独的文件夹保存项目文件, 在这个项目文件夹中新建scrapy项目

scrapy startproject mySpider

使用pycharm打开mySpider项目

scrapy.cfg是项目的设置文件, 并发数, 管道文件等设置.

/mySpider中\_\_init\_\_.py

item.py

pip install --upgrade pip

pip install Scrapy

### Windows 安装方式

* Python 2 / 3
* 升级pip版本：pip install --upgrade pip
* 通过pip 安装 Scrapy 框架pip install Scrapy

### Ubuntu

* 需要9.10或以上版本安装方式
* Python 2 / 3
* 安装非Python的依赖 sudo apt-get install python-dev python-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-dev libssl-dev
* 通过pip 安装 Scrapy 框架 sudo pip install scrapy

安装后, 只要在命令终端输入 scrapy, 提示类似以下结果, 代表已经安装成功

具体Scrapy安装流程参考：

<http://doc.scrapy.org/en/latest/intro/install.html#intro-install-platform-notes> 里面有各个平台的安装方法

## 入门案例

### 学习目标

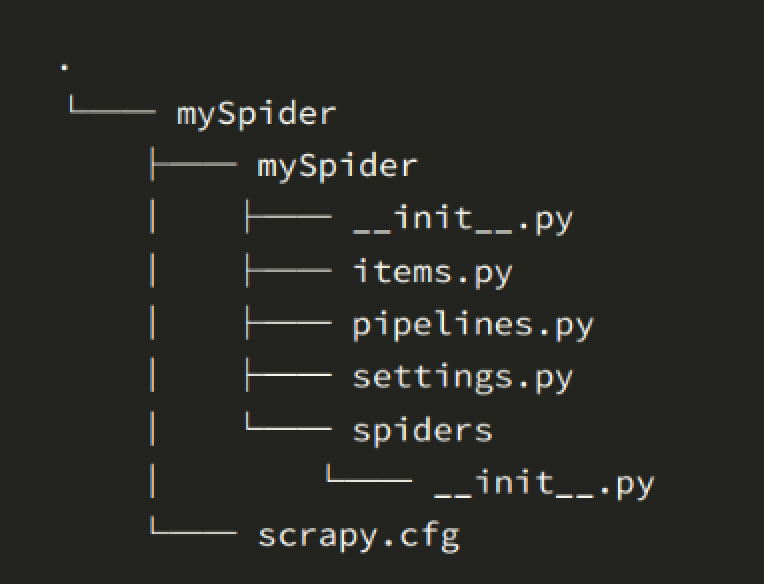
* 创建一个Scrapy项目
* 定义提取的结构化数据(Item)
* 编写爬取网站的 Spider 并提取出结构化数据(Item)
* 编写 Item Pipelines 来存储提取到的Item(即结构化数据)

### 一. 新建项目(scrapy startproject)

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

scrapy startproject mySpider

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



下面来简单介绍一下各个主要文件的作用：

scrapy.cfg ：项目的配置文件

mySpider/ ：项目的Python模块, 将会从这里引用代码

mySpider/items.py ：项目的目标文件

mySpider/pipelines.py ：项目的管道文件

mySpider/settings.py ：项目的设置文件

mySpider/spiders/ ：存储爬虫代码目录

settings设置文件

BOT\_NAME = 'mySpider'

SPIDER\_MODULES = ['mySpider.spiders']

NEWSPIDER\_MODULE = 'mySpider.spiders'

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

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

*# Obey robots.txt rules*

*#是否遵循robots协议*

*# ROBOTSTXT\_OBEY = True*

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

*#爬虫的并发量, 可以同时下载的数据*

*# CONCURRENT\_REQUESTS = 32*

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

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

*# See also autothrottle settings and docs*

*#**下载延迟, 一般不启动, 与time.sleep功能类似*

*#DOWNLOAD\_DELAY = 3*

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

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

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

*# Disable cookies (enabled by default)*

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

*#COOKIES\_ENABLED = False*

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# Override the default request headers:*

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

*#DEFAULT\_REQUEST\_HEADERS = {*

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

*#只显示英文的选项关闭.*

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

*#}*

*# Enable or disable spider middlewares*

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

*#爬虫的中间件一般不用, 下载的中间件用的较多*

*#SPIDER\_MIDDLEWARES = {*

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

*#}*

*# Enable or disable downloader middlewares*

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

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

*#DOWNLOADER\_MIDDLEWARES = {*

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

*#}*

*# Enable or disable extensions*

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

*#EXTENSIONS = {*

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

*#}*

*# Configure item pipelines*

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

*#**管道文件, 决定了下载的数据如何进行处理*

*#ITEM\_PIPELINES = {*

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

*#}*

### 二、明确目标(mySpider/items.py)

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

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

import scrapy

class ItcastItem(scrapy.Item):

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

*#姓名*

name = scrapy.Field()

*#职称*

title = scrapy.Field()

*#个人简介*

info = scrapy.Field()

### 三、制作爬虫 （spiders/itcastSpider.py）

**爬虫功能要分两步：**

#### 1. 爬数据

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

**注意爬虫名和项目名不能相同**

scrapy genspider itcast "itcast.cn"

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

import scrapy

class ItcastSpider(scrapy.Spider):

name = "itcast"

allowed\_domains = ["itcast.cn"]

start\_urls = (

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

)

def parse(self, response):

pass

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

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

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

**将start\_urls的值修改为需要爬取的第一个url**

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

**修改parse()方法**

def parse(self, response):

filename = "teacher.html"

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

##### itcast各科教师爬虫

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

**import** scrapy

*#创建一个爬虫类*

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

*#爬虫名, 在执行时就是使用的这个名字*

name = **"itcast"**

*#允许爬虫爬取的范围, 可以写多个值*

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

*#爬虫起始的url地址, 可以使用元组或列表.*

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

*#定义一个parse方法, 名字不能修改*

*#处理返回的response*

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

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

*#scrapy中的用法, 不能使用response, 要写body*

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

然后运行一下看看, 在mySpider目录下执行：

scrapy crawl itcast

运行出错：

ImportError: No module named 'win32api'

使用pip安装win32

pip install pypiwin32

再次运行, 就能成功了.

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

一个Scrapy爬虫项目里, 可以存在多个爬虫. 各个爬虫在执行时, 就是按照 name 属性来区分.

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

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

# 我们可以指定保存内容的编码格式, 一般情况下, 我们可以在代码最上方添加：

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

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

#### 2. 取数据

* 爬取整个网页完毕, 接下来的就是的取过程了, 首先观察页面源码：

<div class="li\_txt">

<h3> xxx </h3>

<h4> xxxxx </h4>

<p> xxxxxxxx </p>

**是不是一目了然？直接上XPath开始提取数据吧.**

父节点：

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

子节点：

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

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

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

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

from mySpider.items import ItcastItem

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

from mySpider.items import ItcastItem

def parse(self, response):

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

# 存放老师信息的集合

items = []

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

# 将我们得到的数据封装到一个 `ItcastItem` 对象

item = ItcastItem()

#extract()方法返回的都是unicode字符串

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

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

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

#xpath返回的是包含一个元素的列表

item['name'] = name[0]

item['title'] = title[0]

item['info'] = info[0]

items.append(item)

# 直接返回最后数据

return items

##### itcast各科教师爬虫取数据

import scrapy

from mySpider.items import ItcastItem

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

*#创建一个爬虫类*

class ItcastSpider(scrapy.Spider):

*#爬虫名, 在执行时就是使用的这人名字*

name = "itcast"

*#允许爬虫爬取的范围, 可以写多个值*

allowed\_domains = ["itcast.cn"]

*#爬虫起始的url地址, 可以使用元组或列表.*

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

start\_urls = [

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

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

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

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

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

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

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

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

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

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

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

]

*#定义一个parse方法, 名字不能修改*

*#处理返回的response*

*# def parse(self, response):*

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

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

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

def parse(self, response):

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

*# 存放老师信息的集合*

teacherItems = []

*#通过scrapy自带的xpath匹配出所有老师的根节点*

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

*#遍历根节点集合*

for each in teacher\_list:

*# 将我们得到的数据封装到一个 `ItcastItem` 对象*

*#实例化一个对象, 用来保存数据*

item = ItcastItem()

*#extract()方法将匹配出来的结果转换成unicode字符串*

*#不加extract()结果为xpath匹配出来的对象*

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

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

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

*#xpath返回的是包含所有匹配对象的的列表*

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

item['name'] = name[0]

item['title'] = title[0]

item['info'] = info[0]

*#*

teacherItems.append(item)

*# 直接返回最后数据*

return teacherItems

* 我们暂时先不处理管道, 后面会详细介绍.

#### 3. 保存数据

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

# json格式, 默认为Unicode编码

scrapy crawl itcast -o teachers.json

# json lines格式, 默认为Unicode编码

scrapy crawl itcast -o teachers.jsonl

# csv 逗号表达式, 可用Excel打开

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

scrapy crawl itcast -o teachers.csv

# xml格式

scrapy crawl itcast -o teachers.xml

#### itcast各科教师爬虫yield

如果将代码改成下面形式, 结果完全一样.

请思考 yield 在这里的作用：

from mySpider.items import ItcastItem

def parse(self, response):

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

# 存放老师信息的集合

#items = []

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

# 将我们得到的数据封装到一个 `ItcastItem` 对象

item = ItcastItem()

#extract()方法返回的都是unicode字符串

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

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

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

#xpath返回的是包含一个元素的列表

item['name'] = name[0]

item['title'] = title[0]

item['info'] = info[0]

#items.append(item)

#将获取的数据交给pipelines

yield item

# 返回数据, 不经过pipeline

#return items

#### IPython 6.0与yield

pip install ipython **--**upgrade

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

print(b)

fib(5)

1

2

3

5

8

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

yield(b)

m=fib(5)

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

Out[12]: 1

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

Out[13]: 2

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

Out[14]: 3

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

Out[15]: 5

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

Out[16]: 8

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

def fib(n):

a,b,s=0,1,0

while s<n:

a,b=b,a+b

s=s+1

yield(b)

for i in fib(5):

print(i)

1

2

3

5

8

## Scrapy Shell

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

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

### 启动Scrapy Shell

进入项目的根目录, 执行下列命令来启动shell:

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

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

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

### Selectors选择器

Scrapy Selectors 内置 XPath 和 CSS Selector 表达式机制

Selector有四个基本的方法, 最常用的还是xpath:

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

#### XPath表达式的例子及对应的含义:

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

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

//td: 选择所有的 <td> 元素

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

#### 尝试Selector

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

# 启动

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

# 返回 xpath选择器对象列表

response.xpath('//title')

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

# 使用 extract()方法返回 Unicode字符串列表

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

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

# 打印列表第一个元素, 终端编码格式显示

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

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

# 返回 xpath选择器对象列表

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

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

# 返回列表第一个元素的Unicode字符串

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

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

# 按终端编码格式显示

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

职位搜索 | 社会招聘 | Tencent 腾讯招聘

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

职位名称:

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

TEG15-运营开发工程师（深圳）

职位名称详情页:

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

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

职位类别:

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

技术类

新版代码

# 启动

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

# 返回 xpath选择器对象列表

response.xpath('//title')

[<Selector xpath='//title' data='<title>职位搜索 | 社会招聘 | Tencent 腾讯招聘</title'>]

# 使用 extract()方法返回字符串列表

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

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

# 打印列表第一个元素,没有则返回None

print response.xpath('//title').extract\_first()

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

#contains的用法, or的用法, last()的含义

In [6]: response.xpath('//\*[contains(@class,"odd") or contains(@class,"even")]/td[last()]/text()').extract()

Out[6]:

['2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02',

'2017-06-02']

In [4]: response.xpath('//a[contains(@href,"position\_detail.php?")]/text()').extract()

Out[4]:

['19407-移动游戏平台合作（上海）',

'19407-手游商业化与本地化策划（上海）',

'OMG236-腾讯视频平台高级产品经理（深圳）',

'OMG096-科技频道记者（北京）',

'18402-项目管理',

'IEG-招聘经理（深圳）',

'OMG097-视觉设计师（北京）',

'OMG097-策略产品经理/产品运营（北京）',

'OMG097-策略产品经理/产品运营（北京）',

'OMG097-数据产品经理(北京）']

In [5]: response.xpath('//\*[contains(@class,"odd") or contains(@class,"even")]/td[last()-1]/text()').extract()

Out[5]: ['上海', '上海', '深圳', '北京', '深圳', '深圳', '北京', '北京', '北京', '北京']

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

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

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

## Item Pipeline

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

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

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

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

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

### 编写item pipeline

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

import something

class SomethingPipeline(object):

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

# 可选实现, 做参数初始化等

# doing something

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

# item (Item 对象) – 被爬取的item

# spider (Spider 对象) – 爬取该item的spider

# 这个方法必须实现, 每个item pipeline组件都需要调用该方法,

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

return item

def open\_spider(self, spider):

# spider (Spider 对象) – 被开启的spider

# 可选实现, 当spider被开启时, 这个方法被调用.

def close\_spider(self, spider):

# spider (Spider 对象) – 被关闭的spider

# 可选实现, 当spider被关闭时, 这个方法被调用

### 使用pipeline的itcast爬虫

#### ItcastSpider文件

**import** scrapy

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

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

*#创建一个爬虫类*

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

*#爬虫名, 在执行时就是使用的这人名字*

name = **"itcast"**

*#允许爬虫爬取的范围, 可以写多个值*

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

*#爬虫起始的url地址, 可以使用元组或列表.*

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

start\_urls = [

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

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

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

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

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

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

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

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

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

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

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

]

*#定义一个parse方法, 名字不能修改*

*#处理返回的response*

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

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

*# 存放老师信息的集合*

*# teacherItems = []*

*#通过scrapy自带的xpath匹配出所有老师的根节点*

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

*#遍历根节点集合*

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

*# 将我们得到的数据封装到一个 `ItcastItem` 对象*

*#实例化一个对象, 用来保存数据*

item = ItcastItem()

*#extract()方法将匹配出来的结果转换成unicode字符串*

*#不加extract()结果为xpath匹配对象*

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

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

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

*#xpath返回的是包含一个元素的列表*

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

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

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

*#*

*# teacherItems.append(item)*

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

*# 直接返回最后数据*

*# return teacherItems*

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

**yield** item

#### pipeline文件

**item写入JSON文件**

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

# pipelines.py

import json

class ItcastJsonPipeline(object):

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

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

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

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

self.file.write(content)

return item

def close\_spider(self, spider):

self.file.close()

视频演示

import json

class ItcastPipeline(object):

*# \_\_init\_\_方法是可选的, 作为类的初始化方法*

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

*# 创建了一个文件*

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

*# process\_item方法是必须写的, 用来处理item数据*

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

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

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

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

return item

*# close\_spider方法是可选的, 结束时调用这个方法*

*初始化方法和结束 方法可以都写在process\_item方法中*

def close\_spider(self, spider):

self.filename.close()

#### 启用一个Item Pipeline组件

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

BOT\_NAME = 'mySpider'

*#爬虫存放的位置*

SPIDER\_MODULES = ['mySpider.spiders']

NEWSPIDER\_MODULE = 'mySpider.spiders'

*#Override the default request headers:*

DEFAULT\_REQUEST\_HEADERS = {

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

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

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

}

#下载延迟, 一般不启动

DOWNLOAD\_DELAY = 2

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

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

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

*# Disable cookies (enabled by default)*

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

COOKIES\_ENABLED = False

# Configure item pipelines

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

ITEM\_PIPELINES = {

#'mySpider.pipelines.SomePipeline': 300,

"mySpider.pipelines.ItcastJsonPipeline":300

}

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

#### 重新启动爬虫

将parse()方法改为4.2中最后思考中的代码, 然后执行下面的命令：

scrapy crawl itcast

查看当前目录是否生成teacher.json

或者在项目根目录下创建start.py

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

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

执行这个文件即可

## Spiders类

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

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

主要用到的函数及调用顺序为：

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

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

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

### 源码参考

#所有爬虫的基类, 用户定义的爬虫必须从这个类继承

class Spider(object\_ref):

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

#name是spider最重要的属性, 而且是必须的.

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

name = None

#初始化, 提取爬虫名字, start\_ruls

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

if name is not None:

self.name = name

# 如果爬虫没有名字, 中断后续操作则报错

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

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

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

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

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

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

self.start\_urls = []

# 打印Scrapy执行后的log信息

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

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

# 判断对象object的属性是否存在, 不存在做断言处理

def set\_crawler(self, crawler):

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

self.\_crawler = crawler

@property

def crawler(self):

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

return self.\_crawler

@property

def settings(self):

return self.crawler.settings

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

#该方法仅调用一次

def start\_requests(self):

for url in self.start\_urls:

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

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

#Request对象默认的回调函数为parse(), 提交的方式为get

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

return Request(url, dont\_filter=True)

#默认的Request对象回调函数, 处理返回的response.

#生成Item或者Request对象. 用户必须实现这个类

def parse(self, response):

raise NotImplementedError

@classmethod

def handles\_request(cls, request):

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

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

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

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

### 主要属性和方法

* name

定义spider名字的字符串.

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

* allowed\_domains

*可选参数, 包含了spider允许爬取的域名(domain)的列表.*

* start\_urls

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

* start\_requests(self)

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

当spider启动爬取并且未指定start\_urls时, 该方法被调用.

* parse(self, response)

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

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

使用 scrapy.log.msg() 方法记录(log)message. 更多数据请参见 [logging](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\python%E5%AD%A6%E4%B9%A0%E8%B7%AF%E7%BA%BF%E5%9B%BE\05_%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2python%E5%B0%B1%E4%B8%9A%E7%8F%AD%E8%A7%86%E9%A2%91\07%20%E7%88%AC%E8%99%AB\%E8%B5%84%E6%96%99\file\part04\4.7.html)

### 案例：腾讯招聘网自动翻页采集

#### scrapy项目步骤

1、创建项目

scrapy startproject xxx

2、编写item文件

设置需要保存的数据字段

3、进入xxx/spiders

编写爬虫文件, 文件里的name就是爬虫名

4、运行

scrapy crawl itcast

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

#### 使用scrapy shell来进行测试

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

print(response.body)

response.xpath("//title")

返回一个包含匹配选择规则的选择器的列表

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

转换为unicode字符串

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

取出来选择器对象

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

以终端默认的编码格式打印出来

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

text()取出来标签中的内容

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

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

list = list\_1 + list\_2

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

#### 创建一个新的爬虫：

scrapy genspider tencent "tencent.com"

#### 编写items.py

获取职位名称、详细信息、

**import** scrapy

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

*#职位名称*

position\_name = scrapy.Field()

*#详情链接*

position\_link = scrapy.Field()

*#职位类别*

position\_type = scrapy.Field()

*#招聘人数*

people\_num = scrapy.Field()

*#工作地点*

work\_location = scrapy.Field()

*#发布时间*

publish\_time = scrapy.Field()

新版代码

class TencentItem(scrapy.Item):

name = scrapy.Field()

detailLink = scrapy.Field()

positionInfo = scrapy.Field()

peopleNumber = scrapy.Field()

workLocation = scrapy.Field()

publishTime = scrapy.Field()

#### 编写tencent.py

# tencent.py

from mySpider.items import TencentItem

import scrapy

import re

class TencentSpider(scrapy.Spider):

name = "tencent"

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

start\_urls = [

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

]

def parse(self, response):

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

item = TencentItem()

name = each.xpath('./td[1]/a/text()').extract()[0]

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

positionInfo = each.xpath('./td[2]/text()').extract()[0]

peopleNumber = each.xpath('./td[3]/text()').extract()[0]

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

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

#print name, detailLink, catalog, peopleNumber, workLocation,publishTime

item['name'] = name.encode('utf-8')

item['detailLink'] = detailLink.encode('utf-8')

item['positionInfo'] = positionInfo.encode('utf-8')

item['peopleNumber'] = peopleNumber.encode('utf-8')

item['workLocation'] = workLocation.encode('utf-8')

item['publishTime'] = publishTime.encode('utf-8')

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

page = int(curpage) + 10

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

# 发送新的url请求加入待爬队列, 并调用回调函数 self.parse

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

# 将获取的数据交给pipeline

yield item

#### 视频演示: 编写tencentZhaopin.py

**方法一：使用自增的方法处理分页**

**import** scrapy

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

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

name = **"tencentZhaopin"**

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

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

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

offset = 0

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

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

*# 网页中有两种不同的样式*

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

*#初始化模型对象, 类似于python中的字典*

item = TencentspiderItem

*# 职位名称, 直接保存到item的对象中了*

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

*# 详情连接*

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

*# 职位类别*

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

*# 招聘人数*

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

*# 工作地点*

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

*# 发布时间*

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

*#如果得到的是数据, 将数据交给管道文件处理*

**yield** item

*#当一页中所有的数据都处理完成后, 接着处理下一页中的数据*

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

**if** self.offset < 2111:

self.offset += 10

*# else:*

*# break*

*# raise "结束工作"*

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

*# callback回调函数*

*#每次处理完一页的数据之后, 重新发送下一页的请求*

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

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

**方法二：使用re匹配的方法处理分页**

**import** scrapy, re

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

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

name = **"tencentZhaopin"**

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

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

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

offset = 0

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

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

*# 网页中有两种不同的样式*

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

*#初始化模型对象, 类似于python中的字典*

item = TencentspiderItem

*# 职位名称, 直接保存到item的对象中了*

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

*# 详情连接*

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

*# 职位类别*

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

*# 招聘人数*

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

*# 工作地点*

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

*# 发布时间*

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

*#如果得到的是数据, 将数据交给管道文件处理*

**yield** item

*# 从任意地方开始查找匹配一次, 匹配url.*

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

page = int(curpage) + 10

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

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

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

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

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

新版代码

# tencent.py

from mySpider.items import TencentItem

import scrapy

import re

class TencentSpider(scrapy.Spider):

name = "tencent"

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

start\_urls = [

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

]

def parse(self, response):

items = response.xpath('//\*[contains(@class,"odd") or contains(@class,"even")]')

for item in items:

temp = dict(

name=item.xpath("./td[1]/a/text()").extract()[0],

detailLink="http://hr.tencent.com/"+item.xpath("./td[1]/a/@href").extract()[0],

positionInfo=item.xpath('./td[2]/text()').extract()[0] if len(item.xpath('./td[2]/text()').extract())>0 else None,

peopleNumber=item.xpath('./td[3]/text()').extract()[0],

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

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

)

yield temp

now\_page = int(re.search(r"\d+", response.url).group(0))

print("\*" \* 100)

if now\_page < 216:

url = re.sub(r"\d+", str(now\_page + 10), response.url)

print("this is next page url:", url)

print("\*" \* 100)

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

#### 编写pipeline.py文件

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

import json

#class ItcastJsonPipeline(object):

class TencentJsonPipeline(object):

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

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

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

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

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

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

self.file.write(content)

return item

def close\_spider(self, spider):

self.file.close()

#### middleware.py

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

**class** TencentspiderSpiderMiddleware(object):

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

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

*# passed objects.*

@classmethod

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

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

s = cls()

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

**return** s

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

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

*# middleware and into the spider.*

*# Should return None or raise an exception.*

**return None**

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

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

*# it has processed the response.*

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

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

**yield** i

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

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

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

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

*# or Item objects.*

**pass**

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

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

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

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

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

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

**yield** r

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

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

#### 在 setting.py 里设置ITEM\_PIPELINES

ITEM\_PIPELINES = {

#'mySpider.pipelines.SomePipeline': 300,

#"mySpider.pipelines.ItcastJsonPipeline":300

"mySpider.pipelines.TencentJsonPipeline":300

}

### 在setting中设置请求头

DEFAULT\_REQUEST\_HEADERS = {

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

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

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

}

### 执行爬虫：

scrapy crawl tencent

### 请思考 parse()方法的工作机制：

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

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

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

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

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

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

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

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

7. 这一切的一切, Scrapy引擎和调度器将负责到底.

### 常见bug

[scrapy.spidermiddlewares.offsite] DEBUG: Filtered offsite request to 'hr.tencent.com':

解决方式：

domain错误 修改domain为：hr.tencent.com

## CrawlSpiders

通过下面的命令可以快速创建 CrawlSpider模板 的代码：

scrapy genspider -t crawl tencent tencent.com

**-t指定一个模板**

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

class scrapy.spiders.CrawlSpider

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

### 源码参考

class CrawlSpider(Spider):

rules = ()

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

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

self.\_compile\_rules()

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

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

#设置了跟进标志位True

#parse将返回item和跟进了的Request对象

def parse(self, response):

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

#处理start\_url中返回的response, 需要重写

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

return []

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

return results

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

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

if not isinstance(response, HtmlResponse):

return

seen = set()

#抽取之内的所有链接, 只要通过任意一个'规则', 即表示合法

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

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

#使用用户指定的process\_links处理每个连接

if links and rule.process\_links:

links = rule.process\_links(links)

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

for link in links:

seen.add(link)

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

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

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

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

yield rule.process\_request(r)

#处理通过rule提取出的连接, 并返回item以及request

def \_response\_downloaded(self, response):

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

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

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

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

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

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

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

if callback:

#如果是parse调用的, 则会解析成Request对象

#如果是rule callback, 则会解析成Item

cb\_res = callback(response, \*\*cb\_kwargs) or ()

cb\_res = self.process\_results(response, cb\_res)

for requests\_or\_item in iterate\_spider\_output(cb\_res):

yield requests\_or\_item

#如果需要跟进, 那么使用定义的Rule规则提取并返回这些Request对象

if follow and self.\_follow\_links:

#返回每个Request对象

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

yield request\_or\_item

def \_compile\_rules(self):

def get\_method(method):

if callable(method):

return method

elif isinstance(method, basestring):

return getattr(self, method, None)

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

for rule in self.\_rules:

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

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

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

def set\_crawler(self, crawler):

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

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

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

### rules

CrawlSpider使用rules来决定爬虫的爬取规则, 并将匹配后的url请求提交给引擎. 所以在正常情况下, CrawlSpider不需要单独手动返回请求了.

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

class scrapy.spiders.Rule(

link\_extractor,

callback = None,

cb\_kwargs = None,

follow = None,

process\_links = None,

process\_request = None

)

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

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

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

### LinkExtractors

class scrapy.linkextractors.LinkExtractor

Link Extractors 的目的很简单: 提取链接｡

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

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

class scrapy.linkextractors.LinkExtractor(

allow = (),

deny = (),

allow\_domains = (),

deny\_domains = (),

deny\_extensions = None,

restrict\_xpaths = (),

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

attrs = ('href'),

canonicalize = True,

unique = True,

process\_value = None

)

主要参数：

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

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

继续用腾讯招聘为例, 给出配合rule使用CrawlSpider的例子:

1. 首先运行

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

获取响应内容, 打印内容的时候用body

response.body

获取unicode格式的响应内容, 解析的时候用text

response.text

1. 打开网页, 查看页码导航的url地址, 是类似于如下的形式：

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

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

from scrapy.linkextractors import LinkExtractor

# LinkExtractor是匹配整个html文档中的链接

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

创建一个正则匹配规则对象.

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

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

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

使用正则匹配对象从响应文件中取出来所有匹配的links.

page\_lx.extract\_links(response)

1. 没有查到：

[]

1. 注意转义字符的问题, 继续重新匹配：

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

# 或者

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

# 或者

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

page\_lx.extract\_links(response)

### tencent爬虫的CrawlSpider 版本

那么, scrapy shell测试完成之后, 修改以下代码

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

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

rules = [

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

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

]

**这么写对吗？**

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

**scrapy startproject TencentSpider**

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

#### item文件

**import** scrapy

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

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

*# 职位名*

positionname = scrapy.Field()

*# 详情连接*

positionlink = scrapy.Field()

*# 职位类别*

positionType = scrapy.Field()

*# 招聘人数*

peopleNum = scrapy.Field()

*# 工作地点*

workLocation = scrapy.Field()

*# 发布时间*

publishTime = scrapy.Field()

#### Spider

#tencent.py

import scrapy

from scrapy.spiders import CrawlSpider, Rule

from scrapy.linkextractors import LinkExtractor

from mySpider.items import TencentItem

class TencentSpider(CrawlSpider):

name = "tencent"

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

start\_urls = [

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

]

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

rules = [

Rule(page\_lx, callback = "parseContent", follow = True)

]

def parseContent(self, response):

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

name = each.xpath('./td[1]/a/text()').extract()[0]

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

positionInfo = each.xpath('./td[2]/text()').extract()[0]

peopleNumber = each.xpath('./td[3]/text()').extract()[0]

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

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

#print name, detailLink, catalog,recruitNumber,workLocation,publishTime

item = TencentItem()

item['name']=name.encode('utf-8')

item['detailLink']=detailLink.encode('utf-8')

item['positionInfo']=positionInfo.encode('utf-8')

item['peopleNumber']=peopleNumber.encode('utf-8')

item['workLocation']=workLocation.encode('utf-8')

item['publishTime']=publishTime.encode('utf-8')

yield item

# parse() 方法不需要写

# def parse(self, response):

# pass

运行： scrapy crawl tencent

#tencent.py

**import** scrapy

*# 导入CrawlSpider类和Rule*

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

*# 导入链接规则匹配类, 用来提取符合规则的连接*

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

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

**class** TencentSpider(CrawlSpider):

name = **"tencent"**

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

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

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

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

*#职位的详情页匹配规则*

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

rules = [

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

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

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

*#如果有多个匹配规则, 在列表中继续写*

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

]

*# 指定的回调函数, 不需要写parse()方法*

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

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

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

*#fulllist = evenlist + oddlist*

*#for each in fulllist:*

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

item = TencentItem()

*# 职位名称*

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

*# 详情连接*

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

*# 职位类别*

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

*# 招聘人数*

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

*# 工作地点*

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

*# 发布时间*

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

**yield** item

新版代码

import scrapy

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

class TecentSpider(CrawlSpider):

name = 'tecent'

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

start\_urls = ['http://hr.tencent.com/position.php?&start=0']

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

#position.php?&start=10#a

rules = (

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

)

def parse\_item(self, response):

items = response.xpath('//\*[contains(@class,"odd") or contains(@class,"even")]')

for item in items:

temp = dict(

position=item.xpath("./td[1]/a/text()").extract()[0],

detailLink="http://hr.tencent.com/" + item.xpath("./td[1]/a/@href").extract()[0],

type=item.xpath('./td[2]/text()').extract()[0] if len(

item.xpath('./td[2]/text()').extract()) > 0 else None,

need\_num=item.xpath('./td[3]/text()').extract()[0],

location=item.xpath('./td[4]/text()').extract()[0],

publish\_time=item.xpath('./td[5]/text()').extract()[0]

)

print(temp)

yield temp

# parse() 方法不需要重写

# def parse(self, response):

# pass

#### pipeline文件

**import** json

**class** TencentPipeline(object):

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

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

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

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

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

**return** item

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

self.filename.close()

#### settings文件

BOT\_NAME = **'TencentSpider'**

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

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

*# 保存日志信息的文件名*

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

*# 保存日志等级, 低于|等于此等级的信息都被保存*

LOG\_LEVEL = **"DEBUG"**

ITEM\_PIPELINES = {

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

}

#### 运行：

scrapy crawl tencent

### Logging日志文件

Scrapy提供了log功能, 可以通过 logging 模块使用.

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

LOG\_FILE = "TencentSpider.log"

LOG\_LEVEL = "INFO"

#### Log levels

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

#### logging设置

通过在setting.py中进行以下设置可以被用来配置logging:

1. LOG\_ENABLED 默认: True, 启用logging
2. LOG\_ENCODING 默认: 'utf-8', logging使用的编码
3. LOG\_FILE 默认: None, 在当前目录里创建logging输出文件的文件名
4. LOG\_LEVEL 默认: 'DEBUG', log的最低级别
5. LOG\_STDOUT 默认: False 如果为 True, 进程所有的标准输出(及错误)将会被重定向到log中. 例如, 执行 print "hello" , 其将会在Scrapy log中显示.

## Request/Response与模拟登陆

### Request

Request 部分源码：

# 部分代码

class Request(object\_ref):

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

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

dont\_filter=False, errback=None):

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

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

self.\_set\_url(url)

self.\_set\_body(body)

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

self.priority = priority

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

self.callback = callback

self.errback = errback

self.cookies = cookies or {}

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

self.dont\_filter = dont\_filter

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

@property

def meta(self):

if self.\_meta is None:

self.\_meta = {}

return self.\_meta

其中, 比较常用的参数：

url: 就是需要请求, 并进行下一步处理的url

callback: 指定该请求返回的Response, 由那个函数来处理.

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

headers: 请求时, 包含的头文件. 一般不需要. 内容一般如下：

# 自己写过爬虫的肯定知道

Host: media.readthedocs.org

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

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

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

Accept-Encoding: gzip, deflate

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

Cookie: \_ga=GA1.2.1612165614.1415584110;

Connection: keep-alive

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

Cache-Control: max-age=0

meta: 比较常用, 在不同的请求之间传递数据使用的. 字典dict型

request\_with\_cookies = Request(

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

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

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

)

encoding: 使用默认的 'utf-8' 就行.

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

errback: 指定错误处理函数

### Response

# 部分代码

class Response(object\_ref):

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

self.headers = Headers(headers or {})

self.status = int(status)

self.\_set\_body(body)

self.\_set\_url(url)

self.request = request

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

@property

def meta(self):

try:

return self.request.meta

except AttributeError:

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

"is not tied to any request")

大部分参数和上面的差不多：

status: 响应码

\_set\_body(body)： 响应体

\_set\_url(url)：响应url

self.request = request

#### 发送POST请求

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

class mySpider(scrapy.Spider):

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

def start\_requests(self):

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

# FormRequest 是Scrapy发送POST请求的方法

yield scrapy.FormRequest(

url = url,

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

callback = self.parse\_page

)

def parse\_page(self, response):

# do something

#### 模拟登陆

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

<http://docs.pythontab.com/scrapy/scrapy0.24/topics/request-response.html#topics-request-response-ref-request-userlogin>

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

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

下面是使用这种方法的爬虫例子:

import scrapy

class LoginSpider(scrapy.Spider):

name = 'example.com'

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

def parse(self, response):

return scrapy.FormRequest.from\_response(

response,

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

callback=self.after\_login

)

def after\_login(self, response):

# check login succeed before going on

if "authentication failed" in response.body:

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

return

# continue scraping with authenticated session...

#### 知乎爬虫案例参考：

##### zhihuSpider.py爬虫代码

#!/usr/bin/env python

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

from scrapy.spiders import CrawlSpider, Rule

from scrapy.selector import Selector

from scrapy.linkextractors import LinkExtractor

from scrapy import Request, FormRequest

from zhihu.items import ZhihuItem

class ZhihuSipder(CrawlSpider) :

name = "zhihu"

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

start\_urls = [

"http://www.zhihu.com"

]

rules = (

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

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

)

headers = {

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

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

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

"Connection": "keep-alive",

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

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

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2125.111 Safari/537.36",

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

}

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

def start\_requests(self):

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

def post\_login(self, response):

print 'Preparing login'

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

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

print xsrf

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

#登陆成功后, 会调用after\_login回调函数

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

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

headers = self.headers, #注意此处的headers

formdata = {

'\_xsrf': xsrf,

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

'password': '123456'

},

callback = self.after\_login,

dont\_filter = True

)]

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

def after\_login(self, response) :

for url in self.start\_urls :

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

def parse\_page(self, response):

problem = Selector(response)

item = ZhihuItem()

item['url'] = response.url

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

print item['name']

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

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

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

return item

##### Item类设置

from scrapy.item import Item, Field

class ZhihuItem(Item):

# define the fields for your item here like:

# name = scrapy.Field()

url = Field() #保存抓取问题的url

title = Field() #抓取问题的标题

description = Field() #抓取问题的描述

answer = Field() #抓取问题的答案

name = Field() #个人用户的名称

##### setting.py 设置抓取间隔

BOT\_NAME = 'zhihu'

SPIDER\_MODULES = ['zhihu.spiders']

NEWSPIDER\_MODULE = 'zhihu.spiders'

DOWNLOAD\_DELAY = 0.25 #设置下载间隔为250ms

## Downloader Middlewares

### 反反爬虫相关机制

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

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

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

### 通常防止爬虫被反主要有以下几个策略：

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

DOWNLOADER\_MIDDLEWARES = {

'scrapy\_crawlera.CrawleraMiddleware': 600

}

CRAWLERA\_ENABLED = True

CRAWLERA\_USER = '注册/购买的UserKey'

CRAWLERA\_PASS = '注册/购买的Password'

### 设置下载中间件（Downloader Middlewares）

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

下载中间件一般写在settings同级目录下.

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

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

这里是一个例子:

DOWNLOADER\_MIDDLEWARES = {

'mySpider.middlewares.MyDownloaderMiddleware': 543,

}

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

class scrapy.contrib.downloadermiddleware.DownloaderMiddleware

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

* 当每个request通过下载中间件时, 该方法被调用.
* 此方法必须要写
* process\_request() 必须返回以下其中之一：一个 None 、一个 Response 对象、一个 Request 对象或 raise IgnoreRequest:
  + 如果其返回 None , Scrapy将继续处理该request, 执行其他的中间件的相应方法, 直到合适的下载器处理函数(download handler)被调用, 该request被执行(其response被下载).
  + 如果其返回 Response对象, Scrapy将不会调用任何其他的process\_request() 或 process\_exception() 方法, 或相应地下载函数； 其将返回该response. 已安装的中间件的 process\_response() 方法则会在每个response返回时被调用.
  + 如果其返回 Request 对象, Scrapy则停止调用 process\_request方法并重新调度返回的request. 当新返回的request被执行后, 相应地中间件链将会根据下载的response被调用.
  + 如果其raise一个 IgnoreRequest 异常, 则下载中间件的 process\_exception() 方法会被调用. 如果没有任何一个方法处理该异常, 则request的errback(Request.errback)方法会被调用. 如果没有代码处理抛出的异常, 则该异常被忽略且不记录(不同于其他异常那样).
* 参数:
  + request (Request 对象) – 处理的request
  + spider (Spider 对象) – 该request对应的spider

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

当下载器完成http请求, 传递响应给引擎的时候调用

* process\_request() 必须返回以下其中之一: 返回一个 Response 对象、 返回一个 Request 对象或raise一个 IgnoreRequest 异常.
  + 如果其返回一个 Response (可以与传入的response相同, 也可以是全新的对象), 该response会被在链中的其他中间件的 process\_response() 方法处理.
  + 如果其返回一个 Request 对象, 则中间件链停止, 返回的request会被重新调度下载. 处理类似于 process\_request() 返回request所做的那样.
  + 如果其抛出一个 IgnoreRequest 异常, 则调用request的errback(Request.errback). 如果没有代码处理抛出的异常, 则该异常被忽略且不记录(不同于其他异常那样).
* 参数:
  + request (Request 对象) – response所对应的request
  + response (Response 对象) – 被处理的response
  + spider (Spider 对象) – response所对应的spider

### 使用案例给豆瓣电影爬虫添加下载中间件：

#### 1. 创建middlewares.py文件.

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

# middlewares.py

#!/usr/bin/env python

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

import random

import base64

from settings import USER\_AGENTS

from settings import PROXIES

# 随机的User-Agent

class RandomUserAgent(object):

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

useragent = random.choice(USER\_AGENTS)

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

class RandomProxy(object):

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

proxy = random.choice(PROXIES)

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

# 没有代理账户验证的代理使用方式

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

else:

# 对账户密码进行base64编码转换

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

# 对应到代理服务器的信令格式里

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

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

视频演示

# middlewares.py

#!/usr/bin/env python

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

**import** random

**import** base64

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

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

*# 随机的User-Agent*

**class** RandomUserAgent(object):

*#此方法是必须要调用的, 且是固定的, 不能修改名称*

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

useragent = random.choice(USER\_AGENTS)

*#print(useragent)*

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

**class** RandomProxy(object):

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

proxy = random.choice(PROXIES)

*# print(proxy)*

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

*# 没有代理账户验证的代理使用方式*

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

**else**:

*# 对账户密码进行base64编码转换*

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

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

*# 对应到代理服务器的信令格式里*

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

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

为什么HTTP代理要使用base64编码：

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

CONNECT 59.64.128.198:21 HTTP/1.1

Host: 59.64.128.198:21

Proxy-Authorization: Basic bGV2I1TU5OTIz

User-Agent: OpenFetion

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

HTTP/1.0 200 Connection established

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

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

* 添加USER\_AGENTS：

　　USER\_AGENTS = [

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

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

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

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

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

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

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

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

]

* 添加代理IP设置PROXIES：

免费代理IP可以网上搜索, 或者付费购买一批可用的私密代理IP：

PROXIES = [

*#有用户名和密码验证的代理*

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

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

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

*#没有用户名和密码验证的代理*

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

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

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

]

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

COOKIES\_ENABLED = False

* 设置下载延迟

DOWNLOAD\_DELAY = 3

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

DOWNLOADER\_MIDDLEWARES = {

#'mySpider.middlewares.MyCustomDownloaderMiddleware': 543,

'mySpider.middlewares.RandomUserAgent': 1,

'mySpider.middlewares.ProxyMiddleware': 100

}

DOWNLOADER\_MIDDLEWARES = {

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

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

}

## Settings

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

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

### 内置设置参考手册

#### BOT\_NAME

* + 默认: 'scrapybot'
  + 当您使用 startproject 命令创建项目时其也被自动赋值.

#### CONCURRENT\_ITEMS

* + 默认: 100
  + Item Processor(即 Item Pipeline) 同时处理(每个response的)item的最大值.

#### CONCURRENT\_REQUESTS

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

#### DEFAULT\_REQUEST\_HEADERS

* + 默认: 如下

{

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

'Accept-Language': 'en',

}

Scrapy HTTP Request使用的默认header.

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

#### DEPTH\_LIMIT

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

#### DOWNLOAD\_DELAY

* + 默认: 0
  + 下载器在下载同一个网站下一个页面前需要等待的时间. 该选项可以用来限制爬取速度, 减轻服务器压力. 同时也支持小数:
  + DOWNLOAD\_DELAY = 0.25 # 250 ms of delay
  + 默认情况下, Scrapy在两个请求间不等待一个固定的值, 而是使用0.5到1.5之间的一个随机值 \* DOWNLOAD\_DELAY 的结果作为等待间隔.

#### DOWNLOAD\_TIMEOUT

* + 默认: 180
  + 下载器超时时间(单位: 秒).

#### ITEM\_PIPELINES

* + 默认: {}
  + 保存项目中启用的pipeline及其顺序的字典. 该字典默认为空, 值(value)任意, 不过值(value)习惯设置在0-1000范围内, 值越小优先级越高.

ITEM\_PIPELINES = {

'mySpider.pipelines.SomethingPipeline': 300,

'mySpider.pipelines.ItcastJsonPipeline': 800,

}

#### LOG\_ENABLED

* + 默认: True
  + 是否启用logging.

#### LOG\_ENCODING

* + 默认: 'utf-8'
  + logging使用的编码.

#### LOG\_LEVEL

* + 默认: 'DEBUG'
  + log的最低级别. 可选的级别有: CRITICAL、 ERROR、WARNING、INFO、DEBUG . 调试的时候一般写debug信息. 后期运维的话一般设置为info信息.

#### USER\_AGENT

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

#### PROXIES： 代理设置

* + 示例：

PROXIES = [

{'ip\_port': '111.11.228.75:80', 'password': ''},

{'ip\_port': '120.198.243.22:80', 'password': ''},

{'ip\_port': '111.8.60.9:8123', 'password': ''},

{'ip\_port': '101.71.27.120:80', 'password': ''},

{'ip\_port': '122.96.59.104:80', 'password': ''},

{'ip\_port': '122.224.249.122:8088', 'password':''},

]

#### COOKIES\_ENABLED = False

* + 禁用Cookies

# 5. Scrapy实战项目

## (案例一)手机App抓包爬虫

### 1. items.py

class DouyuspiderItem(scrapy.Item):

name = scrapy.Field()# 存储照片的名字

imagesUrls = scrapy.Field()# 照片的url路径

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

### 2. spiders/douyu.py

import scrapy

import json

from douyuSpider.items import DouyuspiderItem

class DouyuSpider(scrapy.Spider):

name = "douyu"

allowd\_domains = ["http://capi.douyucdn.cn"]

offset = 0

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

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

def parse(self, response):

# 返回从json里获取 data段数据集合

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

for each in data:

item = DouyuspiderItem()

item["name"] = each["nickname"]

item["imagesUrls"] = each["vertical\_src"]

yield item

self.offset += 20

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

### 3. 设置setting.py

ITEM\_PIPELINES = {'douyuSpider.pipelines.ImagesPipeline': 1}

# Images 的存放位置, 之后会在pipelines.py里调用

IMAGES\_STORE = "/Users/Power/lesson\_python/douyuSpider/Images"

# user-agent

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

### 4. pipelines.py

import scrapy

import os

from scrapy.pipelines.images import ImagesPipeline

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

class ImagesPipeline(ImagesPipeline):

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

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

image\_url = item["imagesUrls"]

yield scrapy.Request(image\_url)

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

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

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

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

item["imagesPath"] = self.IMAGES\_STORE + "/" + item["name"]

return item

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

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

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

### 执行程序

py2 main.py

### 视频演示

### 通过Fiddler进行手机抓包

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

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

设置手机wifi代理地址为电脑的地址, 端口号为8888

就能够抓取到手机app的流量请求了.

在其中找到发送数据对应的url地址

找到传输数据的js文件, 获得其url地址即可.

#### 用Fiddler对Android应用进行抓包

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

#### 用Fiddler对iPhone手机应用进行抓包

基本流程差不多, 只是手机设置不太一样：

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

代理地址(电脑IP)：192.168.xx.xxx

端口号：8888

### 模拟斗鱼手机客户端爬虫

"""

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

"""

#### 1. items.py

**import** scrapy

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

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

nickname = scrapy.Field()*# 存储照片的名字*

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

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

#### 2. spiders/douyu.py

**import** scrapy, json

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

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

name = **"douyumeinv"**

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

offset = 0

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

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

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

*# 返回从json里获取 data段数据集合*

*# 把json格式的数据转换为python格式, data段是列表*

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

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

item = DouyuItem()

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

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

**yield** item

self.offset += 20

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

#### 3. 设置setting.py

BOT\_NAME = **'douyu'**

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

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

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

*#在抓包软件中查看*

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

ITEM\_PIPELINES = {

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

}

*# Images 的存放位置, 之后会在pipelines.py里调用*

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

#### 4. pipelines.py

**import** scrapy, os

*#获取设置文件, 上面设置了图片的保存路径*

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

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

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

**class** ImagesPipeline(ImagesPipeline):

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

*# return item*

*# 获取settings文件里设置的变量值*

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

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

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

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

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

*#发送图片的请求*

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

*#通过此函数对获得的图片进行处理*

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

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

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

*#使用用户的昵称来重命名下载的图片*

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

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

**return** item

#### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

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

#### 执行程序

py2 main.py

## (案例二)阳光热线问政平台爬虫

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

爬取投诉帖子的编号、帖子的url、帖子的标题, 和帖子里的内容.

response.url获取响应的url地址

### items.py

import scrapy

class DongguanItem(scrapy.Item):

# 每个帖子的标题

title = scrapy.Field()

# 每个帖子的编号

number = scrapy.Field()

# 每个帖子的文字内容

content = scrapy.Field()

# 每个帖子的url

url = scrapy.Field()

### spiders/sunwz.py

##### Spider 版本

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

import scrapy

from dongguan.items import DongguanItem

class SunSpider(CrawlSpider):

name = 'sun'

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

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

offset = 0

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

def parse(self, response):

# 取出每个页面里帖子链接列表

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

# 迭代发送每个帖子的请求, 调用parse\_item方法处理

for link in links:

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

# 设置页码终止条件, 并且每次发送新的页面请求调用parse方法处理

if self.offset <= 71130:

self.offset += 30

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

# 处理每个帖子里

def parse\_item(self, response):

item = DongguanItem()

# 标题

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

# 编号

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

# 文字内容, 默认先取出有图片情况下的文字内容列表

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

# 如果没有内容, 则取出没有图片情况下的文字内容列表

if len(content) == 0:

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

# content为列表, 通过join方法拼接为字符串, 并去除首尾空格

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

else:

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

# 链接

item['url'] = response.url

yield item

##### CrawlSpider 版本

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

import scrapy

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

from dongguan.items import DongguanItem

import time

class SunSpider(CrawlSpider):

name = 'sun'

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

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

# 每一页的匹配规则

pagelink = LinkExtractor(allow=('type=4'))

# 每个帖子的匹配规则

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

rules = [

# 本案例为特殊情况, 需要调用deal\_links方法处理每个页面里的链接

Rule(pagelink, process\_links = "deal\_links", follow = True),

Rule(contentlink, callback = 'parse\_item')

]

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

def deal\_links(self, links):

for link in links:

link.url = link.url.replace("?","&").replace("Type&", "Type?")

print link.url

return links

def parse\_item(self, response):

print response.url

item = DongguanItem()

# 标题

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

# 编号

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

# 文字内容, 默认先取出有图片情况下的文字内容列表

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

# 如果没有内容, 则取出没有图片情况下的文字内容列表

if len(content) == 0:

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

# content为列表, 通过join方法拼接为字符串, 并去除首尾空格

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

else:

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

# 链接

item['url'] = response.url

yield item

#### pipelines.py

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

# 文件处理类库, 可以指定编码格式

import codecs

import json

class JsonWriterPipeline(object):

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

# 创建一个只写文件, 指定文本编码格式为utf-8

self.filename = codecs.open('sunwz.json', 'w', encoding='utf-8')

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

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

self.filename.write(content)

return item

def spider\_closed(self, spider):

self.file.close()

#### settings.py

ITEM\_PIPELINES = {

'dongguan.pipelines.DongguanPipeline': 300,

}

# 日志文件名和处理等级

LOG\_FILE = "dg.log"

LOG\_LEVEL = "DEBUG"

#### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

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

#### 执行程序

py2 main.py

视频演示

### items.py

**import** scrapy

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

*# 每个帖子的标题*

title = scrapy.Field()

*# 每个帖子的编号*

number = scrapy.Field()

*# 每个帖子的文字内容*

content = scrapy.Field()

*# 每个帖子的url*

url = scrapy.Field()

### Spiders/Crawl Spider

#### Spider 版本

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

**import** scrapy

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

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

name = **'dongguan'**

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

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

offset = 0

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

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

*# 取出每一页里的所有帖子的链接, 返回一个列表*

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

*# 迭代取出集合里的链接*

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

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

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

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

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

self.offset += 30

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

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

*# 处理每个帖子的response内容*

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

item = DongguanItem()

*# 标题*

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

*# 编号*

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

*# 链接*

item[**'url'**] = response.url

*# 有图片和无图片时内容所在的位置是不同的, 先使用有图片情况下的匹配规则, 如果有内容, 返回所有内容的列表集合*

content = response.xpath(**'//div[@class="contentext"]/text()'**).extract()

*# 如果没有内容, 则返回空列表, 则使用无图片情况下的匹配规则*

**if** len(content) == 0:

content = response.xpath(**'//div[@class="c1 text14\_2"]/text()'**).extract()

*# content为列表, 通过join方法拼接为字符串, 并去除首尾空格*

item[**'content'**] = **""**.join(content).strip()

**else**:

item[**'content'**] = **""**.join(content).strip()

*# 把数据交给管道文件进行处理*

**yield** item

#### CrawlSpider 版本

# -\*- coding: utf-8 -\*-

**import** scrapy

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

**from** newdongguan.items **import** DongguanItem

**class** DongdongSpider(CrawlSpider):

name = **'dongdong'**

allowed\_domains = [**'wz.sun0769.com'**]

start\_urls = [**'http://wz.sun0769.com/index.php/question/questionType?type=4&page='**]

*# 每一页的匹配规则*

pagelink = LinkExtractor(allow=(**"type=4"**))

*# 每一页里的每个帖子的匹配规则*

contentlink = LinkExtractor(allow=(**r"/html/question/\d+/\d+.shtml"**))

rules = (

*# 本案例中返回的pagelink被web服务器篡改, 需要调用process\_links来处理提取出来的url, 这也是反爬虫的方式之一, 短时间内大量的访问会返回一些假的url地址.*

*#如果没有回调函数, follow默认为True*

Rule(pagelink, process\_links = **"deal\_links"**),

*#CrawlSpider中的callback函数必须用''括起来.*

Rule(contentlink, callback = **"parse\_item"**)

)

*# 需要重新处理每个页面里的链接, 将链接里的'Type&type=4?page=xxx'替换为'Type?type=4&page=xxx'（或者是Type&page=xxx?type=4'替换为'Type?page=xxx&type=4'）, 否则无法发送这个链接*

*# links 是当前response里提取出来的链接列表*

**def** deal\_links(self, links):

**for** each **in** links:

each.url = each.url.replace(**"?"**,**"&"**).replace(**"Type&"**,**"Type?"**)

*#或使用re.sub(s1,s2,s3)*

*#返回修改后的links链接列表*

**return** links

**def** parse\_item(self, response):

item = DongguanItem()

*# 标题*

item[**'title'**] = response.xpath(**'//div[contains(@class, "pagecenter p3")]//strong/text()'**).extract()[0]

*# 编号*

item[**'number'**] = item[**'title'**].split(**' '**)[-1].split(**":"**)[-1]

*# 链接, 使用response.url来获取链接*

item[**'url'**] = response.url

*# 有图片和无图片时内容所在的位置是不同的, 先使用有图片情况下的匹配规则, 如果有内容, 返回所有内容的列表集合*

content = response.xpath(**'//div[@class="contentext"]/text()'**).extract()

*# 如果没有内容, 则返回空列表, 则使用无图片情况下的匹配规则*

**if** len(content) == 0:

content = response.xpath(**'//div[@class="c1 text14\_2"]/text()'**).extract()

item[**'content'**] = **""**.join(content).strip()

**else**:

item[**'content'**] = **""**.join(content).strip()

*# 把数据交给管道文件进行处理*

**yield** item

### pipelines.py

# -\*- coding: utf-8 -\*-

**import** codecs, json

*# codecs是文件处理类库, 可以指定编码格式*

**class** NewdongguanPipeline(object):

**def** \_\_init\_\_(self):

*# 创建一个文件, 指定文本存储的格式为utf-8*

*#在整个爬虫运行期间, 管道文件是始终打开的, 所以这里不用使用追加写入的方式.*

self.filename = codecs.open(**"donggguan.json"**, **"w"**, encoding = **"utf-8"**)

**def** process\_item(self, item, spider):

*# 中文默认使用ascii码来存储, 禁用后默认为Unicode字符串*

content = json.dumps(dict(item), ensure\_ascii=**False**) + **",\n"**

self.filename.write(content)

*#如果在\_\_init\_\_方法中不指定创建的文件的格式, 就要在这里指定写入的内容的格式*

*# self.filename.write(content.encode("utf-8"))*

**return** item

**def** close\_spider(self, spider):

self.filename.close()

### settings.py

BOT\_NAME = **'newdongguan'**

SPIDER\_MODULES = [**'newdongguan.spiders'**]

NEWSPIDER\_MODULE = **'newdongguan.spiders'**

ITEM\_PIPELINES = {

'dongguan.pipelines.DongguanPipeline': 300,

}

#设置请求头

#提供了一个单独的USER-AGENT

#可以在DEFAULT\_REQUEST\_HEADERS中设置User-Agent, 也可以直接在USER-AGENT中设置.

USER\_AGENT = **"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"**

ITEM\_PIPELINES = {

**'newdongguan.pipelines.NewdongguanPipeline'**: 1,

}

# 日志文件名和处理等级

LOG\_FILE = **"dongguan.log"**

LOG\_LEVEL = **"DEBUG"**

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl sunwz'.split())

### 执行程序

py2 main.py

## (案例三)新浪网分类资讯爬虫

爬取新浪网导航页所有下所有大类、小类、小类里的子链接, 以及子链接页面的新闻内容.

### items.py

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### spiders/sina.py

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

import scrapy

import os

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaSpider(scrapy.Spider):

name= "sina"

allowed\_domains= ["sina.com.cn"]

start\_urls= [

"http://news.sina.com.cn/guide/"

]

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类目录的路径和目录名

parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在, 则创建目录

if(not os.path.exists(parentFilename)):

os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头, 如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类, 将存储目录放在本大类目录下

if(if\_belong):

subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在, 则创建目录

if(not os.path.exists(subFilename)):

os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求, 得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url, 再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾, 如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类, 获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] = meta\_1['subUrls']

item['subTitle'] = meta\_1['subTitle']

item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求, 得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法, 获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()')

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head

item['content']= content

yield item

### pipelines.py

from scrapy import signals

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaPipeline(object):

def process\_item(self, item, spider):

sonUrls = item['sonUrls']

# 文件名为子链接url中间部分, 并将 / 替换为 \_, 保存为 .txt格式

filename = sonUrls[7:-6].replace('/','\_')

filename += ".txt"

fp = open(item['subFilename']+'/'+filename, 'w')

fp.write(item['content'])

fp.close()

return item

### settings.py

BOT\_NAME = 'Sina'

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

ITEM\_PIPELINES = {

'Sina.pipelines.SinaPipeline': 300,

}

LOG\_LEVEL = 'DEBUG'

### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl sina'.split())

### 执行程序

py2 main.py

## (案例四)cosplay图片下载器爬虫

### items.py

class CoserItem(scrapy.Item):

url = scrapy.Field()

name = scrapy.Field()

info = scrapy.Field()

image\_urls = scrapy.Field()

images = scrapy.Field()

### spiders/coser.py

# -\*- coding: utf-8 -\*-

from scrapy.selector import Selector

import scrapy

from scrapy.contrib.loader import ItemLoader

from Cosplay.items import CoserItem

class CoserSpider(scrapy.Spider):

name = "coser"

allowed\_domains = ["bcy.net"]

start\_urls = (

'http://bcy.net/cn125101',

'http://bcy.net/cn126487',

'http://bcy.net/cn126173'

)

def parse(self, response):

sel = Selector(response)

for link in sel.xpath("//ul[@class='js-articles l-works']/li[@class='l-work--big']/article[@class='work work--second-created']/h2[@class='work\_\_title']/a/@href").extract():

link = 'http://bcy.net%s' % link

request = scrapy.Request(link, callback=self.parse\_item)

yield request

def parse\_item(self, response):

l = ItemLoader(item=CoserItem(), response=response)

l.add\_xpath('name', "//h1[@class='js-post-title']/text()")

l.add\_xpath('info', "//div[@class='post\_\_info']/div[@class='post\_\_type post\_\_info-group']/span/text()")

urls = l.get\_xpath('//img[@class="detail\_std detail\_clickable"]/@src')

urls = [url.replace('/w650', '') for url in urls]

l.add\_value('image\_urls', urls)

l.add\_value('url', response.url)

return l.load\_item()

#### pipelines.py

import requests

from Cosplay import settings

import os

class ImageDownloadPipeline(object):

def process\_item(self, item, spider):

if 'image\_urls' in item:

images = []

dir\_path = '%s/%s' % (settings.IMAGES\_STORE, spider.name)

if not os.path.exists(dir\_path):

os.makedirs(dir\_path)

for image\_url in item['image\_urls']:

us = image\_url.split('/')[3:]

image\_file\_name = '\_'.join(us)

file\_path = '%s/%s' % (dir\_path, image\_file\_name)

images.append(file\_path)

if os.path.exists(file\_path):

continue

with open(file\_path, 'wb') as handle:

response = requests.get(image\_url, stream=True)

for block in response.iter\_content(1024):

if not block:

break

handle.write(block)

item['images'] = images

return item

#### settings.py

ITEM\_PIPELINES = {'Cosplay.pipelines.ImageDownloadPipeline': 1}

IMAGES\_STORE = '../Images'

DOWNLOAD\_DELAY = 0.25 # 250 ms of delay

#### 在项目根目录下新建main.py文件,用于调试

from scrapy import cmdline

cmdline.execute('scrapy crawl coser'.split())

#### 执行程序

py2 main.py

使用在sublime中使用 :vsp打开一个水平方向并排的窗口

## (案例五)将数据保存在MongoDB中

爬取豆瓣电影top250[movie.douban.com/top250](file:///D:\David\Desktop\IT%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6%E5%A4%B9\python%E5%AD%A6%E4%B9%A0%E8%B7%AF%E7%BA%BF%E5%9B%BE\05_%E4%BC%A0%E6%99%BA%E6%92%AD%E5%AE%A2python%E5%B0%B1%E4%B8%9A%E7%8F%AD%E8%A7%86%E9%A2%91\07%20%E7%88%AC%E8%99%AB\%E8%B5%84%E6%96%99\file\part05\movie.douban.com\top250)的电影数据, 并保存在MongoDB中.

### items.py

class DoubanspiderItem(scrapy.Item):

# 电影标题

title = scrapy.Field()

# 电影评分

score = scrapy.Field()

# 电影信息

content = scrapy.Field()

# 简介

info = scrapy.Field()

### spiders/douban.py

import scrapy

from doubanSpider.items import DoubanspiderItem

class DoubanSpider(scrapy.Spider):

name = "douban"

allowed\_domains = ["movie.douban.com"]

start = 0

url = 'https://movie.douban.com/top250?start='

end = '&filter='

start\_urls = [url + str(start) + end]

def parse(self, response):

item = DoubanspiderItem()

movies = response.xpath("//div[@class=\'info\']")

for each in movies:

title = each.xpath('div[@class="hd"]/a/span[@class="title"]/text()').extract()

content = each.xpath('div[@class="bd"]/p/text()').extract()

score = each.xpath('div[@class="bd"]/div[@class="star"]/span[@class="rating\_num"]/text()').extract()

info = each.xpath('div[@class="bd"]/p[@class="quote"]/span/text()').extract()

item['title'] = title[0]

# 以;作为分隔, 将content列表里所有元素合并成一个新的字符串

item['content'] = ';'.join(content)

item['score'] = score[0]

item['info'] = info[0]

# 提交item

yield item

if self.start <= 225:

self.start += 25

yield scrapy.Request(self.url + str(self.start) + self.end, callback=self.parse)

### pipelines.py

from scrapy.conf import settings

import pymongo

class DoubanspiderPipeline(object):

def \_\_init\_\_(self):

# 获取setting主机名、端口号和数据库名

host = settings['MONGODB\_HOST']

port = settings['MONGODB\_PORT']

dbname = settings['MONGODB\_DBNAME']

# pymongo.MongoClient(host, port) 创建MongoDB链接

client = pymongo.MongoClient(host=host,port=port)

# 指向指定的数据库

mdb = client[dbname]

# 获取数据库里存放数据的表名

self.post = mdb[settings['MONGODB\_DOCNAME']]

def process\_item(self, item, spider):

data = dict(item)

# 向指定的表里添加数据

self.post.insert(data)

return item

### settings.py

BOT\_NAME = 'doubanSpider'

SPIDER\_MODULES = ['doubanSpider.spiders']

NEWSPIDER\_MODULE = 'doubanSpider.spiders'

ITEM\_PIPELINES = {

'doubanSpider.pipelines.DoubanspiderPipeline' : 300

}

# Crawl responsibly by identifying yourself (and your website) on the user-agent

USER\_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.116 Safari/537.36'

# MONGODB 主机环回地址127.0.0.1

MONGODB\_HOST = '127.0.0.1'

# 端口号, 默认是27017

MONGODB\_PORT = 27017

# 设置数据库名称

MONGODB\_DBNAME = 'DouBan'

# 存放本次数据的表名称

MONGODB\_DOCNAME = 'DouBanMovies'

### 运行

启动MongoDB数据库需要两个命令：

mongod：是mongoDB数据库进程本身, 启动mongodb的服务

mongo：是命令行shell客户端

sudo mongod # 首先启动数据库服务, 再执行Scrapy

#启动mongodb数据库服务之后, 保持服务窗口不关闭, 要再打开一个窗口来执行命令

sudo mongo # 启动数据库shell

在mongo shell下使用命令:

# 查看当前数据库

> db

# 列出所有的数据库

> show dbs

# 连接DouBan数据库

> use DouBan

# 列出所有表

> show collections

# 查看表里的数据

> db.DouBanMoives.find()

> db.DouBanMoives.findone()

# 删除当前数据库

> db.dropDatabase()

视频演示

### items.py

**import** scrapy

**class** DoubanItem(scrapy.Item):

*# 电影标题*

title = scrapy.Field()

*# 电影评分*

score = scrapy.Field()

*# 电影信息*

content = scrapy.Field()

*# 简介*

info = scrapy.Field()

### spiders/douban.py

**import** scrapy

**from** douban.items **import** DoubanItem

**class** DoubamovieSpider(scrapy.Spider):

name = **"doubanmovie"**

allowed\_domains = [**"movie.douban.com"**]

offset = 0

url = **"https://movie.douban.com/top250?start="**

start\_urls = (

url+str(offset),

)

**def** parse(self, response):

item = DoubanItem()

*#包含每一页的25个元素*

movies = response.xpath(**"//div[@class='info']"**)

**for** each **in** movies:

*# 标题*

item[**'title'**] = each.xpath(**".//span[@class='title'][1]/text()"**).extract()[0]

*# 信息*

content = each.xpath(**".//div[@class='bd']/p/text()"**).extract()

*# 以;作为分隔, 将content列表里所有元素合并成一个新的字符串*

item[**'content'**] = **';'**.join(content)

*# 评分*

item[**'score'**] = each.xpath(**".//div[@class='star']/span[@class='rating\_num']/text()"**).extract()[0]

*# 简介*

info = each.xpath(**".//p[@class='quote']/span/text()"**).extract()

**if** len(info) != 0:

item[**'quote'**] = info[0]

*#发送到管道文件进行处理*

**yield** item

**if** self.offset < 225:

self.offset += 25

**yield** scrapy.Request(self.url + str(self.offset), callback = self.parse)

### settings.py

BOT\_NAME = **'douban'**

SPIDER\_MODULES = [**'douban.spiders'**]

NEWSPIDER\_MODULE = **'douban.spiders'**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

USER\_AGENT = **"Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0;"**

DOWNLOAD\_DELAY = 2.5

COOKIES\_ENABLED = **False**

ITEM\_PIPELINES = {

**'douban.pipelines.DoubanPipeline'**: 300,

}

*# MONGODB 主机名*

MONGODB\_HOST = **"127.0.0.1"**

*# MONGODB 端口号*

MONGODB\_PORT = 27017

*# 数据库名称*

MONGODB\_DBNAME = **"DouBan"**

*# 存放数据的表名称*

MONGODB\_SHEETNAME = **'DouBanMovies'**

MONGODB\_DOCNAME = **'DouBanMovies'**

### pipelines.py

**import** pymongo

**from** scrapy.conf **import** settings

**class** DoubanPipeline(object):

**def** \_\_init\_\_(self):

*# 获取setting主机名、端口号和数据库名*

host = settings[**"MONGODB\_HOST"**]

port = settings[**"MONGODB\_PORT"**]

dbname = settings[**"MONGODB\_DBNAME"**]

sheetname= settings[**"MONGODB\_SHEETNAME"**]

*# pymongo.MongoClient(host, port)创建MONGODB数据库链接*

client = pymongo.MongoClient(host = host, port = port)

*# 指定数据库*

mydb = client[dbname]

*# 存放数据的数据库表名*

self.sheet = mydb[sheetname]

**def** process\_item(self, item, spider):

data = dict(item)

*# 向指定的表里添加数据*

self.sheet.insert(data)

**return** item

## (案例六)人人网三种scrapy模拟登陆策略

注意：模拟登陆时, 必须保证settings.py里的 COOKIES\_ENABLED(Cookies中间件) 处于开启状态. 如果是不需要登陆就能采集数据的网站, 要禁用cookie.

COOKIES\_ENABLED = True 或 # COOKIES\_ENABLED = False

### 策略一：直接POST数据（比如需要登陆的账户信息)

只要是需要提供post数据的, 就可以用这种方法. 下面示例里post的数据是账户密码：

# -\*- coding: utf-8 -\*-

import scrapy

class Renren1Spider(scrapy.Spider):

name = "renren1"

allowed\_domains = ["renren.com"]

def start\_requests(self):

url = 'http://www.renren.com/PLogin.do'

# FormRequest 是Scrapy发送POST请求的方法

yield scrapy.FormRequest(

url = url,

formdata = {"email" : "mr\_mao\_hacker@163.com", "password" : "axxxxxxxe"},

callback = self.parse\_page)

def parse\_page(self, response):

with open("mao2.html", "w") as filename:

filename.write(response.body)

### 策略二：标准的模拟登陆步骤

正统模拟登录方法：

1. 首先发送登录页面的get请求, 获取到页面里的登录必须的参数（比如说zhihu登陆界面的 \_xsrf）
2. 然后和账户密码一起post到服务器, 登录成功

# -\*- coding: utf-8 -\*-

import scrapy

class Renren2Spider(scrapy.Spider):

name = "renren2"

allowed\_domains = ["renren.com"]

start\_urls = (

"http://www.renren.com/PLogin.do",

)

# 处理start\_urls里的登录url的响应内容, 提取登陆需要的参数（如果需要的话)

def parse(self, response):

# 提取登陆需要的参数

#\_xsrf = response.xpath("//\_xsrf").extract()[0]

# 发送post请求参数, 把账户密码和其它相关的参数发送到服务器, 并调用指定回调函数处理

yield scrapy.FormRequest.from\_response(

response,

formdata = {"email" : "mr\_mao\_hacker@163.com", "password" : "axxxxxxxe"},#, "\_xsrf" = \_xsrf},

*#调用回调函数来处理登陆成功后的页面*

callback = self.parse\_page

)

# 获取登录成功后的页面, 访问需要登录后才能访问的页面, 这里的response是登陆之后的response

def parse\_page(self, response):

print(**"=========1==="** + response.url)

*#with open("mao.html", "w") as filename:*

*# filename.write(response.body)*

*#可以定义多个url, 多次迭代来发送请求*

url = **"http://www.renren.com/422167102/profile"**

*#发送一个get的请求, 这时是已经登陆后发送的请求.*

**yield** scrapy.Request(url, callback = self.parse\_newpage)

# 处理响应内容

def parse\_newpage(self, response):

print(**"===========2===="** + response.url)

with open("xiao.html", "w") as filename:

filename.write(response.body)

### 策略三：直接使用保存登陆状态的Cookie模拟登陆

如果实在没办法了, 可以用这种方法模拟登录, 虽然麻烦一点, 但是成功率100%

**取登陆后的多个url**

# -\*- coding: utf-8 -\*-

import scrapy

class RenrenSpider(scrapy.Spider):

name = "renren"

allowed\_domains = ["renren.com"]

*#虽然没有被scrapy自己的start request方法调用, 但我们可以手动去调用它.*

*#定义一系列登陆后才能查看的页面, 例如是好友的主页.*

start\_urls = (

'http://www.renren.com/111111',

'http://www.renren.com/222222',

'http://www.renren.com/333333',

)

*#抓包, 在raw中查看登陆后的cookie值. 使用正则替换, 查找(.\*?)=(.\*?); 替换为"\1": "\2",\n*

cookies = {

"anonymid" : "ixrna3fysufnwv",

"\_r01\_" : "1",

"ap" : "327550029",

"JSESSIONID" : "abciwg61A\_RvtaRS3GjOv",

"depovince" : "GW",

"springskin" : "set",

"jebe\_key" : "f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198628950",

"t" : "691808127750a83d33704a565d8340ae9",

"societyguester" : "691808127750a83d33704a565d8340ae9",

"id" : "327550029",

"xnsid" : "f42b25cf",

"loginfrom" : "syshome"

}

# 可以重写Spider类的start\_requests方法, 附带Cookie值, 发送POST请求

*#重写系统中的方法. 定义一个与系统中同名的方法, 就会覆盖系统中的方法.*

def start\_requests(self):

for url in self.start\_urls:

*#get请求*

*#yield scrapy.Request(url, callback = self.parse)*

*#post请求*

yield scrapy.FormRequest(url, cookies = self.cookies, callback = self.parse\_page)

# 处理响应内容

def parse\_page(self, response):

print "===========" + response.url

with open("deng.html", "w") as filename:

filename.write(response.body)

import scrapy

*# 实在没办法了, 可以用这种方法模拟登录, 麻烦一点, 成功率100%*

class RenrenSpider(scrapy.Spider):

name = "renren"

allowed\_domains = ["renren.com"]

*#抓包, 在raw中查看登陆后的cookie值. 使用正则替换, 查找(.\*?)=(.\*?); 替换为"\1": "\2",\n*

cookies = {

"anonymid" : "ixrna3fysufnwv",

"\_r01\_" : "1",

"ap" : "327550029",

"JSESSIONID" : "abciwg61A\_RvtaRS3GjOv",

"depovince" : "GW",

"springskin" : "set",

"jebe\_key" : "f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198628950",

"jebe\_key" : "f6fb270b-d06d-42e6-8b53-e67c3156aa7e%7Cc13c37f53bca9e1e7132d4b58ce00fa3%7C1484060607478%7C1%7C1486198619601",

"ver" : "7.0",

"XNESSESSIONID" : "e703b11f8809",

"jebecookies" : "98c7c881-779f-4da8-a57c-7464175cd469|||||",

"ick\_login" : "4b4a254a-9f25-4d4a-b686-a41fda73e173",

"\_de" : "BF09EE3A28DED52E6B65F6A4705D973F1383380866D39FF5",

"p" : "ea5541736f993365a23d04c0946c10e29",

"first\_login\_flag" : "1",

"ln\_uact" : "mr\_mao\_hacker@163.com",

"ln\_hurl" : "http://hdn.xnimg.cn/photos/hdn521/20140529/1055/h\_main\_9A3Z\_e0c300019f6a195a.jpg",

"t" : "691808127750a83d33704a565d8340ae9",

"societyguester" : "691808127750a83d33704a565d8340ae9",

"id" : "327550029",

"xnsid" : "f42b25cf",

"loginfrom" : "syshome"

}

*#重写系统中的方法. 定义一个与系统中同名的方法, 就会覆盖系统中的方法.* 重写Spider类的start\_requests方法, 附带Cookie值, 发送POST请求

def start\_requests(self):

*#get请求*

*#yield scrapy.Request(url, callback = self.parse)*

*#不使用starturl, 而是使用自定义的url*

url = "http://www.renren.com/410043129/profile"

*#post请求*

yield scrapy.FormRequest(url, cookies = self.cookies, callback = self.parse\_page)

def parse\_page(self, response):

print("===========" + response.url)

with open("deng.html", "w") as filename:

filename.write(response.body)

## 附: 通过Fiddler进行手机抓包方法

通过Fiddler抓包工具, 可以抓取手机的网络通信, 但前提是手机和电脑处于同一局域网内（WI-FI或热点）, 然后进行以下设置：

### 用Fiddler对Android应用进行抓包

1. 打开Fiddler设置
2. 在Connections里设置允许连接远程计算机, 确认后重新启动Fiddler
3. 在命令提示符下输入ipconfig查看本机IP
4. 打开Android设备的"设置"->"WLAN", 找到你要连接的网络, 在上面长按, 然后选择"修改网络", 弹出网络设置对话框, 然后勾选"显示高级选项".
5. 在"代理"后面的输入框选择"手动", 在"代理服务器主机名"后面的输入框输入电脑的ip地址, 在"代理服务器端口"后面的输入框输入8888, 然后点击"保存"按钮.
6. 启动Android设备中的浏览器, 访问网页即可在Fiddler中可以看到完成的请求和响应数据.

### 用Fiddler对iPhone手机应用进行抓包

基本流程差不多, 只是手机设置不太一样：

iPhone手机：点击设置 > 无线局域网 > 无线网络 > HTTP代理 > 手动：

代理地址(电脑IP)：192.168.xx.xxx

端口号：8888

# 6. scrapy-redis分布式组件

## scrapy-redis简介

### Scrapy 和 scrapy-redis的区别

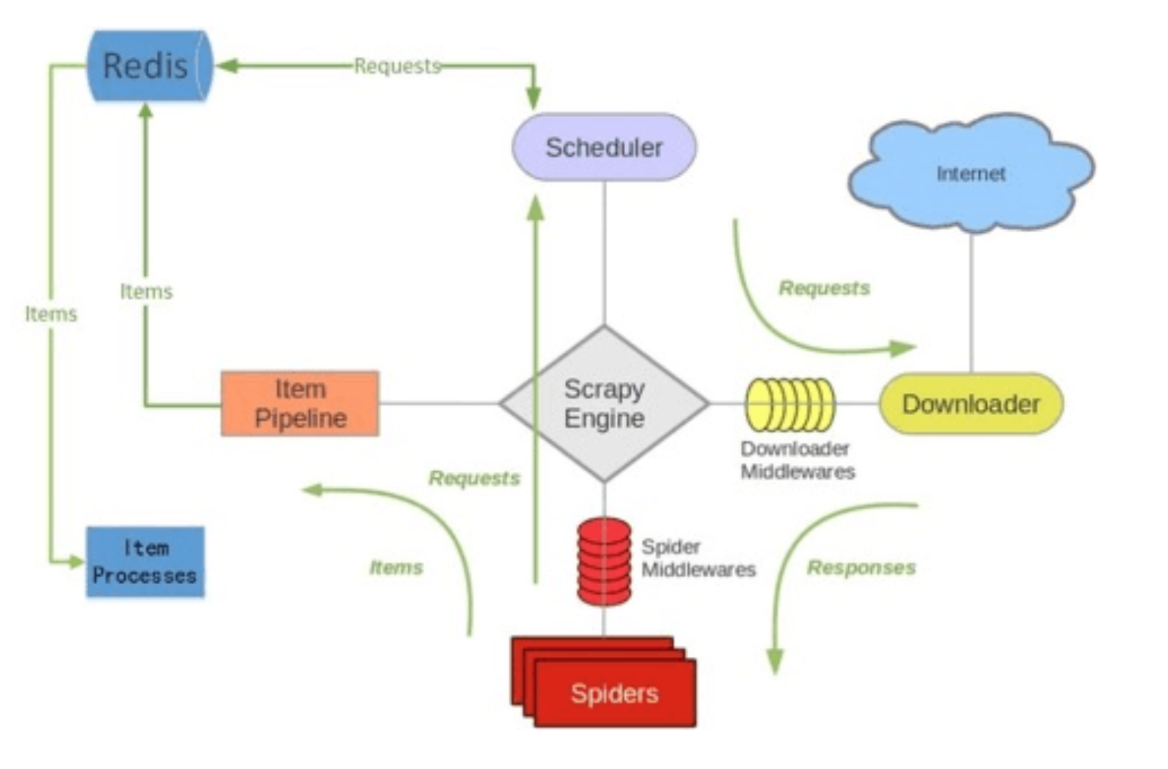
Scrapy 是一个通用的爬虫框架, 但是不支持分布式, Scrapy-redis是为了更方便地实现Scrapy分布式爬取, 而提供了一些以redis为基础的组件(仅有组件). 数据存储在redis数据库中, 是一个基于内存的数据库.

pip install scrapy-redis

Scrapy-redis提供了下面四种组件（components）：(四种组件意味着这四个模块都要做相应的修改)

* Scheduler
* Duplication Filter
* Item Pipeline
* Base Spider

### scrapy-redis架构



如上图所⽰示, scrapy-redis在scrapy的架构上增加了redis, 基于redis的特性拓展了如下组件：

#### Scheduler：

Scrapy改造了python本来的collection.deque(双向队列)形成了自己的Scrapy queue ([https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py)](https://github.com/scrapy/queuelib/blob/master/queuelib/queue.py)), 但是Scrapy多个spider不能共享待爬取队列Scrapy queue, 即Scrapy本身不支持爬虫分布式, scrapy-redis 的解决是把这个Scrapy queue换成redis数据库（也是指redis队列）, 从同一个redis-server存放要爬取的request, 便能让多个spider去同一个数据库里读取.

Scrapy中跟"待爬队列"直接相关的就是调度器Scheduler, 它负责对新的request进行入列操作（加入Scrapy queue）, 取出下一个要爬取的request（从Scrapy queue中取出）等操作. 它把待爬队列按照优先级建立了一个字典结构, 比如：

{

优先级0 : 队列0

优先级1 : 队列1

优先级2 : 队列2

}

然后根据request中的优先级, 来决定该入哪个队列, 出列时则按优先级较小的优先出列. 为了管理这个比较高级的队列字典, Scheduler需要提供一系列的方法. 但是原来的Scheduler已经无法使用, 所以使用Scrapy-redis的scheduler组件.

#### Duplication Filter

Scrapy中用集合实现这个request去重功能, Scrapy中把已经发送的request指纹放入到一个集合中, 把下一个request的指纹拿到集合中比对, 如果该指纹存在于集合中, 说明这个request发送过了, 如果没有则继续操作. 这个核心的判重功能是这样实现的：

def request\_seen(self, request):

# self.request\_figerprints就是一个指纹集合

fp = self.request\_fingerprint(request)

# 这就是判重的核心操作

if fp in self.fingerprints:

return True

self.fingerprints.add(fp)

if self.file:

self.file.write(fp + os.linesep)

在scrapy-redis中去重是由Duplication Filter组件来实现的, 它通过redis的set 不重复的特性, 巧妙的实现了Duplication Filter去重. scrapy-redis调度器从引擎接受request, 将request的指纹存⼊redis的set检查是否重复, 并将不重复的request push写⼊redis的 request queue.

引擎请求request(Spider发出的）时, 调度器从redis的request queue队列⾥里根据优先级pop 出⼀个request 返回给引擎, 引擎将此request发给spider处理.

#### Item Pipeline：

引擎将(Spider返回的)爬取到的Item给Item Pipeline, scrapy-redis 的Item Pipeline将爬取到的 Item 存⼊redis的 items queue.

修改过Item Pipeline可以很方便的根据 key 从 items queue 提取item, 从⽽实现 items processes集群.

#### Base Spider

不在使用scrapy原有的Spider类, 重写的RedisSpider继承了Spider和RedisMixin这两个类, RedisMixin是用来从redis读取url的类.

当我们生成一个Spider继承RedisSpider时, 调用setup\_redis函数, 这个函数会去连接redis数据库, 然后会设置signals(信号)：

* 一个是当spider空闲时候的signal, 会调用spider\_idle函数, 这个函数调用schedule\_next\_request函数, 保证spider是一直活着的状态, 并且抛出DontCloseSpider异常.
* 一个是当抓到一个item时的signal, 会调用item\_scraped函数, 这个函数会调用schedule\_next\_request函数, 获取下一个request.

## 源码分析参考: Connection

官方站点：<https://github.com/rolando/scrapy-redis>

scrapy-redis的官方文档写的比较简洁, 没有提及其运行原理, 所以如果想全面的理解分布式爬虫的运行原理, 还是得看scrapy-redis的源代码才行.

scrapy-redis工程的主体还是是redis和scrapy两个库, 工程本身实现的东西不是很多, 这个工程就像胶水一样, 把这两个插件粘结了起来. 下面我们来看看, scrapy-redis的每一个源代码文件都实现了什么功能, 最后如何实现分布式的爬虫系统：

1. [connection.py](https://github.com/rolando/scrapy-redis/blob/master/src/scrapy_redis/connection.py)

负责根据setting中配置实例化redis连接. 被dupefilter和scheduler调用, 总之涉及到redis存取的都要使用到这个模块.

# 这里引入了redis模块, 这个是redis-python库的接口, 用于通过python访问redis数据库,

# 这个文件主要是实现连接redis数据库的功能, 这些连接接口在其他文件中经常被用到

import redis

import six

from scrapy.utils.misc import load\_object

DEFAULT\_REDIS\_CLS = redis.StrictRedis

# 可以在settings文件中配置套接字的超时时间、等待时间等

# Sane connection defaults.

DEFAULT\_PARAMS = {

'socket\_timeout': 30,

'socket\_connect\_timeout': 30,

'retry\_on\_timeout': True,

}

# 要想连接到redis数据库, 和其他数据库差不多, 需要一个ip地址、端口号、用户名密码（可选）和一个整形的数据库编号

# Shortcut maps 'setting name' -> 'parmater name'.

SETTINGS\_PARAMS\_MAP = {

'REDIS\_URL': 'url',

'REDIS\_HOST': 'host',

'REDIS\_PORT': 'port',

}

def get\_redis\_from\_settings(settings):

"""Returns a redis client instance from given Scrapy settings object.

This function uses ``get\_client`` to instantiate the client and uses

``DEFAULT\_PARAMS`` global as defaults values for the parameters. You can

override them using the ``REDIS\_PARAMS`` setting.

Parameters

----------

settings : Settings

A scrapy settings object. See the supported settings below.

Returns

-------

server

Redis client instance.

Other Parameters

----------------

REDIS\_URL : str, optional

Server connection URL.

REDIS\_HOST : str, optional

Server host.

REDIS\_PORT : str, optional

Server port.

REDIS\_PARAMS : dict, optional

Additional client parameters.

"""

params = DEFAULT\_PARAMS.copy()

params.update(settings.getdict('REDIS\_PARAMS'))

# XXX: Deprecate REDIS\_\* settings.

for source, dest in SETTINGS\_PARAMS\_MAP.items():

val = settings.get(source)

if val:

params[dest] = val

# Allow ``redis\_cls`` to be a path to a class.

if isinstance(params.get('redis\_cls'), six.string\_types):

params['redis\_cls'] = load\_object(params['redis\_cls'])

# 返回的是redis库的Redis对象, 可以直接用来进行数据操作的对象

return get\_redis(\*\*params)

# Backwards compatible alias.

from\_settings = get\_redis\_from\_settings

def get\_redis(\*\*kwargs):

"""Returns a redis client instance.

Parameters

----------

redis\_cls : class, optional

Defaults to ``redis.StrictRedis``.

url : str, optional

If given, ``redis\_cls.from\_url`` is used to instantiate the class.

\*\*kwargs

Extra parameters to be passed to the ``redis\_cls`` class.

Returns

-------

server

Redis client instance.

"""

redis\_cls = kwargs.pop('redis\_cls', DEFAULT\_REDIS\_CLS)

url = kwargs.pop('url', None)

if url:

return redis\_cls.from\_url(url, \*\*kwargs)

else:

return redis\_cls(\*\*kwargs)

## 源码分析参考: Dupefilter

### dupefilter.py

负责执行requst的去重, 实现的很有技巧性, 使用redis的set数据结构. 但是注意scheduler并不使用其中用于在这个模块中实现的dupefilter键做request的调度, 而是使用queue.py模块中实现的queue.

当request不重复时, 将其存入到queue中, 调度时将其弹出.

import logging

import time

from scrapy.dupefilters import BaseDupeFilter

from scrapy.utils.request import request\_fingerprint

from .connection import get\_redis\_from\_settings

DEFAULT\_DUPEFILTER\_KEY = "dupefilter:%(timestamp)s"

logger = logging.getLogger(\_\_name\_\_)

# TODO: Rename class to RedisDupeFilter.

class RFPDupeFilter(BaseDupeFilter):

"""Redis-based request duplicates filter.

This class can also be used with default Scrapy's scheduler.

"""

logger = logger

def \_\_init\_\_(self, server, key, debug=False):

"""Initialize the duplicates filter.

Parameters

----------

server : redis.StrictRedis

The redis server instance.

key : str

Redis key Where to store fingerprints.

debug : bool, optional

Whether to log filtered requests.

"""

self.server = server

self.key = key

self.debug = debug

self.logdupes = True

@classmethod

def from\_settings(cls, settings):

"""Returns an instance from given settings.

This uses by default the key ``dupefilter:<timestamp>``. When using the

``scrapy\_redis.scheduler.Scheduler`` class, this method is not used as

it needs to pass the spider name in the key.

Parameters

----------

settings : scrapy.settings.Settings

Returns

-------

RFPDupeFilter

A RFPDupeFilter instance.

"""

server = get\_redis\_from\_settings(settings)

# XXX: This creates one-time key. needed to support to use this

# class as standalone dupefilter with scrapy's default scheduler

# if scrapy passes spider on open() method this wouldn't be needed

# TODO: Use SCRAPY\_JOB env as default and fallback to timestamp.

key = DEFAULT\_DUPEFILTER\_KEY % {'timestamp': int(time.time())}

debug = settings.getbool('DUPEFILTER\_DEBUG')

return cls(server, key=key, debug=debug)

@classmethod

def from\_crawler(cls, crawler):

"""Returns instance from crawler.

Parameters

----------

crawler : scrapy.crawler.Crawler

Returns

-------

RFPDupeFilter

Instance of RFPDupeFilter.

"""

return cls.from\_settings(crawler.settings)

def request\_seen(self, request):

"""Returns True if request was already seen.

Parameters

----------

request : scrapy.http.Request

Returns

-------

bool

"""

fp = self.request\_fingerprint(request)

# This returns the number of values added, zero if already exists.

added = self.server.sadd(self.key, fp)

return added == 0

def request\_fingerprint(self, request):

"""Returns a fingerprint for a given request.

Parameters

----------

request : scrapy.http.Request

Returns

-------

str

"""

return request\_fingerprint(request)

def close(self, reason=''):

"""Delete data on close. Called by Scrapy's scheduler.

Parameters

----------

reason : str, optional

"""

self.clear()

def clear(self):

"""Clears fingerprints data."""

self.server.delete(self.key)

def log(self, request, spider):

"""Logs given request.

Parameters

----------

request : scrapy.http.Request

spider : scrapy.spiders.Spider

"""

if self.debug:

msg = "Filtered duplicate request: %(request)s"

self.logger.debug(msg, {'request': request}, extra={'spider': spider})

elif self.logdupes:

msg = ("Filtered duplicate request %(request)s"

" - no more duplicates will be shown"

" (see DUPEFILTER\_DEBUG to show all duplicates)")

msg = "Filtered duplicate request: %(request)s"

self.logger.debug(msg, {'request': request}, extra={'spider': spider})

self.logdupes = False

这个文件看起来比较复杂, 重写了scrapy本身已经实现的request判重功能. 因为本身scrapy单机跑的话, 只需要读取内存中的request队列或者持久化的request队列（scrapy默认的持久化似乎是json格式的文件, 不是数据库）就能判断这次要发出的request url是否已经请求过或者正在调度（本地读就行了）. 而分布式跑的话, 就需要各个主机上的scheduler都连接同一个数据库的同一个request池来判断这次的请求是否是重复的了.

在这个文件中, 通过继承BaseDupeFilter重写他的方法, 实现了基于redis的判重. 根据源代码来看, scrapy-redis使用了scrapy本身的一个fingerprint接request\_fingerprint, 这个接口很有趣, 根据scrapy文档所说, 他通过hash来判断两个url是否相同（相同的url会生成相同的hash结果）, 但是当两个url的地址相同, get型参数相同但是顺序不同时, 也会生成相同的hash结果（这个真的比较神奇. . . ）所以scrapy-redis依旧使用url的fingerprint来判断request请求是否已经出现过.

这个类通过连接redis, 使用一个key来向redis的一个set中插入fingerprint（这个key对于同一种spider是相同的, redis是一个key-value的数据库, 如果key是相同的, 访问到的值就是相同的, 这里使用spider名字+DupeFilter的key就是为了在不同主机上的不同爬虫实例, 只要属于同一种spider, 就会访问到同一个set, 而这个set就是他们的url判重池）, 如果返回值为0, 说明该set中该fingerprint已经存在（因为集合是没有重复值的）, 则返回False, 如果返回值为1, 说明添加了一个fingerprint到set中, 则说明这个request没有重复, 于是返回True, 还顺便把新fingerprint加入到数据库中了. DupeFilter判重会在scheduler类中用到, 每一个request在进入调度之前都要进行判重, 如果重复就不需要参加调度, 直接舍弃就好了, 不然就是白白浪费资源.

## 源码分析参考: Picklecompat

### picklecompat.py

"""A pickle wrapper module with protocol=-1 by default."""

try:

import cPickle as pickle # PY2

except ImportError:

import pickle

def loads(s):

return pickle.loads(s)

def dumps(obj):

return pickle.dumps(obj, protocol=-1)

这里实现了loads和dumps两个函数, 其实就是实现了一个序列化器.

因为redis数据库不能存储复杂对象（key部分只能是字符串, value部分只能是字符串, 字符串列表, 字符串集合和hash）, 所以我们存啥都要先串行化成文本才行.

这里使用的就是python的pickle模块, 一个兼容py2和py3的串行化工具. 这个serializer主要用于一会的scheduler存reuqest对象.

## 源码分析参考: Pipelines

### pipelines.py

这是是用来实现分布式处理的作用. 它将Item存储在redis中以实现分布式处理. 由于在这里需要读取配置, 所以就用到了from\_crawler()函数.

from scrapy.utils.misc import load\_object

from scrapy.utils.serialize import ScrapyJSONEncoder

from twisted.internet.threads import deferToThread

from . import connection

default\_serialize = ScrapyJSONEncoder().encode

class RedisPipeline(object):

"""Pushes serialized item into a redis list/queue"""

def \_\_init\_\_(self, server,

key='%(spider)s:items',

serialize\_func=default\_serialize):

self.server = server

self.key = key

self.serialize = serialize\_func

@classmethod

def from\_settings(cls, settings):

params = {

'server': connection.from\_settings(settings),

}

if settings.get('REDIS\_ITEMS\_KEY'):

params['key'] = settings['REDIS\_ITEMS\_KEY']

if settings.get('REDIS\_ITEMS\_SERIALIZER'):

params['serialize\_func'] = load\_object(

settings['REDIS\_ITEMS\_SERIALIZER']

)

return cls(\*\*params)

@classmethod

def from\_crawler(cls, crawler):

return cls.from\_settings(crawler.settings)

def process\_item(self, item, spider):

return deferToThread(self.\_process\_item, item, spider)

def \_process\_item(self, item, spider):

key = self.item\_key(item, spider)

data = self.serialize(item)

self.server.rpush(key, data)

return item

def item\_key(self, item, spider):

"""Returns redis key based on given spider.

Override this function to use a different key depending on the item

and/or spider.

"""

return self.key % {'spider': spider.name}

pipelines文件实现了一个item pipieline类, 和scrapy的item pipeline是同一个对象, 通过从settings中拿到我们配置的REDIS\_ITEMS\_KEY作为key, 把item串行化之后存入redis数据库对应的value中（这个value可以看出出是个list, 我们的每个item是这个list中的一个结点）, 这个pipeline把提取出的item存起来, 主要是为了方便我们延后处理数据.

## 源码分析参考: Queue

### queue.py

该文件实现了几个容器类, 可以看这些容器和redis交互频繁, 同时使用了我们上边picklecompat中定义的序列化器. 这个文件实现的几个容器大体相同, 只不过一个是队列, 一个是栈, 一个是优先级队列, 这三个容器到时候会被scheduler对象实例化, 来实现request的调度. 比如我们使用SpiderQueue最为调度队列的类型, 到时候request的调度方法就是先进先出, 而实用SpiderStack就是先进后出了.

从SpiderQueue的实现看出来, 他的push函数就和其他容器的一样, 只不过push进去的request请求先被scrapy的接口request\_to\_dict变成了一个dict对象（因为request对象实在是比较复杂, 有方法有属性不好串行化）, 之后使用picklecompat中的serializer串行化为字符串, 然后使用一个特定的key存入redis中（该key在同一种spider中是相同的）. 而调用pop时, 其实就是从redis用那个特定的key去读其值（一个list）, 从list中读取最早进去的那个, 于是就先进先出了. 这些容器类都会作为scheduler调度request的容器, scheduler在每个主机上都会实例化一个, 并且和spider一一对应, 所以分布式运行时会有一个spider的多个实例和一个scheduler的多个实例存在于不同的主机上, 但是, 因为scheduler都是用相同的容器, 而这些容器都连接同一个redis服务器, 又都使用spider名加queue来作为key读写数据, 所以不同主机上的不同爬虫实例公用一个request调度池, 实现了分布式爬虫之间的统一调度.

from scrapy.utils.reqser import request\_to\_dict, request\_from\_dict

from . import picklecompat

class Base(object):

"""Per-spider queue/stack base class"""

def \_\_init\_\_(self, server, spider, key, serializer=None):

"""Initialize per-spider redis queue.

Parameters:

server -- redis connection

spider -- spider instance

key -- key for this queue (e.g. "%(spider)s:queue")

"""

if serializer is None:

# Backward compatibility.

# TODO: deprecate pickle.

serializer = picklecompat

if not hasattr(serializer, 'loads'):

raise TypeError("serializer does not implement 'loads' function: %r"

% serializer)

if not hasattr(serializer, 'dumps'):

raise TypeError("serializer '%s' does not implement 'dumps' function: %r"

% serializer)

self.server = server

self.spider = spider

self.key = key % {'spider': spider.name}

self.serializer = serializer

def \_encode\_request(self, request):

"""Encode a request object"""

obj = request\_to\_dict(request, self.spider)

return self.serializer.dumps(obj)

def \_decode\_request(self, encoded\_request):

"""Decode an request previously encoded"""

obj = self.serializer.loads(encoded\_request)

return request\_from\_dict(obj, self.spider)

def \_\_len\_\_(self):

"""Return the length of the queue"""

raise NotImplementedError

def push(self, request):

"""Push a request"""

raise NotImplementedError

def pop(self, timeout=0):

"""Pop a request"""

raise NotImplementedError

def clear(self):

"""Clear queue/stack"""

self.server.delete(self.key)

class SpiderQueue(Base):

"""Per-spider FIFO queue"""

def \_\_len\_\_(self):

"""Return the length of the queue"""

return self.server.llen(self.key)

def push(self, request):

"""Push a request"""

self.server.lpush(self.key, self.\_encode\_request(request))

def pop(self, timeout=0):

"""Pop a request"""

if timeout > 0:

data = self.server.brpop(self.key, timeout)

if isinstance(data, tuple):

data = data[1]

else:

data = self.server.rpop(self.key)

if data:

return self.\_decode\_request(data)

class SpiderPriorityQueue(Base):

"""Per-spider priority queue abstraction using redis' sorted set"""

def \_\_len\_\_(self):

"""Return the length of the queue"""

return self.server.zcard(self.key)

def push(self, request):

"""Push a request"""

data = self.\_encode\_request(request)

score = -request.priority

# We don't use zadd method as the order of arguments change depending on

# whether the class is Redis or StrictRedis, and the option of using

# kwargs only accepts strings, not bytes.

self.server.execute\_command('ZADD', self.key, score, data)

def pop(self, timeout=0):

"""

Pop a request

timeout not support in this queue class

"""

# use atomic range/remove using multi/exec

pipe = self.server.pipeline()

pipe.multi()

pipe.zrange(self.key, 0, 0).zremrangebyrank(self.key, 0, 0)

results, count = pipe.execute()

if results:

return self.\_decode\_request(results[0])

class SpiderStack(Base):

"""Per-spider stack"""

def \_\_len\_\_(self):

"""Return the length of the stack"""

return self.server.llen(self.key)

def push(self, request):

"""Push a request"""

self.server.lpush(self.key, self.\_encode\_request(request))

def pop(self, timeout=0):

"""Pop a request"""

if timeout > 0:

data = self.server.blpop(self.key, timeout)

if isinstance(data, tuple):

data = data[1]

else:

data = self.server.lpop(self.key)

if data:

return self.\_decode\_request(data)

\_\_all\_\_ = ['SpiderQueue', 'SpiderPriorityQueue', 'SpiderStack']

## 源码分析参考: Scheduler

### scheduler.py

此扩展是对scrapy中自带的scheduler的替代（在settings的SCHEDULER变量中指出）, 正是利用此扩展实现crawler的分布式调度. 其利用的数据结构来自于queue中实现的数据结构.

scrapy-redis所实现的两种分布式：爬虫分布式以及item处理分布式就是由模块scheduler和模块pipelines实现. 上述其它模块作为为二者辅助的功能模块

import importlib

import six

from scrapy.utils.misc import load\_object

from . import connection

# TODO: add SCRAPY\_JOB support.

class Scheduler(object):

"""Redis-based scheduler"""

def \_\_init\_\_(self, server,

persist=False,

flush\_on\_start=False,

queue\_key='%(spider)s:requests',

queue\_cls='scrapy\_redis.queue.SpiderPriorityQueue',

dupefilter\_key='%(spider)s:dupefilter',

dupefilter\_cls='scrapy\_redis.dupefilter.RFPDupeFilter',

idle\_before\_close=0,

serializer=None):

"""Initialize scheduler.

Parameters

----------

server : Redis

The redis server instance.

persist : bool

Whether to flush requests when closing. Default is False.

flush\_on\_start : bool

Whether to flush requests on start. Default is False.

queue\_key : str

Requests queue key.

queue\_cls : str

Importable path to the queue class.

dupefilter\_key : str

Duplicates filter key.

dupefilter\_cls : str

Importable path to the dupefilter class.

idle\_before\_close : int

Timeout before giving up.

"""

if idle\_before\_close < 0:

raise TypeError("idle\_before\_close cannot be negative")

self.server = server

self.persist = persist

self.flush\_on\_start = flush\_on\_start

self.queue\_key = queue\_key

self.queue\_cls = queue\_cls

self.dupefilter\_cls = dupefilter\_cls

self.dupefilter\_key = dupefilter\_key

self.idle\_before\_close = idle\_before\_close

self.serializer = serializer

self.stats = None

def \_\_len\_\_(self):

return len(self.queue)

@classmethod

def from\_settings(cls, settings):

kwargs = {

'persist': settings.getbool('SCHEDULER\_PERSIST'),

'flush\_on\_start': settings.getbool('SCHEDULER\_FLUSH\_ON\_START'),

'idle\_before\_close': settings.getint('SCHEDULER\_IDLE\_BEFORE\_CLOSE'),

}

# If these values are missing, it means we want to use the defaults.

optional = {

# TODO: Use custom prefixes for this settings to note that are

# specific to scrapy-redis.

'queue\_key': 'SCHEDULER\_QUEUE\_KEY',

'queue\_cls': 'SCHEDULER\_QUEUE\_CLASS',

'dupefilter\_key': 'SCHEDULER\_DUPEFILTER\_KEY',

# We use the default setting name to keep compatibility.

'dupefilter\_cls': 'DUPEFILTER\_CLASS',

'serializer': 'SCHEDULER\_SERIALIZER',

}

for name, setting\_name in optional.items():

val = settings.get(setting\_name)

if val:

kwargs[name] = val

# Support serializer as a path to a module.

if isinstance(kwargs.get('serializer'), six.string\_types):

kwargs['serializer'] = importlib.import\_module(kwargs['serializer'])

server = connection.from\_settings(settings)

# Ensure the connection is working.

server.ping()

return cls(server=server, \*\*kwargs)

@classmethod

def from\_crawler(cls, crawler):

instance = cls.from\_settings(crawler.settings)

# FIXME: for now, stats are only supported from this constructor

instance.stats = crawler.stats

return instance

def open(self, spider):

self.spider = spider

try:

self.queue = load\_object(self.queue\_cls)(

server=self.server,

spider=spider,

key=self.queue\_key % {'spider': spider.name},

serializer=self.serializer,

)

except TypeError as e:

raise ValueError("Failed to instantiate queue class '%s': %s",

self.queue\_cls, e)

try:

self.df = load\_object(self.dupefilter\_cls)(

server=self.server,

key=self.dupefilter\_key % {'spider': spider.name},

debug=spider.settings.getbool('DUPEFILTER\_DEBUG'),

)

except TypeError as e:

raise ValueError("Failed to instantiate dupefilter class '%s': %s",

self.dupefilter\_cls, e)

if self.flush\_on\_start:

self.flush()

# notice if there are requests already in the queue to resume the crawl

if len(self.queue):

spider.log("Resuming crawl (%d requests scheduled)" % len(self.queue))

def close(self, reason):

if not self.persist:

self.flush()

def flush(self):

self.df.clear()

self.queue.clear()

def enqueue\_request(self, request):

if not request.dont\_filter and self.df.request\_seen(request):

self.df.log(request, self.spider)

return False

if self.stats:

self.stats.inc\_value('scheduler/enqueued/redis', spider=self.spider)

self.queue.push(request)

return True

def next\_request(self):

block\_pop\_timeout = self.idle\_before\_close

request = self.queue.pop(block\_pop\_timeout)

if request and self.stats:

self.stats.inc\_value('scheduler/dequeued/redis', spider=self.spider)

return request

def has\_pending\_requests(self):

return len(self) > 0

这个文件重写了scheduler类, 用来代替scrapy.core.scheduler的原有调度器. 其实对原有调度器的逻辑没有很大的改变, 主要是使用了redis作为数据存储的媒介, 以达到各个爬虫之间的统一调度. scheduler负责调度各个spider的request请求, scheduler初始化时, 通过settings文件读取queue和dupefilters的类型（一般就用上边默认的）, 配置queue和dupefilters使用的key（一般就是spider name加上queue或者dupefilters, 这样对于同一种spider的不同实例, 就会使用相同的数据块了）. 每当一个request要被调度时, enqueue\_request被调用, scheduler使用dupefilters来判断这个url是否重复, 如果不重复, 就添加到queue的容器中（先进先出, 先进后出和优先级都可以, 可以在settings中配置）. 当调度完成时, next\_request被调用, scheduler就通过queue容器的接口, 取出一个request, 把他发送给相应的spider, 让spider进行爬取工作.

## 源码分析参考: Spider

### spider.py

设计的这个spider从redis中读取要爬的url,然后执行爬取, 若爬取过程中返回更多的url, 那么继续进行直至所有的request完成. 之后继续从redis中读取url, 循环这个过程.

分析：在这个spider中通过connect signals.spider\_idle信号实现对crawler状态的监视. 当idle时, 返回新的make\_requests\_from\_url(url)给引擎, 进而交给调度器调度.

from scrapy import signals

from scrapy.exceptions import DontCloseSpider

from scrapy.spiders import Spider, CrawlSpider

from . import connection

# Default batch size matches default concurrent requests setting.

DEFAULT\_START\_URLS\_BATCH\_SIZE = 16

DEFAULT\_START\_URLS\_KEY = '%(name)s:start\_urls'

class RedisMixin(object):

"""Mixin class to implement reading urls from a redis queue."""

# Per spider redis key, default to DEFAULT\_START\_URLS\_KEY.

redis\_key = None

# Fetch this amount of start urls when idle. Default to DEFAULT\_START\_URLS\_BATCH\_SIZE.

redis\_batch\_size = None

# Redis client instance.

server = None

def start\_requests(self):

"""Returns a batch of start requests from redis."""

return self.next\_requests()

def setup\_redis(self, crawler=None):

"""Setup redis connection and idle signal.

This should be called after the spider has set its crawler object.

"""

if self.server is not None:

return

if crawler is None:

# We allow optional crawler argument to keep backwards

# compatibility.

# XXX: Raise a deprecation warning.

crawler = getattr(self, 'crawler', None)

if crawler is None:

raise ValueError("crawler is required")

settings = crawler.settings

if self.redis\_key is None:

self.redis\_key = settings.get(

'REDIS\_START\_URLS\_KEY', DEFAULT\_START\_URLS\_KEY,

)

self.redis\_key = self.redis\_key % {'name': self.name}

if not self.redis\_key.strip():

raise ValueError("redis\_key must not be empty")

if self.redis\_batch\_size is None:

self.redis\_batch\_size = settings.getint(

'REDIS\_START\_URLS\_BATCH\_SIZE', DEFAULT\_START\_URLS\_BATCH\_SIZE,

)

try:

self.redis\_batch\_size = int(self.redis\_batch\_size)

except (TypeError, ValueError):

raise ValueError("redis\_batch\_size must be an integer")

self.logger.info("Reading start URLs from redis key '%(redis\_key)s' "

"(batch size: %(redis\_batch\_size)s)", self.\_\_dict\_\_)

self.server = connection.from\_settings(crawler.settings)

# The idle signal is called when the spider has no requests left,

# that's when we will schedule new requests from redis queue

crawler.signals.connect(self.spider\_idle, signal=signals.spider\_idle)

def next\_requests(self):

"""Returns a request to be scheduled or none."""

use\_set = self.settings.getbool('REDIS\_START\_URLS\_AS\_SET')

fetch\_one = self.server.spop if use\_set else self.server.lpop

# XXX: Do we need to use a timeout here?

found = 0

while found < self.redis\_batch\_size:

data = fetch\_one(self.redis\_key)

if not data:

# Queue empty.

break

req = self.make\_request\_from\_data(data)

if req:

yield req

found += 1

else:

self.logger.debug("Request not made from data: %r", data)

if found:

self.logger.debug("Read %s requests from '%s'", found, self.redis\_key)

def make\_request\_from\_data(self, data):

# By default, data is an URL.

if '://' in data:

return self.make\_requests\_from\_url(data)

else:

self.logger.error("Unexpected URL from '%s': %r", self.redis\_key, data)

def schedule\_next\_requests(self):

"""Schedules a request if available"""

for req in self.next\_requests():

self.crawler.engine.crawl(req, spider=self)

def spider\_idle(self):

"""Schedules a request if available, otherwise waits."""

# XXX: Handle a sentinel to close the spider.

self.schedule\_next\_requests()

raise DontCloseSpider

class RedisSpider(RedisMixin, Spider):

"""Spider that reads urls from redis queue when idle."""

@classmethod

def from\_crawler(self, crawler, \*args, \*\*kwargs):

obj = super(RedisSpider, self).from\_crawler(crawler, \*args, \*\*kwargs)

obj.setup\_redis(crawler)

return obj

class RedisCrawlSpider(RedisMixin, CrawlSpider):

"""Spider that reads urls from redis queue when idle."""

@classmethod

def from\_crawler(self, crawler, \*args, \*\*kwargs):

obj = super(RedisCrawlSpider, self).from\_crawler(crawler, \*args, \*\*kwargs)

obj.setup\_redis(crawler)

return obj

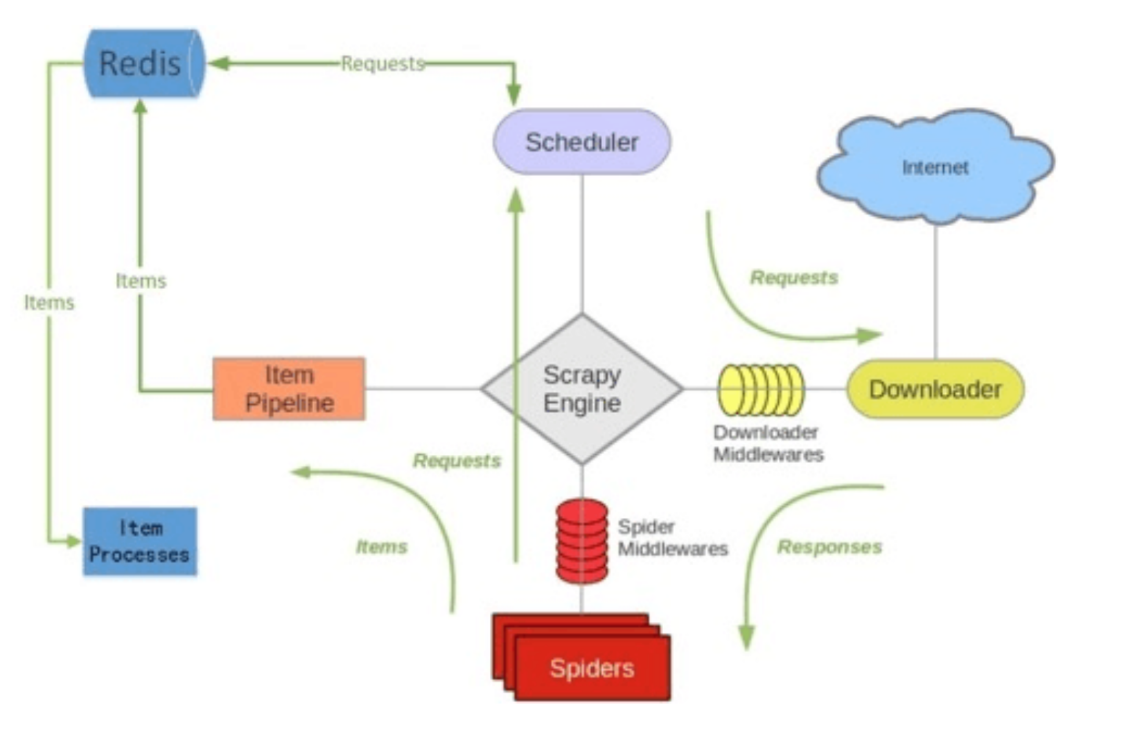
spider的改动也不是很大, 主要是通过connect接口, 给spider绑定了spider\_idle信号, spider初始化时, 通过setup\_redis函数初始化好和redis的连接, 之后通过next\_requests函数从redis中取出strat url, 使用的key是settings中REDIS\_START\_URLS\_AS\_SET定义的（注意了这里的初始化url池和我们上边的queue的url池不是一个东西, queue的池是用于调度的, 初始化url池是存放入口url的, 他们都存在redis中, 但是使用不同的key来区分, 就当成是不同的表吧）, spider使用少量的start url, 可以发展出很多新的url, 这些url会进入scheduler进行判重和调度. 直到spider跑到调度池内没有url的时候, 会触发spider\_idle信号, 从而触发spider的next\_requests函数, 再次从redis的start url池中读取一些url.

## 总结

最后总结一下scrapy-redis的总体思路：这个工程通过重写scheduler和spider类, 实现了调度、spider启动和redis的交互. 实现新的dupefilter和queue类, 达到了判重和调度容器和redis的交互, 因为每个主机上的爬虫进程都访问同一个redis数据库, 所以调度和判重都统一进行统一管理, 达到了分布式爬虫的目的. 当spider被初始化时, 同时会初始化一个对应的scheduler对象, 这个调度器对象通过读取settings, 配置好自己的调度容器queue和判重工具dupefilter. 每当一个spider产出一个request的时候, scrapy内核会把这个reuqest递交给这个spider对应的scheduler对象进行调度, scheduler对象通过访问redis对request进行判重, 如果不重复就把他添加进redis中的调度池. 当调度条件满足时, scheduler对象就从redis的调度池中取出一个request发送给spider, 让他爬取. 当spider爬取的所有暂时可用url之后, scheduler发现这个spider对应的redis的调度池空了, 于是触发信号spider\_idle, spider收到这个信号之后, 直接连接redis读取strart url池, 拿去新的一批url入口, 然后再次重复上边的工作.

# 7. scrapy-redis实战

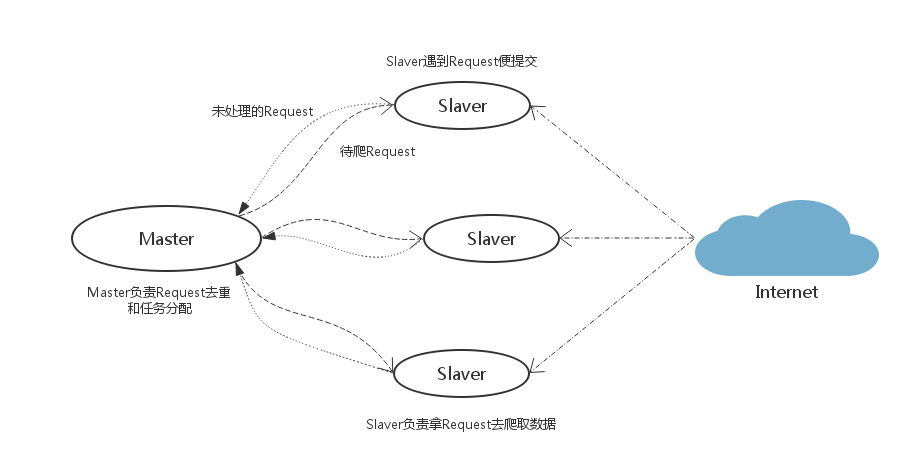
## 从零搭建Redis-Scrapy分布式爬虫



### Scrapy-Redis分布式策略：

假设有四台电脑：Windows 10、Mac OS X、Ubuntu 16.04、CentOS 7.2, 任意一台电脑都可以作为 Master端 或 Slaver端, 比如：

* Master端(核心服务器) ：使用 Windows 10, 搭建一个Redis数据库, 不负责爬取, 只负责url指纹判重、Request的分配, 以及数据的存储
* Slaver端(爬虫程序执行端) ：使用 Mac OS X 、Ubuntu 16.04、CentOS 7.2, 负责执行爬虫程序, 运行过程中提交新的Request给Master



1. 首先Slaver端从Master端拿任务（Request、url）进行数据抓取, Slaver抓取数据的同时, 产生新任务的Request便提交给 Master 处理；
2. Master端只有一个Redis数据库, 负责将未处理的Request去重和任务分配, 将处理后的Request加入待爬队列, 并且存储爬取的数据.

Scrapy-Redis默认使用的就是这种策略, 我们实现起来很简单, 因为任务调度等工作Scrapy-Redis都已经帮我们做好了, 我们只需要继承RedisSpider、指定redis\_key就行了.

redis\_key是统一管理slaver端爬取的指令.

缺点是, Scrapy-Redis调度的任务是Request对象, 里面信息量比较大（不仅包含url, 还有callback函数、headers等信息）, 可能导致的结果就是会降低爬虫速度、而且会占用Redis大量的存储空间, 所以如果要保证效率, 那么就需要一定硬件水平.

### 一、安装Redis

安装Redis：<http://redis.io/download>

安装完成后, 拷贝一份Redis安装目录下的redis.conf到任意目录, 建议保存到：/etc/redis/redis.conf（Windows系统可以无需变动)

## Installation

Download, extract and compile Redis with:

$ wget http://download.redis.io/releases/redis-3.2.9.tar.gz

$ tar xzf redis-3.2.9.tar.gz

$ cd redis-3.2.9

$ make

The binaries that are now compiled are available in the src directory. Run Redis with:

$ src/redis-server

You can interact with Redis using the built-in client:

$ src/redis-cli

redis> set foo bar

OK

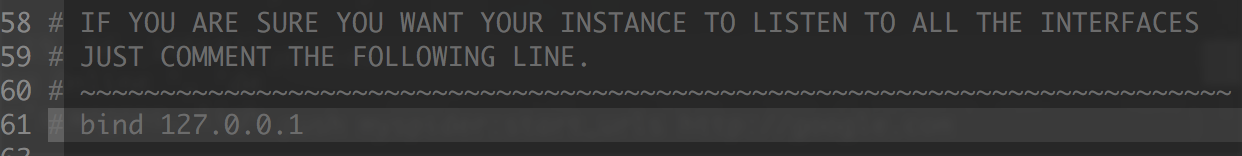
redis> get foo

### 二、修改配置文件 redis.conf

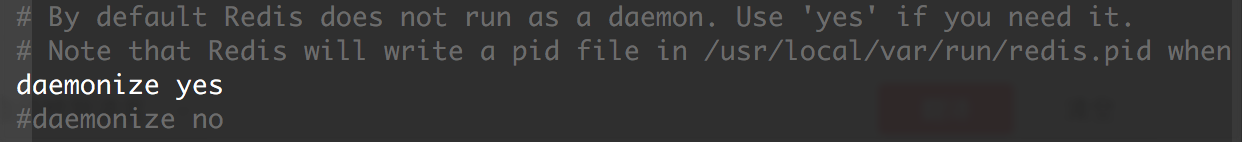
打开你的redis.conf配置文件, 示例:

* 非Windows系统: sudo vi /etc/redis/redis.conf
* Windows系统：C:\Intel\Redis\conf\redis.conf

1. Master端redis.conf里注释bind 127.0.0.1, Slave端才能远程连接到Master端的Redis数据库.



* + daemonize yno表示Redis默认不作为守护进程运行, 即在运行redis-server /etc/redis/redis.conf时, 将显示Redis启动提示画面；
    - daemonize yes则默认后台运行, 不必重新启动新的终端窗口执行其他命令, 看个人喜好和实际需要.



windows用户必须要以管理员模式运行命令行

redis-server C:\Intel\Redis\conf\redis.conf

redis-cli

如果显示redis 127.0.0.1:6379>, 则表示服务已经启动了.

set key1 "hello world"

get key1

想要在其它电脑上连接本地的redis数据库, 首先要知道本地的IP地址, ipconfig, 只要两台电脑之间能够ping通, 就能够连接. 假设本机ip为192.168.21.64, 在远程电脑上使用以下命令

sudo redis-cl -h 192.168.21.64

显示

192.168.21.64:6379>

get key1

此时就能连接到本机数据库, 显示hello world

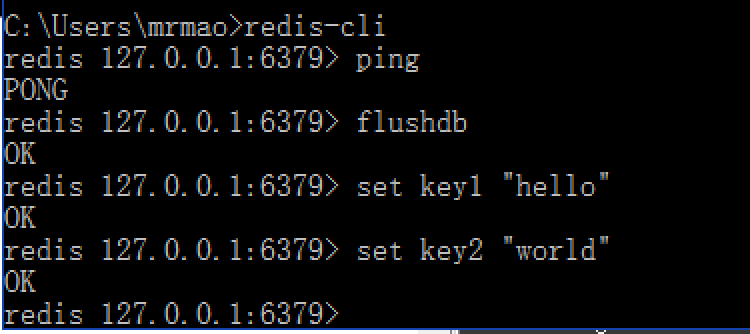
flushdb

清空数据库

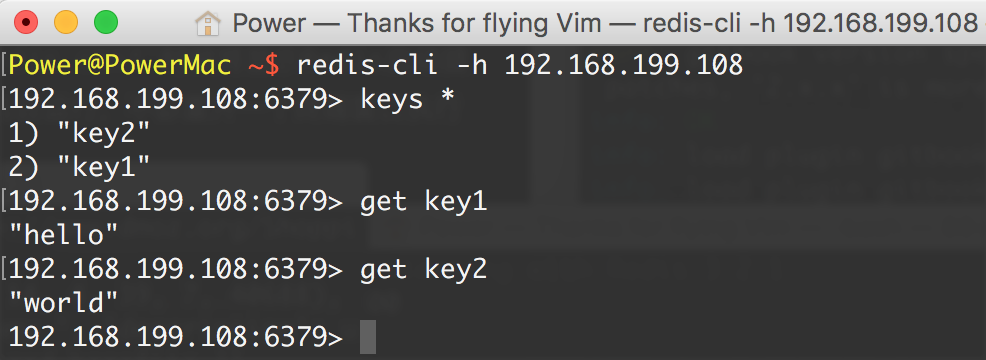
### 三、测试Slave端远程连接Master端

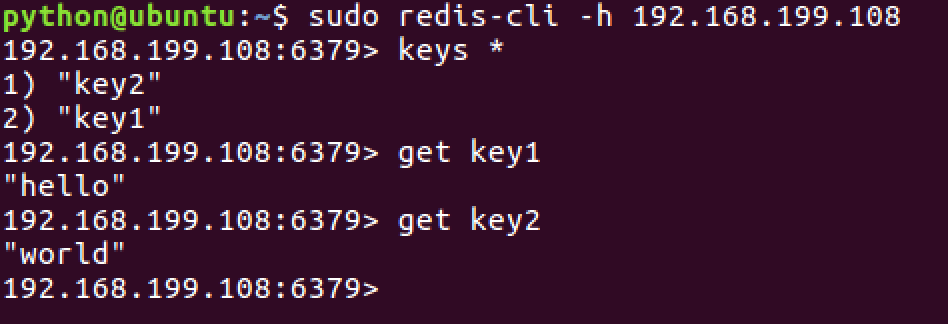
测试中, Master端Windows 10 的IP地址为：192.168.199.108

1. Master端按指定配置文件启动 redis-server, 示例：
   * 非Windows系统：sudo redis-server /etc/redis/redis/conf
   * Windows系统：命令提示符(管理员)模式下执行 redis-server C:\Intel\Redis\conf\redis.conf读取默认配置即可.
2. Master端启动本地redis-cli：



1. slave端启动redis-cli -h 192.168.199.108, -h 参数表示连接到指定主机的redis数据库



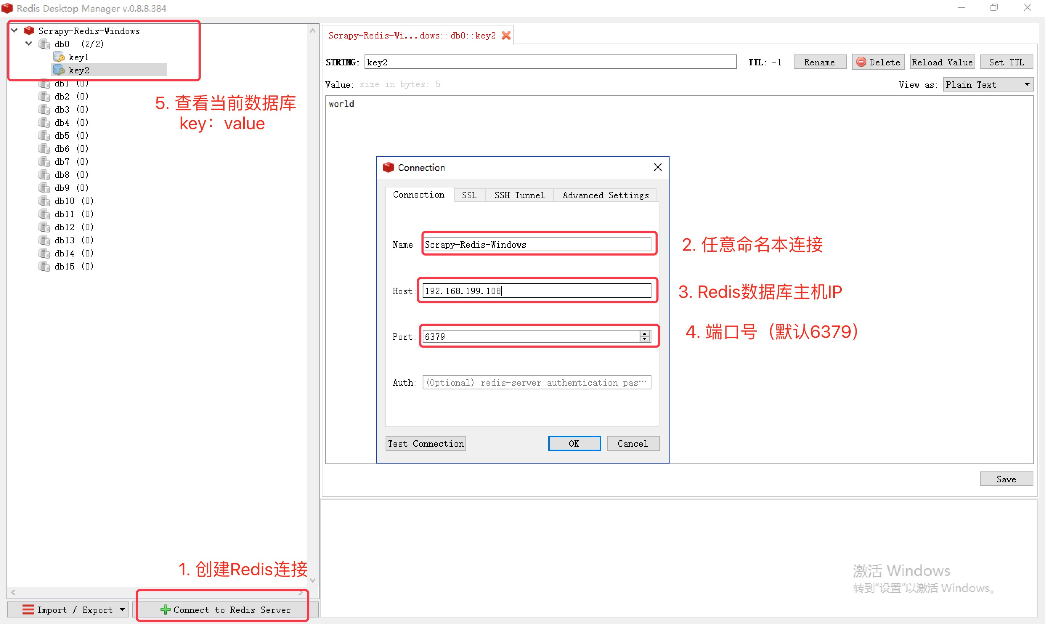


注意：Slave端无需启动redis-server, Master端启动即可. 只要 Slave 端读取到了 Master 端的 Redis 数据库, 则表示能够连接成功, 可以实施分布式.

### 四、Redis数据库桌面管理工具

这里推荐 Redis Desktop Manager, 支持 Windows、Mac OS X、Linux 等平台：

下载地址：<https://redisdesktop.com/download>



### 五、安装scrapy-redis

pip install scrapy-redis

## 源码自带项目说明

### 使用scrapy-redis的example来修改

先从github上拿到scrapy-redis的示例, 然后将里面的example-project目录移到指定的地址：

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例, 改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapyredis-project

我们clone到的 scrapy-redis 源码中有自带一个example-project项目, 这个项目包含3个spider, 分别是dmoz, myspider\_redis, mycrawler\_redis.

注意example-project的名字可以任意修改, 但是其中example文件夹不能随意修改, 一旦修改了example文件夹, 就要在example-project文件夹下的scrapy.cfg文件中修改对应的文件夹名称. example/settings.py中的SPIDER\_MODULES的名称也要修改.

### 一、dmoz (class DmozSpider(CrawlSpider))

这个爬虫继承的是CrawlSpider, 它是用来说明Redis的持续性, 当我们第一次运行dmoz爬虫, 然后Ctrl + C停掉之后, 再运行dmoz爬虫, 之前的爬取记录是保留在Redis里的.

分析起来, 其实这就是一个 scrapy-redis 版 CrawlSpider 类, 需要设置Rule规则, 以及callback不能写parse()方法.

实际上就是一个Crawl Scrapy的项目, 爬取<http://www.dmoz.org/>网站上的分类目录

##### items文件

**from** scrapy.item **import** Item, Field

**from** scrapy.loader **import** ItemLoader

**from** scrapy.loader.processors **import** MapCompose, TakeFirst, Join

*#定义了多个字段*

**class** ExampleItem(Item):

name = Field()

description = Field()

link = Field()

crawled = Field()

spider = Field()

url = Field()

*#使用ItemLoader方法, 与item方法有些许的不同*

**class** ExampleLoader(ItemLoader):

default\_item\_class = ExampleItem

default\_input\_processor = MapCompose(**lambda** s: s.strip())

default\_output\_processor = TakeFirst()

description\_out = Join()

##### pipeline文件

**from** datetime **import** datetime

**class** ExamplePipeline(object):

**def** process\_item(self, item, spider):

*#数据给过管道后会添加以下两个字段*

*#添加了linux的时间戳, 即数据获取的时间, UTC时间, 全球标准时间. 英国格林威治的时间. 中国时间为utc时间+8*

item[**"crawled"**] = datetime.utcnow()

*#添加了一个爬虫名的字段, 有很多个爬虫, 给数据添加一个爬虫名的字段后就知道是哪个爬虫爬取的了.*

item[**"spider"**] = spider.name

**return** item

##### settings文件

*#爬虫位置*

SPIDER\_MODULES = [**'example.spiders'**]

NEWSPIDER\_MODULE = **'example.spiders'**

*#默认的user\_agent, 要进行修改*

USER\_AGENT = **'scrapy-redis (+https://github.com/rolando/scrapy-redis)'**

*#使用了scrapy\_redis中的去重方法*

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

*#使用了scrapy-redis中的调度组件*

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

*#允许执行项目时可以暂停, 暂停后不会清空redis数据库的记录*

SCHEDULER\_PERSIST = **True**

*#如果三个都注释掉, 就使用默认的优先级策略. 默认的是scrapy请求队列形式, 是按优先级顺序排列的队列形式, 一般来说使用把三个都注释掉, 使用默认的即可. 出队列的顺序不影响我们的结果.*

*#在使用redis分布式的爬虫时, 如果使用scrapy的请求队列形式, 虽然不影响请求的发送, 但在redis数据库中却不会显示出请求队列. 必须使用redis的三种请求队列之一才能在redis数据库中显示出请求队列.*

*#按sorted排序顺序出队列*

SCHEDULER\_QUEUE\_CLASS = **"scrapy\_redis.queue.SpiderPriorityQueue"**

*#队列形式, 规则是请求先进先出*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"*

*#栈形式, 规则是请求先进后出*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"*

ITEM\_PIPELINES = {

*#自定义的pipelines, 先经过我们自定义的pipeline, 添加了utc时间和爬虫名. 可以自定义多个管道文件, 经过一系列的处理, 最后再放到redis数据库中.*

**'example.pipelines.ExamplePipeline'**: 300,

*#scrapy\_redis.pipelines.RedisPipeline将数据存储在Redis数据库中, 必须要启动, 如果不想把数据存储在Redis数据库中, 也可以使用自定义的pipeline进行数据的存储. 此时把放在redis数据库的优先级设置到最后.*

**'scrapy\_redis.pipelines.RedisPipeline'**: 400,

}

*#定义数据存储的位置, 如果不写这两个参数, 则默认是保存在本地的.*

*#指定数据库的主机IP*

REDIS\_HOST = **"111.186.110.33"**

*#指定远程数据库的端口号*

REDIS\_PORT = 6379

*#LOG日志*

LOG\_LEVEL = **'DEBUG'**

*# Introduce an artifical delay to make use of parallelism. to speed up the*

*# crawl.*

*#下载延迟*

DOWNLOAD\_DELAY = 2

##### dmoz爬虫文件

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

*#一个简单的CrawlSpider*

*#使用的是CralSpider的爬虫, 不是redis的爬虫, 故不支持分布式.*

**class** DmozSpider(CrawlSpider):

*"""Follow categories and extract links."""*

name = **'dmoz'**

allowed\_domains = [**'dmoz.org'**]

start\_urls = [**'http://www.dmoz.org/'**]

rules = [

Rule(LinkExtractor(

restrict\_css=(**'.top-cat'**, **'.sub-cat'**, **'.cat-item'**)

), callback=**'parse\_directory'**, follow=**True**),

]

**def** parse\_directory(self, response):

**for** div **in** response.css(**'.title-and-desc'**):

**yield** {

**'name'**: div.css(**'.site-title::text'**).extract\_first(),

**'description'**: div.css(**'.site-descr::text'**).extract\_first().strip(),

**'link'**: div.css(**'a::attr(href)'**).extract\_first(),

}

**main.py**

from scrapy import cmdline

cmdline.execute('scrapy crawl dmoz'.split())

##### 执行dmoz爬虫

**执行方式：scrapy crawl dmoz**

### 二、myspider\_redis (class MySpider(RedisSpider))

这个爬虫继承了RedisSpider, 它能够支持分布式的抓取, 采用的是basic spider, 需要写parse函数.

其次就是不再有start\_urls了, 取而代之的是redis\_key, scrapy-redis将key从Redis里pop出来, 成为请求的url地址.

**from** scrapy\_redis.spiders **import** RedisSpider

**class** MySpider(RedisSpider):

*"""Spider that reads urls from redis queue (myspider:start\_urls)."""*

name = **'myspider\_redis'**

*#启动所有slaver端爬虫的指令, 下面的格式是参考格式, 建议采用这种格式*

*#myspider为当前爬虫类的名字*

*#使用lpush myspider:start\_urls http://www.dmoz.org/命令来启动爬虫*

redis\_key = **'myspider:start\_urls'**

*#指定爬取的域范围*

*# allowd\_domains = ['dmoz.org']*

*# 可选：等效于allowd\_domains(), \_\_init\_\_方法按规定格式写, 使用时只需要修改super()里的类名参数即可*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

*#通过执行时输入的start\_urls动态的获取爬取的域的范围*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

*# 修改这里的类名为当前类名, super相当于调用了父类的初始化方法*

super(MySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

**def** parse(self, response):

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

**main.py**

from scrapy import cmdline

cmdline.execute('scrapy runspider myspider\_redis.py'.split())

}

注意：

RedisSpider类 不需要写allowd\_domains和start\_urls：

1. scrapy-redis将从在构造方法\_\_init\_\_()里动态定义爬虫爬取域范围, 也可以选择直接写allowd\_domains.
2. 必须指定redis\_key, 即启动爬虫的命令, 参考格式：redis\_key = 'myspider:start\_urls'
3. 根据指定的格式, start\_urls将在 Master端的 redis-cli 里 lpush 到 Redis数据库里, RedisSpider 将在数据库里获取start\_urls.

执行方式：

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条）, 爬虫（们）将处于等待准备状态：

scrapy runspider myspider\_redis.py

1. 在Master端的redis-cli输入push指令, 参考格式：

$redis > lpush myspider:start\_urls <http://www.dmoz.org/>

只要是连接到这个redis数据库上的slaver端, 都会开始分配url并且执行爬取任务. 是由master端通过redis数据库进行任务的分配的.

1. Slaver端爬虫获取到请求, 开始爬取.

### 三、mycrawler\_redis (class MyCrawler(RedisCrawlSpider))

这个RedisCrawlSpider类爬虫继承了RedisCrawlSpider, 能够支持分布式的抓取. 因为采用的是crawlSpider, 所以需要遵守Rule规则, 以及callback不能写parse()方法.

同样也不再有start\_urls了, 取而代之的是redis\_key, scrapy-redis将key从Redis里pop出来, 成为请求的url地址.

**from** scrapy.spiders **import** Rule

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

**class** MyCrawler(RedisCrawlSpider):

*"""Spider that reads urls from redis queue (myspider:start\_urls)."""*

name = **'mycrawler\_redis'**

redis\_key = **'mycrawler:start\_urls'**

rules = (

*# follow all links,可以写多个Rule, callback的名字不能是parse*

Rule(LinkExtractor(), callback=**'parse\_page'**, follow=**True**),

)

*# \_\_init\_\_方法必须按规定写, 使用时只需要修改super()里的类名参数即可*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

*# 修改这里的类名为当前类名*

super(MyCrawler, self).\_\_init\_\_(\*args, \*\*kwargs)

**def** parse\_page(self, response):

**return** {

**'name'**: response.css(**'title::text'**).extract\_first(),

**'url'**: response.url,

}

注意：

同样的, RedisCrawlSpider类不需要写allowd\_domains和start\_urls：

1. scrapy-redis将从在构造方法\_\_init\_\_()里动态定义爬虫爬取域范围, 也可以选择直接写allowd\_domains.
2. 必须指定redis\_key, 即启动爬虫的命令, 参考格式：redis\_key = 'myspider:start\_urls'
3. 根据指定的格式, start\_urls将在 Master端的 redis-cli 里 lpush 到 Redis数据库里, RedisSpider 将在数据库里获取start\_urls.

**执行方式：**

1. 通过runspider方法执行爬虫的py文件（也可以分次执行多条）, 爬虫（们）将处于等待准备状态：

scrapy runspider mycrawler\_redis.py

1. 在Master端的redis-cli输入push指令, 参考格式：

$redis > lpush mycrawler:start\_urls http://www.dmoz.org/

1. 爬虫获取url, 开始执行.

### 总结：

1. 如果只是用到Redis的去重和保存功能, 就选第一种；
2. 如果要写分布式, 则根据情况, 选择第二种、第三种；
3. 通常情况下, 会选择用第三种方式编写深度聚焦爬虫.

## 有缘网分布式爬虫项目1

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例, 改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapy-youyuan

### 修改settings.py

下面列举了修改后的配置文件中与scrapy-redis有关的部分, middleware、proxy等内容在此就省略了.

# -\*- coding: utf-8 -\*-

# 指定使用scrapy-redis的调度器

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# 指定使用scrapy-redis的去重

DUPEFILTER\_CLASS = 'scrapy\_redis.dupefilters.RFPDupeFilter'

# 指定排序爬取地址时使用的队列,

# 默认的 按优先级排序(Scrapy默认), 由sorted set实现的一种非FIFO、LIFO方式.

SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.SpiderPriorityQueue'

# 可选的 按先进先出排序（FIFO）

# SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.SpiderQueue'

# 可选的 按后进先出排序（LIFO）

# SCHEDULER\_QUEUE\_CLASS = 'scrapy\_redis.queue.SpiderStack'

# 在redis中保持scrapy-redis用到的各个队列, 从而允许暂停和暂停后恢复, 也就是不清理redis queues

SCHEDULER\_PERSIST = True

# 只在使用SpiderQueue或者SpiderStack是有效的参数, 指定爬虫关闭的最大间隔时间

# SCHEDULER\_IDLE\_BEFORE\_CLOSE = 10

# 通过配置RedisPipeline将item写入key为 spider.name : items 的redis的list中, 供后面的分布式处理item

# 这个已经由 scrapy-redis 实现, 不需要我们写代码

ITEM\_PIPELINES = {

'example.pipelines.ExamplePipeline': 300,

'scrapy\_redis.pipelines.RedisPipeline': 400

}

# 指定redis数据库的连接参数

# REDIS\_PASS是我自己加上的redis连接密码（默认不做）

REDIS\_HOST = '127.0.0.1'

REDIS\_PORT = 6379

#REDIS\_PASS = 'redisP@ssw0rd'

# LOG等级

LOG\_LEVEL = 'DEBUG'

#默认情况下,RFPDupeFilter只记录第一个重复请求. 将DUPEFILTER\_DEBUG设置为True会记录所有重复的请求.

DUPEFILTER\_DEBUG =True

# 覆盖默认请求头, 可以自己编写Downloader Middlewares设置代理和UserAgent

DEFAULT\_REQUEST\_HEADERS = {

'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',

'Accept-Language': 'zh-CN,zh;q=0.8',

'Connection': 'keep-alive',

'Accept-Encoding': 'gzip, deflate, sdch'

}

### 查看pipeline.py

# -\*- coding: utf-8 -\*-

from datetime import datetime

class ExamplePipeline(object):

def process\_item(self, item, spider):

#utcnow() 是获取UTC时间

item["crawled"] = datetime.utcnow()

# 爬虫名

item["spider"] = spider.name

return item

### 修改items.py

增加我们最后要保存的youyuanItem项, 这里只写出来一个非常简单的版本

# -\*- coding: utf-8 -\*-

from scrapy.item import Item, Field

class youyuanItem(Item):

# 个人头像链接

header\_url = Field()

# 用户名

username = Field()

# 内心独白

monologue = Field()

# 相册图片链接

pic\_urls = Field()

# 年龄

age = Field()

# 网站来源 youyuan

source = Field()

# 个人主页源url

source\_url = Field()

# 获取UTC时间

crawled = Field()

# 爬虫名

spider = Field()

### 编写 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫, 之后就可以运行爬虫了. 这里的提供一个简单的版本：

# -\*- coding:utf-8 -\*-

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

# 使用redis去重

from scrapy.dupefilters import RFPDupeFilter

from example.items import youyuanItem

import re

#

class YouyuanSpider(CrawlSpider):

name = 'youyuan'

allowed\_domains = ['youyuan.com']

# 有缘网的列表页

start\_urls = ['http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/']

# 搜索页面匹配规则, 根据response提取链接

list\_page\_lx = LinkExtractor(allow=(r'http://www.youyuan.com/find/.+'))

# 北京、18~25岁、女性 的 搜索页面匹配规则, 根据response提取链接

page\_lx = LinkExtractor(allow =(r'http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/'))

# 个人主页 匹配规则, 根据response提取链接

profile\_page\_lx = LinkExtractor(allow=(r'http://www.youyuan.com/\d+-profile/'))

rules = (

# 匹配find页面, 跟进链接, 跳板

Rule(list\_page\_lx, follow=True),

# 匹配列表页成功, 跟进链接, 跳板

Rule(page\_lx, follow=True),

# 匹配个人主页的链接, 形成request保存到redis中等待调度, 一旦有响应则调用parse\_profile\_page()回调函数处理, 不做继续跟进

Rule(profile\_page\_lx, callback='parse\_profile\_page', follow=False),

)

# 处理个人主页信息, 得到我们要的数据

def parse\_profile\_page(self, response):

item = youyuanItem()

item['header\_url'] = self.get\_header\_url(response)

item['username'] = self.get\_username(response)

item['monologue'] = self.get\_monologue(response)

item['pic\_urls'] = self.get\_pic\_urls(response)

item['age'] = self.get\_age(response)

item['source'] = 'youyuan'

item['source\_url'] = response.url

#print "Processed profile %s" % response.url

yield item

# 提取头像地址

def get\_header\_url(self, response):

header = response.xpath('//dl[@class=\'personal\_cen\']/dt/img/@src').extract()

if len(header) > 0:

header\_url = header[0]

else:

header\_url = ""

return header\_url.strip()

# 提取用户名

def get\_username(self, response):

usernames = response.xpath("//dl[@class=\'personal\_cen\']/dd/div/strong/text()").extract()

if len(usernames) > 0:

username = usernames[0]

else:

username = "NULL"

return username.strip()

# 提取内心独白

def get\_monologue(self, response):

monologues = response.xpath("//ul[@class=\'requre\']/li/p/text()").extract()

if len(monologues) > 0:

monologue = monologues[0]

else:

monologue = "NULL"

return monologue.strip()

# 提取相册图片地址

def get\_pic\_urls(self, response):

pic\_urls = []

data\_url\_full = response.xpath('//li[@class=\'smallPhoto\']/@data\_url\_full').extract()

if len(data\_url\_full) <= 1:

pic\_urls.append("");

else:

for pic\_url in data\_url\_full:

pic\_urls.append(pic\_url)

if len(pic\_urls) <= 1:

return "NULL"

# 每个url用|分隔

return '|'.join(pic\_urls)

# 提取年龄

def get\_age(self, response):

age\_urls = response.xpath("//dl[@class=\'personal\_cen\']/dd/p[@class=\'local\']/text()").extract()

if len(age\_urls) > 0:

age = age\_urls[0]

else:

age = "0"

age\_words = re.split(' ', age)

if len(age\_words) <= 2:

return "0"

age = age\_words[2][:-1]

# 从age字符串开始匹配数字, 失败返回None

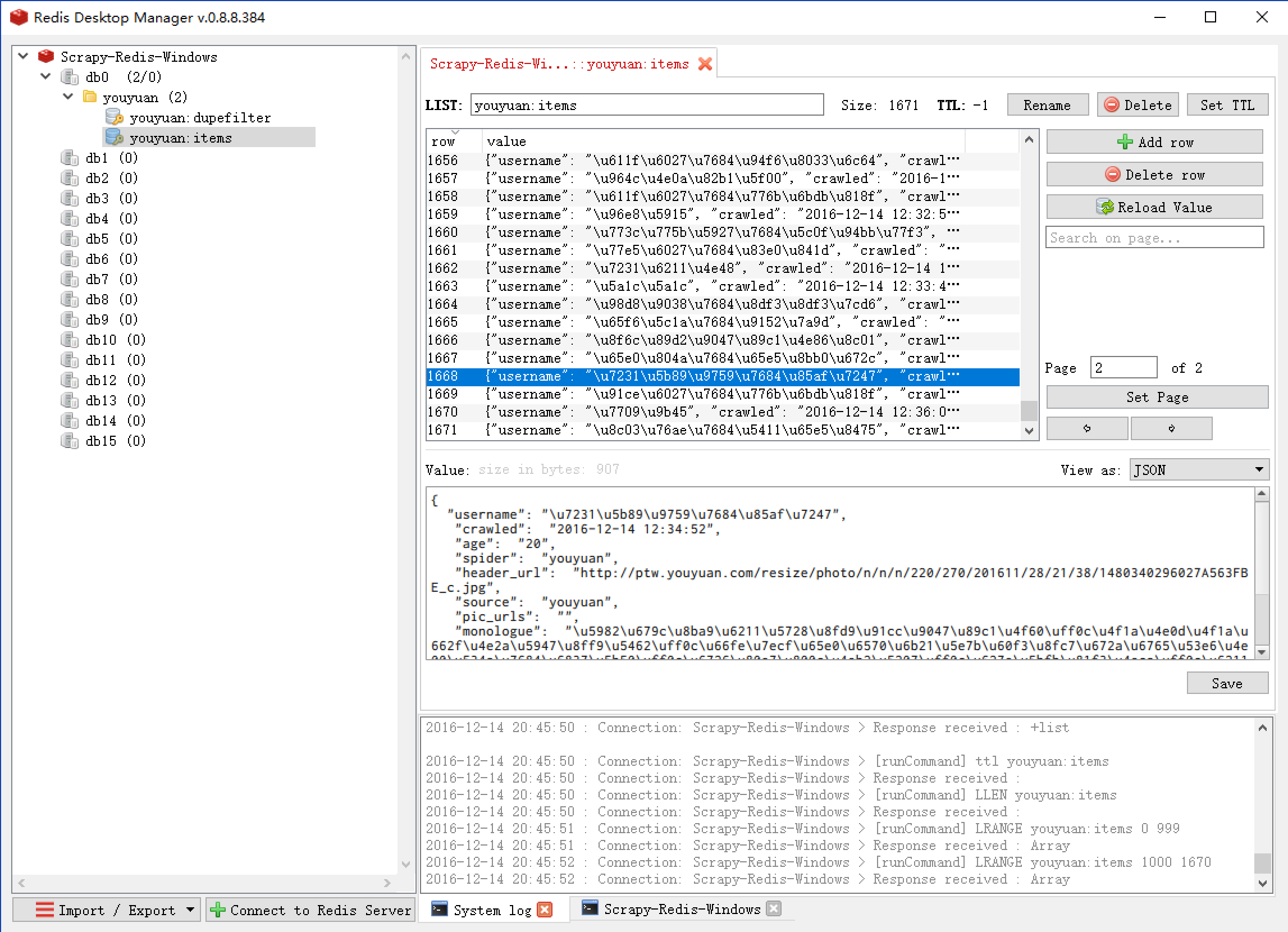
if re.compile(r'[0-9]').match(age):

return age

return "0"

### 运行程序：

1. Master端打开 Redis： redis-server
2. Slave端直接运行爬虫： scrapy crawl youyuan
3. 多个Slave端运行爬虫顺序没有限制.



将项目修改成 RedisCrawlSpider 类的分布式爬虫, 并尝试在多个Slave端运行.

## 有缘网分布式爬虫项目2

### 修改 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫, 使其具有分布式：

# -\*- coding:utf-8 -\*-

from scrapy.linkextractors import LinkExtractor

#from scrapy.spiders import CrawlSpider, Rule

# 1. 导入RedisCrawlSpider类, 不使用CrawlSpider

from scrapy\_redis.spiders import RedisCrawlSpider

from scrapy.spiders import Rule

from scrapy.dupefilters import RFPDupeFilter

from example.items import youyuanItem

import re

# 2. 修改父类 RedisCrawlSpider

# class YouyuanSpider(CrawlSpider):

class YouyuanSpider(RedisCrawlSpider):

name = 'youyuan'

# 3. 取消 allowed\_domains() 和 start\_urls

##### allowed\_domains = ['youyuan.com']

##### start\_urls = ['http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/']

# 4. 增加redis-key

redis\_key = 'youyuan:start\_urls'

list\_page\_lx = LinkExtractor(allow=(r'http://www.youyuan.com/find/.+'))

page\_lx = LinkExtractor(allow =(r'http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/'))

profile\_page\_lx = LinkExtractor(allow=(r'http://www.youyuan.com/\d+-profile/'))

rules = (

Rule(list\_page\_lx, follow=True),

Rule(page\_lx, follow=True),

Rule(profile\_page\_lx, callback='parse\_profile\_page', follow=False),

)

# 5. 增加\_\_init\_\_()方法, 动态获取allowed\_domains()

def \_\_init\_\_(self, \*args, \*\*kwargs):

domain = kwargs.pop('domain', '')

self.allowed\_domains = filter(None, domain.split(','))

super(youyuanSpider, self).\_\_init\_\_(\*args, \*\*kwargs)

# 处理个人主页信息, 得到我们要的数据

def parse\_profile\_page(self, response):

item = youyuanItem()

item['header\_url'] = self.get\_header\_url(response)

item['username'] = self.get\_username(response)

item['monologue'] = self.get\_monologue(response)

item['pic\_urls'] = self.get\_pic\_urls(response)

item['age'] = self.get\_age(response)

item['source'] = 'youyuan'

item['source\_url'] = response.url

yield item

# 提取头像地址

def get\_header\_url(self, response):

header = response.xpath('//dl[@class=\'personal\_cen\']/dt/img/@src').extract()

if len(header) > 0:

header\_url = header[0]

else:

header\_url = ""

return header\_url.strip()

# 提取用户名

def get\_username(self, response):

usernames = response.xpath("//dl[@class=\'personal\_cen\']/dd/div/strong/text()").extract()

if len(usernames) > 0:

username = usernames[0]

else:

username = "NULL"

return username.strip()

# 提取内心独白

def get\_monologue(self, response):

monologues = response.xpath("//ul[@class=\'requre\']/li/p/text()").extract()

if len(monologues) > 0:

monologue = monologues[0]

else:

monologue = "NULL"

return monologue.strip()

# 提取相册图片地址

def get\_pic\_urls(self, response):

pic\_urls = []

data\_url\_full = response.xpath('//li[@class=\'smallPhoto\']/@data\_url\_full').extract()

if len(data\_url\_full) <= 1:

pic\_urls.append("");

else:

for pic\_url in data\_url\_full:

pic\_urls.append(pic\_url)

if len(pic\_urls) <= 1:

return "NULL"

return '|'.join(pic\_urls)

# 提取年龄

def get\_age(self, response):

age\_urls = response.xpath("//dl[@class=\'personal\_cen\']/dd/p[@class=\'local\']/text()").extract()

if len(age\_urls) > 0:

age = age\_urls[0]

else:

age = "0"

age\_words = re.split(' ', age)

if len(age\_words) <= 2:

return "0"

age = age\_words[2][:-1]

if re.compile(r'[0-9]').match(age):

return age

return "0"

### 分布式爬虫执行方式：

#### 6. 在Master端启动redis-server：

redis-server

#### 7. 在Slave端分别启动爬虫, 不分先后：

scrapy runspider youyuan.py

#### 8. 在Master端的redis-cli里push一个start\_urls

redis-cli> lpush youyuan:start\_urls http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/

#### 9. 爬虫启动, 查看redis数据库数据.

## 视频演示 有缘网

## 第1步：有缘网爬虫CrawlSpider版写入到本地文件

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例, 改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapy-youyuan

或者使用命令从头开始创建项目

scrapy startproject youyuan

cd youyuan/spiders

scrapy genspider -t crawl yy "youyuan.com"

### 修改settings.py

下面列举了修改后的配置文件中与scrapy-redis有关的部分, middleware、proxy等内容在此就省略了.

非分布式的版本可以使用redis的方法和redis数据库, 但是没有分布式的功能.

# -\*- coding: utf-8 -\*-

BOT\_NAME = **'youyuan'**

SPIDER\_MODULES = [**'youyuan.spiders'**]

NEWSPIDER\_MODULE = **'youyuan.spiders'**

*#允许执行项目时可以暂停, 暂停后不会清空redis数据库的记录*

*# 在redis中保持scrapy-redis用到的各个队列, 从而允许暂停和暂停后恢复, 也就是不清理redis queues*

SCHEDULER\_PERSIST = **True**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#USER\_AGENT = 'youyuan (+http://www.yourdomain.com)'*

*# Obey robots.txt rules*

#ROBOTSTXT\_OBEY = **True**

*# Configure a delay for requests for the same website (default: 0)*

*# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay*

*# See also autothrottle settings and docs*

DOWNLOAD\_DELAY = 3

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

COOKIES\_ENABLED = **False**

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# 覆盖默认请求头, 可以自己编写Downloader Middlewares设置代理和UserAgent*

DEFAULT\_REQUEST\_HEADERS = {

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8'**,

**'Accept-Language'**: **'zh-CN,zh;q=0.8'**,

**'Connection'**: **'keep-alive'**,

**'Accept-Encoding'**: **'gzip, deflate, sdch'**

}

*# Configure item pipelines*

*# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html*

*# 通过配置RedisPipeline将item写入key为 spider.name : items 的redis的list中, 供后面的分布式处理item*

*# 这个已经由 scrapy-redis 实现, 不需要我们写代码*

ITEM\_PIPELINES = {

**'youyuan.pipelines.YouyuanPipeline'**: 300,

}

*# LOG等级*

#LOG\_LEVEL = **'DEBUG'**

*#默认情况下,RFPDupeFilter只记录第一个重复请求. 将DUPEFILTER\_DEBUG设置为True会记录所有重复的请求.*

#DUPEFILTER\_DEBUG =**True**

### 查看pipeline.py

**写入到本地文件**

**import** json

**class** YouyuanPipeline(object):

**def** \_\_init\_\_(self):

*#self.lists = []*

self.filename = open(**"youyuan.json"**, **"w"**)

**def** process\_item(self, item, spider):

content = json.dumps(dict(item), ensure\_ascii=**False**) + **"\n"**

self.filename.write(content.encode(**"utf-8"**))

*#self.lists.append(content.encode("utf-8"))*

**return** item

**def** close\_spider(self, spider):

self.filename.close()

### 修改items.py

增加我们最后要保存的youyuanItem项, 这里只写出来一个非常简单的版本

# -\*- coding: utf-8 -\*-

from scrapy.item import Item, Field

class youyuanItem(Item):

# 个人头像链接

header\_url = Field()

# 用户名

username = Field()

# 内心独白

monologue = Field()

# 相册图片链接

pic\_urls = Field()

# 年龄

age = Field()

# 网站来源 youyuan

source = Field()

# 个人主页源url

source\_url = Field()

### 编写 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫, 之后就可以运行爬虫了. 这里的提供一个简单的版本：

# -\*- coding:utf-8 -\*-

**import** scrapy, re

**from** scrapy.linkextractors **import** LinkExtractor

**from** scrapy.spiders **import** CrawlSpider, Rule

**from** youyuan.items **import** YouyuanItem

**class** YySpider(CrawlSpider):

name = **'yy'**

allowed\_domains = [**'youyuan.com'**]

*# 有缘网的列表页*

start\_urls = [**'http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/'**]

*# 第一级匹配规则：北京市18~25岁女性的每一页链接匹配规则*

page\_links = LinkExtractor(allow = (**r"youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/"**))

*# 第二级匹配规则：每个女性个人主页的匹配规则*

profile\_links = LinkExtractor(allow = (**r"youyuan.com/\d+-profile/"**))

*# 没有callback函数时默认follow=True*

rules = (

Rule(page\_links),

Rule(profile\_links, callback = **"parse\_item"**),

)

**def** parse\_item(self, response):

item = YouyuanItem()

item[**'username'**] = self.get\_username(response)

*# 年龄*

item[**'age'**] = self.get\_age(response)

*# 头像图片的链接*

item[**'header\_url'**] = self.get\_header\_url(response)

*# 相册图片的链接*

item[**'images\_url'**] = self.get\_images\_url(response)

*# 内心独白*

item[**'content'**] = self.get\_content(response)

*# 籍贯*

item[**'place\_from'**] = self.get\_place\_from(response)

*# 学历*

item[**'education'**] = self.get\_education(response)

*# 　兴趣爱好*

item[**'hobby'**] = self.get\_hobby(response)

*# 个人主页*

item[**'source\_url'**] = response.url

*# 数据来源网站*

item[**'sourec'**] = **"youyuan"**

**yield** item

*#用户名*

**def** get\_username(self, response):

username = response.xpath(**"//dl[@class='personal\_cen']//div[@class='main']/strong/text()"**).extract()

**if** len(username):

username = username[0]

**else**:

username = **"NULL"**

**return** username.strip()

*#年龄*

**def** get\_age(self, response):

age = response.xpath(**"//dl[@class='personal\_cen']//dd/p/text()"**).extract()

**if** len(age):

*#不能使用split(' ')的方法, 因为有的一个空格, 有的两个空格.*

*#这里的u表示什么？？*

age = re.findall(**u"\d+岁"**, age[0])[0]

*#使用r会报错*

*# age = re.findall(r"\d+岁", age[0])[0]*

**else**:

age = **"NULL"**

**return** age.strip()

*#头像图片链接*

**def** get\_header\_url(self, response):

header\_url = response.xpath(**"//dl[@class='personal\_cen']/dt/img/@src"**).extract()

**if** len(header\_url):

header\_url = header\_url[0]

**else**:

header\_url = **"NULL"**

**return** header\_url.strip()

*#相册链接*

**def** get\_images\_url(self, response):

images\_url = response.xpath(**"//div[@class='ph\_show']/ul/li/a/img/@src"**).extract()

**if** len(images\_url):

images\_url = images\_url

**else**:

images\_url = **"NULL"**

**return** images\_url

*#内心独白*

**def** get\_content(self, response):

content = response.xpath(**"//div[@class='pre\_data']/ul/li/p/text()"**).extract()

**if** len(content):

content = content[0]

**else**:

content = **"NULL"**

**return** content.strip()

*#籍贯*

**def** get\_place\_from(self, response):

place\_from = response.xpath(**"//div[@class='pre\_data']/ul/li[2]//ol[1]/li[1]/span/text()"**).extract()

**if** len(place\_from):

place\_from = place\_from[0]

**else**:

place\_from = **"NULL"**

**return** place\_from.strip()

*#学历*

**def** get\_education(self, response):

education = response.xpath(**"//div[@class='pre\_data']/ul/li[3]//ol[2]/li[2]/span/text()"**).extract()

**if** len(education):

education = education[0]

**else**:

education = **"NULL"**

**return** education.strip()

*#爱好*

**def** get\_hobby(self, response):

hobby = response.xpath(**"//dl[@class='personal\_cen']//ol/li/text()"**).extract()

**if** len(hobby):

*#使用逗号把列表中各个元素合并为一个元素*

hobby = **","**.join(hobby).replace(**" "**,**""**)

**else**:

hobby = **"NULL"**

**return** hobby.strip()

### 运行程序：

scrapy crawl youyuan

## 第2步：有缘网爬虫CrawlSpider版写入到redis数据库

# clone github scrapy-redis源码文件

git clone https://github.com/rolando/scrapy-redis.git

# 直接拿官方的项目范例, 改名为自己的项目用（针对懒癌患者)

mv scrapy-redis/example-project ~/scrapy-youyuan

或者使用命令从头开始创建项目

scrapy startproject youyuan

cd youyuan/spiders

scrapy genspider -t crawl yy "youyuan.com"

想要把数据写入到redis数据库中, 而不是写入到本地文件, 只需要在settings中添加redis的去重方法、指定redis的调度器、指定redis的队列方式, 添加redis的pipeline即可. 这时也可以使用scrapy的队列方法, 但是在爬虫运行期间, redis数据库中就不会显示出队列这一项了.

写分布式爬虫项目, 就按正常的scrapy项目写就行, 写完了在settings中添加以上三个内容, 修改pipeline文件即可成为分布式的代码.

### 修改settings.py

下面列举了修改后的配置文件中与scrapy-redis有关的部分, middleware、proxy等内容在此就省略了.

非分布式的版本可以使用redis的方法和redis数据库, 但是没有分布式的功能.

# -\*- coding: utf-8 -\*-

BOT\_NAME = **'youyuan'**

SPIDER\_MODULES = [**'youyuan.spiders'**]

NEWSPIDER\_MODULE = **'youyuan.spiders'**

*#可以使用redis的方法和redis数据库, 但是没有分布式的功能*

*# 指定使用scrapy-redis的去重方法*

DUPEFILTER\_CLASS = **"scrapy\_redis.dupefilter.RFPDupeFilter"**

*## 指定使用scrapy-redis的调度器*

SCHEDULER = **"scrapy\_redis.scheduler.Scheduler"**

*# 指定排序爬取地址时使用的队列, 默认的 按优先级排序(Scrapy默认), 由sorted set实现的一种非FIFO、LIFO方式.*

*#scrapy-redis请求队列形式, 是按优先级顺序, 先进的优先级就高*

SCHEDULER\_QUEUE\_CLASS = **"scrapy\_redis.queue.SpiderPriorityQueue"**

*#队列形式, 可选的, 按先进先出排序（FIFO）*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"*

*#栈形式, 可选的, 按后进先出排序（LIFO）*

*#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"*

*#允许执行项目时可以暂停, 暂停后不会清空redis数据库的记录*

*# 在redis中保持scrapy-redis用到的各个队列, 从而允许暂停和暂停后恢复, 也就是不清理redis queues*

SCHEDULER\_PERSIST = **True**

*# Crawl responsibly by identifying yourself (and your website) on the user-agent*

*#USER\_AGENT = 'youyuan (+http://www.yourdomain.com)'*

*# Obey robots.txt rules*

ROBOTSTXT\_OBEY = **True**

*# Configure a delay for requests for the same website (default: 0)*

*# See http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay*

*# See also autothrottle settings and docs*

DOWNLOAD\_DELAY = 3

*# The download delay setting will honor only one of:*

*#CONCURRENT\_REQUESTS\_PER\_DOMAIN = 16*

*#CONCURRENT\_REQUESTS\_PER\_IP = 16*

*# Disable cookies (enabled by default)*

COOKIES\_ENABLED = **False**

*# Disable Telnet Console (enabled by default)*

*#TELNETCONSOLE\_ENABLED = False*

*# 覆盖默认请求头, 可以自己编写Downloader Middlewares设置代理和UserAgent*

DEFAULT\_REQUEST\_HEADERS = {

**'Accept'**: **'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8'**,

**'Accept-Language'**: **'zh-CN,zh;q=0.8'**,

**'Connection'**: **'keep-alive'**,

**'Accept-Encoding'**: **'gzip, deflate, sdch'**

}

*# Configure item pipelines*

*# See http://scrapy.readthedocs.org/en/latest/topics/item-pipeline.html*

*# 通过配置RedisPipeline将item写入key为 spider.name : items 的redis的list中, 供后面的分布式处理item*

*# 这个已经由 scrapy-redis 实现, 不需要我们写代码*

*# scrapy\_redis.pipelines.RedisPipeline支持将数据存储到Redis数据库里, 必须启动.*

ITEM\_PIPELINES = {

**'youyuan.pipelines.YouyuanPipeline'**: 300,

**'scrapy\_redis.pipelines.RedisPipeline'** : 400,

}

*# LOG等级*

#LOG\_LEVEL = **'DEBUG'**

*#默认情况下,RFPDupeFilter只记录第一个重复请求. 将DUPEFILTER\_DEBUG设置为True会记录所有重复的请求.*

#DUPEFILTER\_DEBUG =**True**

### 修改pipeline.py

添加两个字段, 并写入到redis数据库中

# -\*- coding: utf-8 -\*-

from datetime import datetime

class ExamplePipeline(object):

def process\_item(self, item, spider):

#utcnow() 是获取UTC时间

item["crawled"] = datetime.utcnow()

# 爬虫名

item["spider"] = spider.name

return item

### 修改items.py

增加我们最后要保存的youyuanItem项, 这里只写出来一个非常简单的版本

# -\*- coding: utf-8 -\*-

**import** scrapy

**class** YouyuanItem(scrapy.Item):

*# 用户名*

username = scrapy.Field()

*# 年龄*

age = scrapy.Field()

*# 头像图片的链接*

header\_url = scrapy.Field()

*# 相册图片的链接*

images\_url = scrapy.Field()

*# 内心独白*

content = scrapy.Field()

*# 籍贯*

place\_from = scrapy.Field()

*# 学历*

education = scrapy.Field()

*#　兴趣爱好*

hobby = scrapy.Field()

*# 个人主页*

source\_url = scrapy.Field()

*# 数据来源网站*

sourec = scrapy.Field()

*# 在pipeline中添加两个字段并存入redis数据库中时使用*

*#UTC时间*

crawled = scrapy.Field()

*# 爬虫名*

spider = scrapy.Field()

### 修改pipeline文件

**添加两个字段**

**from** datetime **import** datetime

**class** YouyuanPipeline(object):

**def** process\_item(self, item, spider):

*# 格林威治时间, 距离中国 +8 时区*

item[**'time'**] = datetime.utcnow()

*# 爬虫名*

item[**'spidername'**] = spider.name

**return** item

### 运行程序：

1. Master端打开 Redis： redis-server
2. Slave端直接运行爬虫： scrapy crawl youyuan
3. 多个Slave端运行爬虫顺序没有限制.

## 第3步：有缘网分布式爬虫RedisCrawlSpider

将项目修改成 RedisCrawlSpider 类的分布式爬虫, 并尝试在多个Slave端运行.

在第2步中, 因为没有用到分布式的爬虫, redis数据是存储在本地的, 故不需要指定redis数据库的地址和端口号. 想要使用分布式执行, 还要在第2步的基础上, 修改slave端的redis数据地址和端口号.

### 修改 spiders/youyuan.py

在spiders目录下增加youyuan.py文件编写我们的爬虫, 使其具有分布式：

此时引入的spider类应该是RedisCralSpider

**from** scrapy.linkextractors **import** LinkExtractor

*#from scrapy.spiders import CrawlSpider, Rule*

**from** scrapy.spiders **import** Rule

**from** scrapy\_redis.spiders **import** RedisCrawlSpider

**from** youyuan.items **import** YouyuanItem

**import** re

*#class YySpider(CrawlSpider):*

**class** YySpider(RedisCrawlSpider):

name = **'yy'**

*#allowed\_domains = ['youyuan.com']*

*#start\_urls = ['http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/']*

redis\_key = **"yyspider:start\_urls"**

*# 动态域范围获取*

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

*# Dynamically define the allowed domains list.*

domain = kwargs.pop(**'domain'**, **''**)

self.allowed\_domains = filter(**None**, domain.split(**','**))

super(YySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

*# 第一级匹配规则：北京市18~25岁女性的每一页链接匹配规则*

page\_links = LinkExtractor(allow = (**r"youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p\d+/"**))

*# 第二级匹配规则：每个女性个人主页的匹配规则*

profile\_links = LinkExtractor(allow = (**r"youyuan.com/\d+-profile/"**))

rules = (

Rule(page\_links),

Rule(profile\_links, callback = **"parse\_item"**),

)

**def** parse\_item(self, response):

item = YouyuanItem()

item[**'username'**] = self.get\_username(response)

*# 年龄*

item[**'age'**] = self.get\_age(response)

*# 头像图片的链接*

item[**'header\_url'**] = self.get\_header\_url(response)

*# 相册图片的链接*

item[**'images\_url'**] = self.get\_images\_url(response)

*# 内心独白*

item[**'content'**] = self.get\_content(response)

*# 籍贯*

item[**'place\_from'**] = self.get\_place\_from(response)

*# 学历*

item[**'education'**] = self.get\_education(response)

*# 　兴趣爱好*

item[**'hobby'**] = self.get\_hobby(response)

*# 个人主页*

item[**'source\_url'**] = response.url

*# 数据来源网站*

item[**'sourec'**] = **"youyuan"**

**yield** item

*#用户名*

**def** get\_username(self, response):

username = response.xpath(**"//dl[@class='personal\_cen']//div[@class='main']/strong/text()"**).extract()

**if** len(username):

username = username[0]

**else**:

username = **"NULL"**

**return** username.strip()

*#年龄*

**def** get\_age(self, response):

age = response.xpath(**"//dl[@class='personal\_cen']//dd/p/text()"**).extract()

**if** len(age):

*#不能使用split(' ')的方法, 因为有的一个空格, 有的两个空格.*

*#这里的u表示什么？？*

age = re.findall(**u"\d+岁"**, age[0])[0]

*#使用r会报错*

*# age = re.findall(r"\d+岁", age[0])[0]*

**else**:

age = **"NULL"**

**return** age.strip()

*#头像图片链接*

**def** get\_header\_url(self, response):

header\_url = response.xpath(**"//dl[@class='personal\_cen']/dt/img/@src"**).extract()

**if** len(header\_url):

header\_url = header\_url[0]

**else**:

header\_url = **"NULL"**

**return** header\_url.strip()

*#相册链接*

**def** get\_images\_url(self, response):

images\_url = response.xpath(**"//div[@class='ph\_show']/ul/li/a/img/@src"**).extract()

**if** len(images\_url):

images\_url = images\_url

**else**:

images\_url = **"NULL"**

**return** images\_url

*#内心独白*

**def** get\_content(self, response):

content = response.xpath(**"//div[@class='pre\_data']/ul/li/p/text()"**).extract()

**if** len(content):

content = content[0]

**else**:

content = **"NULL"**

**return** content.strip()

*#籍贯*

**def** get\_place\_from(self, response):

place\_from = response.xpath(**"//div[@class='pre\_data']/ul/li[2]//ol[1]/li[1]/span/text()"**).extract()

**if** len(place\_from):

place\_from = place\_from[0]

**else**:

place\_from = **"NULL"**

**return** place\_from.strip()

*#学历*

**def** get\_education(self, response):

education = response.xpath(**"//div[@class='pre\_data']/ul/li[3]//ol[2]/li[2]/span/text()"**).extract()

**if** len(education):

education = education[0]

**else**:

education = **"NULL"**

**return** education.strip()

*#爱好*

**def** get\_hobby(self, response):

hobby = response.xpath(**"//dl[@class='personal\_cen']//ol/li/text()"**).extract()

**if** len(hobby):

*#使用逗号把列表中各个元素合并为一个元素*

hobby = **","**.join(hobby).replace(**" "**,**""**)

**else**:

hobby = **"NULL"**

**return** hobby.strip()

### 把代码上传到所有slave端

tar -cvf youyuan.tar youyuan

sftp 远程的用户名@ip

sftp python@1921682184

ls显示远程的文件和目录

lls显示要地的文件和目录

put youyun.tar

tar -xvf youyuan.tar

c是打包collection

x是解包

z是压缩, 对应gz

### 分布式爬虫执行方式：

#### 6. 在Master端启动redis-server：

redis-server

#### 7. 在Slave端分别启动爬虫, 不分先后：

修改slave端settings中的REDIS\_HOST为Master端的ip

REDIS\_HOST = "192.168.21.64"

REDIS\_PORT = 6379

scrapy runspider youyuan.py

#### 8. 在Master端的redis-cli里push一个start\_urls

redis-cli> lpush youyuan:start\_urls http://www.youyuan.com/find/beijing/mm18-25/advance-0-0-0-0-0-0-0/p1/

#### 9. 爬虫启动, 查看redis数据库数据.

sudo redis-cli

keys \* 显示所有数据库

yy是数据库名, 0是从第一个开始, -1表示最后一个. 即取数据库中的所有数据.

lrange "yy:items" 0 -1

## 处理Redis里的数据

有缘网的数据爬回来了, 但是放在Redis里没有处理. 之前我们配置文件里面没有定制自己的ITEM\_PIPELINES, 而是使用了RedisPipeline, 所以现在这些数据都被保存在redis的youyuan:items键中, 所以我们需要另外做处理.

在scrapy-youyuan目录下可以看到一个process\_items.py文件, 这个文件就是scrapy-redis的example提供的从redis读取item进行处理的模版.

假设我们要把youyuan:items中保存的数据读出来写进MongoDB或者MySQL, 那么我们可以自己写一个process\_youyuan\_profile.py文件, 然后保持后台运行就可以不停地将爬回来的数据入库了.

### 存入MongoDB

1. 启动MongoDB数据库：sudo mongod
2. 执行下面程序：py2 process\_youyuan\_mongodb.py

# process\_youyuan\_mongodb.py

# -\*- coding: utf-8 -\*-

import json

import redis

import pymongo

def main():

# 指定Redis数据库信息

rediscli = redis.StrictRedis(host='192.168.199.108', port=6379, db=0)

# 指定MongoDB数据库信息

mongocli = pymongo.MongoClient(host='localhost', port=27017)

# 创建数据库名

db = mongocli['youyuan']

# 创建表名

sheet = db['beijing\_18\_25']

while True:

# FIFO模式为 blpop, LIFO模式为 brpop, 获取键值

source, data = rediscli.blpop(["youyuan:items"])

item = json.loads(data)

sheet.insert(item)

try:

print u"Processing: %(name)s <%(link)s>" % item

except KeyError:

print u"Error procesing: %r" % item

if \_\_name\_\_ == '\_\_main\_\_':

main()

**视频演示**

# process\_youyuan\_mongodb.py

# -\*- coding: utf-8 -\*-

**import** redis, pymongo, json

*#操作redis数据库和mongo数据库, 之前向redis数据库存储数据其实是一个json对象, 现在保存为MongoDB数据, 还要转换回去.*

**def** process\_item():

*# 创建redis数据库连接*

rediscli = redis.Redis(host = **"127.0.0.1"**, port = 6379, db = **"0"**)

*# 创建MongoDB数据库连接*

mongocli = pymongo.MongoClient(host = **"127.0.0.1"**, port = 27017)

*# 创建mongodb数据库名称*

dbname = mongocli[**"youyuan"**]

*# 创建mongodb数据库youyuan的表名称*

sheetname = dbname[**"beijing\_18\_25\_mm"**]

*#查看有多少条记录*

offset = 0

*#使用循环向MongoDB中添加多个数据*

**while True**:

*# redis 数据表名 和 数据*

source, data = rediscli.blpop(**"yy:items"**)

offset += 1

*# 将json对象转换为Python对象*

data = json.loads(data)

*# 将数据插入到sheetname表里*

sheetname.insert(data)

print(offset)

**if** \_\_name\_\_ == **"\_\_main\_\_"**:

process\_item()

mongod启动mongodb服务

执行python process\_item\_for\_mongodb.py

mongo启动mongo shell环境

db

show dbs

use youyuan

show collections

db.beijing\_18\_25\_mm.find()

### 存入 MySQL

1. 启动mysql：mysql.server start（更平台不一样）
2. 登录到root用户：mysql -uroot -p
3. 创建数据库youyuan:create database youyuan;
4. 切换到指定数据库：use youyuan
5. 创建表beijing\_18\_25以及所有字段的列名和数据类型.
6. 执行下面程序：py2 process\_youyuan\_mysql.py

#process\_youyuan\_mysql.py

# -\*- coding: utf-8 -\*-

import json

import redis

import MySQLdb

def main():

# 指定redis数据库信息

rediscli = redis.StrictRedis(host='192.168.199.108', port = 6379, db = 0)

# 指定mysql数据库

mysqlcli = MySQLdb.connect(host='127.0.0.1', user='power', passwd='xxxxxxx', db = 'youyuan', port=3306, use\_unicode=True)

while True:

# FIFO模式为 blpop, LIFO模式为 brpop, 获取键值

source, data = rediscli.blpop(["youyuan:items"])

item = json.loads(data)

try:

# 使用cursor()方法获取操作游标

cur = mysqlcli.cursor()

# 使用execute方法执行SQL INSERT语句

cur.execute("INSERT INTO beijing\_18\_25 (username, crawled, age, spider, header\_url, source, pic\_urls, monologue, source\_url) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s )", [item['username'], item['crawled'], item['age'], item['spider'], item['header\_url'], item['source'], item['pic\_urls'], item['monologue'], item['source\_url']])

# 提交sql事务

mysqlcli.commit()

#关闭本次操作

cur.close()

print "inserted %s" % item['source\_url']

except MySQLdb.Error,e:

print "Mysql Error %d: %s" % (e.args[0], e.args[1])

if \_\_name\_\_ == '\_\_main\_\_':

main()

## 新浪网分类资讯爬虫1

思考：如何将已有的Scrapy爬虫项目, 改写成scrapy-redis分布式爬虫.

要求：将所有对应的大类的 标题和urls、小类的 标题和urls、子链接url、文章名以及文章内容, 存入Redis数据库.

以下为原Scrapy爬虫项目源码：

### items.py

# -\*- coding: utf-8 -\*-

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### pipelines.py

# -\*- coding: utf-8 -\*-

from scrapy import signals

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaPipeline(object):

def process\_item(self, item, spider):

sonUrls = item['sonUrls']

# 文件名为子链接url中间部分, 并将 / 替换为 \_, 保存为 .txt格式

filename = sonUrls[7:-6].replace('/','\_')

filename += ".txt"

fp = open(item['subFilename']+'/'+filename, 'w')

fp.write(item['content'])

fp.close()

return item

### settings.py

# -\*- coding: utf-8 -\*-

BOT\_NAME = 'Sina'

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

ITEM\_PIPELINES = {

'Sina.pipelines.SinaPipeline': 300,

}

LOG\_LEVEL = 'DEBUG'

### spiders/sina.py

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

import scrapy

import os

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaSpider(scrapy.Spider):

name= "sina"

allowed\_domains= ["sina.com.cn"]

start\_urls= [

"http://news.sina.com.cn/guide/"

]

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类目录的路径和目录名

parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在, 则创建目录

if(not os.path.exists(parentFilename)):

os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头, 如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类, 将存储目录放在本大类目录下

if(if\_belong):

subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在, 则创建目录

if(not os.path.exists(subFilename)):

os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求, 得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url, 再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾, 如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类, 获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] = meta\_1['subUrls']

item['subTitle'] = meta\_1['subTitle']

item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求, 得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法, 获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()')

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head

item['content']= content

yield item

### 执行：

scrapy crawl sina

## 新浪网分类资讯爬虫-分布式版

将已有的新浪网分类资讯Scrapy爬虫项目, 修改为基于RedisSpider类的scrapy-redis分布式爬虫项目

注：items数据直接存储在Redis数据库中, 这个功能已经由scrapy-redis自行实现. 除非单独做额外处理(比如直接存入本地数据库等), 否则不用编写pipelines.py代码.

### items.py文件

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class SinaItem(scrapy.Item):

# 大类的标题 和 url

parentTitle = scrapy.Field()

parentUrls = scrapy.Field()

# 小类的标题 和 子url

subTitle = scrapy.Field()

subUrls = scrapy.Field()

# 小类目录存储路径

# subFilename = scrapy.Field()

# 小类下的子链接

sonUrls = scrapy.Field()

# 文章标题和内容

head = scrapy.Field()

content = scrapy.Field()

### settings.py文件

# settings.py

SPIDER\_MODULES = ['Sina.spiders']

NEWSPIDER\_MODULE = 'Sina.spiders'

USER\_AGENT = 'scrapy-redis (+https://github.com/rolando/scrapy-redis)'

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

SCHEDULER\_PERSIST = True

SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderPriorityQueue"

#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderQueue"

#SCHEDULER\_QUEUE\_CLASS = "scrapy\_redis.queue.SpiderStack"

ITEM\_PIPELINES = {

# 'Sina.pipelines.SinaPipeline': 300,

'scrapy\_redis.pipelines.RedisPipeline': 400,

}

LOG\_LEVEL = 'DEBUG'

# Introduce an artifical delay to make use of parallelism. to speed up the

# crawl.

DOWNLOAD\_DELAY = 1

REDIS\_HOST = "192.168.13.26"

REDIS\_PORT = 6379

### spiders/sina.py

# sina.py

# -\*- coding: utf-8 -\*-

from Sina.items import SinaItem

from scrapy\_redis.spiders import RedisSpider

#from scrapy.spiders import Spider

import scrapy

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

#class SinaSpider(Spider):

class SinaSpider(RedisSpider):

name= "sina"

redis\_key = "sinaspider:start\_urls"

#allowed\_domains= ["sina.com.cn"]

#start\_urls= [

# "http://news.sina.com.cn/guide/"

#]#起始urls列表

def \_\_init\_\_(self, \*args, \*\*kwargs):

domain = kwargs.pop('domain', '')

self.allowed\_domains = filter(None, domain.split(','))

super(SinaSpider, self).\_\_init\_\_(\*args, \*\*kwargs)

def parse(self, response):

items= []

# 所有大类的url 和 标题

parentUrls = response.xpath('//div[@id=\"tab01\"]/div/h3/a/@href').extract()

parentTitle = response.xpath("//div[@id=\"tab01\"]/div/h3/a/text()").extract()

# 所有小类的ur 和 标题

subUrls = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/@href').extract()

subTitle = response.xpath('//div[@id=\"tab01\"]/div/ul/li/a/text()').extract()

#爬取所有大类

for i in range(0, len(parentTitle)):

# 指定大类的路径和目录名

#parentFilename = "./Data/" + parentTitle[i]

#如果目录不存在, 则创建目录

#if(not os.path.exists(parentFilename)):

# os.makedirs(parentFilename)

# 爬取所有小类

for j in range(0, len(subUrls)):

item = SinaItem()

# 保存大类的title和urls

item['parentTitle'] = parentTitle[i]

item['parentUrls'] = parentUrls[i]

# 检查小类的url是否以同类别大类url开头, 如果是返回True (sports.sina.com.cn 和 sports.sina.com.cn/nba)

if\_belong = subUrls[j].startswith(item['parentUrls'])

# 如果属于本大类, 将存储目录放在本大类目录下

if(if\_belong):

#subFilename =parentFilename + '/'+ subTitle[j]

# 如果目录不存在, 则创建目录

#if(not os.path.exists(subFilename)):

# os.makedirs(subFilename)

# 存储 小类url、title和filename字段数据

item['subUrls'] = subUrls[j]

item['subTitle'] =subTitle[j]

#item['subFilename'] = subFilename

items.append(item)

#发送每个小类url的Request请求, 得到Response连同包含meta数据 一同交给回调函数 second\_parse 方法处理

for item in items:

yield scrapy.Request( url = item['subUrls'], meta={'meta\_1': item}, callback=self.second\_parse)

#对于返回的小类的url, 再进行递归请求

def second\_parse(self, response):

# 提取每次Response的meta数据

meta\_1= response.meta['meta\_1']

# 取出小类里所有子链接

sonUrls = response.xpath('//a/@href').extract()

items= []

for i in range(0, len(sonUrls)):

# 检查每个链接是否以大类url开头、以.shtml结尾, 如果是返回True

if\_belong = sonUrls[i].endswith('.shtml') and sonUrls[i].startswith(meta\_1['parentUrls'])

# 如果属于本大类, 获取字段值放在同一个item下便于传输

if(if\_belong):

item = SinaItem()

item['parentTitle'] =meta\_1['parentTitle']

item['parentUrls'] =meta\_1['parentUrls']

item['subUrls'] =meta\_1['subUrls']

item['subTitle'] =meta\_1['subTitle']

#item['subFilename'] = meta\_1['subFilename']

item['sonUrls'] = sonUrls[i]

items.append(item)

#发送每个小类下子链接url的Request请求, 得到Response后连同包含meta数据 一同交给回调函数 detail\_parse 方法处理

for item in items:

yield scrapy.Request(url=item['sonUrls'], meta={'meta\_2':item}, callback = self.detail\_parse)

# 数据解析方法, 获取文章标题和内容

def detail\_parse(self, response):

item = response.meta['meta\_2']

content = ""

head = response.xpath('//h1[@id=\"main\_title\"]/text()').extract()

content\_list = response.xpath('//div[@id=\"artibody\"]/p/text()').extract()

# 将p标签里的文本内容合并到一起

for content\_one in content\_list:

content += content\_one

item['head']= head[0] if len(head) > 0 else "NULL"

item['content']= content

yield item

### 执行：

slave端：

scrapy runspider sina.py

Master端：

redis-cli> lpush sinaspider:start\_urls http://news.sina.com.cn/guide/

## IT桔子分布式项目

IT桔子是关注IT互联网行业的结构化的公司数据库和商业信息服务提供商, 于2013年5月21日上线.

IT桔子致力于通过信息和数据的生产、聚合、挖掘、加工、处理, 帮助目标用户和客户节约时间和金钱、提高效率, 以辅助其各类商业行为, 包括风险投资、收购、竞争情报、细分行业信息、国外公司产品信息数据服务等.

用于需自行对所发表或采集的内容负责,因所发表或采集的内容引发的一切纠纷、损失, 由该内容的发表或采集者承担全部直接或间接(连带)法律责任, IT桔子不承担任何法律责任.

项目采集地址：[http://www.itjuzi.com/company](http://www.itjuzi.com/company/)

要求：采集页面下所有创业公司的公司信息, 包括以下但不限于：

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

class CompanyItem(scrapy.Item):

# 公司id (url数字部分)

info\_id = scrapy.Field()

# 公司名称

company\_name = scrapy.Field()

# 公司口号

slogan = scrapy.Field()

# 分类

scope = scrapy.Field()

# 子分类

sub\_scope = scrapy.Field()

# 所在城市

city = scrapy.Field()

# 所在区域

area = scrapy.Field()

# 公司主页

home\_page = scrapy.Field()

# 公司标签

tags = scrapy.Field()

# 公司简介

company\_intro = scrapy.Field()

# 公司全称：

company\_full\_name = scrapy.Field()

# 成立时间：

found\_time = scrapy.Field()

# 公司规模：

company\_size = scrapy.Field()

# 运营状态

company\_status = scrapy.Field()

# 投资情况列表：包含获投时间、融资阶段、融资金额、投资公司

tz\_info = scrapy.Field()

# 团队信息列表：包含成员姓名、成员职称、成员介绍

tm\_info = scrapy.Field()

# 产品信息列表：包含产品名称、产品类型、产品介绍

pdt\_info = scrapy.Field()

### 项目实现：

#### items.py

# items.py

# -\*- coding: utf-8 -\*-

import scrapy

class CompanyItem(scrapy.Item):

# 公司id (url数字部分)

info\_id = scrapy.Field()

# 公司名称

company\_name = scrapy.Field()

# 公司口号

slogan = scrapy.Field()

# 分类

scope = scrapy.Field()

# 子分类

sub\_scope = scrapy.Field()

# 所在城市

city = scrapy.Field()

# 所在区域

area = scrapy.Field()

# 公司主页

home\_page = scrapy.Field()

# 公司标签

tags = scrapy.Field()

# 公司简介

company\_intro = scrapy.Field()

# 公司全称：

company\_full\_name = scrapy.Field()

# 成立时间：

found\_time = scrapy.Field()

# 公司规模：

company\_size = scrapy.Field()

# 运营状态

company\_status = scrapy.Field()

# 投资情况列表：包含获投时间、融资阶段、融资金额、投资公司

tz\_info = scrapy.Field()

# 团队信息列表：包含成员姓名、成员职称、成员介绍

tm\_info = scrapy.Field()

# 产品信息列表：包含产品名称、产品类型、产品介绍

pdt\_info = scrapy.Field()

#### settings.py

# -\*- coding: utf-8 -\*-

BOT\_NAME = 'itjuzi'

SPIDER\_MODULES = ['itjuzi.spiders']

NEWSPIDER\_MODULE = 'itjuzi.spiders'

# Enables scheduling storing requests queue in redis.

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

# Ensure all spiders share same duplicates filter through redis.

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

# REDIS\_START\_URLS\_AS\_SET = True

COOKIES\_ENABLED = False

DOWNLOAD\_DELAY = 1.5

# 支持随机下载延迟

RANDOMIZE\_DOWNLOAD\_DELAY = True

# Obey robots.txt rules

ROBOTSTXT\_OBEY = False

ITEM\_PIPELINES = {

'scrapy\_redis.pipelines.RedisPipeline': 300

}

DOWNLOADER\_MIDDLEWARES = {

# 该中间件将会收集失败的页面, 并在爬虫完成后重新调度. （失败情况可能由于临时的问题, 例如连接超时或者HTTP 500错误导致失败的页面）

'scrapy.downloadermiddlewares.retry.RetryMiddleware': 80,

# 该中间件提供了对request设置HTTP代理的支持. 您可以通过在 Request 对象中设置 proxy 元数据来开启代理.

'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 100,

'itjuzi.middlewares.RotateUserAgentMiddleware': 200,

}

REDIS\_HOST = "192.168.199.108"

REDIS\_PORT = 6379

#### middlewares.py

# -\*- coding: utf-8 -\*-

from scrapy.contrib.downloadermiddleware.useragent import UserAgentMiddleware

import random

# User-Agetn 下载中间件

class RotateUserAgentMiddleware(UserAgentMiddleware):

def \_\_init\_\_(self, user\_agent=''):

self.user\_agent = user\_agent

def process\_request(self, request, spider):

# 这句话用于随机选择user-agent

ua = random.choice(self.user\_agent\_list)

request.headers.setdefault('User-Agent', ua)

user\_agent\_list = [

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/22.0.1207.1 Safari/537.1",

"Mozilla/5.0 (X11; CrOS i686 2268.111.0) AppleWebKit/536.11 (KHTML, like Gecko) Chrome/20.0.1132.57 Safari/536.11",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.6 (KHTML, like Gecko) Chrome/20.0.1092.0 Safari/536.6",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.6 (KHTML, like Gecko) Chrome/20.0.1090.0 Safari/536.6",

"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/19.77.34.5 Safari/537.1",

"Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.9 Safari/536.5",

"Mozilla/5.0 (Windows NT 6.0) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.36 Safari/536.5",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 5.1) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_8\_0) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1063.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1062.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1062.0 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.1 Safari/536.3",

"Mozilla/5.0 (Windows NT 6.2) AppleWebKit/536.3 (KHTML, like Gecko) Chrome/19.0.1061.0 Safari/536.3",

"Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/535.24 (KHTML, like Gecko) Chrome/19.0.1055.1 Safari/535.24",

"Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/535.24 (KHTML, like Gecko) Chrome/19.0.1055.1 Safari/535.24",

"Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/531.21.8 (KHTML, like Gecko) Version/4.0.4 Safari/531.21.10",

"Mozilla/5.0 (Windows; U; Windows NT 5.2; en-US) AppleWebKit/533.17.8 (KHTML, like Gecko) Version/5.0.1 Safari/533.17.8",

"Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US) AppleWebKit/533.19.4 (KHTML, like Gecko) Version/5.0.2 Safari/533.18.5",

"Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.1.17) Gecko/20110123 (like Firefox/3.x) SeaMonkey/2.0.12",

"Mozilla/5.0 (Windows NT 5.2; rv:10.0.1) Gecko/20100101 Firefox/10.0.1 SeaMonkey/2.7.1",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_5\_8; en-US) AppleWebKit/532.8 (KHTML, like Gecko) Chrome/4.0.302.2 Safari/532.8",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_4; en-US) AppleWebKit/534.3 (KHTML, like Gecko) Chrome/6.0.464.0 Safari/534.3",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_6\_5; en-US) AppleWebKit/534.13 (KHTML, like Gecko) Chrome/9.0.597.15 Safari/534.13",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_2) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.835.186 Safari/535.1",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_6\_8) AppleWebKit/535.2 (KHTML, like Gecko) Chrome/15.0.874.54 Safari/535.2",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_6\_8) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.36 Safari/535.7",

"Mozilla/5.0 (Macintosh; U; Mac OS X Mach-O; en-US; rv:2.0a) Gecko/20040614 Firefox/3.0.0 ",

"Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5; en-US; rv:1.9.0.3) Gecko/2008092414 Firefox/3.0.3",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1) Gecko/20090624 Firefox/3.5",

"Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.6; en-US; rv:1.9.2.14) Gecko/20110218 AlexaToolbar/alxf-2.0 Firefox/3.6.14",

"Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10.5; en-US; rv:1.9.2.15) Gecko/20110303 Firefox/3.6.15",

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:2.0.1) Gecko/20100101 Firefox/4.0.1"

]

#### spiders/juzi.py

# -\*- coding: utf-8 -\*-

from bs4 import BeautifulSoup

from scrapy.linkextractors import LinkExtractor

from scrapy.spiders import CrawlSpider, Rule

from scrapy\_redis.spiders import RedisCrawlSpider

from itjuzi.items import CompanyItem

class ITjuziSpider(RedisCrawlSpider):

name = 'itjuzi'

allowed\_domains = ['www.itjuzi.com']

# start\_urls = ['http://www.itjuzi.com/company']

redis\_key = 'itjuzispider:start\_urls'

rules = [

# 获取每一页的链接

Rule(link\_extractor=LinkExtractor(allow=('/company\?page=\d+'))),

# 获取每一个公司的详情

Rule(link\_extractor=LinkExtractor(allow=('/company/\d+')), callback='parse\_item')

]

def parse\_item(self, response):

soup = BeautifulSoup(response.body, 'lxml')

# 开头部分： //div[@class="infoheadrow-v2 ugc-block-item"]

cpy1 = soup.find('div', class\_='infoheadrow-v2')

if cpy1:

# 公司名称：//span[@class="title"]/b/text()[1]

company\_name = cpy1.find(class\_='title').b.contents[0].strip().replace('\t', '').replace('\n', '')

# 口号： //div[@class="info-line"]/p

slogan = cpy1.find(class\_='info-line').p.get\_text()

# 分类：子分类//span[@class="scope c-gray-aset"]/a[1]

scope\_a = cpy1.find(class\_='scope c-gray-aset').find\_all('a')

# 分类：//span[@class="scope c-gray-aset"]/a[1]

scope = scope\_a[0].get\_text().strip() if len(scope\_a) > 0 else ''

# 子分类：# //span[@class="scope c-gray-aset"]/a[2]

sub\_scope = scope\_a[1].get\_text().strip() if len(scope\_a) > 1 else ''

# 城市+区域：//span[@class="loca c-gray-aset"]/a

city\_a = cpy1.find(class\_='loca c-gray-aset').find\_all('a')

# 城市：//span[@class="loca c-gray-aset"]/a[1]

city = city\_a[0].get\_text().strip() if len(city\_a) > 0 else ''

# 区域：//span[@class="loca c-gray-aset"]/a[2]

area = city\_a[1].get\_text().strip() if len(city\_a) > 1 else ''

# 主页：//a[@class="weblink marl10"]/@href

home\_page = cpy1.find(class\_='weblink marl10')['href']

# 标签：//div[@class="tagset dbi c-gray-aset"]/a

tags = cpy1.find(class\_='tagset dbi c-gray-aset').get\_text().strip().strip().replace('\n', ',')

#基本信息：//div[@class="block-inc-info on-edit-hide"]

cpy2 = soup.find('div', class\_='block-inc-info on-edit-hide')

if cpy2:

# 公司简介：//div[@class="block-inc-info on-edit-hide"]//div[@class="des"]

company\_intro = cpy2.find(class\_='des').get\_text().strip()

# 公司全称：成立时间：公司规模：运行状态：//div[@class="des-more"]

cpy2\_content = cpy2.find(class\_='des-more').contents

# 公司全称：//div[@class="des-more"]/div[1]

company\_full\_name = cpy2\_content[1].get\_text().strip()[len('公司全称：'):] if cpy2\_content[1] else ''

# 成立时间：//div[@class="des-more"]/div[2]/span[1]

found\_time = cpy2\_content[3].contents[1].get\_text().strip()[len('成立时间：'):] if cpy2\_content[3] else ''

# 公司规模：//div[@class="des-more"]/div[2]/span[2]

company\_size = cpy2\_content[3].contents[3].get\_text().strip()[len('公司规模：'):] if cpy2\_content[3] else ''

#运营状态：//div[@class="des-more"]/div[3]

company\_status = cpy2\_content[5].get\_text().strip() if cpy2\_content[5] else ''

# 主体信息：

main = soup.find('div', class\_='main')

# 投资情况：//table[@class="list-round-v2 need2login"]

# 投资情况, 包含获投时间、融资阶段、融资金额、投资公司

tz = main.find('table', 'list-round-v2')

tz\_list = []

if tz:

all\_tr = tz.find\_all('tr')

for tr in all\_tr:

tz\_dict = {}

all\_td = tr.find\_all('td')

tz\_dict['tz\_time'] = all\_td[0].span.get\_text().strip()

tz\_dict['tz\_round'] = all\_td[1].get\_text().strip()

tz\_dict['tz\_finades'] = all\_td[2].get\_text().strip()

tz\_dict['tz\_capital'] = all\_td[3].get\_text().strip().replace('\n', ',')

tz\_list.append(tz\_dict)

# 团队信息：成员姓名、成员职称、成员介绍

tm = main.find('ul', class\_='list-prodcase limited-itemnum')

tm\_list = []

if tm:

for li in tm.find\_all('li'):

tm\_dict = {}

tm\_dict['tm\_m\_name'] = li.find('span', class\_='c').get\_text().strip()

tm\_dict['tm\_m\_title'] = li.find('span', class\_='c-gray').get\_text().strip()

tm\_dict['tm\_m\_intro'] = li.find('p', class\_='mart10 person-des').get\_text().strip()

tm\_list.append(tm\_dict)

# 产品信息：产品名称、产品类型、产品介绍

pdt = main.find('ul', class\_='list-prod limited-itemnum')

pdt\_list = []

if pdt:

for li in pdt.find\_all('li'):

pdt\_dict = {}

pdt\_dict['pdt\_name'] = li.find('h4').b.get\_text().strip()

pdt\_dict['pdt\_type'] = li.find('span', class\_='tag yellow').get\_text().strip()

pdt\_dict['pdt\_intro'] = li.find(class\_='on-edit-hide').p.get\_text().strip()

pdt\_list.append(pdt\_dict)

item = CompanyItem()

item['info\_id'] = response.url.split('/')[-1:][0]

item['company\_name'] = company\_name

item['slogan'] = slogan

item['scope'] = scope

item['sub\_scope'] = sub\_scope

item['city'] = city

item['area'] = area

item['home\_page'] = home\_page

item['tags'] = tags

item['company\_intro'] = company\_intro

item['company\_full\_name'] = company\_full\_name

item['found\_time'] = found\_time

item['company\_size'] = company\_size

item['company\_status'] = company\_status

item['tz\_info'] = tz\_list

item['tm\_info'] = tm\_list

item['pdt\_info'] = pdt\_list

return item

#### scrapy.cfg

# Automatically created by: scrapy startproject

#

# For more information about the [deploy] section see:

# https://scrapyd.readthedocs.org/en/latest/deploy.html

[settings]

default = itjuzi.settings

[deploy]

#url = http://localhost:6800/

project = itjuzi

#### 运行：

Slave端：

scrapy runspider juzi.py

Master端：

redis-cli > lpush itjuzispider:start\_urls http://www.itjuzi.com/company