Python 爬虫框架 Scrapy

目　录

第 1 章 Python 爬虫基础

本章工作任务

- 任务 1：为什么学习 Python 爬虫
- 任务 2：什么是爬虫
- 任务 3：urllib 的应用
- 任务 4：cookie 实际应用
- 任务 5：正则表达式

本章技能目标及重难点

编号	技能点描述	级别
1	为什么学习爬虫	★
2	什么是爬虫	★★
3	urllib 的应用	★★★
4	cookie 实际应用	★★
5	正则表达式	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Python 爬虫，需要同学们理解为什么使用爬虫，它的概念、特点。最主要的是需要大家学会如何使用基础的爬虫库。

本章学习建议

本章适合有 Python 基础的学员学习。

本章内容（学习活动）

1.1 为什么学习 Python 爬虫？

现在信息更新的非常快速，又迎来了大数据的时代， 各行各业如果不与时俱进，都将面临优胜劣汰，知识是不断的更新的，只有一技之长，才能立于不败之地。

网络爬虫，即 Web Spider，是一个很形象的名字。目前爬虫开发的语言的主要是 Python，本课程结合几个小的爬虫案例，帮助学员更好的学习爬虫开发。

那么为什么我们要学习 Python 爬虫呢？而不选择 Java？PHP？Nodejs？

1.1.1 Python 语言的流行程度

Spectrum 排名：Python 的排名从去年开始就借助人工智能持续上升，现在它已经成为了第一名。但排在前四名的语言 Python、C、Java 和 C++都拥有广大的用户群体，并且他们的用户总量也十分相近。

现在全世界大约有几百万以上的 Python 语言的用户，大家可以看一下以下图片：

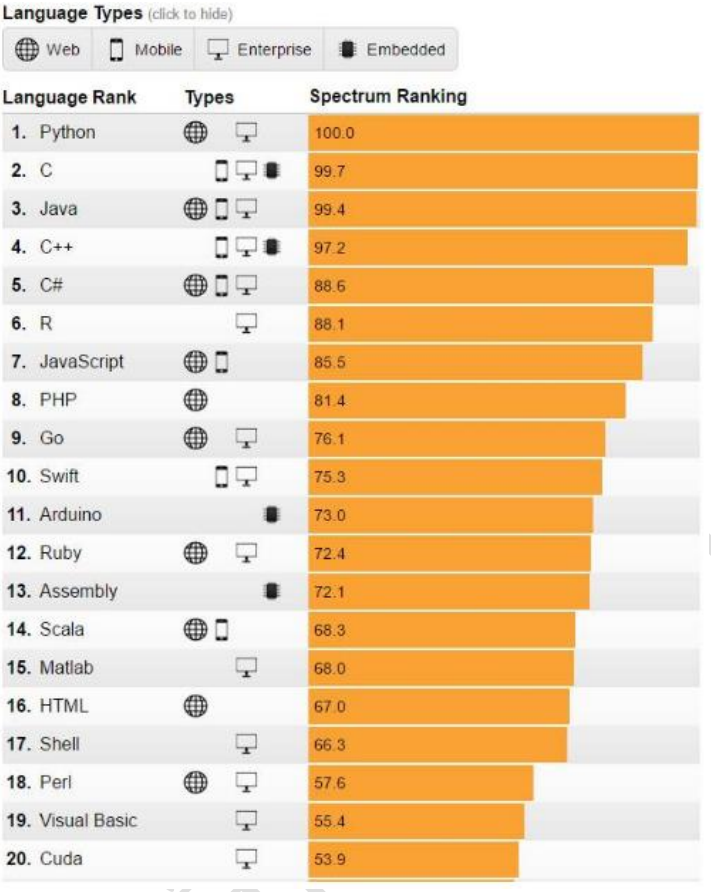

图 1-1　IEEE 发布 2017 年 7 月度编程语言排行榜

图 1-1 为 2016 年 Spectrum 评选出的排名前十的编程语言，Spectrum 的“交互式编程语言排行”让用户可以根据自己的喜好调整不同评价指标所占的权重，从而得到所需的排名。从该图可以看出 Python 在 IEEE 的会员用户语言使用中排名第一。

Aug 2017	Aug 2016	Change	Programming Language	Ratings	Change
1	1		Java	12.961%	-6.05%
2	2		C	6.477%	-4.83%
3	3		C++	5.550%	-0.25%
4	4		C#	4.195%	-0.71%
5	5		Python	3.692%	-0.71%
6	8	↑	Visual Basic .NET	2.569%	+0.05%
7	6	↓	PHP	2.293%	-0.88%
8	7	↓	JavaScript	2.098%	-0.61%
9	9		Perl	1.995%	-0.52%
10	12	↑	Ruby	1.965%	-0.31%
11	14	↑	Swift	1.825%	-0.16%
12	11	↓	Delphi/Object Pascal	1.825%	-0.45%
13	13		Visual Basic	1.809%	-0.24%
14	10	⇊	Assembly language	1.805%	-0.56%
15	17	↑	R	1.766%	+0.16%
16	20	⇈	Go	1.645%	+0.37%
17	18	↑	MATLAB	1.619%	+0.08%
18	15	↓	Objective-C	1.505%	-0.38%
19	22	↑	Scratch	1.481%	+0.43%
20	26	⇈	Dart	1.273%	+0.30%

图 1-2　TIOBE 2017 年 8 月编程语言排行榜

图 1-2 TIOBE 的数据就更能说明 Python 语言的地位，TIOBE 编程语言社区排行榜是编程语言流行趋势的一个指标，每月更新，这份排行榜排名基于互联网上有经验的程序员、 课程和第三方厂商的数量。排名使用著名的搜索引擎（诸如 Google、MSN、Yahoo!、Wikipedia、YouTube 以及 Baidu 等）进行计算。而 Python 语言排名第五。并趋于日益增长趋势。

1.1.2 爬虫框架比较

1、如果是定向爬取几个页面，做一些简单的页面解析，爬取效率不是核心要求，那么用什么语言差异不大。

当然要是页面结构复杂，正则表达式写得巨复杂，尤其是用过那些支持 xpath 的类库/爬虫库后，就会发现此种方式虽然入门门槛低，但扩展性、可维护性等都奇差。因此此种情况下还是推荐采用一些现成的爬虫库，诸如 xpath、多线程支持还是必须考虑的因素。

2、如果是定向爬取，且主要目标是解析 js 动态生成的内容

此时候，页面内容是有 js/ajax 动态生成的，用普通的请求页面->解析的方法就不管用了，需要借助一个类似 firefox、chrome 浏览器的 js 引擎来对页面的 js 代码做动态解析。

此种情况下，推荐考虑 casperJS+phantomjs 或 slimerJS+phantomjs ，当然诸如 selenium 之类的也可以考虑。

3、如果爬虫是涉及大规模网站爬取，效率、扩展性、可维护性等是必须考虑的因素时候

大规模爬虫爬取涉及诸多问题：多线程并发、I/O 机制、分布式爬取、消息通讯、判重机制、任务调度等等，此时候语言和所用框架的选取就具有极大意义了。

PHP 对多线程、异步支持较差，不建议采用。

NodeJS：对一些垂直网站爬取倒可以，但由于分布式爬取、消息通讯等支持较弱，根据自己情况判断。

Python：强烈建议，对以上问题都有较好支持。尤其是 Scrapy 框架值得作为第一选择。优点诸多：支持 xpath；基于 twisted，性能不错；有较好的调试工具；

此种情况下，如果还需要做 js 动态内容的解析，casperjs 就不适合了，只有基于诸如 chrome V8 引擎之类自己做 js 引擎。

至于 C、C++虽然性能不错，但不推荐，尤其是考虑到成本等诸多因素；对于大部分公司还是建议基于一些开源的框架来做，不要自己发明轮子，做一个简单的爬虫容易，但要做一个完备的爬虫挺难的。

1.1.3 为什么 Python 适合写爬虫

1. 抓取网页本身的接口

相比与其他静态编程语言，如 java，c#，C++，python 抓取网页文档的接口更简洁；相比其他动态脚本语言，如 perl，shell，python 的 urllib2 包提供了较为完整的访问网页文档的 API。（当然 ruby 也是很好的选择）

此外，抓取网页有时候需要模拟浏览器的行为，很多网站对于生硬的爬虫抓取都是封杀的。这是我们需要模拟 user agent 的行为构造合适的请求，譬如模拟用户登陆、模拟 session/cookie 的存储和设置。在 python 里都有非常优秀的第三方包帮你搞定，如 Requests，mechanize

2.网页抓取后的处理

抓取的网页通常需要处理，比如过滤 html 标签，提取文本等。python 的 beautifulsoap 提供了简洁的文档处理功能，能用极短的代码完成大部分文档的处理。

3.对页面的解析能力

关于这一条，基本上就是靠特定语言的第三方包来完成网页的解析。如果要从零开始自己实现一个 HTML 解析器，难度和时间上的阻碍都是很大的。而对于复杂的基于大量 Javascript 运算生成的网页或者请求，则可以通过调度浏览器环境来完成。这一条上，Python 是绝对胜任的。

4.对数据库的操作能力（mysql）

对数据库的操作能力上，Python 有官方及第三方的连接库。另外，对于爬虫抓取的数据，存储在 NoSQL 型数据库个人认为更加合适。

5.爬取效率

确实脚本语言的运算速度不高，但是相对于特定网站反爬虫机制强度以及网络 IO 的速度，这几门语言的速度诧异都可以忽略不计，而在于开发者的水平。如果利用好发送网络请求的等待时间处理另外的事情（多线程、多进程或者协程），那么各语言效率上是不成问题的。

6.代码量

这一点上 Python 是占有优势的，众所周知 Python 代码简洁著称，只要开发者水平到位，Python 代码可以像伪代码一样简洁易懂，且代码量较低。

其实以上功能很多语言和工具都能做，但是用 python 能够干得最快，最干净。Life is short，u need python.

接下来我们来介绍什么是爬虫。

1.2 什么是爬虫？

1.2.1 爬虫的由来

随着网络的迅速发展，万维网成为大量信息的载体，如何有效地提取并利用这些信息成为一个巨大的挑战。搜索引擎(Search Engine)，例如传统的通用搜索引擎 AltaVista，Yahoo!和 Google 等，作为一个辅助人们检索信息的工具成为用户访问万维网的入口和指南。但是，这些通用性搜索引擎也存在着一定的局限性，如：

(1)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，通用搜索引擎所返回的结果包含大量用户不关心的网页。

(2)通用搜索引擎的目标是尽可能大的网络覆盖率，有限的搜索引擎服务器资源与无限的网络数据资源之间的矛盾将进一步加深。

(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 ,通用搜索引擎往往对这些信息含量密集且具有一定结构的数据无能为力 ,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。

为了解决上述问题，定向抓取相关网页资源的聚焦爬虫应运而生。聚焦爬虫是一个自动下载网页的程序，它根据既定的抓取目标，有选择的访问万维网上的网页与相关的链接，获取所需要的信息。与通用爬虫(general purpose web crawler)不同，聚焦爬虫并不追求大的覆盖，而将目标定为抓取与某一特定主题内容相关的网页，为面向主题的用户查询准备数据资源。

网络爬虫是一个自动提取网页的程序，它为搜索引擎从万维网上下载网页，是搜索引擎的重要组成。传统爬虫从一个或若干初始网页的 URL 开始，获得初始网页上的 URL，在抓取网页的过程中，不断从当前页面上抽取新的 URL 放入队列,直到满足系统的一定停止条件。聚焦爬虫的工作流程较为复杂，需要根据一定的网页分析算法过滤与主题无关的链接，保留有用的链接并将其放入等待抓取的 URL 队列。然后，它将根据一定的搜索策略从队列中选择下一步要抓取的网页 URL，并重复上述过程，直到达到系统的某一条件时停止。另外，所有被爬虫抓取的网页将会被系统存贮，进行一定的分析、过滤，并建立索引，以便之后的查询和检索；对于聚焦爬虫来说，这一过程所得到的分析结果还可能对以后的抓取过程给出反馈和指导。聚集爬虫的逻辑结构图，如图 1-3 所示。

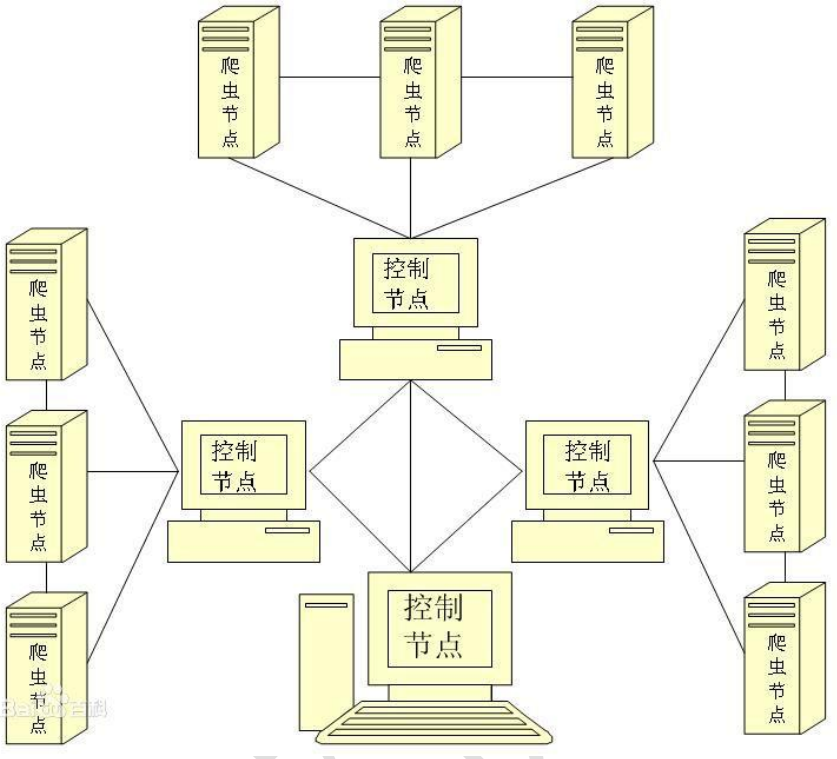

图 1-3　聚集爬虫的逻辑结构图

相对于通用网络爬虫，聚焦爬虫还需要解决三个主要问题：

(1) 对抓取目标的描述或定义；

(2) 对网页或数据的分析与过滤；

(3) 对 URL 的搜索策略。

1.2.2 爬虫的定义

网络爬虫，即 Web Spider，是一个很形象的名字。把互联网比喻成一个蜘蛛网，那么 Spider 就是在网上爬来爬去的蜘蛛。网络蜘蛛是通过网页的链接地址来寻找网页的。

从网站某一个页面（通常是首页）开始，读取网页的内容，找到在网页中的其它链接地址，然后通过这些链接地址寻找下一个网页，这样一直循环下去，直到把这个网站所有的网页都抓取完为止。

如果把整个互联网当成一个网站，那么网络蜘蛛就可以用这个原理把互联网上所有的网页都抓取下来。这样看来，网络爬虫就是一个爬行程序，一个抓取网页的程序。网络爬虫的基本操作是抓取网页。

1.2.3 爬虫的工作过程

在用户浏览网页的过程中，我们可能会看到许多好看的图片，比如 http://image.baidu.com/ ，我们会看到几张的图片以及百度搜索框，这个过程其实就是用户输入网址之后，经过 DNS 服务器，找到服务器主机，向服务器发出一个请求，服务器经过解析之后，发送给用户的浏览器 HTML、JS、CSS 等文件，浏览器解析出来，用户便可以看到形形色色的图片了。

因此，用户看到的网页实质是由 HTML 代码构成的，爬虫爬来的便是这些内容，通过分析和过滤这些 HTML 代码，实现对图片、文字等资源的获取。

想象你是一只蜘蛛，现在你被放到了互联“网”上。那么，你需要把所有的网页都看一遍。怎么办呢？没问题呀，你就随便从某个地方开始，比如说人民日报的首页，我们用$表示吧。

在人民日报的首页，你看到那个页面引向的各种链接。于是你很开心地从爬到了“国内新闻”那个页面。太好了，这样你就已经爬完了俩页面（首页和国内新闻）！暂且不用管爬下来的页面怎么处理的，你就想象你把这个页面完完整整抄成了个 html 放到了你身上。

突然你发现， 在国内新闻这个页面上，有一个链接链回“首页”。作为一只聪明的蜘蛛，你肯定知道你不用爬回去的吧，因为你已经看过了啊。所以，你需要用你的脑子，存下你已经看过的页面地址。这样，每次看到一个可能需要爬的新链接，你就先查查你脑子里是不是已经去过这个页面地址。如果去过，那就别去了。

好的，理论上如果所有的页面可以从首页达到的话，那么可以证明你一定可以爬完所有的网页。但是为什么爬虫事实上是个非常复杂的东西——搜索引擎公司通常有一整个团队来维护和开发。

因为就现在网络资源的大小而言，即使很大的搜索引擎也只能获取网络上可得到资源的一小部分。2001 年由劳伦斯河盖尔斯共同做的一项研究指出，没有一个搜索引擎抓取的内容达到网络的 16%。网络爬虫通常仅仅下载网页内容的一部分，并且不管你的带宽有多大，只要你的机器下载网页的速度是瓶颈的话，那么你只有加快这个速度。那么我们就需要提高效率——使用多线程。这就是我们为什么使用 Scrapy 的原因。

1.2.4 爬虫存在的意义

爬虫不仅仅只是获取数据，他有着许多强大的作用，例如：

- 爬取数
- 分析并推送
- 资源批量下载
- 数据监控
- 社会计算方面的统计和预测
- 机器学习

- 网络爬虫
- 建立机器翻译的语料库
- 搭建大数据的数据库

1.3 Python 爬虫入门

1.3.1 Urllib 库的基本使用

1. 分分钟扒一个网页下来

怎样扒网页呢？其实就是根据 URL 来获取它的网页信息，虽然我们在浏览器中看到的是一幅幅优美的画面，但是其实是由浏览器解释才呈现出来的，实质它是一段 HTML 代码，加 JS、CSS，如果把网页比作一个人，那么 HTML 便是他的骨架，JS 便是他的肌肉，CSS 便是它的衣服。所以最重要的部分是存在于 HTML 中的，下面我们就写个例子来扒一个网页下来。代码如 Code1-4 所示。

```
import urllib2
'''
传入一个 url 链接，协议是 http 协议
url，URL 链接
data，访问 url 时候传送的数据包，默认 None
timeout，设置超时，默认 socket_GOLBA_DEFAULT_TIMEOUT
返回一个 response 对象
'''

response = urllib2.urlopen('http://www.ixiaochouyu.com')
# read 可以获取到的网页内容
print response.read().decode('gbk').encode('utf-8')
```

Code 1-4 爬取北风网的基础代码

是的你没看错，真正的程序就两行，把它保存成 demo.py，进入该文件的目录，执行如下命令查看运行结果，感受一下，使用命令 python demo.py，运行效果如 Code1-5 所示。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gbk" />
<meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=1.0,
maximum-scale=1.0,user-scalable=no" id="viewport">
<meta name="Keywords" content="java 培训,大数据培训,IT 培训,IT 培训机构,web 前端培训,html5
培训" />
<meta name="Description" content="0 基础学 IT,高薪包就业！北风网拥有 IT 在线教育培训课程包括
java 培训,html5  培训,web 前端培训,大数据培训,软件测试培训,android 培训，ios 培训等,是国内领先的
IT 培训机构。" />
<meta property="qc:admins" content="5264211717612516567 6375" />
<meta property="wb:webmaster" content="33a538e7275dc61f" />
<meta name="baidu-site-verification" content="X58MEu1ru1" />
<title>北风网-中国知名 IT 培训机构，IT 在线教育十强</title>
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="themes/newixiaochouyu/css/new1611/reset.css">
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/animate.css">
<link rel="stylesheet"
href="themes/newixiaochouyu/css/new1611/h_release/jquery.mmenu.all.css" />
<link rel="stylesheet" type="text/css"
```

```
href="themes/newixiaochouyu/css/new1611/h_release/font.css"/>
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/h_release/component.css"/>
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/h_release/menu.css"/>
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/h_release/subjectTemplate1.0.css"/>
<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/ibfIndex2.0.css">
<!-- <script type="text/javascript" src="themes/newixiaochouyu/js/zepto.js"></script> -->
<script type="text/javascript"
src="themes/newixiaochouyu/js/new1611/jquery-1.11.1.min.js"></script>
<script type="text/javascript" src="themes/newixiaochouyu/js/new1611/ibfbase.js"></script>
<script type="text/javascript"
src="themes/newixiaochouyu/js/new1611/subjectTemplate2.0.js" ></script>
<script type="text/javascript"
src="themes/newixiaochouyu/js/new1611/template-native1.0.js"></script>
<style>
    .basehead {
        position: absolute;

        z-index: 2;
    }
```

```
        @media (max-width:969px) {
                .headerPhone {
                        position: absolute;


                        z-index: 2;
                }
        }
        .indexwrap2 {
       margin-top: 40px;
}
</style>
</head>
<body style="background:#f2f2f2;">
<div class="wrap">
<script>
                        //弹出 53kf 对话框
        function open53kf(href){


                href=href?href:'/kf.html'
                window.open (href,'newwindow','height=550,width=720');
        }
</script>


<link rel="stylesheet" type="text/css"
```

```
href="themes/newixiaochouyu/css/new1611/announcement.css">


<link rel="stylesheet" type="text/css"
href="themes/newixiaochouyu/css/new1611/pages/fixedadnew2.css">


<div class="events-announcement">
```

Code 1-5　爬取北风网 HTML 代码运行效果

看，这个网页的源码已经被我们扒下来了，是不是很酸爽？

2． 分析扒网页的方法

那么我们来分析这两行代码，第一行

```
response = urllib2.urlopen('http://www.ixiaochouyu.com')
```

首先我们调用的是 urllib2 库里面的 urlopen 方法，传入一个 URL，这个网址是北风网首页，协议是 HTTP 协议，当然你也可以把 HTTP 换做 FTP,FILE,HTTPS 等等，只是代表了一种访问控制协议，urlopen 一般接受三个参数，它的参数如下：

```
urlopen(url, data, timeout)
```

第一个参数 url 即为 URL，第二个参数 data 是访问 URL 时要传送的数据，第三个 timeout 是设置超时时间。

第二三个参数是可以不传送的，data 默认为空 None，timeout 默认为 socket._GLOBAL_DEFAULT_TIMEOUT

第一个参数 URL 是必须要传送的，在这个例子里面我们传送了百度的 URL，执行 urlopen 方法之后，返回一个 response 对象，返回信息便保存在这里面。

```
print response.read()
```

response 对象有一个 read 方法，可以返回获取到的网页内容。

如果不加 read 直接打印会是什么？答案如下：

```
<addinfourl at 56705352L whose fp = <socket._fileobject object at 0x0000000003519D68>>
```

直接打印出了该对象的描述，所以记得一定要加 read 方法，否则它不出来内容可就不怪我咯！

3． 构造 Request

其实上面的 urlopen 参数可以传入一个 request 请求,它其实就是一个 Request 类的实例，构造时需要传入 Url,Data 等等的内容。比如上面的两行代码，我们可以这么改写

```
# coding:utf-8

import urllib2

#  构建 Request

request = urllib2.Request('http://www.ixiaochouyu.com/')

response = urllib2.urlopen(request)

# read，可以获取到的网页内容

print response.read()
```

运行结果是完全一样的，只不过中间多了一个 request 对象，推荐大家这么写，因为在构建请求时还需要加入好多内容，通过构建一个 request，服务器响应请求得到应答，这样显得逻辑上清晰明确。

4． POST 和 GET 数据传送

上面的程序演示了最基本的网页抓取，不过，现在大多数网站都是动态网页，需要你动态地传递参数给它，它做出对应的响应。所以，在访问时，我们需要传递数据给它。最常见的情况是什么？对了，就是登录注册的时候呀。

把数据用户名和密码传送到一个 URL，然后你得到服务器处理之后的响应，这个该怎么办？下面让我来为小伙伴们揭晓吧！

数据传送分为 POST 和 GET 两种方式，两种方式有什么区别呢？

最重要的区别是 GET 方式是直接以链接形式访问，链接中包含了所有的参数，当然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，不过如果你想直接查看提交了什么就不太方便了，大家可以酌情选择。

POST 方式：

上面我们说了 data 参数是干嘛的？对了，它就是用在这里的，我们传送的数据就是这个参数 data，下面演示一下 POST 方式，代码如 Code1-6 所示。

```
# coding: utf-8

import urllib2

import urllib
```

```
url = "http://123.207.11.209/index.php/Home/Index/checkLogin"
# 建立数据包
value = {'admin_name': 'root_python', 'admin_password': 'zmzyroot'}
req = urllib2.Request(url)
data = urllib.urlencode(value)
# 传入 cookie
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor())
response = opener.open(req, data)
print response.read()
```

Code 1-6 POST 方式代码（1）

我们引入了 urllib 库，现在我们模拟登陆 CSDN，当然上述代码可能登陆不进去，因为还要做一些设置头部 header 的工作，或者还有一些参数没有设置全，还没有提及到在此就不写上去了，在此只是说明登录的原理。我们需要定义一个字典，名字为 values，参数我设置了 username 和 password，下面利用 urllib 的 urlencode 方法将字典编码，命名为 data，构建 request 时传入两个参数，url 和 data，运行程序，即可实现登陆，返回的便是登陆后呈现的页面内容。当然你可以自己搭建一个服务器来测试一下。

注意上面字典的定义方式还有一种，下面的写法是等价的，如 Code1-7 所示。

```
# coding: utf-8
import urllib2
import urllib

url = "http://123.207.11.209/index.php/Home/Index/checkLogin"
# 建立数据包
value = {}
```

```
value['admin_name'] = 'root_python'
value['admin_password'] = 'zmzyroot'
req = urllib2.Request(url)
data = urllib.urlencode(value)
# 传入 cookie
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor())
response = opener.open(req, data)
print response.read()
```

Code 1-7　POST 方式代码（2）

以上方法便实现了 POST 方式的传送

GET 方式：

至于 GET 方式我们可以直接把参数写到网址上面，直接构建一个带参数的 URL 出来即可，如 Code1-8 所示。

```
# coding: utf-8
import urllib
import urllib2

values = {}
data = urllib.urlencode(values)
url = "http://www.baidu.com/s?wd=python"
request = urllib2.Request(url, data)
response = urllib2.urlopen(request)
print response.read()
```

Code 1-8　GET 方式代码

你可以 print geturl，打印输出一下 url，发现其实就是原来的 url 加？然后加编码后的参数

http://passport.csdn.net/account/login?username=1016903103%40qq.com&password=XXXX

和我们平常 GET 访问方式一模一样，这样就实现了数据的 GET 方式传送。

1.3.2 Urllib 库的高级用法

1. 设置 Headers

有些网站不会同意程序直接用上面的方式进行访问，如果识别有问题，那么站点根本不会响应，所以为了完全模拟浏览器的工作，我们需要设置一些 Headers 的属性。

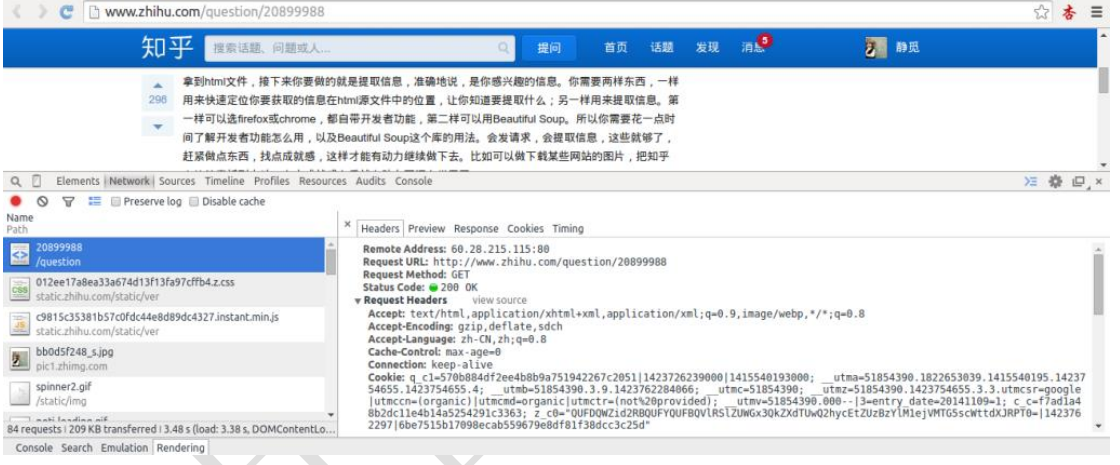

图 1-9 调试操作

首先，打开我们的浏览器，调试浏览器 F12，我用的是 Chrome，打开网络监听，示意如下，比如知乎，点登录之后，我们会发现登陆之后界面都变化了，出现一个新的界面，实质上这个页面包含了许许多多的内容，这些内容也不是一次性就加载完成的，实质上是执行了好多次请求，一般是首先请求 HTML 文件，然后加载 JS，CSS 等等，经过多次请求之后，网页的骨架和肌肉全了，整个网页的效果也就出来了，操作效果如图 1-9 所示。

拆分这些请求，我们只看一第一个请求，你可以看到，有个 Request URL，还有 headers，下面便是 response，图片显示得不全，小伙伴们可以亲身实验一下。那么这个头中包含了许许多多是信息，有文件编码啦，压缩方式啦，请求的 agent 啦等等。

其中，agent 就是请求的身份，如果没有写入请求身份，那么服务器不一定会响应，所以可以在 headers 中设置 agent,例如下面的例子，这个例子只是说明了怎样设置的 headers，小伙伴们看一下设置格式就好，如 Code1-10 所示。

```
# coding: utf-8
import urllib
import urllib2
values = {}
data = urllib.urlencode(values)
user_agent = 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)
Maxthon/5.0 Chrome/47.0.2526.73'
headers = {'User-Agent': user_agent}
url = "http://www.baidu.com/s?wd=python"
request = urllib2.Request(url, data)
response = urllib2.urlopen(request)
print response.read().decode('utf-8')
print url
```

Code 1-10 设置 header 代码

这样，我们设置了一个 headers，在构建 request 时传入，在请求时，就加入了 headers 传送，服务器若识别了是浏览器发来的请求，就会得到响应。

另外，我们还有对付“反盗链”的方式，对付防盗链，服务器会识别 headers 中的 referer 是不是它自己，如果不是，有的服务器不会响应，所以我们还可以在 headers 中加入 referer

例如我们可以构建下面的 headers，代码如 Code1-11 所示。

```
Headers = { ' User-Agent':  'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT) ',  ' Referer':  '
http://www.zhihu.com/articles'}
```

Code 1-11 设置 referer 代码

同上面的方法，在传送请求时把 headers 传入 Request 参数里，这样就能应付防盗链了。

另外 headers 的一些属性，下面的需要特别注意一下：

- User-Agent：有些服务器或 Proxy 会通过该值来判断是否是浏览器发出的请求
- Content-Type：在使用 REST 接口时，服务器会检查该值，用来确定 HTTP Body 中的内容该怎样解析。
- application/xml ： 在 XML RPC，如 RESTful/SOAP 调用时使用

- application/json ： 在 JSON RPC 调用时使用
- application/x-www-form-urlencoded ： 浏览器提交 Web 表单时使用

在使用服务器提供的 RESTful 或 SOAP 服务时， Content-Type 设置错误会导致服务器拒绝服务，其他的有必要的可以审查浏览器的 headers 内容，在构建时写入同样的数据即可。

2. Proxy（代理）的设置

urllib2 默认会使用环境变量 http_proxy 来设置 HTTP Proxy。假如一个网站它会检测某一段时间某个 IP 的访问次数，如果访问次数过多，它会禁止你的访问。所以你可以设置一些代理服务器来帮助你做工作，每隔一段时间换一个代理，网站君都不知道是谁在捣鬼了，这酸爽！

下面一段代码说明了代理的设置用法，如 Code1-12 所示。

```
# coding: utf-8
import urllib
import urllib2

url = 'http://www.ixiaochouyu.com'
user_agent = 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko) Maxthon/5.0 Chrome/47.0.2526.73'
headers = {'User-Agent': user_agent}
values = {}
data = urllib.urlencode(values)
enable_proxy = True
proxy_handler = urllib2.ProxyHandler({"http":"http://some-proxy.com:8080"})
null_proxy_handler = urllib2.ProxyHandler({})
if enable_proxy:
    opener = urllib2.build_opener(proxy_handler)
else:
    opener = urllib2.build_opener(null_proxy_handler)
print urllib2.install_opener(opener)
```

Code 1-12 代理设置代码

3. Timeout 设置

上一节已经说过 urlopen 方法了，第三个参数就是 timeout 的设置，可以设置等待多久超时，为了解决一些网站实在响应过慢而造成的影响。

例如下面的代码,如果第二个参数 data 为空那么要特别指定是 timeout 是多少，写明形参，如果 data 已经传入，则不必声明，代码如 Code1-13 和 1-14 所示。

```
# coding: utf-8
import urllib2
response = urllib2.urlopen('https://github.com/', timeout=10)
print response.read()
```

Code 1-13 Timeout 代码设置

4. 使用 HTTP 的 PUT 和 DELETE 方法

http 协议有六种请求方法，get,head,put,delete,post,options，我们有时候需要用到 PUT 方式或者 DELETE 方式请求。

PUT：这个方法比较少见。HTML 表单也不支持这个。本质上来讲， PUT 和 POST 极为相似，都是向服务器发送数据，但它们之间有一个重要区别，PUT 通常指定了资源的存放位置，而 POST 则没有，POST 的数据存放位置由服务器自己决定。

DELETE：删除某一个资源。基本上这个也很少见，不过还是有一些地方比如 amazon 的 S3 云服务里面就用的这个方法来删除资源。 如果要使用 HTTP PUT 和 DELETE ，只能使用比较低层的 httplib 库。虽然如此，我们还是能通过下面的方式，使 urllib2 能够发出 PUT 或 DELETE 的请求，不过用的次数的确是少，在这里提一下。

操作代码，如 Code1-14 所示。

```
# coding: utf-8
import urllib2
request = urllib2.Request('http://www.ixiaochouyu.com/')
request.get_method = 'PUT'   # DELETE/PUT
response = urllib2.urlopen(request)
```

Code 1-14 使用 HTTP 的 PUT 和 DELETE 方法

5. 使用 DebugLog

可以通过下面的方法把 Debug Log 打开，这样收发包的内容就会在屏幕上打印出来，方便调试，这个也不太常用，仅提一下。

```
# coding: utf-8
import urllib2
import urllib
# 设置 HTTP 处理机制
httpHandler = urllib2.HTTPHandler(debuglevel=1)
# 设置 HTTPS 处理机制
httpsHandler = urllib2.HTTPSHandler(debuglevel=1)
# 构建 opener
```

```
opener = urllib2.build_opener(httpHandler,httpsHandler)
# 载入 opener
urllib2.install_opener(opener)
values = {}
values['keyword'] = 'aaaa'
data = urllib.urlencode(values)
response = urllib2.urlopen('https://www.baidu.com',data)
print response
```

Code 1-15 使用 DebugLog

1.3.3 URLError 异常处理

大家好，本节在这里主要说的是 URLError 还有 HTTPError，以及对它们的一些处理。

1. URLError

首先解释下 URLError 可能产生的原因：

- 网络无连接，即本机无法上网
- 连接不到特定的服务器
- 服务器不存在

在代码中，我们需要用 try-except 语句来包围并捕获相应的异常。下面是一个例子，先感受下它的风骚，代码如 Code1-16 所示。

```
# coding:utf-8
import urllib2
request = urllib2.Request('http://sadas.csdn.net')
try:
    print urllib2.urlopen(request).read()
# 捕获异常
except urllib2.URLError,e:
    print e.reason
else:
    print 'ok'
```

Code 1-16 使用 try-except

我们利用了 urlopen 方法访问了一个不存在的网址，运行结果如下 Cod1-17 所示。

```
[Errno 11001] getaddrinfo failed
```

Code 1-17 异常报错

它说明了错误代号是 11004，错误原因是 getaddrinfo failed。

2. HTTPError

HTTPError 是 URLError 的子类，在你利用 urlopen 方法发出一个请求时，服务器上都会对应一个应答对象 response，其中它包含一个数字”状态码”。举个例子，假如 response 是一个”重定向”，需定位到别的地址获取文档，urllib2 将对此进行处理。

其他不能处理的，urlopen 会产生一个 HTTPError，对应相应的状态吗，HTTP 状态码表示 HTTP 协议所返回的响应的状态。下面将状态码归结如下表 1-18 所示：

状态码	状态码英文名称	描述
100	Continue 继续	客户端应继续其请求
101	Switching Protocols 切换协议	服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到 HTTP 的新版本协议
200	OK 请求成功	一般用于 GET 与 POST 请求
201	Created 已创建	成功请求并创建了新的资源
202	Accepted 已接受	已经接受请求，但未处理完成
203	Non-Authoritative Information 非授权信息	请求成功。但返回的 meta 信息不在原始的服务器，而是一个副本
204	No Content 无内容	服务器成功处理，但未返回内容。在未更新网页的情况下，可确保浏览器继续显示当前文档
205	Reset Content 重置内容	服务器处理成功，用户终端（例如：浏览器）应重置文档视图。可通过此返回码清除浏览器的表单域
206	Partial Content 部分内容	服务器成功处理了部分 GET 请求
300	Multiple Choices 多种选择	请求的资源可包括多个位置，相应可返回一个资源特征与地址的列表用于用户终端（例如：浏览器）选择
301	Moved Permanently 永久移动	请求的资源已被永久的移动到新 URI，返回信息会包括新的 URI，浏览器会自动定向到新 URI。今后任何新的请求都应使用新的 URI 代替
302	Found 临时移动	与 301 类似。但资源只是临时被移动。客户端应继续使用原有 URI
303	See Other 查看其它地址	与 301 类似。使用 GET 和 POST 请求查看
304	Not Modified 未修改	所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源
305	Use Proxy 使用代理	所请求的资源必须通过代理访问
306	Unused	已经被废弃的 HTTP 状态码

307	Temporary Redirect 临时重定向	与 302 类似。使用 GET 请求重定向
400	Bad Request	客户端请求的语法错误，服务器无法理解
401	Unauthorized	请求要求用户的身份认证
402	Payment Required	保留，将来使用
403	Forbidden	服务器理解请求客户端的请求，但是拒绝执行此请求
404	Not Found	服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面
405	Method Not Allowed	客户端请求中的方法被禁止
406	Not Acceptable	服务器无法根据客户端请求的内容特性完成请求
407	Proxy Authentication Required	请求要求代理的身份认证，与 401 类似，但请求者应当使用代理进行授权
408	Request Time-out	服务器等待客户端发送的请求时间过长，超时
409	Conflict	服务器完成客户端的 PUT 请求是可能返回此代码，服务器处理请求时发生了冲突
410	Gone	客户端请求的资源已经不存在。410 不同于 404，如果资源以前有现在被永久删除了可使用 410 代码，网站设计人员可通过 301 代码指定资源的新位置
411	Length Required	服务器无法处理客户端发送的不带 Content-Length 的请求信息
412	Precondition Failed	客户端请求信息的先决条件错误
413	Request Entity Too Large	由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。如果只是服务器暂时无法处理，则会包含一个 Retry-After 的响应信息
414	Request-URI Too Large	请求的 URI 过长（URI 通常为网址），服务器无法处理
415	Unsupported Media Type	服务器无法处理请求附带的媒体格式
416	Requested range not satisfiable	客户端请求的范围无效
417	Expectation Failed	服务器无法满足 Expect 的请求头信息
500	Internal Server Error	服务器内部错误，无法完成请求
501	Not Implemented	服务器不支持请求的功能，无法完成请求
502	Bad Gateway	充当网关或代理的服务器，从远端服务器接收到了一个无效的请求

503	Service Unavailable	由于超载或系统维护 ,服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的 Retry-After 头信息中
504	Gateway Time-out	充当网关或代理的服务器 ,未及时从远端服务器获取请求
505	HTTP Version not supported	服务器不支持请求的 HTTP 协议的版本 ,无法完成处理

表 1-18 HTTP 状态码

HTTPError 实例产生后会有一个 code 属性 ,这就是是服务器发送的相关错误号。因为 urllib2 可以为你处理重定向，也就是 3 开头的代号可以被处理，并且 100-299 范围的号码指示成功，所以你只能看到 400-599 的错误号码。

下面我们写一个例子来感受一下，捕获的异常是 HTTPError，它会带有一个 code 属性，就是错误代号，另外我们又打印了 reason 属性，这是它的父类 URLError 的属性，代码如 Code1-19 所示。

```
# coding:utf-8
import urllib2
request = urllib2.Request('http://my.csdn.net/sadas')
try:
    print urllib2.urlopen(request).read()
#  捕获 HttpError 异常
except urllib2.HTTPError, e:
    print e.code
    print e.reason
```

Code 1-19 参考代码

运行结果如下 Code1-20 所示。

```
403
Forbidden
```

Code 1-20 参考代码运行效果

错误代号是 403，错误原因是 Forbidden，说明服务器禁止访问。

我们知道，HTTPError 的父类是 URLError，根据编程经验，父类的异常应当写到子类异常的后面，如果子类捕获不到，那么可以捕获父类的异常，所以上述的代码可以这么改写，如 Code1-21 所示。

```
# coding:utf-8
import urllib2
request = urllib2.Request('http://my.csdn.net/sadas')
```

```
try:
    print urllib2.urlopen(request).read()
# 捕获 HttpError 异常
except urllib2.HTTPError, e:
    print e.code
    print e.reason
# 捕获 URLError 异常
except urllib2.URLError, e:
    print e.reason
else:
    print 'ok'
```

Code 1-21 参考代码改进

首先对异常的属性进行判断，以免出现属性输出报错的现象。

以上，就是对 URLError 和 HTTPError 的相关介绍，以及相应的错误处理办法。

1.3.4 Cookie 实际应用

大家好哈，上一节我们研究了一下爬虫的异常处理问题，那么接下来我们一起来看一下 Cookie 的使用。

为什么要使用 Cookie 呢？

Cookie，指某些网站为了辨别用户身份、进行 session 跟踪而储存在用户本地终端上的数据（通常经过加密）

比如说有些网站需要登录后才能访问某个页面，在登录之前，你想抓取某个页面内容是不允许的。那么我们可以利用 Urllib2 库保存我们登录的 Cookie，然后再抓取其他页面就达到目的了。

在此之前呢，我们必须先介绍一个 opener 的概念。

1. Opener

当你获取一个 URL 你使用一个 opener(一个 urllib2.OpenerDirector 的实例)。在前面，我们都是使用的默认的 opener，也就是 urlopen。它是一个特殊的 opener，可以理解成 opener 的一个特殊实例，传入的参数仅仅是 url，data，timeout。

如果我们需要用到 Cookie，只用这个 opener 是不能达到目的的，所以我们需要创建更一般的 opener 来实现对 Cookie 的设置。

2. Cookielib

cookielib 模块的主要作用是提供可存储 cookie 的对象，以便于与 urllib2 模块配合使用来访问 Internet 资源。Cookielib 模块非常强大，我们可以利用本模块的 CookieJar 类的对象来捕获 cookie 并在后续连接请求时重新发送，比如可以实现模拟登录功能。该模块主要的对象有 CookieJar、FileCookieJar、MozillaCookieJar、LWPCookieJar。

它们的关系：CookieJar —-派生—->FileCookieJar —-派生—->MozillaCookieJar 和 LWPCookieJar

3. 获取 Cookie 保存到变量

首先，我们先利用 CookieJar 对象实现获取 cookie 的功能，存储到变量中，参考代码如下，如 Code1-22 所示。

```
# coding:utf-8
import urllib2
import cookielib

# 声明 CookieJar 对象
cookie = cookielib.CookieJar()
# 利用 urllib2 库的 HTTPCookieProcessor 来创建 cookie 处理器
handler = urllib2.HTTPCookieProcessor(cookie)
# handler 来构建 opener
opener = urllib2.build_opener(handler)
urllib2.install_opener(opener)
# open 方法传入 url
response =
opener.open('https://passport.csdn.net/account/login?from=http://my.csdn.net/my/mycsdn')
for item in cookie:
    print 'Name:' + item.name
    print 'Value:' + item.value
```

Code 1-22 CookielJar 操作代码

我们使用以上方法将 cookie 保存到变量中，然后打印出了 cookie 中的值，运行结果如下，如 Code1-23 所示。

```
Name:JSESSIONID
Value:03E6C0F8279E92CE49F2214FF01C557F.tomcat1
Name:LSSC
Value:LSSC-100936-EHNHdPhkyuaHvTPjrGcrKk1koW0p9P-passport.csdn.net
```

Code 1-23 运行结果

4. 保存 Cookie 到文件

在上面的方法中，我们将 cookie 保存到了 cookie 这个变量中，如果我们想将 cookie 保存到文件中该怎么做呢？这时，我们就要用到 FileCookieJar 这个对象了，在这里我们使用它的子类 MozillaCookieJar 来实现 Cookie 的保存，实现代码如 Code1-24 所示。

```
# coding:utf-8
import urllib2
import cookielib
# 定义文件
filename = 'cookie.txt'
# 声明一个 MozillaCookieJar 对象来保存 cookie 到文件中
cookie = cookielib.MozillaCookieJar(filename)
# 利用 urllib2 库的 HTTPCookieProcessor 来创建 cookie 处理器
handler = urllib2.HTTPCookieProcessor(cookie)
# handler 来构建 opener
opener = urllib2.build_opener(handler)
urllib2.install_opener(opener)
# open 方法传入 url
response =
opener.open('https://passport.csdn.net/account/login?from=http://my.csdn.net/my/mycsdn')
# 保存 cookie 到文件
# ignore_discard，即使 cookie 被丢弃，是否保存
# ignore_expires，如果在该文件中 cookie 已存在，是否覆盖
cookie.save(ignore_discard=True, ignore_expires=True)
```

Code 1-24 4.保存 Cookie 到文件

由此可见，ignore_discard 的意思是即使 cookies 将被丢弃也将它保存下来，ignore_expires 的意思是如果在该文件中 cookies 已经存在，则覆盖原文件写入，在这里，我们将这两个全部设置为 True。运行之后，cookies 将被保存到 cookie.txt 文件中，我们查看一下内容，如下 Code1-25 所示。

```
# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This is a generated file!  Do not edit.

passport.csdn.net  FALSE  /  FALSE  JSESSIONID
    73E0FCD65A4BE1B7AEDDED6D9C9DC179.tomcat2
```

```
passport.csdn.net   FALSE    /     TRUE      1510889006   LSSC
      LSSC-96904-coeZcPZkCa9Y9tvckHIrBBv7CATige-passport.csdn.net
```

Code 1-25 cookie 文件内容

5. 从文件中获取 Cookie 并访问

那么我们已经做到把 Cookie 保存到文件中了，如果以后想使用，可以利用下面的方法来读取 cookie 并访问网站，感受一下，代码如 Code1-26 所示。

```
# coding:utf-8
import urllib2
import cookielib

# 创建 MozillaCookieJar 对象
cookie = cookielib.MozillaCookieJar()
# 从文件中读取 cookie 的内容到变量
cookie.load('cookie.txt', ignore_expires=True, ignore_discard=True)
# 创建请求的 request
request =
urllib2.Request('https://passport.csdn.net/account/login?from=http://my.csdn.net/my/mycsdn'
)
# 利用 urllib2 创建一个 opener，传入 cookie
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookie))
response = opener.open(request)
print response.read().decode('utf-8')
```

Code 1-26 从文件读取 cookie 再访问

设想，如果我们的 cookie.txt 文件中保存的是某个人登录百度的 cookie，那么我们提取出这个 cookie 文件内容，就可以用以上方法模拟这个人的账号登录百度。

6. 利用 cookie 模拟网站登录

```
# coding:utf-8
import urllib
import urllib2
import cookielib
filename = 'cookie.txt'
# 声明一个 MozillaCookieJar 对象用来保存 cookie 到文件中
cookie = cookielib.MozillaCookieJar(filename)
# 生成 opener
```

```
opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookie))
# 建立数据包
values = {}
values['username'] = 'zmzy'
values['password'] = '123456'
values['lt'] = 'LT-512088-b2ce4SgDBOyOCU9b7sKWKkC1EB6aoT'
values['execution'] = 'e2s1'
values['_eventId'] = 'submit'
data = urllib.urlencode(values)
url = "https://passport.csdn.net/account/login"
# 模拟登录，并把 cookie 保存到变量
result = opener.open(url, data)
# 保存 cookie 到 cookie.txt 中
cookie.save(ignore_discard=True, ignore_expires=True)
# 请求登录后的一些页面
geturl = 'http://msg.csdn.net/letters'
response = opener.open(geturl)
print response.read()
```

Code 1-27 从文件读取 cookie 再访问

上面我们以 csdn 为例，利用 cookie 实现模拟登录，并将 cookie 信息保存到文本文件中，来感受一下 cookie 大法吧！实现代码如上 Code1-27 所示。

以上程序的原理如下

创建一个带有 cookie 的 opener，在访问登录的 URL 时，将登录后的 cookie 保存下来，然后利用这个 cookie 来访问其他网址。

如登录之后才能查看的成绩查询呀，本学期课表呀等等网址，模拟登录就这么实现啦，是不是很酷炫？

好，小伙伴们要加油哦！我们现在可以顺利获取网站信息了，接下来就是把网站里面有效内容提取出来，下一节我们去会会正则表达式！

1.3.5 正则表达式

在前面我们已经搞定了怎样获取页面的内容，不过还差一步，这么多杂乱的代码夹杂文字我们怎样把它提取出来整理呢？下面就开始介绍一个十分强大的工具，正则表达式！

1. 了解正则表达式

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。 正则表达式是用来匹配字符串非常强大的工具，在其他编程语言中同样有正则表达式的概念，Python 同样不例外，利用了正则表达式，我们想要从返回的页面内容提取出我们想要的内容就易如反掌了。

正则表达式的大致匹配过程是：

1) 依次拿出表达式和文本中的字符比较，

2) 如果每一个得到字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。

3) 如果表达式中有量词或边界，这个过程会稍微有一些不同。

2. 正则表达式的语法规则

下面是 Python 中正则表达式的一些匹配规则，如图 1-28 所示

语法	说明	表达式实例	完整匹配的字符串
	字符		
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	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 abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	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。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 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。	示例将在下文中介绍。	
	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。 它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。 如果|没有被包括在()中，则它的范围是整个正则表达式。	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，分组表达式作为一个整体，可以后接数量词。表达式中的|仅在该组中有效。	(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。匹配模式将在下文中介绍。	(?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。 不消耗字符串内容。	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。 不消耗字符串内容。	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。 不消耗字符串内容。	(?<=\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。 不消耗字符串内容。	(?<!\d)a	前面不是数字的a
(?(id/name) yes-pattern |no-pattern)	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。 |no-patern可以省略。	(\d)abc(?(1)\d|abc)	1abc2 abcabc

图 1-28 正则表达式

3. 数量词的贪婪模式与非贪婪模式

正则表达式通常用于在文本中查找匹配的字符串。Python 里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；非贪婪的则相反，总是尝试匹配尽可能少的字符。例如：正则表达式“ab” 如果用于查找“abbbc”，将找到“abbb”。而如果使用非贪婪的数量词“ab?”，将找到“a”。

注：我们一般使用非贪婪模式来提取。

4. 反斜杠问题

与大多数编程语言相同，正则表达式里使用”\”作为转义字符，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符”\”，那么使用编程语言表示的正则表达式里将需要 4 个反斜杠”\\”：前两个

和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。

Python 里的原生字符串很好地解决了这个问题，这个例子中的正则表达式可以使用 r”\” 表示。同样，匹配一个数字的 ”\d” 可以写成 r”\d” 。有了原生字符串，妈妈也不用担心是不是漏写了反斜杠，写出来的表达式也更直观勒。

5. Python Re 模块

Python 自带了 re 模块，它提供了对正则表达式的支持。主要用到的方法列举如下：

1.返回 pattern 对象

```
re.compile(string[,flag])
```

2.以下为匹配所用函数

```
re.match(pattern, string[, flags])
re.search(pattern, string[, flags])
re.split(pattern, string[, maxsplit])
re.findall(pattern, string[, flags])
re.finditer(pattern, string[, flags])
re.sub(pattern, repl, string[, count])
re.subn(pattern, repl, string[, count])
```

在介绍这几个方法之前，我们先来介绍一下 pattern 的概念，pattern 可以理解为一个匹配模式，那么我们怎么获得这个匹配模式呢？很简单，我们需要利用 re.compile 方法就可以。例如

```
pattern = re.compile(r'hello')
```

在参数中我们传入了原生字符串对象，通过 compile 方法编译生成一个 pattern 对象，然后我们利用这个对象来进行进一步的匹配。

另外大家可能注意到了另一个参数 flags，在这里解释一下这个参数的含义：

参数 flag 是匹配模式，取值可以使用按位或运算符’|’ 表示同时生效，比如 re.I | re.M。

可选值有：

- re.I(全拼：IGNORECASE): 忽略大小写（括号内是完整写法，下同）
- re.M(全拼：MULTILINE): 多行模式，改变'^'和'$'的行为（参见上图）
- re.S(全拼：DOTALL): 点任意匹配模式，改变'.'的行为
- re.L(全拼：LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定
- re.U(全拼：UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于 unicode 定义的字符属性

- re.X(全拼：VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。

在刚才所说的另外几个方法例如 re.match 里我们就需要用到这个 pattern 了，下面我们一一介绍。

（1）re.match(pattern, string[, flags])

这个方法将会从 string（我们要匹配的字符串）的开头开始，尝试匹配 pattern，一直向后匹配，如果遇到无法匹配的字符，立即返回 None，如果匹配未结束已经到达 string 的末尾，也会返回 None。两个结果均表示匹配失败，否则匹配 pattern 成功，同时匹配终止，不再对 string 向后匹配。下面我们通过一个例子理解一下。如 Code1-29 所示

```
# -*- coding:utf-8 -*-

# 导入 re 模块
import re

# 将正则表达式编译成 Pattern 对象，注意 hello 前面的 r 的意思是"原生字符串"
pattern = re.compile(r'hello')

# 使用 re.match 匹配文本，获取匹配结果，无法匹配时将返回 None
result1 = re.match(pattern, 'hello')
result2 = re.match(pattern, 'helloo ZMZY!')
result3 = re.match(pattern, 'helo ZMZY!')
result4 = re.match(pattern, 'hello ZMZY!')

# 如果 1 匹配成功
if result1:
    # 使用 Match 获取分组信息
    print result1.group()
else:
    print '1 匹配失败!'

# 如果 2 匹配成功
if result2:
    # 使用 Match 获取分组信息
    print result2.group()
```

```
else:
    print '2 匹配失败!'

# 如果 3 匹配成功
if result3:
    # 使用 Match 获取分组信息
    print result3.group()
else:
    print '3 匹配失败!'

# 如果 4 匹配成功
if result4:
    # 使用 Match 获取分组信息
    print result4.group()
else:
    print '4 匹配失败!'
```

Code 1-29 match 应用代码

运行结果，如 Code1-30 所示

```
hello
hello
3 匹配失败!
hello
```

Code 1-30 match 代码运行结果

匹配分析

(1) 第一个匹配，pattern 正则表达式为‘ hello’ ，我们匹配的目标字符串 string 也为 hello，从头至尾完全匹配，匹配成功。

(2) 第二个匹配，string 为 helloo CQC，从 string 头开始匹配 pattern 完全可以匹配，pattern 匹配结束，同时匹配终止，后面的 o CQC 不再匹配，返回匹配成功的信息。

(3) 第三个匹配，string 为 helo CQC，从 string 头开始匹配 pattern，发现到 ‘o’ 时无法完成匹配，匹配终止，返回 None

(4) 第四个匹配，同第二个匹配原理，即使遇到了空格符也不会受影响。

我们还看到最后打印出了 result.group()，这个是什么意思呢？下面我们说一下关于 match 对象的的属性和方法 Match 对象是一次匹配的结果，包含了很多关于此次匹配的信息，可以使用 Match 提供的可读属性或方法来获取这些信息，如表 1-31 所示。

属性	描述
string	匹配时使用的文本
re	匹配时使用的 Pattern 对象
pos	文本中正则表达式开始搜索的索引。值与 Pattern.match()和 Pattern.search()方法的同名参数相同
endpos	文本中正则表达式结束搜索的索引。值与 Pattern.match()和 Pattern.search()方法的同名参数相同
lastindex	最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组，将为 None.
lastgroup	最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组 ,将为 None

方法	描述
group([group1, ..])	获得一个活多个分组截获的字符串；指定多个参数时将以元组形式返回。group1 可以使用编号也可以使用别名；编号 0 代表整个匹配的子串；不填写参数时，返回 group(0)；没有截获字符串的组返回 None；截获了多次的组返回最后一次截获的子串
groups([default])	以元组形式返回全部分组截获的字符串。相当于调用 group(1,2,...last)。default 表示没有截获字符串的组以这个值替代，默认为 None。
groupdict([default])	返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。default 含义同上
start([group])	返回指定的组截获的子串在 string 中的起始索引（子串第一个字符的索引）。group 默认值为 0。
end([group])	返回指定的组截获的子串在 string 中的结束索引（子串最后一个字符的索引+1）。group 默认值为 0。
span([group])	返回(start(group), end(group))。
expand(template)	将匹配到的分组代入 template 中然后返回。template 中可以使用\id 或\g、\g 引用分组，但不能使用编号 0。\id 与\g 是等价的；但\10 将被认为是第 10 个

	分组，如果你想表达\1 之后是字符串’ 0’ ，只能使用\g0。

表 1-31 match 对象的的属性和方法

下面我们用一个例子来体会一下，代码如 Code1-32 所示。

```
# 导入 re 模块
import re
# 匹配如下内容：单词+空格+单词+任意字符
m = re.match(r'(\w+) (\w+)(?P<sign>.*)', 'hello world!')
print "m.string:", m.string
print "m.re:", m.re
print "m.pos:", m.pos
print "m.endpos:", m.endpos
print "m.lastindex:", m.lastindex
print "m.lastgroup:", m.lastgroup
print "m.group():", m.group()
print "m.group(1,2):", m.group(1, 2)
print "m.groups():", m.groups()
print "m.groupdict():", m.groupdict()
print "m.start(2):", m.start(2)
print "m.end(2):", m.end(2)
print "m.span(2):", m.span(2)
print r"m.expand(r'\g \g\g'):", m.expand(r'\2 \1\3')
```

Code 1-32 match 对象的实际应用代码

运行结果如 Code1-33 所示。

```
m.string: hello world!
m.re: <_sre.SRE_Pattern object at 0x0000000003409140>
m.pos: 0
m.endpos: 12
m.lastindex: 3
m.lastgroup: sign
m.group(): hello world!
m.group(1,2): ('hello', 'world')
m.groups(): ('hello', 'world', '!')
m.groupdict(): {'sign': '!'}
m.start(2): 6
m.end(2): 11
```

```
m.span(2): (6, 11)
m.expand(r'\g \g\g'): world hello!
```

Code 1-33 match 对象代码运行结果

（2）re.search(pattern, string[, flags])

search 方法与 match 方法极其类似，区别在于 match() 函数只检测 re 是不是在 string 的开始位置匹配，search() 会扫描整个 string 查找匹配，match（）只有在 0 位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match() 就返回 None。同样，search 方法的返回对象同样 match() 返回对象的方法和属性。我们用一个例子感受一下，代码如 Code1-34 所示。

```
# 导入 re 模块
import re
# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'world')
# 使用 search()查找匹配的子串，不存在能匹配的子串时将返回 None
# 这个例子中使用 match()无法成功匹配
match = re.search(pattern, 'hello world!')
if match:
    # 使用 Match 获得分组信息
    print match.group()
```

Code 1-34 serach 应用代码

输入出结果为 world。

（3）re.split(pattern, string[, flags])

按照能够匹配的子串将 string 分割后返回列表。maxsplit 用于指定最大分割次数，不指定将全部分割。我们通过下面的例子感受一下。

```
import re
pattern = re.compile(r'\d+')
print re.split(pattern,'one1two2three3four4')
```

输出结果为：['one', 'two', 'three', 'four', '']

（4）re.findall(pattern, string[, flags])

搜索 string，以列表形式返回全部能匹配的子串。我们通过这个例子来感受一下。

```
import re
pattern = re.compile(r'\d+')
print re.findall(pattern,'one1two2three3four4')
```

输出结果为：['1', '2', '3', '4']

（5）re.finditer(pattern, string[, flags])

搜索 string，返回一个顺序访问每一个匹配结果（Match 对象）的迭代器。我们通过下面的例子来感受一下。

```
import re
pattern = re.compile(r'\d+')
for m in re.finditer(pattern,'one1two2three3four4'):
    print m.group(),
```

输出结果为：1 2 3 4

（6）re.sub(pattern, string[, flags])

使用 repl 替换 string 中每一个匹配的子串后返回替换后的字符串。 当 repl 是一个字符串时，可以使用 \id 或 \g、\g 引用分组，但不能使用编号 0。 当 repl 是一个方法时，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。

```
import re
pattern = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'
print re.sub(pattern,r'\2 \1', s)
def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
print re.sub(pattern,func, s)
```

输出结果为：

say i, world hello!

I Say, Hello World!

（7）re.subn(pattern, string[, flags])

返回 (sub(repl, string[, count]), 替换次数)。

```
import re
pattern = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'
print re.subn(pattern,r'\2 \1', s)
def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
print re.subn(pattern,func, s)
```

输出结果为：

('say i, world hello!', 2)

('I Say, Hello World!', 2)

（8）Python Re 模块的另一种使用方式

在上面我们介绍了 7 个工具方法，例如 match，search 等等，不过调用方式都是 re.match，re.search 的方式，其实还有另外一种调用方式，可以通过 pattern.match，pattern.search 调用，这样调用便不用将 pattern 作为第一个参数传入了，大家想怎样调用皆可。

函数 API 列表，代码如图 1-35 所示。

```
match(string[, pos[, endpos]]) | re.match(pattern, string[, flags])
search(string[, pos[, endpos]]) | re.search(pattern, string[, flags])
split(string[, maxsplit]) | re.split(pattern, string[, maxsplit])
findall(string[, pos[, endpos]]) | re.findall(pattern, string[, flags])
finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags])
sub(rep1, string[, count]) | re.sub(pattern, rep1, string[, count])
subn(rep1, string[, count]) | re.subn(pattern, rep1, string[, count])
```

图 1-35 函数 API 列表

具体的调用方法不必详说了，原理都类似，只是参数的变化不同。小伙伴们尝试一下吧~

小伙伴们加油，即使这一节看得云里雾里的也没关系，接下来我们会通过一些实战例子来帮助大家熟练掌握正则表达式的。

第 2 章 爬虫库

本章工作任务

- 任务 1：熟练掌握 Requests 库安装配置及用法
- 任务 2：熟练掌握 Beautiful Soup 库安装配置及用法
- 任务 3：熟练掌握 Xpath 语法与 lxml 库的安装配置及用法

本章技能目标及重难点

编号	技能点描述	级别
1	Requests 库安装配置及用法	★★
2	Beautiful Soup 库安装配置及用法	★★
3	Xpath 语法与 lxml 库的安装配置及用法	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习爬虫利器的安装配置及使用用法，需要同学们理解正确应用 Requests、BeautifulSoup、Xpath 与 lxml，它的概念、特点。最主要的是需要大家学会合理应用相应利器。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

2.1 Urlib 进阶版之 Requests

之前我们用了 urllib 库，这个作为入门的工具还是不错的，对了解一些爬虫的基本理念，掌握爬虫爬取的流程有所帮助。入门之后，我们就需要学习一些更加高级的内容和工具来方便我们的爬取。那么这一节来简单介绍一下 requests 库的基本用法。

2.1.1 Requests 安装

利用 pip 安装

```
pip install requests
```

或者利用 easy_install

```
easy_install requests
```

通过以上两种方法均可以完成安装。

2.1.2 Requests 引入案例

首先我们引入一个小例子来感受一下。

```
import requests
r = requests.get('http://www.ixiaochouyu.com')
print type(r)
print r.status_code
print r.encoding
#print r.text
print r.cookies
```

以上代码我们请求了本站点的网址，然后打印出了返回结果的类型，状态码，编码方式，Cookies 等内容。

运行结果如下。

```
<class 'requests.models.Response'>
200
UTF-8
<RequestsCookieJar[]>
```

怎样，是不是很方便。别急，更方便的在后面呢。

2.1.3 基本请求

requests 库提供了 http 所有的基本请求方式。例如

```
r = requests.post("http://httpbin.org/post")
r = requests.put("http://httpbin.org/put")
r = requests.delete("http://httpbin.org/delete")
r = requests.head("http://httpbin.org/get")
r = requests.options("http://httpbin.org/get")
```

嗯，一句话搞定。

Requests 的上述方法还支持，如下参数：

- Headers：字典形式的头文件信息数据包
- Cookies：字典或者 CookieJar 对象形式数据包
- Files：字典形式多部分编码上传，类似于“name “：文件类似于对象”或者{“name”:文件元祖}
- auth：身份验证元组,允许基本/精华/自定义 HTTP 身份验证
- timeout：设置等待服务发送数据的时间
- allow_redirects :布尔形式。允许/拒绝 GET/OPTIONS/POST/PUT/PATCH/DELETE/HEAD 重定向。默认为“TRUE”
- proxies：字典形式的 URL 代理映射协议。
- verify：SSL 证书是否需要验证。CA_BUNDLE 路径也可以提供。默认为“TRUE”。
- stream：如果为 False，就立即下载响应内容。
- cert：如果为字符串，则传入 SSL 客户机证书文件路径（.pem）。如果为元祖，则使用（“证书”，“关键词”）方式

2.1.4 基本 GET 请求

最基本的 GET 请求可以直接用 get 方法

```
r = requests.get("http://httpbin.org/get")
```

如果想要加参数，可以利用 params 参数

```
import requests
```

```
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.get("http://httpbin.org/get", params=payload)
print r.url
```

运行结果

```
http://httpbin.org/get?key2=value2&key1=value1
```

如果想添加 headers，可以传 headers 参数

```
import requests
payload = {'key1': 'value1', 'key2': 'value2'}
headers = {'content-type': 'application/json'}
r = requests.get("http://httpbin.org/get", params=payload, headers=headers)
print r.url
```

通过 headers 参数可以增加请求头中的 headers 信息。

2.1.5 基本 POST 请求

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。

```
import requests
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.post("http://httpbin.org/post", data=payload)
print r.text
```

运行结果。

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "key1": "value1",
    "key2": "value2"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "23",
    "Content-Type": "application/x-www-form-urlencoded",
```

```
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.9.1"
  },
  "json": null,
  "url": "http://httpbin.org/post"
}
```

可以看到参数传成功了，然后服务器返回了我们传的数据。

有时候我们需要传送的信息不是表单形式的，需要我们传 JSON 格式的数据过去，所以我们可以用 json.dumps() 方法把表单数据序列化。

```
import json
import requests
url = 'http://httpbin.org/post'
payload = {'some': 'data'}
r = requests.post(url, data=json.dumps(payload))
print r.text
```

运行结果。

```
{
 "args": {},
        "data": "{\"some\": \"data\"}",
        "files": {},
        "form": {},
        "headers": {
                "Accept": "*/*",
                "Accept-Encoding": "gzip, deflate",
                "Content-Length": "16",
                "Host": "httpbin.org",
                "User-Agent": "python-requests/2.9.1"
        },
        "json": {
                "some": "data"
        },
        "url": "http://httpbin.org/post"
}
```

通过上述方法，我们可以 POST JSON 格式的数据，如果想要上传文件，那么直接用 file 参数即可，新建一个 a.txt 的文件，内容写上 Hello World!

```
import requests
url = ' http://httpbin.org/post '
files = {'file': open('test.txt', 'rb')}
r = requests.post(url, files=files)
print r.text
```

可以看到运行结果如下。

```
{
"args": {},
                "data": "{\"some\": \"data\"}",
                "files": {},
                "form": {},
                "headers": {
                        "Accept": "*/*",
                        "Accept-Encoding": "gzip, deflate",
                        "Content-Length": "16",
                        "Host": "httpbin.org",
                        "User-Agent": "python-requests/2.9.1"
                },
                "json": {
                        "some": "data"
                },
                "url": "http://httpbin.org/post"
}
```

这样我们便成功完成了一个文件的上传。

requests 是支持流式上传的，这允许你发送大的数据流或文件而无需先把它们读入内存。要使用流式上传，仅需为你的请求体提供一个类文件对象即可。

```
with open('massive-body') as f:
requests.post('http://some.url/streamed', data=f)
```

这是一个非常实用方便的功能。

2.1.6 Cookies

如果一个响应中包含了 cookie，那么我们可以利用 cookies 变量来拿到。

```
import requests
url = 'http://www.ixiaochouyu.com'
r = requests.get(url)
print r.cookies
print r.cookies['example_cookie_name']
```

以上程序仅是样例，可以用 cookies 变量来得到站点的 cookies，另外可以利用 cookies 变量来向服务器发送 cookies 信息。

```
import requests
url = 'http://httpbin.org/cookies'
cookies = dict(cookies_are='working')
r = requests.get(url, cookies=cookies)
print r.text
```

运行结果

```
'{"cookies": {"cookies_are": "working"}}'
```

可以已经成功向服务器发送了 cookies

2.1.7 超时配置

可以利用 timeout 变量来配置最大请求时间。

```
requests.get('http://github.com', timeout=0.001)
```

注：timeout 仅对连接过程有效，与响应体的下载无关。

也就是说，这个时间只限制请求的时间。即使返回的 response 包含很大内容，下载需要一定时间，然而这并没有什么卵用。

2.1.8 会话对象

在以上的请求中，每次请求其实都相当于发起了一个新的请求。也就是相当于我们每个请求都用了不同的浏览器单独打开的效果。也就是它并不是指的一个会话，即使请求的是同一个网址。比如

```
import requests
requests.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = requests.get("http://httpbin.org/cookies")
print(r.text)
```

结果是

```
{
```

```
  "cookies": {}
}
```

很明显，这不在一个会话中，无法获取 cookies，那么在一些站点中，我们需要保持一个持久的会话怎么办呢？就像用一个浏览器逛淘宝一样，在不同的选项卡之间跳转，这样其实就是建立了一个长久会话。

解决方案如下

```
import requests
s = requests.Session()
s.get('http://httpbin.org/cookies/set/sessioncookie/123456789')
r = s.get("http://httpbin.org/cookies")
print(r.text)
```

在这里我们请求了两次，一次是设置 cookies，一次是获得 cookies

运行结果

```
{
  "cookies": {
    "sessioncookie": "123456789"
  }
}
```

发现可以成功获取到 cookies 了，这就是建立一个会话到作用。体会一下。

那么既然会话是一个全局的变量，那么我们肯定可以用来全局的配置了。

```
import requests
s = requests.Session()
s.headers.update({'x-test': 'true'})
r = s.get('http://httpbin.org/headers', headers={'x-test2': 'true'})
print r.text
```

通过 s.headers.update 方法设置了 headers 的变量。然后我们又在请求中设置了一个 headers，那么会出现什么结果？

很简单，两个变量都传送过去了。

运行结果

```
{
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
```

```
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.9.1",
    "X-Test": "true",
    "X-Test2": "true"
  }
}
```

如果 get 方法传的 headers 同样也是 x-test 呢？

```
r = s.get('http://httpbin.org/headers', headers={'x-test': 'true'})
```

嗯，它会覆盖掉全局的配置

```
{
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.9.1",
    "X-Test": "true"
  }
}
```

那如果不想要全局配置中的一个变量了呢？很简单，设置为 None 即可

```
r = s.get('http://httpbin.org/headers', headers={'x-test': None})
```

运行结果

```
{
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.9.1"
  }
}
```

嗯，以上就是 session 会话的基本用法。

2.1.9 SSL 证书验证

现在随处可见 https 开头的网站，Requests 可以为 HTTPS 请求验证 SSL 证书，就像 web 浏览器一样。要想检查某个主机的 SSL 证书，你可以使用 verify 参数

现在 12306 证书不是无效的嘛，来测试一下

```
import requests
r = requests.get('https://kyfw.12306.cn/otn/', verify=True)
print r.text
```

结果

```
requests.exceptions.SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:590)
```

果真如此

来试下 github 的

```
import requests
r = requests.get('https://github.com', verify=True)
print r.text
```

嗯，正常请求，内容我就不输出了。

如果我们想跳过刚才 12306 的证书验证，把 verify 设置为 False 即可

```
import requests
r = requests.get('https://kyfw.12306.cn/otn/', verify=False)
print r.text
```

发现就可以正常请求了。在默认情况下 verify 是 True，所以如果需要的话，需要手动设置下这个变量。

2.1.10 代理

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
proxies = {
  "https": "http://41.118.132.69:4433"
}
r = requests.post("http://httpbin.org/post", proxies=proxies)
print r.text
```

也可以通过环境变量 HTTP_PROXY 和 HTTPS_PROXY 来配置代理

```
export HTTP_PROXY="http://10.10.1.10:3128"
export HTTPS_PROXY="http://10.10.1.10:1080"
```

通过以上方式，可以方便地设置代理。

2.1.11 总结

以上总结了一下 requests 的基本用法，如果你对爬虫有了一定的基础，那么肯定可以很快上手，在此就不多赘述了。练习才是王道，大家尽快投注于实践中吧。

2.2 开启 Beautiful Soup 之旅

上一节我们介绍了 Requests，它的内容其实还是蛮多的，如果但是仅仅只有 Requests 并不能实现我们的 Python 爬虫功能，我们还有一个更强大的工具，叫 Beautiful Soup，有了它我们可以很方便地提取出 HTML 或 XML 标签中的内容 ,实在是方便 ,这一节就让我们一起来感受一下 Beautiful Soup 的强大吧。

2.2.1 Beautiful Soup 的简介

简单来说，Beautiful Soup 是 python 的一个库，最主要的功能是从网页抓取数据。

官方解释是这么说的。Beautiful Soup 提供一些简单的、python 式的函数用来处理导航、搜索、修改分析树等功能。它是一个工具箱，通过解析文档为用户提供需要抓取的数据，因为简单，所以不需要多少代码就可以写出一个完整的应用程序。

Beautiful Soup 自动将输入文档转换为 Unicode 编码，输出文档转换为 utf-8 编码。你不需要考虑编码方式，除非文档没有指定一个编码方式，这时，Beautiful Soup 就不能自动识别编码方式了。然后，你仅仅需要说明一下原始编码方式就可以了。

Beautiful Soup 已成为和 lxml、html6lib 一样出色的 python 解释器，为用户灵活地提供不同的解析策略或强劲的速度。

废话不多说，我们来试一下吧~

2.2.2 Beautiful Soup 安装

Beautiful Soup 3 目前已经停止开发，推荐在现在的项目中使用 Beautiful Soup 4，不过它已经被移植到 BS4 了，也就是说导入时我们需要 import bs4 。所以这里我们用的版本是 Beautiful Soup 4.3.2 (简称 BS4)，另外据说 BS4 对 Python3 的支持不够好，不过我用的是 Python2.7.7，如果有小伙伴用的是 Python3 版本，可以考虑下载 BS3 版本。

如果你用的是新版的 Centos 或 Ubuntu,那么可以通过系统的软件包管理来安装 ,不过它不是最新版本，目前是 4.2.1 版本。

如果想安装最新的版本，请直接下载安装包来手动安装，也是十分方便的方法。在这里我安装的是 Beautiful Soup 4.3.2。

这里提供两个 Beautiful Soup 的下载链接：

Beautiful Soup 3.2.1 （https://pypi.python.org/pypi/BeautifulSoup/3.2.1）

Beautiful Soup 4.3.2 （https://pypi.python.org/pypi/beautifulsoup4/4.3.2）

下载完成之后解压，运行 python setup.py install 命令即可完成安装。

如下 Code1-36 所示，证明安装成功了。

```
Collecting BeautifulSoup
Installing collected package: BeautifulSoup
Successfully installed BeautifulSoup-3.2.1
```

Code1-36 Beautiful Soup 安装成功效果图

然后需要安装 lxml，运行命令 apt-get install Python-lxml 即可进行安装。

Beautiful Soup 支持 Python 标准库中的 HTML 解析器,还支持一些第三方的解析器，如果我们不安装它，则 Python 会使用 Python 默认的解析器，lxml 解析器更加强大，速度更快，推荐安装。

2.2.3 创建 Beautiful Soup 对象

首先必须要导入 bs4 库

```
from bs4 import BeautifulSoup
```

我们创建一个字符串，后面的例子我们便会用它来演示，代码如 Code1-37 所示。

```
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>
"""
```

Code 1-37 BeautifulSoup 简单代码图

创建 beautifulsoup 对象

```
soup = BeautifulSoup(html)
```

另外，我们还可以用本地 HTML 文件来创建对象，例如

```
soup = BeautifulSoup(open('index.html'))
```

上面这句代码便是将本地 index.html 文件打开，用它来创建 soup 对象

下面我们来打印一下 soup 对象的内容，格式化输出

```
print soup.prettify()
```

输出结果便是 index.html 中的 html 代码，并且格式化打印出了它的内容，这个函数经常用到，小伙伴们要记好咯。

2.2.4 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

下面我们进行一一介绍

1. Tag

Tag 是什么？通俗点讲就是 HTML 中的一个个标签，例如<title>The Dormouse's story</title>、<title>The Dormouse's story</title>

上面的 title a 等等 HTML 标签加上里面包括的内容就是 Tag，下面我们来感受一下怎样用 Beautiful Soup 来方便地获取 Tags

下面每一段代码中注释部分即为运行结果。

```
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>
```

我们可以利用 soup 加标签名轻松地获取这些标签的内容，是不是感觉比正则表达式方便多了？不过有一点是，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，我们在后面进行介绍。

我们可以验证一下这些对象的类型。

```
print type(soup.a)              #<class 'bs4.element.Tag'>
```

对于 Tag，它有两个重要的属性，是 name 和 attrs，下面我们分别来感受一下。

1) name

```
print soup.name       #[document]
print soup.head.name#head
```

soup 对象本身比较特殊，它的 name 即为 [document]，对于其他内部标签，输出的值便为标签本身的名称。

2) attrs

```
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。

如果我们想要单独获取某个属性，可以这样，例如我们获取它的 class 叫什么。

```
print soup.p['class']           #['title']
```

还可以这样，利用 get 方法，传入属性的名称，二者是等价的。

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

我们可以对这些属性和内容等等进行修改，例如。

```
soup.p['class']="newClass"
print soup.p      #<p class="newClass" name="dromouse"><b>The Dormouse's story</b></p>
```

还可以对这个属性进行删除，例如。

```
del soup.p['class']
print soup.p              #<p name="dromouse"><b>The Dormouse's story</b></p>
```

不过，对于修改删除的操作，不是我们的主要用途，在此不做详细介绍了，如果有需要，请查看前面提供的官方文档。

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
```

这样我们就轻松获取到了标签里面的内容，想想如果用正则表达式要多麻烦。它的类型是一个 NavigableString，翻译过来叫 可以遍历的字符串，不过我们最好还是称它英文名字吧。

来检查一下它的类型。

```
print type(soup.p.string)          #<class 'bs4.element.NavigableString'>
```

3. BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容.大部分时候,可以把它当作 Tag 对象，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         #{} 空字典
```

4. Comment

Comment 对象是一个特殊类型的 NavigableString 对象，其实输出的内容仍然不包括注释符号，但是如果不好好处理它，可能会对我们的文本处理造成意想不到的麻烦。

我们找一个带注释的标签。

```
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，我们发现它已经把注释符号去掉了，所以这可能会给我们带来不必要的麻烦。

另外我们打印输出下它的类型，发现它是一个 Comment 类型，所以，我们在使用前最好做一下判断，判断代码如下：

```
if type(soup.a.string)==bs4.element.Comment:
    print soup.a.string
```

上面的代码中，我们首先判断了它的类型，是否为 Comment 类型，然后再进行其他操作，如打印输出。

2.2.5 遍历文档树

1. 直接子节点

a) contents

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>
```

b) 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>
```

我们怎样获得里面的内容呢？很简单，遍历一下就好了，代码及结果如下。

```
for child in   soup.body.children:
print child
```

运行结果：

```
<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
```

2. 所有子孙节点

.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
```

运行结果如下，可以发现，所有的节点都被打印出来了，先生最外层的 HTML 标签，其次从 head 标签一个个剥离，以此类推，代码运行结果如 Code1-38 所示

```
<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>,
```

Code 1-38 所有子孙节点代码运行效果图

3. 节点内容

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
```

如果 tag 包含了多个子节点,tag 就无法确定，string 方法应该调用哪个子节点的内容, .string 的输出结果是 None。

```
print soup.html.string # None
```

4. 多个内容

a) string

获取多个内容，不过需要遍历获取，比如下面的例子。

```
for string in soup.strings:
    print(repr(string))
    # u"The Dormouse's story"
    # u'\n\n'
    # u"The Dormouse's story"
    # u'\n\n'
    # u'Once upon a time there were three little sisters; and their names were\n'
    # u'Elsie'
    # u',\n'
    # u'Lacie'
    # u' and\n'
    # u'Tillie'
    # u';\nand they lived at the bottom of a well.'
    # u'\n\n'
    # u'...'
    # u'\n'
```

b) stripped_string

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

```
for string in soup.stripped_strings:
    print(repr(string))
```

```
# u"The Dormouse's story"
# u"The Dormouse's story"
# u'Once upon a time there were three little sisters; and their names were'
# u'Elsie'
# u','
# u'Lacie'
# u'and'
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'...'
```

5. 父节点

操作示例代码如下。

```
p = soup.p
print p.parent.name          #body
content = soup.head.title.string
print content.parent.name  #title
```

6. 全部父节点

通过元素的 .parents 属性可以递归得到元素的所有父辈节点，例如。

```
content = soup.head.title.string
for parent in   content.parents:
        print parent.name
```

运行结果：

```
title
head
html
[document]
```

7. 兄弟节点

兄弟节点可以理解为和本节点处在统一级的节点，.next_sibling 属性获取了该节点的下一个兄弟节点，.previous_sibling 则与之相反，如果节点不存在，则返回 None

注意：实际文档中的 tag 的 .next_sibling 和 .previous_sibling 属性通常是字符串或空白，因为空白或者换行也可以被视作一个节点，所以得到的结果可能是空白或者换行。

```
print soup.p.next_sibling
#        实际该处为空白
print soup.p.prev_sibling
```

```
#None    没有前一个兄弟节点，返回 None
print soup.p.next_sibling.next_sibling
#<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>
#下一个节点的下一个兄弟节点是我们可以看到的节点
```

8. 全部兄弟节点

通过 .next_siblings 和 .previous_siblings 属性可以对当前节点的兄弟节点迭代输出。

```
for sibling in soup.a.next_siblings:
    print(repr(sibling))
    # u',\n'
    # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
    # u' and\n'
    # <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
    # u'; and they lived at the bottom of a well.'
    # None
```

9. 前后节点

与 .next_sibling .previous_sibling 不同，它并不是针对于兄弟节点，而是在所有节点，不分层次。

比如 head 节点为<head><title>The Dormouse's story</title></head>

那么它的下一个节点便是 title，它是不分层次关系的。

```
print soup.head.next_element
#<title>The Dormouse's story</title>
```

10. 所有前后节点

通过 .next_elements 和 .previous_elements 的迭代器就可以向前或向后访问文档的解析内容，就好像文档正在被解析一样。

```
print(repr(element))
# u'Tillie'
# u';\nand they lived at the bottom of a well.'
# u'\n\n'
# <p class="story">...</p>
# u'...'
```

```
# u'\n'
# None
```

以上是遍历文档树的基本用法。

2.2.6 搜索文档树

(1) find_all(name , attrs , recursive , text , **kwargs)

find_all() 方法搜索当前 tag 的所有 tag 子节点,并判断是否符合过滤器的条件

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>]
```

D.传 True

True 可以匹配任何值,下面代码查找到所有的 tag,但是不会返回字符串节点。

```
for tag in soup.find_all(True):
    print(tag.name)
# html
# head
# title
# body
# p
# b
# p
# a
# a
for tag in soup.find_all(True):
    print(tag.name)
# html
# head
# title
# body
# p
# b
# p
# a
# a
```

E.传方法

如果没有合适过滤器,那么还可以定义一个方法,方法只接受一个元素参数 [4] ,如果这个方法返回 True 表示当前元素匹配并且被找到,如果不是则反回 False

下面方法校验了当前元素,如果包含 class 属性却不包含 id 属性,那么将返回 True:

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

将这个方法作为参数传入 find_all() 方法,将得到所有<p>标签:

```
soup.find_all(has_class_but_no_id)
# [<p class="title"><b>The Dormouse's story</b></p>,
#    <p class="story">Once upon a time there were...</p>,
```

```
#  <p class="story">...</p>]
```

2）keyword 参数

注意：如果一个指定名字的参数不是搜索内置的参数名,搜索时会把该参数当作指定名字 tag 的属性来搜索,如果包含一个名字为 id 的参数,Beautiful Soup 会搜索每个 tag 的”id”属性

```
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

如果传入 href 参数,Beautiful Soup 会搜索每个 tag 的”href”属性

```
soup.find_all(href=re.compile("elsie"))
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

使用多个指定名字的参数可以同时过滤 tag 的多个属性

```
soup.find_all(href=re.compile("elsie"), id='link1')
# [<a class="sister" href="http://example.com/elsie" id="link1">three</a>]
```

在这里我们想用 class 过滤，不过 class 是 python 的关键词，这怎么办？加个下划线就可以

```
soup.find_all("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>]
```

有些 tag 属性在搜索不能使用,比如 HTML5 中的 data-* 属性

```
data_soup = BeautifulSoup('<div data-foo="value">foo!</div>')
data_soup.find_all(data-foo="value")
# SyntaxError: keyword can't be an expression
```

但是可以通过 find_all() 方法的 attrs 参数定义一个字典参数来搜索包含特殊属性的 tag。

```
data_soup.find_all(attrs={"data-foo": "value"})
# [<div data-foo="value">foo!</div>]
```

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, True

```
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"]
```

4) limit 参数

find_all() 方法返回全部的搜索结构,如果文档树很大那么搜索会很慢.如果我们不需要全部结果,可以使用 limit 参数限制返回结果的数量.效果与 SQL 中的 limit 关键字类似,当搜索到的结果数量达到 limit 的限制时,就停止搜索返回结果.

文档树中有 3 个 tag 符合搜索条件,但结果只返回了 2 个,因为我们限制了返回数量

```
soup.find_all("a", limit=2)
# [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
#  <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

5) recursive 参数

调用 tag 的 find_all() 方法时,Beautiful Soup 会检索当前 tag 的所有子孙节点,如果只想搜索 tag 的直接子节点,可以使用参数 recursive=False .

一段简单的文档:

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

是否使用 recursive 参数的搜索结果:

```
soup.html.find_all("title")
# [<title>The Dormouse's story</title>]

soup.html.find_all("title", recursive=False)
# []
```

(2) find(name , attrs , recursive , text , **kwargs)

它与 find_all() 方法唯一的区别是 find_all() 方法的返回结果是值包含一个元素的列表,而 find() 方法直接返回结果

(3) find_parents() find_parent()

find_all() 和 find() 只搜索当前节点的所有子节点,孙子节点等. find_parents() 和 find_parent() 用来搜索当前节点的父辈节点,搜索方法与普通 tag 的搜索方法相同,搜索文档搜索文档包含的内容

（4）find_next_siblings() find_next_sibling()

这 2 个方法通过 .next_siblings 属性对当 tag 的所有后面解析的兄弟 tag 节点进行迭代, find_next_siblings() 方法返回所有符合条件的后面的兄弟节点,find_next_sibling() 只返回符合条件的后面的第一个 tag 节点

（5）find_previous_siblings() find_previous_sibling()

这 2 个方法通过 .previous_siblings 属性对当前 tag 的前面解析的兄弟 tag 节点进行迭代, find_previous_siblings() 方法返回所有符合条件的前面的兄弟节点, find_previous_sibling() 方法返回第一个符合条件的前面的兄弟节点

（6）find_all_next() find_next()

这 2 个方法通过 .next_elements 属性对当前 tag 的之后的 tag 和字符串进行迭代, find_all_next() 方法返回所有符合条件的节点, find_next() 方法返回第一个符合条件的节点

（7）find_all_previous() 和 find_previous()

这 2 个方法通过 .previous_elements 属性对当前节点前面的 tag 和字符串进行迭代, find_all_previous() 方法返回所有符合条件的节点, find_previous()方法返回第一个符合条件的节点

2.2.7 CSS 选择器

我们在写 CSS 时，标签名不加任何修饰，类名前加点，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，代码如下。

```
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>]

print soup.select('p a[href="http://example.com/elsie"]')
#[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

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()
```

好，这就是另一种与 find_all 方法有异曲同工之妙的查找方法，是不是感觉很方便？

2.2.8 总结

本篇内容比较多，把 Beautiful Soup 的方法进行了大部分整理和总结，不过这还不算完全，仍然有 Beautiful Soup 的修改删除功能，不过这些功能用得比较少，只整理了查找提取的方法，希望对大家有帮助！小伙伴们加油！

熟练掌握了 Beautiful Soup，一定会给你带来太多方便，加油吧！

2.3 Xpath 语言与 lxml 库的用法

2.3.1 安装 lxml

xml 的详细介绍，官网链接：http://lxml.de/，是一种使用 Python 编写的库，可以迅速、灵活地处理 XML

直接执行如下命令

```
pip install lxml
```

就可完成安装，如果安装过程中提示， Microsoft Visual C++库没安装，则需要下载 C++支持的库，链接为：https://www.microsoft.com/en-us/download/details.aspx?id=44266

2.3.2 Xpath 语法

XPath 是一门在 XML 文档中查找信息的语言。XPath 可用来在 XML 文档中对元素和属性进行遍历。XPath 是 W3C XSLT 标准的主要元素，并且 XQuery 和 XPointer 都构建于 XPath 表达之上。

节点关系

（1）父（Parent）

每个元素以及属性都有一个父。在下面的例子中，book 元素是 title、author、year 以及 price 元素的父：

```
<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：

```
<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：

```
<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：

```
<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：

```
<bookstore>
  <book>
   <title>Harry Potter</title>
   <author>J K. Rowling</author>
   <year>2005</year>
```

```
    <price>29.99</price>
  </book>
</bookstore>
```

选取节点

XPath 使用路径表达式在 XML 文档中选取节点。节点是通过沿着路径或者 step 来选取的。

下面列出了最有用的路径表达式：

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，而不考虑他们的位置。
.	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

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。谓语被嵌在方括号中。

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。

路径表达式	结果
//book/title | //book/price	选取 book 元素的所有 title 和 price 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 运算符

下面列出了可用在 XPath 表达式中的运算符：

运算符	描述	实例	返回值
|	计算两个节点集	//book | //cd	返回所有拥有 book 和 cd 元素的节点集
+	加法	6 + 4	10
–	减法	6 – 4	2
*	乘法	6 * 4	24

div	除法	8 div 4	2
=	等于	price=9.80	如果 price 是 9.80，则返回 true。如果 price 是 9.90，则返回 false。
!=	不等于	price!=9.80	如果 price 是 9.90，则返回 true。如果 price 是 9.80，则返回 false。
<	小于	price<9.80	如果 price 是 9.00，则返回 true。如果 price 是 9.90，则返回 false。
<=	小于或等于	price<=9.80	如果 price 是 9.00，则返回 true。如果 price 是 9.90，则返回 false。
>	大于	price>9.80	如果 price 是 9.90，则返回 true。如果 price 是 9.80，则返回 false。
>=	大于或等于	price>=9.80	如果 price 是 9.90，则返回 true。如果 price 是 9.70，则返回 false。
or	或	price=9.80 or price=9.70	如果 price 是 9.80，则返回 true。如果 price 是 9.50，则返回 false。
and	与	price>9.00 and price<9.90	如果 price 是 9.80，则返回 true。如果 price 是 8.50，则返回 false。
mod	计算除法的余数	5 mod 2	1

2.3.3 lxml 用法

初步使用

首先我们利用它来解析 HTML 代码，先来一个小例子来感受一下它的基本用法。

```
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>
    </ul>
 </div>
'''
html = etree.HTML(text)
```

```
result = etree.tostring(html)
print(result)
```

首先我们使用 lxml 的 etree 库，然后利用 etree.HTML 初始化，然后我们将其打印出来。

其中，这里体现了 lxml 的一个非常实用的功能就是自动修正 html 代码，大家应该注意到了，最后一个 li 标签，其实我把尾标签删掉了，是不闭合的。不过，lxml 因为继承了 libxml2 的特性，具有自动修正 HTML 代码的功能。

所以输出结果是这样的。

```
<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>
```

不仅补全了 li 标签，还添加了 body，html 标签。

文件读取

除了直接读取字符串，还支持从文件读取内容。比如我们新建一个文件叫做 hello.html，内容为如下。

```
<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>
```

利用 parse 方法来读取文件。

```
from lxml import etree
html = etree.parse('hello.html')
result = etree.tostring(html, pretty_print=True)
print(result)
```

同样可以得到相同的结果。

2.3.4 总结

XPath 是一个非常好用的解析方法 ,同时也作为爬虫学习的基础 ,在后面的 selenium 以及 scrapy 框架中都会涉及到这部分知识，希望大家可以把它的语法掌握清楚，为后面的深入研究做好铺垫。

2.4 动态网页的抓取

2.4.1 什么是动态网页？

部分不懂前端的学员可能对这个概念不是很理解，所以这里先解释一下什么是动态网页：

1. 正常情况下，网站服务器给我们直接返回 html 源码。
2. html 源码里面会指明我们还需要去请求的其他文件如 css， js 和 image 等。
3. 这些请求在浏览器获取到 html 之后浏览器会主动分析这些请求然后依次去请求，
4. 然后浏览器会去执行 js 和 css 等文件，这时候 js 文件实际上是可以直接操作 html 内容的，js 可以修改我们的 html 源码。
5. 我们直接通过 requests.get 方法或者 urllib 获取到的 html 源码实际上是浏览器处理之前的原始 html。

2.4.2 Selenium 简介

selenium 是一个用于 Web 应用自动化程序测试的工具，测试直接运行在浏览器中，就像真正的用户在操作一样

selenium2 支持通过驱动真实浏览器（FirfoxDriver，IternetExplorerDriver，OperaDriver，ChromeDriver）

selenium2 支持通过驱动无界面浏览器（HtmlUnit，PhantomJs）

Window 安装：

第一种方法是：下载源码安装，下载地址（https://pypi.python.org/pypi/selenium）解压并把整个目录放到 C:\Python27\Lib\site-packages 下面

第二种方法是：可以直接在 C:\Python27\Scripts 下输入命令安装 pip install -U selenium

Linux 安装：

```
sudo pip install selenium
```

2.4.3 PhantomJS 简介

PhantomJS 是一个基于 WebKit（WebKit 是一个开源的浏览器引擎，Chrome，Safari 就是用的这个浏览器引擎） 的服务器端 JavaScript API，

主要应用场景是：无需浏览器的 Web 测试，页面访问自动化，屏幕捕获，网络监控。

Windows 安装：

下载源码安装，下载地址（http://phantomjs.org/download.html）解压并把解压缩的路径添加到环境变量中即可，我自己的放到了 C:\Python27\Scripts 下面

Linux 安装：

```
sudo apt-get install PhantomJS
```

2.4.4 动态爬虫

这是我学习爬虫比较深入的一步了，大部分的网页抓取用 urllib2 都可以搞定，但是涉及到 JavaScript 的时候，urlopen 就完全傻逼了，所以不得不用模拟浏览器，方法也有很多，此处我采用的是 selenium2+phantomjs，原因在于：

selenium2 支持所有主流的浏览器和 phantomjs 这些无界面的浏览器，我开始打算用 Chrome，但是发现需要安装一个什么 Chrome 驱动，于是就弃用了，选择 phantomjs，而且这个名字听起来也比较洋气。

python 可以使用 selenium 执行 javascript，selenium 可以让浏览器自动加载页面，获取需要的数据。selenium 自己不带浏览器，可以使用第三方浏览器如 Firefox，Chrome 等，也可以使用 headless 浏览器如 PhantomJS 在后台执行。

在工作用遇到一个问题,当加载一个手机端的 URL 时候，会加载不上，需要我们在请求头中设置一个 User-Agent，设置完以后就可以打开了（Windows 下执行，linux 下执行的话就不用加 executable_path='C:\Python27\Scripts\phantomjs.exe'）

```
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
dcap = dict(DesiredCapabilities.PHANTOMJS)  #设置 userAgent
dcap["phantomjs.page.settings.userAgent"] = ("Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36")
obj = webdriver.PhantomJS(executable_path='C:\Python27\Scripts\phantomjs.exe',desired_capabilities=dcap) #加载网址
obj.get('http://www.ixiaochouyus.com')#打开网址
obj.save_screenshot("1.png")  #截图保存
obj.quit()  # 关闭浏览器。当出现异常时记得在任务浏览器中关闭 PhantomJS，因为会有多个 PhantomJS 在运行状态，影响电脑性能
```

一、超时设置

webdriver 类中有三个和时间相关的方法：

```
1.pageLoadTimeout     设置页面完全加载的超时时间，完全加载即完全渲染完成，同步和异步脚本都执行完
2.setScriptTimeout    设置异步脚本的超时时间
3.implicitlyWait      识别对象的智能等待时间
```

下面我们以获取校花网 title 为例来验证效果，因为校花网中图片比较多，所以加载的时间比较长，更能时间我们的效果。

```
from selenium import webdriver
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
try:
    obj.get('http://www.ixiaochouyu.com')
    print obj.title
except Exception as e:
    print e
```

二、元素的定位

对象的定位是通过属性定位来实现的，这种属性就像人的身份证信息一样，或是其他的一些信息来找到这个对象，那我们下面就介绍下 Webdriver 提供的几个常用的定位方法

```
<input id="kw" name="wd" class="s_ipt" value="" maxlength="255" autocomplete="off">
```

上面这个是百度的输入框，我们可以发现我们可以用 id 来定位这个标签，然后就可以进行后面的操作了。

```
from selenium import webdriver
```

```
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
try:
    obj.get('http://www.baidu.com')
    obj.find_element_by_id('kw')                    #通过 ID 定位
    obj.find_element_by_class_name('s_ipt')         #通过 class 属性定位
    obj.find_element_by_name('wd')                  #通过标签 name 属性定位
    obj.find_element_by_tag_name('input')           #通过标签属性定位
    obj.find_element_by_css_selector('#kw')         #通过 css 方式定位
    obj.find_element_by_xpath("//input[@id='kw']")  #通过 xpath 方式定位
    obj.find_element_by_link_text("贴吧")           #通过 xpath 方式定位
    print obj.find_element_by_id('kw').tag_name     #获取标签的类型
except Exception as e:
    print e
```

三、浏览器的操作

1、调用启动的浏览器不是全屏的，有时候会影响我们的某些操作，所以我们可以设置全屏

```
from selenium import webdriver
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
obj.maximize_window()  #设置全屏
try:
    obj.get('http://www.baidu.com')
    obj.save_screenshot('2.png')  # 截取全屏，并保存
except Exception as e:
    print e
```

2、设置浏览器宽、高

```
from selenium import webdriver
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
obj.set_window_size('480','800') #设置浏览器宽 480，高 800
try:
    obj.get('http://www.baidu.com')
    obj.save_screenshot('3.png')  # 截取全屏，并保存
except Exception as e:
    print e
```

3、操作浏览器前进、后退

```
from selenium import webdriver
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
try:
    obj.get('http://www.baidu.com')     #访问百度首页
    obj.save_screenshot('1.png')
    obj.get('http://www.sina.com.cn') #访问新浪首页
    obj.save_screenshot('2.png')
    obj.back()                          #回退到百度首页
    obj.save_screenshot('3.png')
    obj.forward()                       #前进到新浪首页
    obj.save_screenshot('4.png')
except Exception as e:
    print e
```

四、操作测试对象

定位到元素以后，我们就应该对相应的对象进行某些操作，以达到我们某些特定的目的，那我们下面就介绍下 Webdriver 提供的几个常用的操作方法

```
from selenium import webdriver
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
try:
    obj.get('http://www.baidu.com')
    print obj.find_element_by_id("cp").text   # 获取元素的文本信息
    obj.find_element_by_id('kw').clear()                    #用于清除输入框的内容
    obj.find_element_by_id('kw').send_keys('Hello')   #在输入框内输入 Hello
    obj.find_element_by_id('su').click()                    #用于点击按钮
    obj.find_element_by_id('su').submit()                   #用于提交表单内容

except Exception as e:
    print e
```

五、键盘事件

1、键盘按键用法

```
from selenium.webdriver.common.keys import Keys
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
```

```
obj.set_page_load_timeout(5)
try:
    obj.get('http://www.baidu.com')
    obj.find_element_by_id('kw').send_keys(Keys.TAB)    #用于清除输入框的内容,相当于 clear()
    obj.find_element_by_id('kw').send_keys('Hello')    #在输入框内输入 Hello
    obj.find_element_by_id('su').send_keys(Keys.ENTER) #通过定位按钮，通过 enter（回车）代替 click()

except Exception as e:
    print e
```

2、键盘组合键使用

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
obj.set_page_load_timeout(5)
try:
    obj.get('http://www.baidu.com')
    obj.find_element_by_id('kw').send_keys(Keys.TAB)    #用于清除输入框的内容,相当于 clear()
    obj.find_element_by_id('kw').send_keys('Hello')    #在输入框内输入 Hello
    obj.find_element_by_id('kw').send_keys(Keys.CONTROL,'a')    #ctrl + a 全选输入框内容
    obj.find_element_by_id('kw').send_keys(Keys.CONTROL,'x')    #ctrl + x 剪切输入框内容
except Exception as e:
    print e
```

六、中文乱码问题

selenium2 在 python 的 send_keys( )中输入中文会报错 ,其实在中文前面加一个 u 变成 unicode 就能搞定了

七、鼠标事件

1、鼠标右击

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
try:
    obj.get("http://pan.baidu.com")
    obj.find_element_by_id('TANGRAM__PSP_4__userName').send_keys('13201392325')    #定位并输入用户名
    obj.find_element_by_id('TANGRAM__PSP_4__password').send_keys('18399565576lu') #定位并输入密码
    obj.find_element_by_id('TANGRAM__PSP_4__submit').submit()                     #提交表单内容
```

```
    f = obj.find_element_by_xpath('/html/body/div/div[2]/div[2]/....')          #定位到要点击的标签
    ActionChains(obj).context_click(f).perform()                    #对定位到的元素进行右键点击操作
except Exception as e:
    print e
```

2、鼠标双击

```
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
obj = webdriver.PhantomJS(executable_path="D:\Python27\Scripts\phantomjs.exe")
try:
    obj.get("http://pan.baidu.com")
    obj.find_element_by_id('TANGRAM__PSP_4__userName').send_keys('13201392325')    #定位并输入用户名
    obj.find_element_by_id('TANGRAM__PSP_4__password').send_keys('18399565576lu') #定位并输入密码
    obj.find_element_by_id('TANGRAM__PSP_4__submit').submit()                      #提交表单内容
    f = obj.find_element_by_xpath('/html/body/div/div[2]/div[2]/....')          #定位到要点击的标签
    ActionChains(obj).double_click(f).perform()                    #对定位到的元素进行双击操作
except Exception as e:
    print e
```

八、知乎案例：

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver import ActionChains
import time
import sys

driver = webdriver.PhantomJS(executable_path=' D:\Python27\Scripts\phantomjs.exe ')
driver.get("http://www.zhihu.com/#signin")
#driver.find_element_by_name('email').send_keys('your email')
driver.find_element_by_xpath('//input[@name="password"]').send_keys('your password')
#driver.find_element_by_xpath('//input[@name="password"]').send_keys(Keys.RETURN)
time.sleep(2)
driver.get_screenshot_as_file('show.png')
#driver.find_element_by_xpath('//button[@class="sign-button"]').click()
driver.find_element_by_xpath('//form[@class="zu-side-login-box"]').submit()
```

```
try:
    dr=WebDriverWait(driver,5)
    dr.until(lambda    the_driver:the_driver.find_element_by_xpath('//a[@class="zu-top-nav-userinfo
"]').is_displayed())
except:
    print '登录失败'
    sys.exit(0)
driver.get_screenshot_as_file('show.png')
#user=driver.find_element_by_class_name('zu-top-nav-userinfo ')
#webdriver.ActionChains(driver).move_to_element(user).perform() #移动鼠标到我的用户名
loadmore=driver.find_element_by_xpath('//a[@id="zh-load-more"]')
actions = ActionChains(driver)
actions.move_to_element(loadmore)
actions.click(loadmore)
actions.perform()
time.sleep(2)
driver.get_screenshot_as_file('show.png')
print driver.current_url
print driver.page_source
driver.quit()
```

2.4.5 总结

Selenium 和 Phantomjs 除了能解决动态页面的问题以外，用 selenium 用来模拟登陆也比 urllib2 简单得多。

第 3 章 Scrapy 框架

本章工作任务

- 任务 1：为什么使用 Scrapy
- 任务 2：安装和配置 Scrapy
- 任务 3： Scrapy 命令行工具
- 任务 4：编写第一个 Scrapy 爬虫

本章技能目标及重难点

编号	技能点描述	级别
1	为什么使用 Scrapy？	★
2	安装和配置 Scrapy	★★
3	Scrapy 命令行工具	★★★
4	编写第一个 Scrapy 爬虫	★★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 爬虫框架，需要同学们理解为什么使用 Scrapy，它的概念、特点。最主要的是需要大家学会如何建立一个简单的爬虫项目。

本章学习建议

本章适合有 Python 基础的学员学习。

本章内容（学习活动）

3.1 为什么使用 Scrapy？

3.1.1 什么是 Scrapy

Scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。

其最初是为了页面抓取（更确切来说，网络抓取)所设计的， 也可以应用在获取 API 所返回的数据(例如 Amazon Associates Web Services ）或者通用的网络爬虫。

Scrapy 其实是 Search+Python。Scrapy 使用 Twisted 这个异步网络库来处理网络通讯，架构清晰，并且包含了各种中间件接口，可以灵活的完成各种需求。

3.1.2 Scrapy 和 PySpider 比较

Pyspiders 是国内某大神开发了个 WebUI 的[Pyspider](GitHub - binux/pyspider: A Powerful Spider(Web Crawler) System in Python.)，具有以下特性：

1. python 脚本控制，可以用任何你喜欢的 html 解析包（内置 pyquery ）
2. WEB 界面编写调试脚本，起停脚本，监控执行状态，查看活动历史，获取结果产出
3. 支持 MySQL, MongoDB, SQLite
4. 支持抓取 JavaScript 的页面
5. 组件可替换，支持单机/分布式部署，支持 Docker 部署
6. 强大的调度控制

从内容上讲，两者具有功能差不多。不同点如下所示。

1.Scrapy 原生不支持 js 渲染，需要单独下载[scrapy-splash](GitHub - scrapy-plugins/scrapy-splash: Scrapy+Splash for JavaScript integration),而 PySpider 内置支持[scrapyjs](GitHub - scrapy-plugins/scrapy-splash: Scrapy+Splash for JavaScript integration)。

2.PySpider 内置 pyquery 选择器，Scrapy 有 XPath 和 CSS 选择器，这两个大家可能更熟一点。

3.Scrapy 全部命令行操作，PySpider 有较好的 WebUI

4.Scrapy 对千万级 URL 去重支持很好，采用[布隆过滤](海量大数据处理单机方案)来做，而 Spider 用的是数据库来去重。

5.PySpider 更加容易调试，Scrapy 默认的 debug 模式信息量太大，warn 模式信息量太少，由于异步框架出错后是不会停掉其他任务的，也就是出错了还会接着跑

从整体上来说，pyspider 比 scrapy 简单，并且 pyspider 可以在线提供爬虫服务，也就是所说的 SaaS，想要做个简单的爬虫推荐使用它，但自定义程度相对 Scrapy 低，社区人数和文档都没有 Scrapy 强，但 Scrapy 要学习的相关知识也较多，故而完成一个爬虫的时间较长。

3.1.3 Scrapy 整体结构

1、引擎(Scrapy Engine)

用来处理整个系统的数据流处理，触发事务。

2、调度器(Scheduler)

用来接受引擎发过来的请求，压入队列中，并在引擎再次请求的时候返回。

3、下载器(Downloader)

用于下载网页内容，并将网页内容返回给蜘蛛。

4、蜘蛛(Spiders)

蜘蛛是主要干活的 ,用它来制订特定域名或网页的解析规则。编写用于分析 response 并提取 item(即获取到的 item)或额外跟进的 URL 的类。 每个 spider 负责处理一个特定(或一些)网站。

蜘蛛的整个抓取流程（周期）是这样的：

1. 首先获取第一个 URL 的初始请求，当请求返回后调取一个回调函数。第一个请求是通过调用 start_requests()方法。该方法默认从 start_urls 中的 Url 中生成请求，并执行解析来调用回调函数。
2. 在回调函数中，你可以解析网页响应并返回项目对象和请求对象或两者的迭代。这些请求也将包含一个回调，然后被 Scrapy 下载，然后有指定的回调处理。
3. 在回调函数中，你解析网站的内容，同程使用的是 Xpath 选择器（但是你也可以使用 BeautifuSoup, lxml 或其他任何你喜欢的程序），并生成解析的数据项。
4. 最后，从蜘蛛返回的项目通常会进驻到项目管道。

5、项目管道(Item Pipeline)

主要责任是负责处理有蜘蛛从网页中抽取的项目，他的主要任务是清晰、验证和存储数据。当页面被蜘蛛解析后，将被发送到项目管道，并经过几个特定的次序处理数据。每个项目管道的组件都是有一个简单的方法组成的 Python 类。他们获取了项目并执行他们的方法，同时他们还需要确定的是是否需要在项目管道中继续执行下一步或是直接丢弃掉不处理。

项目管道通常执行的过程有：

1. 清洗 HTML 数据
2. 验证解析到的数据（检查项目是否包含必要的字段）
3. 检查是否是重复数据（如果重复就删除）
4. 将解析到的数据存储到数据库中

6、下载器中间件(Downloader Middlewares)

位于 Scrapy 引擎和下载器之间的钩子框架，主要是处理 Scrapy 引擎与下载器之间的请求及响应。它提供了一个自定义的代码的方式来拓展 Scrapy 的功能。下载中间器是一个处理请求和响应的钩子框架。他是轻量级的，对 Scrapy 尽享全局控制的底层的系统。

7、蜘蛛中间件(Spider Middlewares)

介于 Scrapy 引擎和蜘蛛之间的钩子框架，主要工作是处理蜘蛛的响应输入和请求输出。它提供一个自定义代码的方式来拓展 Scrapy 的功能。蛛中间件是一个挂接到 Scrapy 的蜘蛛处理机制的框架，你可以插入自定义的代码来处理发送给蜘蛛的请求和返回蜘蛛获取的响应内容和项目。

8、调度中间件(Scheduler Middlewares)

介于 Scrapy 引擎和调度之间的中间件，从 Scrapy 引擎发送到调度的请求和响应。他提供了一个自定义的代码来拓展 Scrapy 的功能。

3.1.4 数据处理流程

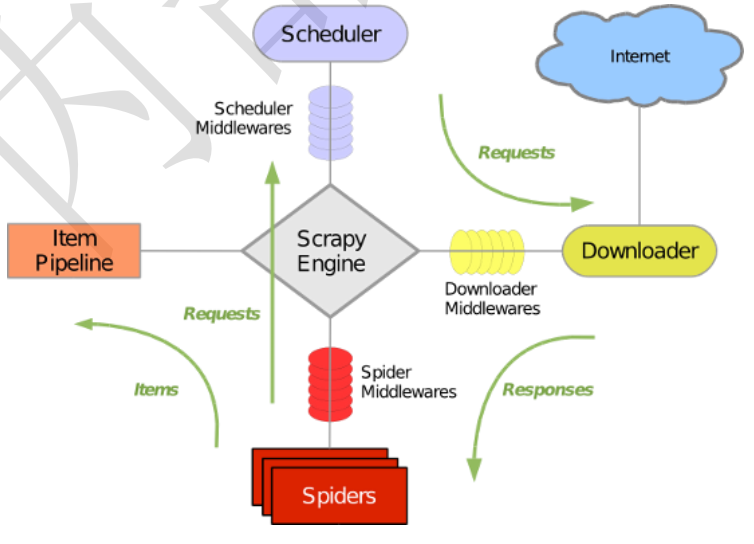

图 3-1 Scrapy 执行流程图

如图 3-1 所示 ,显示的是 Scrapy 爬虫执行流程 ,绿线是数据流向 ,首先从初始 URL 开始 ,Scheduler 会将其交给 Downloader 进行下载，下载之后会交给 Spider 进行分析，Spider 分析出来的结果有两种：一种是需要进一步抓取的链接，例如之前分析的“下一页”的链接，这些东西会被传回 Scheduler；另一种是需要保存的数据，它们则被送到 Item Pipeline 那里，那是对数据进行后期处理（详细分析、过滤、存储等）的地方。另外，在数据流动的通道里还可以安装各种中间件，进行必要的处理。

Scrapy 中的数据流由执行引擎控制，其过程如下:

1. 引擎打开一个网站(open a domain)，找到处理该网站的 Spider 并向该 spider 请求第一个要爬取的 URL(s)。
2. 引擎从 Spider 中获取到第一个要爬取的 URL 并在调度器(Scheduler)以 Request 调度。
3. 引擎向调度器请求下一个要爬取的 URL。
4. 调度器返回下一个要爬取的 URL 给引擎，引擎将 URL 通过下载中间件(请求(request)方向)转发给下载器(Downloader)。
5. 一旦页面下载完毕，下载器生成一个该页面的 Response，并将其通过下载中间件(返回(response)方向)发送给引擎。
6. 引擎从下载器中接收到 Response 并通过 Spider 中间件(输入方向)发送给 Spider 处理。
7. Spider 处理 Response 并返回爬取到的 Item 及(跟进的)新的 Request 给引擎。
8. 引擎将(Spider 返回的)爬取到的 Item 给 Item Pipeline 将(Spider 返回的)Request 给调度器。
9. (从第二步)重复直到调度器中没有更多地 request，引擎关闭该网站。

3.2 安装和配置 Scrapy

3.2.1 安装 pywin32

在 windows 下，必须安装 pywin32，安装地址：https://sourceforge.net/projects/pywin32/files/pywin32/，显示如下图 3-11 所示。

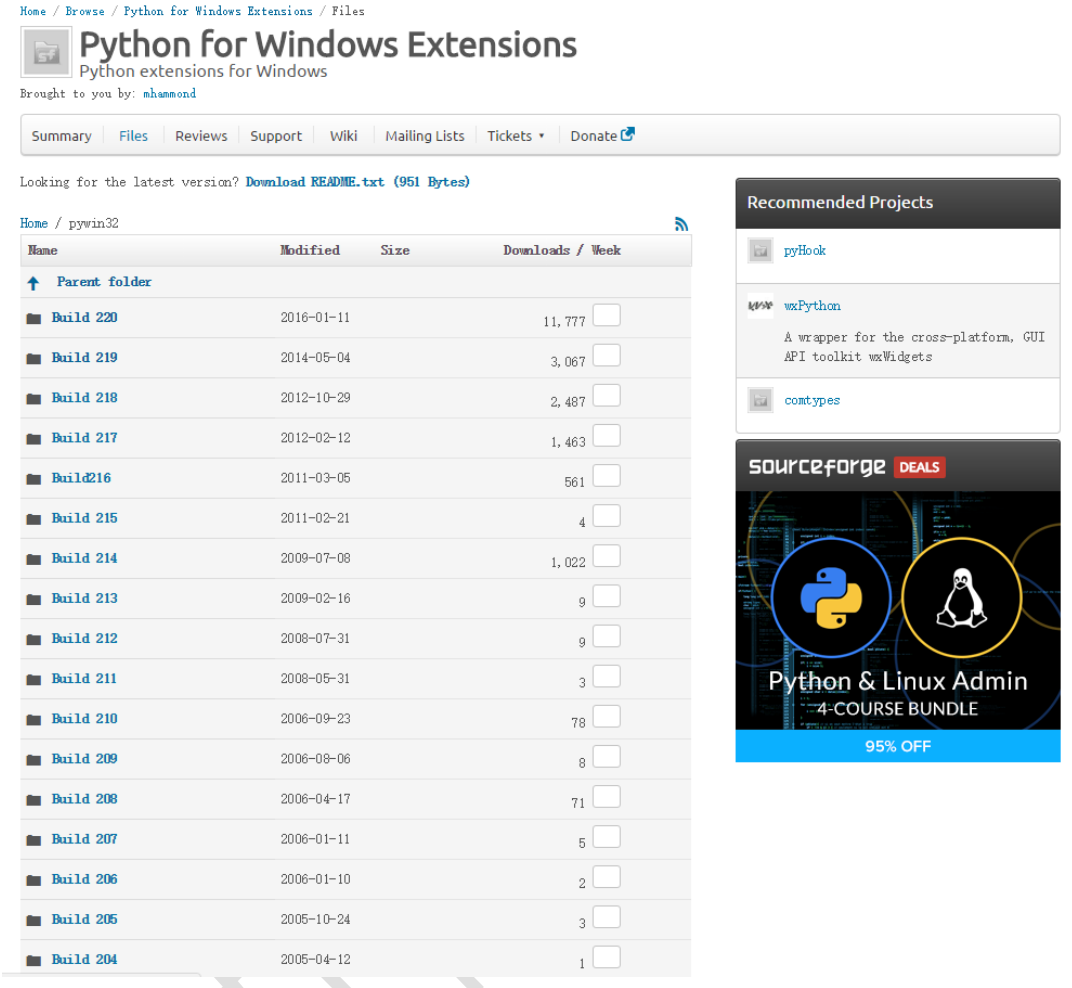

图 3-12 pywin32 各版本界面图

本次我们安装选择安装的是 Build 220，点击进入该链接：https://sourceforge.net/projects/pywin32/files/pywin32/Build%20220/，显示如下图 3-13 所示。

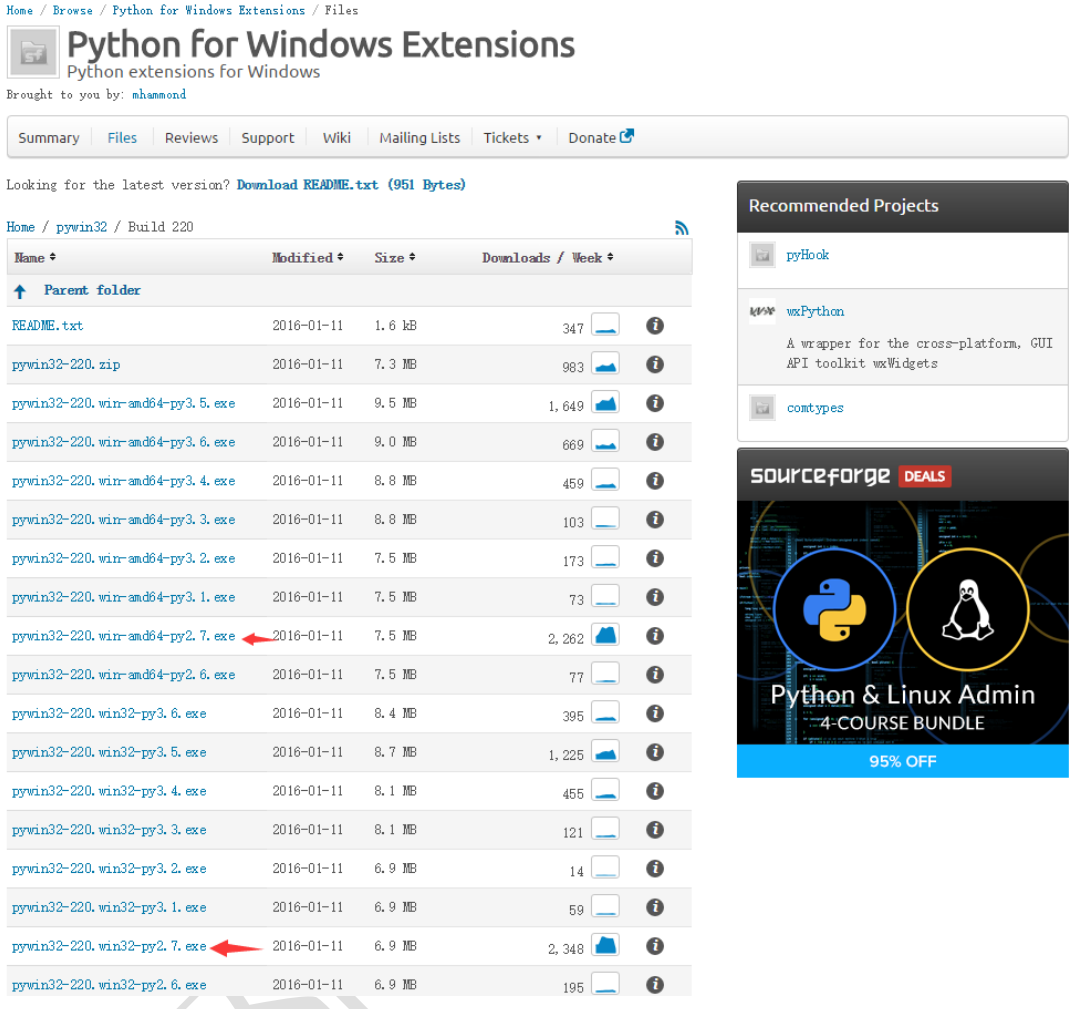

图 3-13 pywin3 220 下载界面图

32 位操作系统的同学，点击 pywin32-220.win32-py2.7.exe 下载链接，64 位操作系统的同学，点击 pywin32-220.win-amd64-py2.7.exe 下载链接。

为什么要安装 pywin32？因为在 window 环境中，Scrapy 要调用 Window 的基础命令，需要 win32api 的支持，才能使用。

下载完成后，直接双击安装即可，安装完毕之后验证：

在 python 命令行下输入

```
import win32com
```

如果没有提示错误，则证明安装成功，如 Code 3-14 所示

```
>>> import win32com

>>>
```

Code 3-14 pywin32 验证

3.2.2 安装 zope.interface

可以使用 pip 和 easy_install 来安装 zope.interface，操作方式如 Code 3-15 所示。

```
>>> pip install zope.interface

>>> easy_install zope.interface
```

Code 3-15 zope.interface 安装

现在也有 exe 版本，链接为：https://pypi.python.org/pypi/zope.interface/4.1.0#downloads，显示如下图 3-16 所示。

File	Type	Py Version	Uploaded on	Size
zope.interface-4.1.0-py2.6-win-amd64.egg (md5)	Python Egg	2.6	2014-02-06	269KB
zope.interface-4.1.0-py2.6-win32.egg (md5)	Python Egg	2.6	2014-02-06	268KB
zope.interface-4.1.0-py2.7-win-amd64.egg (md5)	Python Egg	2.7	2014-02-06	268KB
zope.interface-4.1.0-py2.7-win32.egg (md5)	Python Egg	2.7	2014-02-06	267KB
zope.interface-4.1.0-py3.2-win-amd64.egg (md5)	Python Egg	3.2	2014-02-06	274KB
zope.interface-4.1.0-py3.2-win32.egg (md5)	Python Egg	3.2	2014-02-06	273KB
zope.interface-4.1.0-py3.3-win-amd64.egg (md5)	Python Egg	3.3	2015-09-10	292KB
zope.interface-4.1.0-py3.3-win32.egg (md5)	Python Egg	3.3	2015-09-10	291KB
zope.interface-4.1.0.tar.gz (md5)	Source		2014-02-06	834KB
zope.interface-4.1.0.win-amd64-py2.6.exe (md5)	MS Windows installer	2.6	2014-02-06	352KB
zope.interface-4.1.0.win-amd64-py2.7.exe (md5)	MS Windows installer	2.7	2014-02-06	352KB
zope.interface-4.1.0.win-amd64-py3.2.exe (md5)	MS Windows installer	3.2	2014-02-06	352KB
zope.interface-4.1.0.win-amd64-py3.3.exe (md5)	MS Windows installer	3.3	2015-09-10	350KB
zope.interface-4.1.0.win32-py2.6.exe (md5)	MS Windows installer	2.6	2014-02-06	324KB
zope.interface-4.1.0.win32-py2.7.exe (md5)	MS Windows installer	2.7	2014-02-06	324KB
zope.interface-4.1.0.win32-py3.2.exe (md5)	MS Windows installer	3.2	2014-02-06	324KB
zope.interface-4.1.0.win32-py3.3.exe (md5)	MS Windows installer	3.3	2015-09-10	319KB

Author: Zope Foundation and Contributors
Home Page: http://pypi.python.org/pypi/zope.interface
License: ZPL 2.1
Categories
Development Status :: 5 - Production/Stable
Framework :: Zope3
Intended Audience :: Developers
License :: OSI Approved :: Zope Public License
Operating System :: OS Independent
Programming Language :: Python
Programming Language :: Python :: 2
Programming Language :: Python :: 2.6
Programming Language :: Python :: 2.7
Programming Language :: Python :: 3
Programming Language :: Python :: 3.2
Programming Language :: Python :: 3.3
Programming Language :: Python :: Implementation :: CPython
Programming Language :: Python :: Implementation :: PyPy
Topic :: Software Development :: Libraries :: Python Modules
Package Index Owner: fdrake, J1m, philikon, pcardune, ctheune, hannosch, ccomb, srichter, ajung, chrisw, nadako, faassen, hathawsh, tseaver, icemac, gary, roymath, tlotze, kobold, agroszer, baijum, regebro, menesis, chrism, davisagli, alga, mgedmin
Package Index Maintainer: zope.wineggbuilder, zope.wheelbuilder
DOAP record: zope.interface-4.1.0.xml

图 3-16 zope.interface 安装包下载界面

32 位操作系统的同学，点击 zope.interface-4.1.0.win32-py2.7.exe 下载链接，64 位操作系统的同学，点击 zope.interface-4.1.0.win-amd64-py2.7.exe 下载链接。

下载完成后，直接双击安装即可，安装完毕之后验证：

在 python 命令行下输入

```
import zope.interface
```

如果没有提示错误，则证明安装成功，如 Code3-17 所示

```
>>> import zope.interface
```

Code 3-17 zope.interface 验证

3.2.3 安装 pyOpenSSL

在 Windows 下，是没有预装 pyOpenSSL 的，而在 Linux 下是已经安装好的。

安装地址：https://launchpad.net/pyopenssl，界面显示如下，如图 3-18 所示。

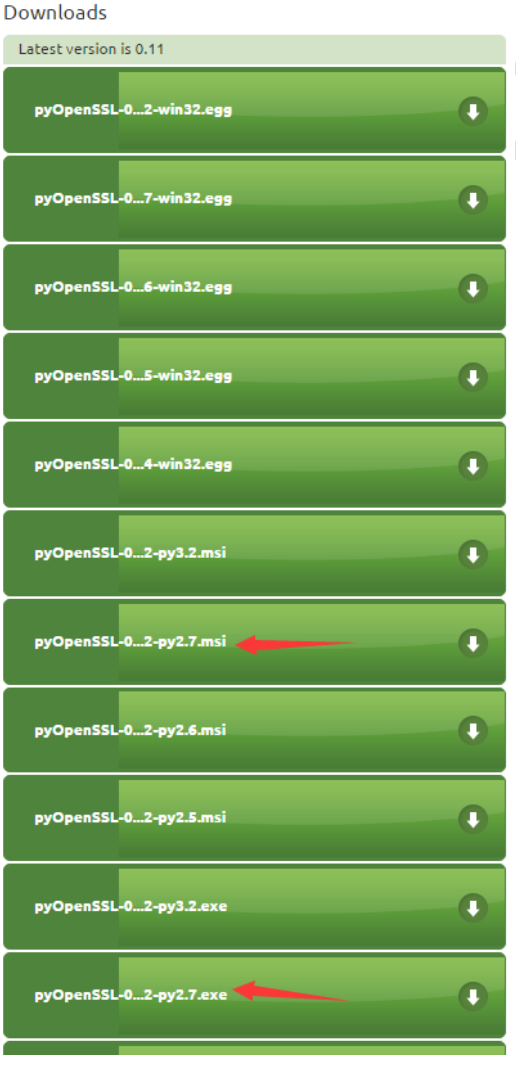

图 3-18 pyOpenSSL 下载界面

3.2.4 安装 Scrapy

终于轮到安装 scrapy 了，直接在 cmd 中输入 easy_install scrapy 回车即可。安装后在 cmd 命令行进行成功与否的验证如 Code 3-19 所示。

```
C:\Users\Administrator>scrapy

Scrapy 1.4.0 - no active project


Usage:

  scrapy <command> [options] [args]


Available commands:

  bench            Run quick benchmark test

  fetch            Fetch a URL using the Scrapy downloader

  genspider        Generate new spider using pre-defined templates

  runspider        Run a self-contained spider (without creating a project)

  settings         Get settings values

  shell            Interactive scraping console

  startproject     Create new project

  version          Print Scrapy version

  view             Open URL in browser, as seen by Scrapy


  [ more ]         More commands available when run from project directory


Use "scrapy <command> -h" to see more info about a command
```

Code 3-19 Scrapy 安装验证

安装完成，开始使用吧！

3.3 为什么要使用命令行工具？

3.3.1 Scrapy 命令行工具是什么

Scrapy 是通过 scrapy 命令行工具进行控制的。 这里我们称之为 “Scrapy tool” 以用来和子命令进行区分。对于子命令，我们称为 “command” 或者 “Scrapy commands”。

Scrapy tool 针对不同的目的提供了多个命令，每个命令支持不同的参数和选项。

3.3.2 默认的 Scrapy 项目结构

在开始对命令行工具以及子命令的探索前，让我们首先了解一下 Scrapy 的项目的目录结构。

虽然可以被修改，但所有的 Scrapy 项目默认有类似于下边的文件结构:

```
scrapy.cfg
myproject/
    __init__.py
    items.py
       middlewares.py
    pipelines.py
    settings.py
    spiders/
        __init__.py
        spider1.py
        spider2.py
        ...
```

1.scrapy.cfg

存放的目录被认为是项目的根目录 。该文件中包含 python 模块名的字段定义了项目的设置。

2. items.py

该文件中包含了 scrapy 数据容器模型代码。

Item 对象是种简单的容器，保存了爬取到得数据。其提供了类似于词典(dictionary-like)的 API 以及用于声明可用字段的简单语法。

3. middlewares.py

该文件中包含下载器中间件和爬虫中间件模型代码。

下载器中间件是介于 Scrapy 的 request/response 处理的钩子框架。 是用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。

爬虫中间件是介入到 Scrapy 的 spider 处理机制的钩子框架，您可以添加代码来处理发送给 Spiders 的 response 及 spider 产生的 item 和 request。。

4. pipelines.py

每个管道组件是实现了简单方法的 Python 类。他们接收到 Item 并通过它执行一些行为，同时也决定此 Item 是否继续通过后续的管道组件，或是被丢弃而不再进行处理。

5. settings.py

Scrapy 设定(settings)提供了定制 Scrapy 组件的方法。您可以控制包括核心(core)，插件(extension)，pipeline 及 spider 组件。

3.4 编写第一个 Scrapy 爬虫

本课程将带您完成下列任务:

1. 创建一个 Scrapy 项目
2. 定义提取的 Item
3. 编写爬取网站的 spider 并提取 Item
4. 编写 Item Pipeline 来存储提取到的 Item(即数据)

3.4.1 创建项目

在开始爬取之前，您必须创建一个新的 Scrapy 项目。 进入您打算存储代码的目录中，运行下列命令:

```
scrapy startproject tutorial
```

该命令将会创建包含下列内容的 tutorial 目录:

```
tutorial/
    scrapy.cfg
    tutorial/
        __init__.py
        items.py
        pipelines.py
```

```
    settings.py
    spiders/
        __init__.py
        ...
```

这些文件分别是:

- scrapy.cfg: 项目的配置文件
- tutorial/: 该项目的 python 模块。之后您将在此加入代码。
- tutorial/items.py: 项目中的 item 文件。
- tutorial/pipelines.py: 项目中的 pipelines 文件。
- tutorial/settings.py: 项目的设置文件。
- tutorial/spiders/: 放置 spider 代码的目录。

3.4.2 定义 Item

Item 是保存爬取到的数据的容器；其使用方法和 python 字典类似，并且提供了额外保护机制来避免拼写错误导致的未定义字段错误。

类似在 ORM 中做的一样，您可以通过创建一个 scrapy.Item 类，并且定义类型为 scrapy.Field 的类属性来定义一个 Item。(如果不了解 ORM,不用担心，您会发现这个步骤非常简单)

首先根据需要从 dmoz.org 获取到的数据对 item 进行建模。我们需要从 dmoz 中获取名字，url，以及网站的描述。对此，在 item 中定义相应的字段。编辑 tutorial 目录中的 items.py 文件，如 Code 3-20 所示。

```
import scrapy


class DmozItem(scrapy.Item):
    title=scrapy.Field()
    link=scrapy.Field()
    desc=scrapy.Field()
```

Code 3-20 items.py 代码

一开始这看起来可能有点复杂，但是通过定义 item， 您可以很方便的使用 Scrapy 的其他方法。而这些方法需要知道您的 item 的定义。

3.4.3 Spider 爬虫

Spider 是用户编写用于从单个网站(或者一些网站)爬取数据的类。

其包含了一个用于下载的初始 URL，如何跟进网页中的链接以及如何分析页面中的内容，提取生成 item 的方法。

为了创建一个 Spider，您必须继承 scrapy.Spider 类，且定义以下三个属性:

- name:用于区别 Spider。该名字必须是唯一的，您不可以为不同的 Spider 设定相同的名字。
- start_urls: 包含了 Spider 在启动时进行爬取的 url 列表。因此，第一个被获取到的页面将是其中之一。后续的 URL 则从初始的 URL 获取到的数据中提取。
- parse()是 spider 的一个方法。被调用时，每个初始 URL 完成下载后生成的 Response 对象将会作为唯一的参数传递给该函数。该方法负责解析返回的数据(response data) ,提取数据(生成 item)以及生成需要进一步处理的 URL 的 Request 对象。

以下为我们的第一个 Spider 代码，保存在 tutorial/spiders 目录下的 dmoz_spider.py 文件中，如 Code 3-21 所示:

```
# -*- coding: utf-8 -*-

import scrapy



class DmozSpider(scrapy.Spider):

    name = 'dmoz'

    allowed_domains = ['dmoz.org']

    start_urls = [

        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Books',

        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Resources'

    ]
```

```
    def parse(self, response):
        filename=response.split("/")[-2]
        with open(filename,'wb') as f:
            f.write(response.body)
```

Code 3-21 demoz_spider.py 代码

3.4.4 爬虫爬取

进入项目的根目录，执行下列命令启动 spider:

```
scrapy crawl dmoz
```

crawl dmoz 启动用于爬取 dmoz.org 的 spider，您将得到类似的输出，如图 3-22 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy crawl dmoz
2017-10-18 10:03:26 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: tutorial)
2017-10-18 10:03:26 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE':
'tutorial.spiders', 'SPIDER_MODULES': ['tutorial.spi
ders'], 'ROBOTSTXT_OBEY': True, 'BOT_NAME': 'tutorial'}
2017-10-18 10:03:27 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-10-18 10:03:27 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
```

```
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-10-18 10:03:27 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-10-18 10:03:27 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2017-10-18 10:03:27 [scrapy.core.engine] INFO: Spider opened
2017-10-18 10:03:27 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min),
scraped 0 items (at 0 items/min)
2017-10-18 10:03:27 [scrapy.extensions.telnet] DEBUG: Telnet console listening on
127.0.0.1:6023
2017-10-18 10:03:28 [scrapy.core.engine] DEBUG: Crawled (403) <GET
http://www.dmoz.org/robots.txt> (referer: None)
2017-10-18 10:03:28 [scrapy.core.engine] DEBUG: Crawled (403) <GET
http://www.dmoz.org/Computers/Programming/Langurages/Python/Books> (refe
rer: None)
2017-10-18 10:03:28 [scrapy.spidermiddlewares.httperror] INFO: Ignoring response <403
http://www.dmoz.org/Computers/Programming/Langurages/
Python/Books>: HTTP status code is not handled or not allowed
2017-10-18 10:03:28 [scrapy.core.engine] DEBUG: Crawled (403) <GET
http://www.dmoz.org/Computers/Programming/Langurages/Python/Resources> (
referer: None)
2017-10-18 10:03:28 [scrapy.spidermiddlewares.httperror] INFO: Ignoring response <403
http://www.dmoz.org/Computers/Programming/Langurages/
Python/Resources>: HTTP status code is not handled or not allowed
2017-10-18 10:03:28 [scrapy.core.engine] INFO: Closing spider (finished)
```

Code 3-22 dmoz 爬虫运行情况

查看包含 [dmoz] 的输出，可以看到输出的 log 中包含定义在 start_urls 的初始 URL，并且与 spider 中是一一对应的。在 log 中可以看到其没有指向其他页面((referer:None))。

除此之外，更有趣的事情发生了。就像我们 parse 方法指定的那样，有两个包含 url 所对应的内容的文件被创建了: Book，Resources，如图 3-23 所示 。

图 3-23 生成文件

刚才发生了什么？Scrapy 为 Spider 的 start_urls 属性中的每个 URL 创建了 scrapy.Request 对象，并将 parse 方法作为回调函数(callback)赋值给了 Request。

Request 对象经过调度，执行生成 scrapy.http.Response 对象并送回给 spider parse() 方法。

3.4.5 提取 Item

从网页中提取数据有很多方法。Scrapy 使用了一种基于 XPath 和 CSS 表达式机制: Scrapy Selectors。 关于 selector 和其他提取机制的信息请参考 Selector 文档。

这里给出 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

上边仅仅是几个简单的 XPath 例子，XPath 实际上要比这远远强大的多。

为了配合 XPath，Scrapy 除了提供了 Selector 之外，还提供了方法来避免每次从 response 中提取数据时生成 selector 的麻烦。

Selector 有四个基本的方法:

- 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 。
- 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.
- extract(): 序列化该节点为 unicode 字符串并返回 list。
- 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。

在 Shell 中尝试 Selector 选择器

为了介绍 Selector 的使用方法，接下来我们将要使用内置的 Scrapy shell。

您需要进入项目的根目录，执行下列命令来启动 shell:

```
scrapy shell http://www.dmoz.org/Computers/Programming/Languages/Python/Books/
```

注解：当您在终端运行 Scrapy 时，请一定记得给 url 地址加上引号，否则包含参数的 url(例如 & 字符)会导致 Scrapy 运行失败。

shell 的输出，如 Code 3-24 所示。

```
2017-10-18 10:08:29 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.0.0.1:6023
2017-10-18 10:08:29 [scrapy.core.engine] INFO: Spider opened
2017-10-18 10:08:30 [scrapy.core.engine] DEBUG: Crawled (403) <GET http://www.dmoz.org/robots.txt> (referer: None)
2017-10-18 10:08:30 [scrapy.core.engine] DEBUG: Crawled (403) <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Books/> (refe
rer: None)
2017-10-18 10:08:31 [traitlets] DEBUG: Using default logger
2017-10-18 10:08:31 [traitlets] DEBUG: Using default logger
[s] Available Scrapy objects:
[s]   scrapy     scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler    <scrapy.crawler.Crawler object at 0x0634EB90>
[s]   item       {}
[s]   request    <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s]   response   <403 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s]   settings   <scrapy.settings.Settings object at 0x0634EF50>
[s]   spider     <DmozSpider 'dmoz' at 0x6573a50>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects (by default, redirects are followed)
[s]   fetch(req)                  Fetch a scrapy.Request and update local objects
[s]   shelp()           Shell help (print this help)
[s]   view(response)    View response in a browser
```

Code 3-24 生成文件

当 shell 载入后，您将得到一个包含 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。

更为重要的是，当输入 response.selector 时， 您将获取到一个可以用于查询返回数据的 selector(选择器)， 以及映射到 response.selector.xpath() 、response.selector.css() 的 快捷方法 (shortcut): response.xpath() 和 response.css() 。

同时，shell 根据 response 提前初始化了变量 sel 。该 selector 根据 response 的类型自动选择最合适的分析规则(XML vs HTML)。

让我们来试试，如 Code 3-25 所示

```
In [1]: sel.xpath('//title')
2017-10-18 10:10:49 [py.warnings] WARNING: shell:1: ScrapyDeprecationWarning: "sel" shortcut
is deprecated. Use "response.xpath()", "response.css()" or "response.selector" instead

Out[1]: [<Selector xpath='//title' data=u'<title>DMOZ</titl
e>'>]

In [2]: sel.xpath('//title').extract()
2017-10-18 10:11:12 [py.warnings] WARNING: shell:1: ScrapyDeprecationWarning: "sel" shortcut
is deprecated. Use "response.xpath()", "response.css()" or "response.selector" instead

Out[2]: [u'<title>DMOZ</title>']

In [3]: sel.xpath('//title/text()')
2017-10-18 10:11:36 [py.warnings] WARNING: shell:1: ScrapyDeprecationWarning: "sel" shortcut
is deprecated. Use "response.xpath()", "response.css()" or "response.selector" instead

Out[3]: [<Selector xpath='//title/text()' data=u'DMOZ'>]

In [4]: sel.xpath('//title/text()').extract()
2017-10-18 10:11:44 [py.warnings] WARNING: shell:1: ScrapyDeprecationWarning: "sel" shortcut
is deprecated. Use "response.xpath()", "response.css()" or "response.selector" instead

Out[4]: [u'DMOZ']
```

Code 3-25 提取数据

现在，我们来尝试从这些页面中提取些有用的数据。

您可以在终端中输入 response.body 来观察 HTML 源码并确定合适的 XPath 表达式。不过，这任务非常无聊且不易。您可以考虑使用 Firefox 的 Firebug 扩展来使得工作更为轻松。详情请参考使用 Firebug 进行爬取和借助 Firefox 来爬取。

在查看了网页的源码后，您会发现网站的信息是被包含在 第二个 <ul> 元素中。

我们可以通过这段代码选择该页面中网站列表里所有 <li> 元素，如 Code 3-26 所示。

```
sel.xpath('//ul/li')
```

Code 3-26 提取数据

网站的描述，如 Code 3-27 所示。

```
sel.xpath('//ul/li/text()').extract()
```

Code 3-27 提取数据

网站的标题，如 Code 3-28 所示。

```
sel.xpath('//ul/li/a/text()').extract()
```

Code 3-28 提取数据

网站的链接，如 Code 3-29 所示。

```
sel.xpath('//ul/li/a/@href').extract()
```

Code 3-29 提取数据

之前提到过，每个.xpath()调用返回 selector 组成的 list，因此我们可以拼接更多的.xpath()来进一步获取某个节点。我们将在下边使用这样的特性，如 Code 3-30 所示。

```
for sel in response.xpath('//ul/li'):
        title=sel.xpath('a/text()').extract()
        link=sel.xpath('a/@href').extract()
        desc=sel.xpath('text()').extract()
```

Code 3-30 xpath()进一步操作

在我们的 spider 中加入这段代码，如 Code 3-31 所示。

```
# -*- coding: utf-8 -*-
import scrapy
class DmozSpider(scrapy.Spider):
    name = 'dmoz'
    allowed_domains = ['dmoz.org']
    start_urls = [
        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Books',
        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Resources'
    ]
```

```
    def parse(self, response):
        for sel in response.xpath('//ul/li'):
            title=sel.xpath('a/text()').extract()
            link=sel.xpath('a/@href').extract()
            desc=sel.xpath('text()').extract()
            print title,link,desc
```

Code 3-31 spider 代码

现在尝试再次爬取 dmoz.org，您将看到爬取到的网站信息被成功输出，使用命令，scrapy crawl dmoz。

3.4.6 使用 Item

Item 对象是自定义的 python 字典。学员可以使用标准的字典语法来获取到其每个字段的值。(字段即是我们之前用 Field 赋值的属性)，代码如 Code 3-32 所示

```
item=DmozItem()
item['title']= ‘Example title’
item['title']
```

Code 3-32 使用 Item

一般来说，Spider 将会将爬取到的数据以 Item 对象返回。所以为了将爬取的数据返回，我们最终的代码将是如 Code 3-33 所示。

```
# -*- coding: utf-8 -*-
import scrapy
from tutorial.items import DmozItem
class DmozSpider(scrapy.Spider):
    name = 'dmoz'
    allowed_domains = ['dmoz.org']
    start_urls = [
        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Books',
        'http://www.dmoz.org/Computers/Programming/Langurages/Python/Resources'
    ]
    def parse(self, response):
        for sel in response.xpath('/html/body/center/h3/p[4]'):
```

```
        item=DmozItem()
        item['title']=sel.xapth('a/text()').extract()
        item['link']=sel.xpath('a/@href').extract()
        # item['desc']=sel.xapth('text()').extract()
        yield item
```

Code 3-33 Item 代码

3.4.7 保存爬取到的数据

最简单存储爬取的数据的方式是使用 Feed exports ,使用命令 ,scrapy crawl dmoz -o items.json。该命令将采用 JSON 格式对爬取的数据进行序列化，生成 items.json 文件。

在类似本篇教程里这样小规模的项目中，这种存储方式已经足够。 如果需要对爬取到的 item 做更多更为复杂的操作，您可以编写 Item Pipeline 。 类似于我们在创建项目时对 Item 做的，用于您编写自己的 tutorial/pipelines.py 也被创建。 不过如果您仅仅想要保存 item，您不需要实现任何的 pipeline。

第 4 章 命令行工具

本章工作任务

- 任务 1：为什么使用命令行工具？
- 任务 2：熟练使用常用的命令
- 任务 3：了解掌握可用的工具命令

本章技能目标及重难点

编号	技能点描述	级别
1	为什么使用命令行工具	★
2	熟练使用常用的命令	★★★
3	了解掌握可用的工具命令	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 命令行工具，需要同学们理解为什么使用命令行工具，它的概念、特点。最主要的是需要大家学会如何使用命令行工具。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

4.1 为什么使用命令行工具

Scrapy 是通过 scrapy 命令行工具进行控制的。 这里我们称之为 “Scrapy tool” 以用来和子命令进行区分。对于子命令，我们称为 “command” 或者 “Scrapy commands”。

Scrapy tool 针对不同的目的提供了多个命令，每个命令支持不同的参数和选项。

4.1.1 默认的 Scrapy 项目结构

在开始对命令行工具以及子命令的探索前，让我们首先了解一下 Scrapy 的项目的目录结构。

虽然可以被修改，但所有的 Scrapy 项目默认有类似于下边的文件结构，如图 4-1 所示。

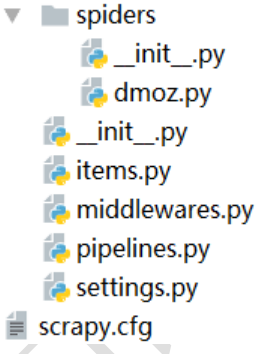

图 4-1 Scrapy 项目目录结构

scrapy.cfg 存放的目录被认为是 项目的根目录 。该文件中包含 python 模块名的字段定义了项目的设置，如 Code 4-2 所示。

```
# Automatically created by: scrapy startproject
#
# For more information about the [deploy] section see:
# https://scrapyd.readthedocs.org/en/latest/deploy.html

[settings]
default = tutorial.settings

[deploy]
#url = http://localhost:6800/
```

```
project = tutorial
```

Code 4-2 scrapy.cfg

4.1.2 使用 Scrapy 工具

您可以以无参数的方式启动 Scrapy 工具。该命令将会给出一些使用帮助以及可用的命令 ,如 Code 4-3 所示。

```
C:\Users\Administrator>scrapy
Scrapy 1.4.0 - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  bench          Run quick benchmark test
  fetch          Fetch a URL using the Scrapy downloader
  genspider      Generate new spider using pre-defined templates
  runspider      Run a self-contained spider (without creating a project)
  settings       Get settings values
  shell          Interactive scraping console
  startproject   Create new project
  version        Print Scrapy version
  view           Open URL in browser, as seen by Scrapy

  [ more ]       More commands available when run from project directory

Use "scrapy <command> -h" to see more info about a command
```

Code 4-3 Scrapy 命令

如果您在 Scrapy 项目中运行，当前激活的项目将会显示在输出的第一行。上面的输出就是响应的例子。如果您在一个项目中运行命令将会得到类似的输出，如 Code4-3 所示。

4.1.3 创建项目

一般来说，使用 scrapy 工具的第一件事就是创建您的 Scrapy 项目：

```
scrapy startproject myproject
```

该命令将会在 myproject 目录中创建一个 Scrapy 项目。

接下来，进入到项目目录中：

```
cd myproject
```

这时候您就可以使用 scrapy 命令来管理和控制您的项目了。

4.1.4 控制项目

您可以在您的项目中使用 scrapy 工具来对其进行控制和管理。

比如，创建一个新的 spider：

```
scrapy genspider mydomain mydomain.com
```

有些 Scrapy 命令(比如 crawl)要求必须在 Scrapy 项目中运行。 您可以通过下边的 commands reference 来了解哪些命令需要在项目中运行，哪些不用。

另外要注意，有些命令在项目里运行时的效果有些许区别。 以 fetch 命令为例，如果被爬取的 url 与某个特定 spider 相关联， 则该命令将会使用 spider 的动作(spider-overridden behaviours)。 (比如 spider 指定的 user_agent)。 该表现是有意而为之的。一般来说， fetch 命令就是用来测试检查 spider 是如何下载页面。

4.2 可用的工具命令(tool commands)

该章节提供了可用的内置命令的列表。每个命令都提供了描述以及一些使用例子。您总是可以通过运行命令来获取关于每个命令的详细内容：

```
scrapy <command> -h
```

您也可以查看所有可用的命令：

```
scrapy -h
```

Scrapy 提供了两种类型的命令。一种必须在 Scrapy 项目中运行(针对项目(Project-specific)的命令)，另外一种则不需要(全局命令)。全局命令在项目中运行时的表现可能会与在非项目中运行有些许差别(因为可能会使用项目的设定)。

全局命令：

- startproject
- settings
- runspider
- shell
- fetch
- view

- version

项目(Project-only)命令：

- crawl
- check
- list
- edit
- parse
- genspider
- deploy
- bench

4.2.1 startproject

- 语法: scrapy startproject <project_name>
- 是否需要项目: no

在 project_name 文件夹下创建一个名为 project_name 的 Scrapy 项目。

```
scrapy startproject myproject
```

4.2.2 settings

- 语法: scrapy settings [options]
- 是否需要项目: no

在项目中运行时，该命令将会输出项目的设定值，否则输出 Scrapy 默认设定，如 Code 4-5 所示。

```
C:\Users\Administrator\PycharmProjects\myproject>scrapy setting --help
Scrapy 1.4.0 - project: myproject

Unknown command: setting

Use "scrapy" to see available commands

C:\Users\Administrator\PycharmProjects\myproject>scrapy settings --help
Usage
=====
  scrapy settings [options]

Get settings values
```

```
Options
=======
--help, -h                  show this help message and exit
--get=SETTING                print raw setting value
--getbool=SETTING            print setting value, interpreted as a boolean
--getint=SETTING             print setting value, interpreted as an integer
--getfloat=SETTING           print setting value, interpreted as a float
--getlist=SETTING            print setting value, interpreted as a list

Global Options
--------------
--logfile=FILE              log file. if omitted stderr will be used
--loglevel=LEVEL, -L LEVEL
                             log level (default: DEBUG)
--nolog                      disable logging completely
--profile=FILE              write python cProfile stats to FILE
--pidfile=FILE              write process ID to FILE
--set=NAME=VALUE, -s NAME=VALUE
                             set/override setting (may be repeated)
--pdb                        enable pdb on failure
```

Code 4-5 setting

4.2.3 runspider

- 语法: scrapy runspider <spider_file.py>
- 是否需要项目: no

在未创建项目的情况下，运行一个编写在 Python 文件中的 spider，如 Code 4-6 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial\tutorial\spiders 的目录

2017-10-18   11:05      <DIR>              .
2017-10-18   11:05      <DIR>              ..
2017-10-18   10:52                 1,018 dmoz.py
2017-10-18   11:05                 1,156 dmoz.pyc
2017-07-14   13:19                   161 __init__.py
2017-10-18   09:51                   169 __init__.pyc
               4 个文件              2,504 字节
```

```
                2 个目录 46,255,362,048 可用字节
C:\Users\Administrator\PycharmProjects\zhibo\tutorial\tutorial\spiders>scrapy ru
nspider dmoz.py
2017-10-18 11:06:32 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: tutorial
)
2017-10-18 11:06:32 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MO
DULE': 'tutorial.spiders', 'SPIDER_LOADER_WARN_ONLY': True, 'SPIDER_MODULES': ['
tutorial.spiders'], 'ROBOTSTXT_OBEY': True, 'BOT_NAME': 'tutorial'}
2017-10-18 11:06:32 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-10-18 11:06:33 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
```

Code 4-6 runspider

4.2.4 shell

- 语法: scrapy shell <url>
- 是否需要项目: no

Scrapy 终端是一个交互终端，供您在未启动 spider 的情况下尝试及调试您的爬取代码。 其本意是用来测试提取数据的代码，不过您可以将其作为正常的 Python 终端，在上面测试任何的 Python 代码。

该终端是用来测试 XPath 或 CSS 表达式,查看他们的工作方式及从爬取的网页中提取的数据。在编写您的 spider 时，该终端提供了交互性测试您的表达式代码的功能，免去了每次修改后运行 spider 的麻烦。

一旦熟悉了 Scrapy 终端后，您会发现其在开发和调试 spider 时发挥的巨大作用。

如果您安装了 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。

启动终端

您可以使用 shell 来启动 Scrapy 终端:

```
scrapy shell <url>
```

<url> 是您要爬取的网页的地址。

使用终端

Scrapy 终端仅仅是一个普通的 Python 终端(或 IPython)。其提供了一些额外的快捷方式。

可用的快捷命令(shortcut)

- shelp() - 打印可用对象及快捷命令的帮助列表
- fetch(request_or_url) - 根据给定的请求(request)或 URL 获取一个新的 response ,并更新相关的对象
- view(response) - 在本机的浏览器打开给定的 response。其会在 response 的 body 中添加一个 tag ，使得外部链接(例如图片及 css)能正确显示。 注意，该操作会在本地创建一个临时文件，且该文件不会被自动删除。

可用的 Scrapy 对象

这些对象有:

- crawler - 当前 Crawler 对象。
- spider - 处理 URL 的 spider。对当前 URL 没有处理的 Spider 时则为一个 Spider 对象。
- request - 最近获取到的页面的 Request 对象。您可以使用 replace() 修改该 request。或者 使用 fetch 快捷方式来获取新的 request。
- response - 包含最近获取到的页面的 Response 对象。
- sel - 根据最近获取到的 response 构建的 Selector 对象。
- settings - 当前的 Scrapy settings

终端会话(shell session)样例

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial\tutorial\spiders>scrapy sh
ell http://www.example.com/some/pasge.html
2017-10-18 11:09:17 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: tutorial
```

```
)
2017-10-18 11:09:17 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MO
DULE': 'tutorial.spiders', 'ROBOTSTXT_OBEY': True, 'DUPEFILTER_CLASS': 'scrapy.d
upefilters.BaseDupeFilter', 'SPIDER_MODULES': ['tutorial.spiders'], 'BOT_NAME':
'tutorial', 'LOGSTATS_INTERVAL': 0}
2017-10-18 11:09:17 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-10-18 11:09:17 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-10-18 11:09:17 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-10-18 11:09:17 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2017-10-18 11:09:17 [scrapy.extensions.telnet] DEBUG: Telnet console listening o
n 127.0.0.1:6023
2017-10-18 11:09:17 [scrapy.core.engine] INFO: Spider opened
2017-10-18 11:09:19 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://www.ex
ample.com/robots.txt> (referer: None)
2017-10-18 11:09:20 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://www.ex
ample.com/some/pasge.html> (referer: None)
```

```
2017-10-18 11:09:21 [traitlets] DEBUG: Using default logger
2017-10-18 11:09:21 [traitlets] DEBUG: Using default logger
[s] Available Scrapy objects:
[s]   scrapy       scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler      <scrapy.crawler.Crawler object at 0x06319B90>
[s]   item         {}
[s]   request      <GET http://www.example.com/some/pasge.html>
[s]   response     <404 http://www.example.com/some/pasge.html>
[s]   settings     <scrapy.settings.Settings object at 0x06319FD0>
[s]   spider       <DefaultSpider 'default' at 0x6545eb0>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects (by default
, redirects are followed)
[s]   fetch(req)                  Fetch a scrapy.Request and update local object
s
[s]   shelp()           Shell help (print this help)
[s]   view(response)    View response in a browser
In [1]:
```

Code 4-7 shell

在浏览器中打开给定的 URL，并以 Scrapy spider 获取到的形式展现。有些时候 spider 获取到的页面和普通用户看到的并不相同。 因此该命令可以用来检查 spider 所获取到的页面，并确认这是您所期望的，如图 4-7 所示。

4.2.5 fetch

- 语法: scrapy fetch <url>
- 是否需要项目: no

使用 Scrapy 下载器(downloader)下载给定的 URL，并将获取到的内容送到标准输出。

该命令以 spider 下载页面的方式获取页面。例如，如果 spider 有 USER_AGENT 属性修改了 User Agent，该命令将会使用该属性。

因此，您可以使用该命令来查看 spider 如何获取某个特定页面。

该命令如果非项目中运行则会使用默认 Scrapy downloader 设定，命令如 Code 4-8 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy fetch
2017-10-18 11:10:39 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: tutorial)
2017-10-18 11:10:39 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE':
```

```
'tutorial.spiders', 'SPIDER_MODULES': ['tutorial.spi
ders'], 'ROBOTSTXT_OBEY': True, 'BOT_NAME': 'tutorial'}
Usage
=====
  scrapy fetch [options] <url>

Fetch a URL using the Scrapy downloader and print its content to stdout. You
may want to use --nolog to disable logging

Options
=======
--help, -h              show this help message and exit
--spider=SPIDER         use this spider
--headers               print response HTTP headers instead of body
--no-redirect           do not handle HTTP 3xx status codes and print response
                        as-is

Global Options
--------------
--logfile=FILE          log file. if omitted stderr will be used
--loglevel=LEVEL, -L LEVEL
                        log level (default: DEBUG)
--nolog                 disable logging completely
--profile=FILE          write python cProfile stats to FILE
--pidfile=FILE          write process ID to FILE
--set=NAME=VALUE, -s NAME=VALUE
                        set/override setting (may be repeated)
--pdb                   enable pdb on failure
```

Code 4-8 fetch 命令参数

显示页面的 HTML 代码。

--help、-h：显示帮助信息和退出

--spider=SPIDER：使用爬虫

--headers：打印 reponse 的 http 头包信息代替主题

--no-redirect：不处理 HTTP 状态和打印信息

--logfile=FILE：日志文件，如果省略错误信息使用

--loglevel=LEVEL，-L LEVEL：日志等级，默认 DEBUG

--nolog：不使用日志

--profile=FILE：写 python 的简单信息到文件中

--set=NAME=VALUE，-s NAME=VALUE：设置或者覆盖设置

--pdb：使 PDB 失败

4.2.6 version

- 语法: scrapy version [-v]
- 是否需要项目: no

输出 Scrapy 版本。配合 -v 运行时，该命令同时输出 Python，Twisted 以及平台的信息，方便 bug 提交，如 Code 4-10 所示。

```
C:\Users\Administrator>scrapy version
Scrapy 1.4.0

C:\Users\Administrator>scrapy version -v
Scrapy       : 1.4.0
lxml         : 3.8.0.0
libxml2      : 2.7.8
cssselect : 1.0.1
parsel       : 1.2.0
w3lib        : 1.17.0
Twisted      : 17.5.0
Python       : 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32
bit (Intel)]
pyOpenSSL : 17.1.0 (OpenSSL 1.1.0f  25 May 2017)
Platform    : Windows-7-6.1.7601-SP1
```

Code 4-9 version

4.2.7 crawl

- 语法: scrapy crawl <spider>
- 是否需要项目: yes

使用 spider 进行爬取，运行效果如 Code 4-11 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial\tutorial>scrapy crawl dmoz
2017-10-18 11:22:07 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: tutorial)
2017-10-18 11:22:07 [scrapy.utils.log] INFO: Overridden settings: {'NEWSPIDER_MODULE':
'tutorial.spiders', 'SPIDER_MODULES': ['tutorial.spi
```

```
ders'], 'ROBOTSTXT_OBEY': True, 'BOT_NAME': 'tutorial'}
2017-10-18 11:22:07 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-10-18 11:22:07 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-10-18 11:22:07 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-10-18 11:22:07 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2017-10-18 11:22:07 [scrapy.core.engine] INFO: Spider opened
2017-10-18 11:22:07 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/min),
scraped 0 items (at 0 items/min)
2017-10-18 11:22:07 [scrapy.extensions.telnet] DEBUG: Telnet console listening on
127.0.0.1:6023
{'downloader/exception_count': 1,
'downloader/request_bytes': 954,
 'downloader/request_count': 4,
 'downloader/request_method_count/GET': 4,
 'downloader/response_bytes': 3525,
```

```
 'downloader/response_count': 3,
'finish_reason': 'finished',
 'finish_time': datetime.datetime(2017, 10, 18, 3, 22, 34, 418000),
 'httperror/response_ignored_count': 2,
'log_count/DEBUG': 5,
 'log_count/INFO': 9,
 'response_received_count': 3,
 'retry/count': 1,
'scheduler/dequeued': 2,
 'scheduler/dequeued/memory': 2,
 'scheduler/enqueued': 2,
 'scheduler/enqueued/memory': 2,
 'start_time': datetime.datetime(2017, 10, 18, 3, 22, 7, 489000)}
2017-10-18 11:22:34 [scrapy.core.engine] INFO: Spider closed (finished)
```

Code 4-10 crawl 运行 spider

4.2.8 check

- 语法: scrapy check [-l] <spider>
- 是否需要项目: yes

运行 contract 检查，运行效果如 Code 4-12 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy check

----------------------------------------------------------------------
Ran 0 contracts in 0.000s

OK
```

Code 4-11 contract 检查

4.2.9 list

- 语法: scrapy list
- 是否需要项目: yes

列出当前项目中所有可用的 spider。每行输出一个 spider，如 Code 4-13 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy list
dmoz
```

Code 4-12 显示 spider 列表

4.2.10 edit

- 语法: scrapy edit <spider>
- 是否需要项目: yes

使用 EDITOR 中设定的编辑器编辑给定的 spider。该命令仅仅是提供一个快捷方式。开发者可以自由选择其他工具或者 IDE 来编写调试 spider。

```
scrapy edit spider1
```

4.2.11 parse

- 语法: scrapy parse <url> [options]
- 是否需要项目: yes

获取给定的 URL 并使用相应的 spider 分析处理。如果您提供--callback 选项 ,则使用 spider 的该方法处理，否则使用 parse。

支持的选项：

- --spider=SPIDER：跳过自动检测 spider 并强制使用特定的 spider
- --a NAME=VALUE：设置 spider 的参数(可能被重复)
- --callback or -c：spider 中用于解析返回(response)的回调函数
- --pipelines：在 pipeline 中处理 item
- --rules or -r：使用 CrawlSpider 规则来发现用来解析返回(response)的回调函数
- --noitems：不显示爬取到的 item
- --nolinks：不显示提取到的链接
- --nocolour：避免使用 pygments 对输出着色
- --depth or -d：指定跟进链接请求的层次数(默认：1)
- --verbose or -v：显示每个请求的详细信息

4.2.12 genspider

- 语法: scrapy genspider [-t template] <name> <domain>
- 是否需要项目: yes

在当前项目中创建 spider。

这仅仅是创建 spider 的一种快捷方法。该方法可以使用提前定义好的模板来生成 spider。您也可以自己创建 spider 的源码文件。如 Code 4-15 所示，查看 genspider 的参数。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy genspider -l
Available templates:
  basic
```

```
  crawl
  csvfeed
  xmlfeed
```

Code 4-13 genspider 列表

如图 4-16 所示，使用 genspider 建立默认的 spider。

```
C:\Users\Administrator\PycharmProjects\zhibo\tutorial>scrapy genspider -d basic
# -*- coding: utf-8 -*-
import scrapy


class $classname(scrapy.Spider):
    name = '$name'
    allowed_domains = ['$domain']
    start_urls = ['http://$domain/']

    def parse(self, response):
        pass
```

Code 4-14 genspider 建立 spider

如图 4-17 所示，使用 genspider 建立自定义的 spider。

```
C:\Users\Administrator\PycharmProjects\zhibo>scrapy genspider -t basic example e
xample.com
Created spider 'example' using template 'basic'
```

Code 4-15 genspider 建立自定义 spider

4.2.13 bench

- 语法: scrapy bench
- 是否需要项目: yes

运行 benchmark 测试。Benchmarking，如 Code 4-18 所示。

```
C:\Users\Administrator\PycharmProjects\zhibo>scrapy bench
2017-10-18 11:30:45 [scrapy.utils.log] INFO: Scrapy 1.4.0 started (bot: scrapybo
t)
2017-10-18 11:30:45 [scrapy.utils.log] INFO: Overridden settings: {'CLOSESPIDER_
TIMEOUT': 10, 'LOG_LEVEL': 'INFO', 'LOGSTATS_INTERVAL': 1}
2017-10-18 11:30:47 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.closespider.CloseSpider',
```

```
 'scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2017-10-18 11:30:47 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2017-10-18 11:30:47 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2017-10-18 11:30:47 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2017-10-18 11:30:47 [scrapy.core.engine] INFO: Spider opened
2017-10-18 11:30:47 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pag
es/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:48 [scrapy.extensions.logstats] INFO: Crawled 61 pages (at 3660
 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:49 [scrapy.extensions.logstats] INFO: Crawled 117 pages (at 336
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:50 [scrapy.extensions.logstats] INFO: Crawled 173 pages (at 336
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:51 [scrapy.extensions.logstats] INFO: Crawled 221 pages (at 288
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:52 [scrapy.extensions.logstats] INFO: Crawled 277 pages (at 336
0 pages/min), scraped 0 items (at 0 items/min)
```

```
2017-10-18 11:30:53 [scrapy.extensions.logstats] INFO: Crawled 325 pages (at 288
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:54 [scrapy.extensions.logstats] INFO: Crawled 381 pages (at 336
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:55 [scrapy.extensions.logstats] INFO: Crawled 437 pages (at 336
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:56 [scrapy.extensions.logstats] INFO: Crawled 485 pages (at 288
0 pages/min), scraped 0 items (at 0 items/min)
2017-10-18 11:30:57 [scrapy.core.engine] INFO: Closing spider (closespider_timeo
ut)
```

Code 4-16bensh

第 5 章 Items

本章工作任务

- 任务 1：为什么要使用 Item？
- 任务 2：如何使用 Item？
- 任务 3：Item 的扩展

本章技能目标及重难点

编号	技能点描述	级别
1	为什么要使用 Item？	★
2	如何使用 Item？	★★★
3	Item 的扩展	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 的 Items，需要同学们学会创建 Item 及相关扩展的 Item 应用。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

5.1 为什么要使用 Item？

爬取的主要目标就是从非结构性的数据源提取结构性数据，例如网页。Scrapy 提供 Item 类来满足这样的需求。

Item 对象是种简单的容器，保存了爬取到得数据。其提供了类似于词典(dictionary-like)的 API 以及用于声明可用字段的简单语法。

5.1.1 声明 Item

Item 使用简单的 class 定义语法以及 Field 对象来声明，如 Code 5-1 所示。

```
import scrapy
class Product(scrapy.Item):
    name=scrapy.Field()
    price=scrapy.Field()
    stock=scrapy.Field()
    last_updated=scrapy.Field(serializer=str)
```

Code 5-1 声明 Item

熟悉 Django 的学员一定会注意到 Scrapy Item 定义方式与 Django Models 很类似，不过没有那么多不同的字段类型(Field type)，更为简单。

5.1.2 Item 字段(Item Fields)

Field 对象指明了每个字段的元数据(metadata)。例如下面例子中 last_updated 中指明了该字段的序列化函数。

您可以为每个字段指明任何类型的元数据。Field 对象对接受的值没有任何限制。也正是因为这个原因，文档也无法提供所有可用的元数据的键(key)参考列表。Field 对象中保存的每个键可以由多个组件使用，并且只有这些组件知道这个键的存在。您可以根据自己的需求，定义使用其他的 Field 键。设置 Field 对象的主要目的就是在一个地方定义好所有的元数据。一般来说，那些依赖某个字段的组件肯定使用了特定的键(key)。您必须查看组件相关的文档，查看其用了哪些元数据键(metadata key)。

需要注意的是，用来声明 item 的 Field 对象并没有被赋值为 class 的属性。不过您可以通过 Item.fields 属性进行访问。

以上就是所有您需要知道的如何声明 item 的内容了。

5.2 如何使用 Item？

接下来以下边声明的 Product item 来演示一些 item 的操作。您会发现 API 和 dict API 非常相似。

5.2.1 创建 item

Code 图 5-2 所示。

```
>>> product=Product(name='Desktop PC',price=1000)
>>> print product
{'name': 'Desktop PC', 'price': 1000}
```

Code 5-2 创建 Item

5.2.2 获取字段的值

get 方法可以设置默认值，如此可以避免报错信息的出现。如 Code 5-3 所示。

```
>>> product['name']
'Desktop PC'
>>> product.get('name')
'Desktop PC'
>>> product['price']
1000
>>> product['last_updated']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\site-packages\scrapy\item.py", line 59, in __getitem__
    return self._values[key]
KeyError: 'last_updated'
>>> product.get('last_updated','not set')
'not set'
>>> product['lala']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\site-packages\scrapy\item.py", line 59, in __getitem__
    return self._values[key]
```

```
KeyError: 'lala'
>>> product.get('lala','unknown filed')
'unknown filed'
>>> 'name' in product # is name field populated?
True
>>> 'last_updated' in product # is last_updated populated?
False
>>> 'lala' in product.fields # is lala a declared field?
False
```

Code 5-3 获取字段值

5.2.3 设置字段的值

当获取一个不存在的属性，会显示错误信息。如表 5-4 所示。

```
>>> product['last_updated']='today'
>>> product['last_updated']
'today'
>>> product['lala']='test'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\site-packages\scrapy\item.py", line 66, in __setitem__
    (self.__class__.__name__, key))
KeyError: 'Product does not support field: lala'
```

Code 5-4 设置字段值

5.2.4 获取所有获取到的值

您可以使用 dict API 来获取所有的值，即 keys()，同时也可以使用 items()。如图 5-5 所示。

```
>>> product.keys()
['price', 'last_updated', 'name']
>>> product.items()
[('price', 1000), ('last_updated', 'today'), ('name', 'Desktop PC')]
```

Code 5-5 获取所有获取到的值

5.2.5 复制 item

可以使用对象方式，也可以使用 copy()方法复制 item。如 Code 5-6 所示。

```
>>> product2=Product(product)
```

```
>>> print product2
{'last_updated': 'today', 'name': 'Desktop PC', 'price': 1000}
>>> product3=product2.copy()
>>> print product3
{'last_updated': 'today', 'name': 'Desktop PC', 'price': 1000}
```

Code 5-6 复制 Item

5.3 Item 扩展

5.3.1 根据 item 创建字典(dict)

使用 dict 方式创建字典，如 Code 5-7 所示。

```
>>> dict(product) #create a dict from all populated values
{'price': 1000, 'last_updated': 'today', 'name': 'Desktop PC'}
```

Code 5-7 根据 item 创建字典(dict)

5.3.2 根据字典(dict)创建 item

传入 dict 来创建 Item，如 Code 5-8 所示。

```
>>> Product({'name':'Laptop PC','price':1500})
{'name': 'Laptop PC', 'price': 1500}
>>> Product({'name':'Laptop PC','lala':1500})#warning:unknown field in dict
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "C:\Python27\lib\site-packages\scrapy\item.py", line 56, in __init__
    self[k] = v
  File "C:\Python27\lib\site-packages\scrapy\item.py", line 66, in __setitem__
    (self.__class__.__name__, key))
KeyError: 'Product does not support field: lala'
```

Code 5-8 根据字典创建 Item

5.3.3 扩展 Item

您可以通过继承原始的 Item 来扩展 item(添加更多的字段或者修改某些字段的元数据)。代码如图 5-9 所示。

```
import scrapy
class Product(scrapy.Item):
    name=scrapy.Field()
```

```
    price=scrapy.Field()
    stock=scrapy.Field()
    last_updated=scrapy.Field(serializer=str)
class DiscountedProduct(Product):
    discount_precent=scrapy.Field(serializer=str)
    discount_expiration_date=scrapy.Field()
```

Code 5-9 扩展 Item

您也可以通过使用原字段的元数据，添加新的值或修改原来的值来扩展字段的元数据，代码如 Code 5-10 所示。

```
class SpecificProduct(Product):
    name=scrapy.Field(Product.fields['name'],serializer=my_serializer)
```

Code 5-10 扩展 Item

这段代码在保留所有原来的元数据值的情况下添加(或者覆盖)了 name 字段的 serializer。

serializer，表示序列化的意思，可以使用自定义的方法，也可以使用内置的 str

5.3.4 字段(Field)对象

Field 仅仅是内置的 dict 类的一个别名，并没有提供额外的方法或者属性。换句话说，Field 对象完完全全就是 Python 字典(dict)。被用来基于类属性(class attribute)的方法来支持 item 声明语法。

class scrapy.item.Field([arg])

第 6 章 Spiders

本章工作任务

- 任务 1：为什么要使用 Spiders？
- 任务 2：如何使用 Spiders？
- 任务 3：Spiders 的扩展

本章技能目标及重难点

编号	技能点描述	级别
1	为什么要使用 Spiders？	★
2	如何使用 Spiders？	★★★
3	Spiders 的扩展	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 的 Spiders，需要同学们学会使用 Spiders 及相关扩展的 Spiders 应用。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

6.1 为什么要使用 Spiders？

6.1.1 Spider 介绍

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。

对 spider 来说，爬取的循环类似下文:

1．以初始的 URL 初始化 Request，并设置回调函数。当该 request 下载完毕并返回时，将生成 response，并作为参数传给该回调函数。

2．spider 中初始的 request 是通过调用 start_requests()来获取的。start_requests()读取 start_urls 中的 URL， 并以 parse 为回调函数生成 Request。

3．在回调函数内分析返回的(网页)内容，返回 Item 对象或者 Request 或者一个包括二者的可迭代容器。 返回的 Request 对象之后会经过 Scrapy 处理，下载相应的内容，并调用设置的 callback 函数(函数可相同)。

4．在回调函数内，您可以使用 选择器(Selectors) (您也可以使用 BeautifulSoup, lxml 或者您想用的任何解析器) 来分析网页内容，并根据分析的数据生成 item。

5．最后，由 spider 返回的 item 将被存到数据库(由某些 Item Pipeline 处理)或使用 Feed exports 存入到文件中。

虽然该循环对任何类型的 spider 都(多少)适用，但 Scrapy 仍然为了不同的需求提供了多种默认 spider。 之后将讨论这些 spider。

6.1.2 创建 Spider

Spider 可以通过接受参数来修改其功能。 spider 参数一般用来定义初始 URL 或者指定限制爬取网站的部分。 您也可以使用其来配置 spider 的任何功能。

在运行 crawl 时添加-a 可以传递 Spider 参数：

```
scrapy crawl myspider -a category=electronics
```

Spider 在构造器(constructor)中获取参数，如表 6-1 所示。

```
import scrapy
class MySpider(scrapy.Spider):
    name='myspider'
    def __init__(self,category=None,*args,**kwargs):
        super(MySpider, self).__init__(*args,**kwargs)
        self.start_urls=['http://www.example.com/categories/%s'%category]
```

Code 6-1 Spider 参数

Spider 参数也可以通过 Scrapyd 的 schedule.json API 来传递。

6.2 如何使用 Spiders？

Scrapy 提供多种方便的通用 spider 供您继承使用。 这些 spider 为一些常用的爬取情况提供方便的特性， 例如根据某些规则跟进某个网站的所有链接、根据 Sitemaps 来进行爬取，或者分析 XML/CSV 源。

下面 spider 的示例中，我们假定您有个项目在 myproject.items 模块中声明了 TestItem，如图 6-2 所示。

```
import scrapy
class TestItem(scrapy.Item):
    id=scrapy.Field()
    name=scrapy.Field()
    description=scrapy.Field()
```

Code 6-2 TestItem

6.2.1 Spider 定义

class scrapy.spider.Spider

Spider 是最简单的 spider。每个其他的 spider 必须继承自该类(包括 Scrapy 自带的其他 spider 以及您自己编写的 spider)。Spider 并没有提供什么特殊的功能。其仅仅请求给定的 start_urls/start_requests，并根据返回的结果(resulting responses)调用 spider 的 parse 方法。

1.参数

name

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。 不过您可以生成多个相同的 spider 实例(instance)，这没有任何限制。 name 是 spider 最重要的属性，而且是必须的。

如果该 spider 爬取单个网站(single domain)，一个常见的做法是以该网站(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 。

allowed_domains

可选。包含了 spider 允许爬取的域名(domain)列表(list)。 当 OffsiteMiddleware 启用时， 域名不在列表中的 URL 不会被跟进。

start_urls

URL 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。 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。

2.方法

start_requests()

该方法必须返回一个可迭代对象(iterable)。该对象包含了 spider 用于爬取的第一个 Request。

当 spider 启动爬取并且未制定 URL 时，该方法被调用。 当指定了 URL 时，make_requests_from_url()将被调用来创建 Request 对象。 该方法仅仅会被 Scrapy 调用一次，因此您可以将其实现为生成器。

该方法的默认实现是使用 start_urls 的 url 生成 Request。

如果您想要修改最初爬取某个网站的 Request 对象，您可以重写(override)该方法。 例如，如果您需要在启动时以 POST 登录某个网站，你可以这么写，如 Code 6-3 所示。

```
def start_request(self):
    return [scrapy.FormRequest('http://www.example.com/login',
                               formdata={'user':'john','pass':'secret'},
                               callback=self.logged_in)]
def logged_in(self,response):
    pass
```

Code 6-3 start_requests

make_requests_from_url(url)

该方法接受一个 URL 并返回用于爬取的 Request 对象。该方法在初始化 request 时被 start_requests()调用，也被用于转化 url 为 request。

默认未被复写(overridden)的情况下，该方法返回的 Request 对象中，parse()作为回调函数，dont_filter 参数也被设置为开启。(详情参见 Request)。

parse(response)

当 response 没有指定回调函数时，该方法是 Scrapy 处理下载的 response 的默认方法。

parse 负责处理 response 并返回处理的数据以及(/或)跟进的 URL。 Spider 对其他的 Request 的回调函数也有相同的要求。

该方法及其他的 Request 回调函数必须返回一个包含 Request 及(或) Item 的可迭代的对象。

参数: response (Response) – 用于分析的 response

log(message[, level, component])

使用 scrapy.log.msg() 方法记录(log)message。log 中自动带上该 spider 的 name 属性。关于 Logging 我会在后面的爬虫小技巧课程中做详细讲解。

closed(reason)

当 spider 关闭时，该函数被调用。 该方法提供了一个替代调用 signals.connect()来监听 spider_closed 信号的快捷方式。

6.2.2 Spider 样例

让我们来看一个例子，如图 6-4 所示。

```
import scrapy
class ExampleSpider(scrapy.Spider):
    name = 'example'
    allowed_domains = ['example.com']
    start_urls = [
        'http://example.com/1.html',
        'http://example.com/2.html',
        'http://example.com/3.html',
                    ]
    def parse(self, response):
        self.log('A response from %s just arrived!' % response.url)
```

Code 6-4 Spider 样例

另一个在单个回调函数中返回多个 Request 以及 Item 的例子，如 Code 6-5 所示。

```
import scrapy
from myproject.items import MyItem
class ExampleSpider(scrapy.Spider):
    name = 'example'
```

```
    allowed_domains = ['example.com']
    start_urls = [
        'http://example.com/1.html',
        'http://example.com/2.html',
        'http://example.com/3.html',
                    ]
    def parse(self, response):
        sel=scrapy.Selector(response)
        for h3 in response.xpath('//h3').extract():
            yield   MyItem(title=h3)
        for url in response.xpath('//a/@href').extract():
            yield scrapy.Request(url,callback=self.parse)
```

Code 6-5 Spider 样例

6.2.3 CrawlSpider

class scrapy.contrib.spiders.CrawlSpider

爬取一般网站常用的 spider。其定义了一些规则(rule)来提供跟进 link 的方便的机制。 也许该 spider 并不是完全适合您的特定网站或项目，但其对很多情况都使用。因此您可以以其为起点，根据需求修改部分方法。当然您也可以实现自己的 spider。

除了从 Spider 继承过来的(您必须提供的)属性外，其提供了一个新的属性:

rules

一个包含一个(或多个) Rule 对象的集合(list)。 每个 Rule 对爬取网站的动作定义了特定表现。Rule 对象在下边会介绍。 如果多个 rule 匹配了相同的链接，则根据他们在本属性中被定义的顺序，第一个会被使用。

该 spider 也提供了一个可复写(overrideable)的方法:

parse_start_url(response)

当 start_url 的请求返回时，该方法被调用。 该方法分析最初的返回值并必须返回一个 Item 对象或者 一个 Request 对象或者 一个可迭代的包含二者对象。

6.2.4 爬取规则(Crawling rules)

class scrapy.contrib.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。其定义了如何从爬取到的页面提取链接。

callback 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。从 link_extractor 中每获取到链接时将会调用该函数。该回调函数接受一个 response 作为其第一个参数，并返回一个包含 Item 以及(或) Request 对象(或者这两者的子类)的列表(list)。（注意：当编写爬虫规则时，请避免使用 parse 作为回调函数。由于 CrawlSpider 使用 parse 方法来实现其逻辑，如果 您覆盖了 parse 方法，crawl spider 将会运行失败。）

cb_kwargs 包含传递给回调函数的参数(keyword argument)的字典。

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。

process_links 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。 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 是一个 callable 或 string(该 spider 中同名的函数将会被调用)。 该规则提取到每个 request 时都会调用该函数。该函数必须返回一个 request 或者 None。 (用来过滤 request)

6.2.5 CrawlSpider 样例

接下来给出配合 rule 使用 CrawlSpider 的例子，如 Code 6-6 所示。

```
import scrapy
from scrapy.contrib.spiders import CrawlSpider,Rule
from scrapy.contrib.linkextractors import LinkExtractor
class MySpider(scrapy.Spider):
    name = 'example'
    allowed_domains = ['example.com']
    start_urls = ['http://example.com/']
    rules=(
        Rule(LinkExtractor(allow=('categroy\.php',),deny=('subsection\.php',))),
        Rule(LinkExtractor(allow=('Item\.php',)),callback='parse_item'),
    )
    def parse_item(self,response):
        self.log('Hi,this is an item page! %s'% response.url)
        item=scrapy.Item()
        item['id']=response.xpath('//td[@id="item_id"]/text()').re(r'ID:(\d+)')
        item['name'] = response.xpath('//td[@id="item_name"]/text()').extract()
        item['description'] = response.xpath('//td[@id="item_description"]/text()').extract()
        return item
```

Code 6-6 CrawlSpider 样例

该 spider 将从 example.com 的首页开始爬取 ,获取 category 以及 item 的链接并对后者使用 parse_item 方法。当 item 获得返回(response)时，将使用 XPath 处理 HTML 并生成一些数据填入 Item 中。

6.3 Spiders 扩展

6.3.1 XMLFeedSpider

class scrapy.contrib.spiders.XMLFeedSpider

XMLFeedSpider 被设计用于通过迭代各个节点来分析 XML 源(XML feed)。迭代器可以从 iternodes，xml，html 选择。鉴于 xml 以及 html 迭代器需要先读取所有 DOM 再分析而引起的性能问题，一般还是推荐使用 iternodes。不过使用 html 作为迭代器能有效应对错误的 XML。

您必须定义下列类属性来设置迭代器以及标签名(tag name):

Iterator

用于确定使用哪个迭代器的 string。可选项有:

- 'iternodes' -一个高性能的基于正则表达式的迭代器
- 'html' -使用 Selector 的迭代器。 需要注意的是该迭代器使用 DOM 进行分析，其需要将所有的 DOM 载入内存， 当数据量大的时候会产生问题。
- 'xml' -使用 Selector 的迭代器。 需要注意的是该迭代器使用 DOM 进行分析，其需要将所有的 DOM 载入内存，当数据量大的时候会产生问题。

默认值为 iternodes 。

Itertag

一个包含开始迭代的节点名的 string。例如，itertag = 'product'

namespaces

一个由 (prefix, url) 元组(tuple)所组成的 list。 其定义了在该文档中会被 spider 处理的可用的 namespace。 prefix 及 uri 会被自动调用 register_namespace()生成 namespace。

您可以通过在 itertag 属性中制定节点的 namespace。代码如 Code 6-7 所示。

```
#coding:utf8
"""
@author:xiaochouyu
"""
import scrapy
```

```
from scrapy.contrib.spiders import XMLFeedSpider
class YourSpider(XMLFeedSpider):
    namespaces = [('n','http://www.sitemaps.org/schemas/sitemap/0.9')]
    itertag = 'n:url'
```

Code 6-7 XMLFeedSpider

除了这些新的属性之外，该 spider 也有以下可以覆盖(overrideable)的方法:

adapt_response(response)

该方法在 spider 分析 response 前被调用。您可以在 response 被分析之前使用该函数来修改内容(body)。该方法接受一个 response 并返回一个 response(可以相同也可以不同)。

parse_node(response, selector)

当节点符合提供的标签名时(itertag)该方法被调用。 接收到的 response 以及相应的 Selector 作为参数传递给该方法。该方法返回一个 Item 对象或者 Request 对象 或者一个包含二者的可迭代对象(iterable)。

process_results(response, results)

当 spider 返回结果(item 或 request)时该方法被调用。设定该方法的目的是在结果返回给框架核心(framework core)之前做最后的处理，例如设定 item 的 ID。其接受一个结果的列表(list of results)及对应的 response。其结果必须返回一个结果的列表(list of results)(包含 Item 或者 Request 对象)。

6.3.2 XMLFeedSpider 样例

该 spider 十分易用。下边是其中一个例子，代码如 Code 6-8 所示。

```
from scrapy import log
from scrapy.contrib.spiders import XMLFeedSpider
from myproject.items import TestItem
class MySpider(XMLFeedSpider):
    name='example.com'
    allowed_domains=['example.com']
    start_urls=['http://www.example.com/feed.xml']
    iterator = 'iternodes'
    itertag = 'item'
    def parse_node(self,response,node):
        log.msg('Hi,this is a <%s> node!:%s'%(self.itertag,''.join(node.extract())))
        item=TestItem()
        item['id']=node.xpath('@id').extract()
```

```
        item['name']=node.xpath('name').extract()
        return item
```

Code 6-8 XMLFeedSpider 例子

简单来说，我们在这里创建了一个 spider，从给定的 start_urls 中下载 feed，并迭代 feed 中每个 item 标签，输出，并在 Item 中存储有些随机数据。

6.3.3 CSVFeedSpider

class scrapy.contrib.spiders.CSVFeedSpider

该 spider 除了其按行遍历而不是节点之外其他和 XMLFeedSpider 十分类似。 而其在每次迭代时调用的是 parse_row()。

delimiter

在 CSV 文件中用于区分字段的分隔符。类型为 string。 默认为 ',' (逗号)。

quotechar

填写一个字符串，表示 CSV 文件中的每个字段的特征，默认为” “（引号）。

headers

在 CSV 文件中包含的用来提取字段的行的列表。参考下边的例子。

parse_row(response, row)

该方法接收一个 response 对象及一个以提供或检测出来的 header 为键的字典(代表每行)。 该 spider 中，您也可以覆盖 adapt_response 及 process_results 方法来进行预处理(pre-processing)及后(post-processing)处理。

6.3.4 CSVFeedSpider 样例

下面的例子和之前的例子很像，但使用了 CSVFeedSpider，如 Code 6-9 所示。

```
#coding:utf8
"""
@author:xiaochouyu
"""
from scrapy import log
from scrapy.contrib.spiders import XMLFeedSpider
from scrapy.contrib.spiders import CSVFeedSpider
from myproject.items import TestItem
class MySpider(XMLFeedSpider):
    name='example.com'
```

```
    allowed_domains=['example.com']
    start_urls=['http://www.example.com/feed.csv']
    delimiter=';'
    quotechar="'"
    headers=['id','name','sescription']
    def parse_row(self,response,row):
        log.msg('Hi,this is a row !:%r' % row)
        item=TestItem()
        item['id']=row.xpath('@id')
        item['name']=row.xpath('name')
        item['description']=row.xpath('description')
        return item
```

Code 6-9 CSVFeed 例子

6.3.5 SitemapSpiderss

class scrapy.contrib.spiders.SitemapSpider

SitemapSpider 使您爬取网站时可以通过 Sitemaps 来发现爬取的 URL。

其支持嵌套的 sitemap，并能从 robots.txt 中获取 sitemap 的 url。

sitemap_urls

包含您要爬取的 url 的 sitemap 的 url 列表(list)。您也可以指定为一个 robots.txt，spider 会从中分析并提取 url。

sitemap_rules

一个包含 (regex, callback) 元组的列表(list):

- regex 是一个用于匹配从 sitemap 提供的 url 的正则表达式。regex 可以是一个字符串或者编译的正则对象(compiled regex object)。
- callback 指定了匹配正则表达式的 url 的处理函数。callback 可以是一个字符串(spider 中方法的名字)或者是 callable。

代码例如：

```
sitemap_rules = [('/product/', 'parse_product')]
```

规则按顺序进行匹配，之后第一个匹配才会被应用。

如果您忽略该属性，sitemap 中发现的所有 url 将会被 parse 函数处理。

sitemap_follow

一个用于匹配要跟进的 sitemap 的正则表达式的列表(list)。其仅仅被应用在 使用 Sitemap index files 来指向其他 sitemap 文件的站点。

默认情况下所有的 sitemap 都会被跟进。

sitemap_alternate_links

指定当一个 url 有可选的链接时，是否跟进。 有些非英文网站会在一个 url 块内提供其他语言的网站链接。

例如:

```
<url>
    <loc>http://example.com/</loc>
    <xhtml:link rel="alternate" hreflang="de" href="http://example.com/de"/>
</url>
```

当 sitemap_alternate_links 设置时，两个 URL 都会被获取。 当 sitemap_alternate_links 关闭时，只有 http://example.com/ 会被获取。

默认 sitemap_alternate_links 关闭。

6.3.6 SitemapSpider 样例

简单的例子:使用 parse 处理通过 sitemap 发现的所有 url，代码如图 6-10 所示。

```
from scrapy.contrib.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls=[ ‘http://www.example.com/sitemap.xml’ ]
    def parse(self,response):
        pass
```

Code 6-10 SitemapSpider 例子

用特定的函数处理某些 url，其他的使用另外的 callback，代码如图 6-11 所示。

```
from scrapy.contrib.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']
    sitemap_rules = [
        ('/product/','parse_product'),
        ('/category/','parse_category')
    ]
    def parse_product(self,response):
```

```
        pass
    def parse_category(selfself,response):
        pass
```

Code 6-11 SitemapSpider 例子

跟进 robots.txt 文件定义的 sitemap 并只跟进包含有..sitemap_shop 的 url，代码如 Code 6-12 所示。

```
from scrapy.contrib.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/','parse_shop'),
    ]
    sitemap_follow = ['/sitemap_shops']
    def parse_shop(self,response):
        pass
```

Code 6-12 SitemapSpider 例子

在 SitemapSpider 中使用其他 url，代码如图 6-13 所示。

```
import scrapy
from scrapy.contrib.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/','parse_shop'),
    ]
    other_urls=['http://www.example.com/about']
    def start_requests(self):
        requests=list(super(MySpider,self).start_requests())
        requests+=[scrapy.Request(x,self.parse_other) for x in self.other_urls]
    def parse_shop(self,response):
        pass
    def parse_other(self,response):
        pass
```

Code 6-13 SitemapSpider 例子

6.3.7 在 spider 中启动 shell 来查看 response

有时您想在 spider 的某个位置中查看被处理的 response，以确认您期望的 response 到达特定位置。

这可以通过 scrapy.shell.inspect_response 函数来实现。

以下是如何在 spider 中调用该函数的例子，如 Code 6-14 所示。

```
import scrapy
class MySpider(scrapy.Spider):
    name="myspider"
    start_urls=[
        "http://example.com",
        "http://example.org",
        "http://example.net"
    ]
    def parse(self,response):
        if ".org" in response.url:
            from scrapy.shell import inspect_response
            inspect_response(response,self)
```

Code 6-14 在 spider 中启动 shell 来查看 response

接着测试提取代码，如 6-16 所示。

```
response.url
  'http://example.org'
response.xpath( '//h1[@class=" fn" ]' )
[]
```

Code 6-16 测试提取代码

咦，看来是没有。那么为何会这样呢？您可以在浏览器里查看 response 的结果，判断是否是您期望的结果，如 Code 6-17 所示。

```
view(response)
True
```

Code 6-17 查看 response

最后您可以点击 Ctrl-D(Windows 下 Ctrl-Z)来退出终端，恢复爬取。

注意: 由于该终端屏蔽了 Scrapy 引擎，您在这个终端中不能使用 fetch 快捷命令(shortcut)。 当您离开终端时，spider 会从其停下的地方恢复爬取，正如上面显示的那样。

第 7 章 选择器

本章工作任务

- 任务 1：为什么使用 Scrapy 的选择器？
- 任务 2：如何使用选择器？
- 任务 3：选择器的扩展

本章技能目标及重难点

编号	技能点描述	级别
1	为什么要使用选择器？	★
2	如何使用选择器？	★★★
3	选择器的扩展	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 的选择器，需要同学们学会使用选择器及相关扩展的选择器应用。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

7.1 为什么使用 Scrapy 的选择器？

7.1.1 为什么必须学会使用 Selector

当抓取网页时，你做的最常见的任务是从 HTML 源码中提取数据。现有的一些库可以达到这个目的：

- BeautifulSoup 是在程序员间非常流行的网页分析库，它基于 HTML 代码的结构来构造一个 Python 对象， 对不良标记的处理也非常合理，但它有一个缺点：慢。
- lxml 是一个基于 ElementTree(不是 Python 标准库的一部分)的 python 化的 XML 解析库(也可以解析 HTML)。

7.1.2 什么是 Scrapy 选择器

Scrapy 提取数据有自己的一套机制。它们被称作选择器(seletors)，因为他们通过特定的 XPath 或者 CSS 表达式来“选择” HTML 文件中的某个部分。

XPath 是一门用来在 XML 文件中选择节点的语言，也可以用在 HTML 上。CSS 是一门将 HTML 文档样式化的语言。选择器由它定义，并与特定的 HTML 元素的样式相关连。

Scrapy 选择器构建于 lxml 库之上，这意味着它们在速度和解析准确性上非常相似。

7.2 如何使用选择器？

7.2.1 构造选择器(selectors)

Scrapy selector 是以 文字(text) 或 TextResponse 构造的 Selector 实例。 其根据输入的类型自动选择最优的分析方法(XML vs HTML)，以文字构造，如 Code7-1 所示。

```
>>> from scrapy.selector import Selector
>>> body = '<html><body><span>good</span></body><html/>'
>>> Selector(text = body).xpath('//span/text()').extract()
[u'good']
```

Code 7-1 构以文字构造

以 response 构造,如图 7-2 所示。

```
>>> from scrapy.http import HtmlResponse
>>> response = HtmlResponse(url = 'http://example.com',body=body)
>>> Selector(response=response).xpath('//span/text()').extract()
[u'good']
```

Code 7-2 以 response 构造

为了方便起见 ,response 对象以.selector 属性提供了一个 selector ,您可以随时使用该快捷方法，如 Code7-3 所示。

```
>>> from scrapy.http import HtmlResponse
>>> response = HtmlResponse(url = 'http://example.com',body=body)
>>> response.selector.xpath('//span/text()').extract()
[u'good']
```

Code 7-3 .selector 属性提供了一个 selector

7.2.2 使用选择器(selectors)

我们将使用 Scrapy shell (提供交互测试)和位于 Scrapy 文档服务器的一个样例页面，来解释如何使用选择器：

https://doc.scrapy.org/en/latest/_static/selectors-sample1.html

这里是它的 HTML 源码，如 Code7-4 所示。

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <base href="http://example.com/">
    <title>Example website</title>
</head>
<body>
    <div id="images">
        <a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>
        <a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>
        <a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>
        <a href="image4.html">Name: My image 4 <br><img src="image4_thumb.jpg"></a>
        <a href="image5.html">Name: My image 5 <br><img src="image5_thumb.jpg"></a>
    </div>
</body>
```

```
</html>
```

Code 7-4 .selector 属性提供了一个 selector

首先，我们打开 shell，命令为：scrapy shell http://doc.scrapy.org/en/latest _static/selectors-sample1.html。

接着，当 shell 载入后，您将获得名为 response 的 shell 变量，其为响应的 response，并且在其 response.selector 属性上绑定了一个 selector。

因为我们处理的是 HTML，选择器将自动使用 HTML 语法分析。

那么，通过查看 HTML code 该页面的源码，我们构建一个 XPath 来选择 title 标签内的文字，如 Code7-5 所示。

```
In [1]: response.selector.xpath('//title/text()')
Out[1]: [<Selector xpath='//title/text()' data=u'Example website'>]
```

Code 7-5 xpath 应用

由于在 response 中使用 XPath、CSS 查询十分普遍，因此，Scrapy 提供了两个实用的快捷方式: response.xpath()及 response.css()，如 Code7-6 所示。

```
In [2]: response.xpath('//title/text()')
Out[2]: [<Selector xpath='//title/text()' data=u'Example website'>]


In [3]: response.css('title:text')
Out[3]: [<Selector xpath=u'descendant-or-sel::title/text()' data=u'Example website'>]
```

Code 7-6 css 应用

如你所见，.xpath()及.css()方法返回一个类 SelectorList 的实例，它是一个新选择器的列表。这个 API 可以用来快速的提取嵌套数据。

为了提取真实的原文数据，你需要调用.extract() 方法如下，如 Code7-7 所示。

```
In [4]: response.xpath('//title/text()').extract()
Out[4]: [u'Example website'>]
```

Code 7-6 提取真实的原文数据

注意 CSS 选择器可以使用 CSS3 伪元素(pseudo-elements)来选择文字或者属性节点，如 Code7-7 所示。

```
In [5]: response.css('title::text').extract()
Out[5]: [u'Example website'>]
```

Code 7-7 css 选择器

现在我们将得到根 URL(base URL)和一些图片链接，如 Code7-8 所示。

```
In [6]: response.xpath('//base/@href').extract()
Out[6]: [u'http://example.com/']

In [7]: response.css('base::attr(href)').extract()
Out[7]: [u'http://example.com/']

In [8]: response.xpath('//a[contains(@href,"images")]/@href').extract()
Out[8]:
[u'image1.html',
u'image2.html',
u'image3.html',
u'image4.html',
u'image5.html',]

In [9]: response.css('a[href*=image]::attr(href)').extract()
Out[9]:
[u'image1.html',
u'image2.html',
u'image3.html',
u'image4.html',
u'image5.html',]

In [10]: response.xpath('//a[contains(@href,"image")]/img/@src').extract()
Out[10]:
[u'image1_thumb.jpg',
u'image2_thumb.jpg',
u'image3_thumb.jpg',
```

```
u'image4_thumb.jpg',
u'image5_thumb.jpg',]

In [11]: response.css('a[href*=image] img::attr(src)').extract()
Out[11]:
[u'image1_thumb.jpg',
u'image2_thumb.jpg',
u'image3_thumb.jpg',
u'image4_thumb.jpg',
u'image5_thumb.jpg',]
```

Code 7-8 根 URL(base URL)和一些图片链接

7.2.3 嵌套选择器(selectors)

选择器方法(.xpath() or .css())返回相同类型的选择器列表，因此你也可以对这些选择器调用选择器方法。下面是一个例子，如 Code7-9 所示。

```
In[12]: links = response.xpath('//a[contains(@href, "images")]')
In[13]: links.extract()
[u'<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>,
u'<a href="image2.html">Name: My image 2 <br><img src="image2_thumb.jpg"></a>,
u'<a href="image3.html">Name: My image 3 <br><img src="image3_thumb.jpg"></a>,
u'<a href="image4.html">Name: My image 4 <br><img src="image4_thumb.jpg"></a>,
u'<a href="image5.html">Name: My image 5 <br><img src="image5_thumb.jpg"></a>]

In[14]: for index, link in enumerate(links):
   ...:         args = (index. link.xpath('@href').extract(), link.xpath('img/@src').extract())
   ...:         print 'Link number %d points to url %s and image %s' % args
   ...:
Link number 0 points to url [u'image1.html'] and images[u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and images[u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and images[u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and images[u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and images[u'image5_thumb.jpg']
```

Code 7-9 根 URL(base URL)和一些图片链接

7.2.4 使用相对 XPaths

记住如果你使用嵌套的选择器，并使用起始为/的 XPath，那么该 XPath 将对文档使用绝对路径，而且对于你调用的 Selector 不是相对路径。

比如，假设你想提取在 元素中的所有<p>元素。首先，你将先得到所有的 元素。

开始时，你可能会尝试使用下面的错误的方法，因为它其实是从整篇文档中，而不仅仅是从那些 元素内部提取所有的<p>元素，如 Code7-10 所示。

```
In[15]: divs = response.xpath('//div')


In[16]: for p in divs.xpath('//p'): # this is wrong - gets all <p> from the whole document.
   ...:             print p.extract()
```

Code 7-10 错误的用法

下面是比较合适的处理方法(注意.//p XPath 的点前缀)，如 Code7-11 所示。

```
In[17]: fro p in divs.xpath('.//p'): # extracts all <p> inside
   ...: ...         print p.extract()
   ...:
```

Code 7-11 合适的处理方法

另一种常见的情况将是提取所有直系<p>的结果，如 Code7-12 所示。

```
In[18]: for p in divs.xpath('p'):
   ...: ...         print p.extract()
```

Code 7-12 另一种常见的情况

7.2.5 使用 EXSLT 扩展

因建于 lxml 之上，Scrapy 选择器也支持一些 EXSLT 扩展，可以在 XPath 表达式中使用这些预先制定的命名空间，如表 7-1 所示。

前缀	命名空间	用途
re	http://exslt.org/regular-expressions	正则表达式
set	http://exslt.org/sets	集合操作

正则表达式

例如在 XPath 的 starts-with()或 contains()无法满足需求时，test()函数可以非常有用。

例如在列表中选择有”class”元素且结尾为一个数字的链接，如图 7-13 所示。

```
In [19]: from scrapy import Selector

In [20]: doc = """
   ...: ... <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>
   ...: ... """

In [21]: sel = Selector(text=doc, type="html")

In [22]: sel.xpath('//li//@href').extract()
Out[22]: [u'link1.htlm', u'link2.html', u'link3.html', u'link4.html', u'link5.html']

In [23]: sel.xpath('//li[re:test(@class, "item-\d$")]//@href').extract()
Out[23]: [u'link1.html', u'link2.html', u'link4.html', u'link5.html']
```

Code 7-13 正则表达式

C 语言库 libxslt 不原生支持 EXSLT 正则表达式 ,因此 lxml 在实现时使用了 Python re 模块的钩子。 因此，在 XPath 表达式中使用 regexp 函数可能会牺牲少量的性能。

集合操作

集合操作可以方便地用于在提取文字元素前从文档树中去除一些部分。

例如使用 itemscopes 组和对应的 itemprops 来提取微数据(来自 http://schema.org/Product 的样本内容)，如 Code7-14 所示。

```
from scrapy.selector import Selector
from scrapy.http import HtmlResponse
```

```
doc = """
    <div itemscope itemtype="http://schema.org/Product">
        <span itemprop="name">Kenmore White 17" Microwave</span>
        <img src="kenmore-microwave-17in.jpg" alt='Kenmore 17" Microwave'
        <div itemprop="aggregateRating"  itemscope
itemtype="http://schema.org/AggregateRating">
            Rated <span itemprop="ratingValue">3.5</span>/5based on <span
itemprop="reviewCount">l1</span> customer reviews
        </div>
        <div itemprop="offers" itemscope itemtype="http://schema.org/0ffer" >
            <span itemprop="price">$55.00</span>
            <link itemprop="availability" href="http://schema.org/InStock" />In stock
        </div>

        Product description:
        <span itemprop="description">
            0.7 cubic feet countertop microwave.Has six preset cooking categories and
convenience features likeadd-A-Minute and Child Lock.
        </span>

        Customer reviews

        <div itemprop-"review" itemscope itemtype="http://schema.org/Review>
            <span itemprop="name">Not a happy camper</span>
            by <span itemprop="author">Ellie</span>,
            <meta itemprop="datePublished content = "2011-04-01">Apri1 1, 2011
            <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
                <meta itemprop="worstRating" content = "1">
                <span itemprop="ratingValue">1<span>/
```

```
                <span itemprop="bestRating">5</span>stars
            </div>
            <span itemprop="description">
                The 1amp burned out and now I have to replaceit: </span>
            </div>
            <div itemprop="review" itemscope itemtype="http://schema.org/Review">
                <span itemprop="name">Value purchases</span>
                -by <span itemprop="author">Lucas</span>
            <meta itemprop="datePublished" content="2011-03-25>March 25，2011
            <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
                <meta itemprop="worstRating" content = "1"/>
                <span itemprop="ratingValue">4</span>/
                <span itemprop="bestRating">5</span>stars
            </div>
            <span itemprop="description">Great microwave for the price.It is small and
            fits in my apartment.</span>
        </div>
    </div>
    """
for scope in Selector(text-doc).xpath('//div[@itemscope]'):
    print "current scope:", scope.xpath('@itemtype').extract()
    props = scope.xpath('''
                set:difference(./descendant::*/@itemprop,
                .//"[@itemscope]/*/@itemprop)''')
    print "    properties:", props.extract()
    print
```

Code 7-14 集合操作

在这里，我们首先在 itemscope 元素上迭代，对于其中的每一个元素，我们寻找所有的 itemprops 元素，并排除那些本身在另一个 itemscope 内的元素。

√7.3 选择器的扩展

7.3.1 内建选择器的参考

class scrapy.selector.Selector(response=None, text=None, type=None)

Selector 的实例是对选择某些内容响应的封装。

response 是 HtmlResponse 或 XmlResponse 的一个对象，将被用来选择和提取数据。

text 是在 response 不可用时的一个 unicode 字符串或 utf-8 编码的文字。将 text 和 response 一起使用是未定义行为。

type 定义了选择器类型，可以是 "html"，"xml" or None (默认).

如果 type 是 None ，选择器会根据 response 类型(参见下面)自动选择最佳的类型，或者在和 text 一起使用时，默认为 "html" 。

如果 type 是 None ，并传递了一个 response ，选择器类型将从 response 类型中推导如下：

- "html" for HtmlResponse type
- "xml" for XmlResponse type
- "html" for anything else

其他情况下，如果设定了 type ，选择器类型将被强制设定，而不进行检测。

xpath(query)

寻找可以匹配 xpath query 的节点，并返回 SelectorList 的一个实例结果，单一化其所有元素。列表元素也实现了 Selector 的接口。

query 是包含 XPATH 查询请求的字符串。

为了方便起见，该方法也可以通过 response.xpath()调用。

css(query)

应用给定的 CSS 选择器，返回 SelectorList 的一个实例。

query 是一个包含 CSS 选择器的字符串。

在后台，通过 cssselect 库和运行 .xpath() 方法，CSS 查询会被转换为 XPath 查询。

为了方便起见，该方法也可以通过 response.css() 调用.

extract()

串行化并将匹配到的节点返回一个 unicode 字符串列表。 结尾是编码内容的百分比。

re(regex)

应用给定的 regex，并返回匹配到的 unicode 字符串列表。

regex 可以是一个已编译的正则表达式，也可以是一个将被 re.compile(regex)编译为正则表达式的字符串。

register_namespace(prefix, uri)

注册给定的命名空间，其将在 Selector 中使用。 不注册命名空间，你将无法从非标准命名空间中选择或提取数据。

remove_namespaces()

移除所有的命名空间，允许使用少量的命名空间 xpaths 遍历文档。

__nonzero__()

如果选择了任意的真实文档，将返回 True ，否则返回 False 。 也就是说， Selector 的布尔值是通过它选择的内容确定的。

7.3.2 SelectorList 对象

class scrapy.selector.SelectorList

SelectorList 类是内建 list 类的子类，提供了一些额外的方法。

xpath(query)

对列表中的每个元素调用.xpath()方法，返回结果为另一个单一化的 SelectorList。

query 和 Selector.xpath()中的参数相同。

css(query)

对列表中的各个元素调用.css() 方法，返回结果为另一个单一化的 SelectorList。

query 和 Selector.css() 中的参数相同。

extract()

对列表中的各个元素调用.extract()方法，返回结果为单一化的 unicode 字符串列表。

re()

对列表中的各个元素调用 .re() 方法，返回结果为单一化的 unicode 字符串列表。

__nonzero__()

列表非空则返回 True，否则返回 False。

7.3.3 在 HTML 响应上的选择器样例

这里是一些 Selector 的样例，用来说明一些概念。在所有的例子中，我们假设已经有一个通过 HtmlResponse 对象实例化的 Selector，如下:

sel = Selector(html_response)

从 HTML 响应主体中提取所有的<h1>元素，返回:class:Selector 对象(即 SelectorList 的一个对

象)的列表:

sel.xpath("//h1")

从 HTML 响应主体上提取所有<h1>元素的文字，返回一个 unicode 字符串的列表:

sel.xpath("//h1").extract() # this includes the h1 tag

sel.xpath("//h1/text()").extract() # this excludes the h1 tag

在所有<p>标签上迭代，打印它们的类属性:

```
for node in sel.xpath("//p"):
    print node.xpath("@class").extract()
```

7.3.4 在 XML 响应上的选择器样例

这里是一些样例，用来说明一些概念。在两个例子中，我们假设已经有一个通过 XmlResponse 对象实例化的 Selector ，如下:

sel = Selector(xml_response)

从 XML 响应主体中选择所有的 元素，返回 Selector 对象(即 SelectorList 对象)的列表:

sel.xpath("//product")

从 Google Base XML feed 中提取所有的价钱，这需要注册一个命名空间:

sel.register_namespace("g", "http://base.google.com/ns/1.0")

sel.xpath("//g:price").extract()

7.3.5 移除命名空间

在处理爬虫项目时，完全去掉命名空间而仅仅处理元素名字，写更多简单/实用的 XPath 会方便很多。你可以为此使用 Selector.remove_namespaces() 方法。

让我们来看一个例子，以 Github 博客的 atom 订阅来解释这个情况。

首先，我们使用想爬取的 url 来打开 shell:

scrapy shell https://github.com/blog.atom

一旦进入 shell，我们可以尝试选择所有的 <link> 对象，可以看到没有结果(因为 Atom XML 命名空间混淆了这些节点)，如 Code7-15 所示。

```
In [1]: response.xpath("//link")
Out[1]: []
```

Code 7-15 移除命名空间

但一旦我们调用 Selector.remove_namespaces()方法，所有的节点都可以直接通过他们的名字来

访问，如 Code7-16 所示。

```
In [2]: response.selector.remove_namespaces()


In [3]: response.xpath("//link")
Out[3]:
[<Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>,
 <Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom'>]
```

Code 7-16 remove_namespaces()

如果你对为什么命名空间移除操作并不总是被调用，而需要手动调用有疑惑。这是因为存在如下两个原因，按照相关顺序如下：

1. 移除命名空间需要迭代并修改文件的所有节点，而这对于 Scrapy 爬取的所有文档操作需要一定的性能消耗
2. 会存在这样的情况，确实需要使用命名空间，但有些元素的名字与命名空间冲突。尽管这些情况非常少见。

7.4 总结

本章节解释了选择器如何工作，并描述了相应的 API。不同于 lxml API 的臃肿，该 API 短小而简洁。这是因为 lxml 库除了用来选择标记化文档外，还可以用到许多任务上。

第 8 章 管道

本章工作任务

- 任务 1：管道的作用
- 任务 2：私人定制自己的管道
- 任务 3：常用管道操作

本章技能目标及重难点

编号	技能点描述	级别
1	管道的作用	★★
2	私人定制自己的管道	★★★
3	常用管道操作	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 爬虫的管道机制，学完同学们就可以私人定制自己的管道。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

8.1 管道的作用

当 Item 在 Spider 中被收集之后，它将会被传递到 Item Pipeline，一些组件会按照一定的顺序执行对 Item 的处理。

每个 item pipeline 组件(有时称之为“Item Pipeline”)是实现了简单方法的 Python 类。他们接收到 Item 并通过它执行一些行为，同时也决定此 Item 是否继续通过 pipeline，或是被丢弃而不再进行处理。

以下是 item pipeline 的一些典型应用：

- 清理 HTML 数据
- 验证爬取的数据(检查 item 包含某些字段)
- 查重(并丢弃)
- 将爬取结果保存到数据库中

8.2 私人订制自己的管道

编写你自己的 item pipeline 很简单，每个 item pipeline 组件是一个独立的 Python 类，同时必须实现以下方法:

- process_item(self, item, spider)

此外，也可以实现以下方法:

- open_spider(self, spider)
- close_spider(spider)
- from_crawler(cls, crawler)

8.2.1 process_item(self, item, spider)

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 (或任何继承类)对象，或是抛出 DropItem 异常，被丢弃的 item 将不会被之后的 pipeline 组件所处理。

参数：

- item (Item 对象) – 被爬取的 item
- spider (Spider 对象) – 爬取该 item 的 spider

8.2.2 open_spider(self, spider)

当 spider 被开启时，这个方法被调用。

参数：

- spider (Spider 对象) – 被开启的 spider

8.2.3 close_spider(self, spider)

当 spider 被关闭时，这个方法被调用。

参数：

- spider (Spider 对象) – 被开启的 spider

8.2.4 from_spider(self, spider)

如果爬虫存在，这类就会从爬虫创建管道实例。它必须返回管道的新实例。爬虫对象提供存取所有 Scrapy 核心部件设置和信号，这个方法对于管道而言，允许他们和链接 scrapy 的功能。

参数：

- spider (Spider 对象) – 被开启的 spider

8.3 Item pipeline 样例

8.3.1 验证价格，同时丢弃没有价格的 item

我们如果爬取一个进口商品网站，我们可能需要计算进口商品的入关税，这样我们就需要对数据进行处理。

让我们来看一下以下这个假设的 pipeline，它为那些不含税(price_excludes_vat 属性)的 item 调整了 price 属性，同时丢弃了那些没有价格的 item，代码如 8-1 所示。

```
from scrapy.exceptions import DropItem
class PricePipeline(object):
    vat_factor=1.15
    def process_item(self,item,spider):
        if item['price']:
            if item['price_excludes_vat']:
                item['price']=item['price']*self.vat_factor
            return item
        else:
            raise DropItem('Missing price in %s'% item)
```

Code 8-1 验证价格，同时丢弃没有价格的 item

8.3.2 将 item 写入 JSON 文件

我们为了更好保存数据，并且有些时候为了测试爬虫爬取数据的有效性，我们可以把数据保存为 JSON 格式的文件。

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.jl 文件，每行包含一个序列化为 JSON 格式的 item，如 Code 8-2 所示。

```
import Json
class JsonWritePipeline(object):
    def __init__(self):
        self.file=open('item.j1','wb')
    def process_item(self,item,spider):
        line=Json.dumps(dict(item))+"\n"
        self.file.write(line)
        return item
```

Code 8-2 将 item 写入 JSON 文件

JsonWriterPipeline 的目的只是为了介绍怎样编写 item pipeline ,如果你想要将所有爬取的 item 都保存到同一个 JSON 文件，你需要使用 Feed exports，这个部分我们将会在爬虫小技巧中进行讲解。

8.3.3 将 item 写入 MongoDb

我们经常会把爬取下来的数据，存储到数据库中。

在这个例子中我们将使用到 pymongo 来操作 MongoDB。MongoDB 的地址和数据库名称是 Scrapy 设置文件中设置了，我们直接调用，以 MongoDB 命名的集合。

这个例子主要是说明如何使用 from_crawler()方法来做清理资源，如 Code 8-3 所示。

```
import Pymongo
class MongoPipeline(object):
    def __init__(self,mongo_uri,mongo_db):
        self.mongo_uri=mongo_uri
        self.mongo_db=mongo_db
    @classmethod
    def from_crawler(cls,crawler):
        return cls(
            mongo_uri=crawler.settings.get('MONGO_URI'),
            mongo_db=crawler.settings.get('MONGO_DATABASE','items')
```

```
        )
    def open_spider(self,spider):
        self.client=Pymongo.MongoClient(self.mongo_uri)
        self.db=self.client[self.mongo_db]
    def close_spider(self,spider):
        self.client.close()
    def process_item(self,item,spider):
        collection_name=item.__class__.__name__
        self.db[collection_name].insert(dict(item))
        return item
```

Code 8-3 将 item 写入 MongoDb

8.3.4 去重

一个用于去重的过滤器，丢弃那些已经被处理过的 item。让我们假设我们的 item 有一个唯一的 id，但是我们 spider 返回的多个 item 中包含有相同的 id，代码如 Code 8-4 所示。

```
from scrapy.exceptions import DropItem
class DuplicatesPipeline(object):
    def __init__(self):
        self.ids_seen=set()
    def process_item(self,item,spider):
        if item['id'] in self.ids_seen:
            raise DropItem('Duplicate item found:%s' % item)
        else:
            self.ids_seen.add(['id'])
            return item
```

Code 8-4 去重

8.3.5 启用一个 Item Pipeline 组件

为了启用一个 Item Pipeline 组件，你必须将它的类添加到 ITEM_PIPELINES 配置，就像下面这个例子，写到 settings 文件中，如图 8-5 所示。

```
ITEM_PIPELINES = {
    # 'myproject.pipelines.MyprojectPipeline': 300,
     'myproject.pipelines.PricePipeline': 300,
     'myproject.pipelines.JsonWriterPipeline': 300,
}
```

图 8-5 启用一个 Item Pipeline 组件

前面的键表示的是引入的自定义管道的路径。

后面的数值表示的是自定义的管道执行的先后顺序。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。

8.4 Feed exports

实现爬虫时最经常提到的需求就是能合适的保存爬取到的数据，或者说，生成一个带有爬取数据的”输出文件”(通常叫做”输出 feed”)，来供其他系统使用。

Scrapy 自带了 Feed 输出，并且支持多种序列化格式(serialization format)及存储方式(storage backends)。

8.4.1 序列化方式(Serialization formats)

feed 输出使用到了 Item exporters 。其自带支持的类型有:

- JSON
- JSON lines
- CSV
- XML

您也可以通过 FEED_EXPORTERS 设置扩展支持的属性。

JSON

- FEED_FORMAT: json
- 使用的 exporter: JsonItemExporter
- 大数据量情况下使用 JSON 请参见 这个警告

JSON lines

- FEED_FORMAT: jsonlines
- 使用的 exporter: JsonLinesItemExporter

CSV

- FEED_FORMAT: csv
- 使用的 exporter: CsvItemExporter

XML

- FEED_FORMAT: xml
- 使用的 exporter: XmlItemExporter

Pickle

- FEED_FORMAT: pickle
- 使用的 exporter: PickleItemExporter

Marshal

- FEED_FORMAT: marshal
- 使用的 exporter: MarshalItemExporter

8.4.2 存储(Storages)

使用 feed 输出时您可以通过使用 URI(通过 FEED_URI 设置) 来定义存储端。feed 输出支持 URI 方式支持的多种存储后端类型。

自带支持的存储后端有:

- 本地文件系统
- FTP
- S3 (需要 boto)
- 标准输出

有些存储后端会因所需的外部库未安装而不可用。例如，S3 只有在 boto 库安装的情况下才可使用。

存储 URI 也包含参数。当 feed 被创建时这些参数可以被覆盖:

- %(time)s - 当 feed 被创建时被 timestamp 覆盖
- %(name)s - 被 spider 的名字覆盖

其他命名的参数会被 spider 同名的属性所覆盖。例如， 当 feed 被创建时， %(site_id)s 将会被 spider.site_id 属性所覆盖。

下面用一些例子来说明:

存储在 FTP，每个 spider 一个目录:

ftp://user:password@ftp.example.com/scraping/feeds/%(name)s/%(time)s.json

存储在 S3，每一个 spider 一个目录:

s3://mybucket/scraping/feeds/%(name)s/%(time)s.json

8.4.3 存储端(Storage backends)

本地文件系统

将 feed 存储在本地系统。

- URI scheme: file
- URI 样例: file:///tmp/export.csv
- 需要的外部依赖库:none

注意: (只有)存储在本地文件系统时，您可以指定一个绝对路径 /tmp/export.csv 并忽略协议(scheme)。不过这仅仅只能在 Unix 系统中工作。

FTP

将 feed 存储在 FTP 服务器。

- URI scheme:ftp
- URI 样例:ftp://user:pass@ftp.example.com/path/to/export.csv
- 需要的外部依赖库:none

S3

将 feed 存储在 Amazon S3 。

- URI scheme: s3
- URI 样例:
  - s3://mybucket/path/to/export.csv
  - s3://aws_key:aws_secret@mybucket/path/to/export.csv
- 需要的外部依赖库: boto

您可以通过在 URI 中传递 user/pass 来完成 AWS 认证，或者也可以通过下列的设置来完成: AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY

标准输出

feed 输出到 Scrapy 进程的标准输出。

- URI scheme: stdout
- URI 样例: stdout:
- 需要的外部依赖库: none

8.4.4 设定(Settings)

这些是配置 feed 输出的设定:

- FEED_URI (必须)
- FEED_FORMAT
- FEED_STORAGES
- FEED_EXPORTERS
- FEED_STORE_EMPTY

FEED_URI

Default：None

输出 feed 的 URI。支持的 URI 协议请参见上面讲的存储端(Storage backends)的内容。

为了启用 feed 输出，该设定是必须的。

FEED_FORMAT

输出 feed 的序列化格式。可用的值请参见上面讲的序列化方式(Serialization formats)的内容。

FEED_STORE_EMPTY

Default：False

是否输出空 feed(没有 item 的 feed)。

FEED_STORAGES

Default：{}

包含项目支持的额外 feed 存储端的字典。 字典的键(key)是 URI 协议(scheme)，值是存储类(storage class)的路径。

FEED_STORAGES_BASE

Default 如下图 8-6 所示。

```
FEED_STORAGES_BASE={
    '':'scrapy.contrib.feedexport.FileFeedStorage',
    'file':'scrapy.contrib.feedexport.StdoutFeedStorage',
    'stdout':'scrapy.contrib.feedexport.S3FeedStorage',
    's3':'scrapy.contrib.feedexport.S3FeedStorage',
    'ftp':'scrapy.contrib.feedexport.FTPFeedStorage',
}
```

Code 8-6 FEED_STORAGES_BASE

包含 Scrapy 内置支持的 feed 存储端的字典。

FEED_EXPORTERS

Default：{}

包含项目支持的额外输出器(exporter)的字典。该字典的键(key)是 URI 协议(scheme) ,值是 Item 输出器(exporter) 类的路径。

FEED_EXPORTERS_BASE

Default 如下 Code 8-7 所示。

```
FEED_STORAGES_BASE={
    'json':'scrapy.contrib.exporter.JsonItemExporter',
    'jsonlines':'scrapy.contrib.exporter.JsonLinesItemExporter',
    'csv':'scrapy.contrib.exporter.CsvItemExporter',
```

```
    'xml':'scrapy.contrib.exporter.XmlItemExporter',
    'marshal':'scrapy.contrib.exporter.MarshalItemExporter',
}
```

Code 8-7 FEED_EXPORTERS_BASE

包含 Scrapy 内置支持的 feed 输出器(exporter)的字典。

8.5 图片管道

Scrapy 提供了一个 item pipeline ，来下载属于某个特定项目的图片，比如，当你抓取产品时，也想把它们的图片下载到本地。

这条管道，被称作图片管道，在 ImagesPipeline 类中实现，提供了一个方便并具有额外特性的方法，来下载并本地存储图片:

- 将所有下载的图片转换成通用的格式（JPG）和模式（RGB）
- 避免重新下载最近已经下载过的图片
- 缩略图生成
- 检测图像的宽/高，确保它们满足最小限制

这个管道也会为那些当前安排好要下载的图片保留一个内部队列，并将那些到达的包含相同图片的项目连接到那个队列中。 这可以避免多次下载几个项目共享的同一个图片。

Pillow 是用来生成缩略图，并将图片归一化为 JPEG/RGB 格式，因此为了使用图片管道，你需要安装这个库。 Python Imaging Library (PIL) 在大多数情况下是有效的，但众所周知，在一些设置里会出现问题，因此我们推荐使用 Pillow 而不是 PIL.

8.5.1 使用图片管道

当使用 ImagesPipeline ，典型的工作流程如下所示:

1. 在一个爬虫里，你抓取一个项目，把其中图片的 URL 放入 image_urls 组内。
2. 项目从爬虫内返回，进入项目管道。
3. 当项目进入 ImagesPipeline，image_urls 组内的 URLs 将被 Scrapy 的调度器和下载器（这意味着调度器和下载器的中间件可以复用）安排下载，当优先级更高，会在其他页面被抓取前处理。项目会在这个特定的管道阶段保持“locker”的状态，直到完成图片的下载（或者由于某些原因未完成下载）。
4. 当图片下载完，另一个组(images)将被更新到结构中。这个组将包含一个字典列表，其中包括下载图片的信息，比如下载路径、源抓取地址（从 image_urls 组获得）和图片的校验码。

images 列表中的图片顺序将和源 image_urls 组保持一致。如果某个图片下载失败，将会记录下错误信息，图片也不会出现在 images 组中。

8.5.2 使用样例

为了使用图片管道，你仅需要 启动它 并用 image_urls 和 images 定义一个项目，Item 代码如 Code 8-8 所示。

```
import scrapy
class MyItem(scrapy.Item):
    image_urls=scrapy.Field()
    images=scrapy.Field()
```

Code 8-8 Item

8.5.3 开启你的图片管道

为了开启你的图片管道，你首先需要在项目中添加它 ITEM_PIPELINES setting，代码如 Code 8-9 所示。

```
ITEM_PIPELINES={
    'scrapy.contrib.pipeline.images.ImagesPipeline':1
}
IMAGES_STORE='/path/to/valid/dir'
```

Code 8-9 Item

并将 IMAGES_STORE 设置为一个有效的文件夹，用来存储下载的图片。否则管道将保持禁用状态，即使你在 ITEM_PIPELINES 设置中添加了它。

8.5.4 图片存储

文件系统是当前官方唯一支持的存储系统，但也支持（非公开的） Amazon S3（https://aws.amazon.com/cn/s3/）。

文件系统存储

图片存储在文件中（一个图片一个文件），并使用它们 URL 的 SHA1 hash 作为文件名。

比如，对下面的图片 URL:

http://www.ixiaochouyu.com/themes/newixiaochouyu/images/job_Python/banner.jpg

它的 SHA1 hash 值为:

0c3eae0a23f71835abfcf79267ee48ec2ca332cb

将被下载并存为下面的文件:

<IMAGES_STORE>/full/0c3eae0a23f71835abfcf79267ee48ec2ca332cb.jpg

其中:

- <IMAGES_STORE> 是定义在 IMAGES_STORE 设置里的文件夹
- full 是用来区分图片和缩略图（如果使用的话）的一个子文件夹。

8.5.5 额外的特性

图片失效期限设置

图像管道避免下载最近已经下载的图片。使用 IMAGES_EXPIRES 设置可以调整失效期限，可以用天数来指定，配置文件 setting 设置，如 Code 8-10 所示。

```
#90 天的图片失效期限
IMAGES_EXPIRES=90
```

Code 8-10 图片失效期限设置

缩略图生成

图片管道可以自动创建下载图片的缩略图。

为了使用这个特性，你需要设置 IMAGES_THUMBS 字典，其关键字为缩略图名字，值为它们的大小尺寸。

代码如 Code 8-11 所示。

```
IMAGES_THUMBS={
    'small'  (50,50),
    'big:(270,270)
}
```

Code 8-11 缩略图片生成

当你使用这个特性时，图片管道将使用下面的格式来创建各个特定尺寸的缩略图:

```
pip install requests
```

<IMAGES_STORE>/thumbs/<size_name>/<image_id>.jpg

其中:

- <size_name> 是 IMAGES_THUMBS 字典关键字（small， big ，等）
- <image_id> 是图像 url 的 SHA1 hash

例如使用 small 和 big 缩略图名字的图片文件:

```
<IMAGES_STORE>/full/0c3eae0a23f71835abfcf79267ee48ec2ca332cb.jpg
```

```
<IMAGES_STORE>/thumbs/small/0c3eae0a23f71835abfcf79267ee48ec2ca332cb.jpg
<IMAGES_STORE>/thumbs/big/0c3eae0a23f71835abfcf79267ee48ec2ca332cb.jpg
```

第一个是从网站下载的完整图片。

滤过小图片

你可以丢掉那些过小的图片，只需在 setting 配置文件中设置 IMAGES_MIN_HEIGHT 和 IMAGES_MIN_WIDTH 中指定最小允许的尺寸。

比如:

```
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

注意：这些尺寸一点也不影响缩略图的生成。

默认情况下，没有尺寸限制，因此所有图片都将处理。

8.5.6 实现定制图片管道

下面是你可以在定制的图片管道里重写的方法：

```
class scrapy.contrib.pipeline.images.ImagesPipeline
```

get_media_requests(item, info)

在工作流程中可以看到，管道会得到图片的 URL 并从项目中下载。为了这么做，你需要重写 get_media_requests() 方法，并对各个图片 URL 返回一个 Request，如 Code 8-12 所示。

```
def get_media_requests(self,item,info) :
    for image_url in item[' image_urls' ]:
        yield scrapy.Request(image_url)
```

Code 8-12 get_media_requests

这些请求将被管道处理，当它们完成下载后，结果将以 2-元素的元组列表形式传送到 item_completed() 方法:

- success 是一个布尔值，当图片成功下载时为 True ，因为某个原因下载失败为``False``
- image_info_or_error 是一个包含下列关键字的字典（如果成功为 True ）或者出问题时为 Twisted Failure 。
  - url - 图片下载的 url。这是从 get_media_requests() 方法返回请求的 url。
  - path - 图片存储的路径（类似 IMAGES_STORE ）
  - checksum - 图片内容的 MD5 hash

item_completed() 接收的元组列表需要保证与 get_media_requests() 方法返回请求的顺

序相一致。下面是 results 参数的一个典型值，如图 8-13 所示。

```
[(True,
    {'checksum' :'2b00042f7481c7b856c4b41@d28f33cf',
    path' :' full/7d97e98f8af71@c7e7fe703abc8f639e0ee507c4.jpg' ,
    'ur1' : 'http://www.examp1e.com/images/product1.jpg'}),
(True,
    {'checksum' :'b9628c4ab9b595f72f288b98c4fde93d' ,
    path' :'ful1/1ca5879492b8fd686df1964ea3c1e2f4520f876f.jpg',
    url' :http://www.example.com/images/product2.jpg'}),
(False,
Failure:.))]
```

Code 8-13 item_completed

默认 get_media_requests() 方法返回 None ，这意味着项目中没有图片可下载。

item_completed(results, items, info)

当一个单独项目中的所有图片请求完成时（要么完成下载，要么因为某种原因下载失败），ImagesPipeline.item_completed() 方法将被调用。

item_completed() 方法需要返回一个输出，其将被送到随后的项目管道阶段，因此你需要返回（或者丢弃）项目，如你在任意管道里所做的一样。

这里是一个 item_completed() 方法的例子，其中我们将下载的图片路径（传入到 results 中）存储到 image_paths 项目组中，如果其中没有图片，我们将丢弃项目，如图 8-14 所示。

```
from scrapy.exceptions import DropItem
def item_completed( self,results, item,info) :
    image_paths= [x[' path'] for ok,x in results if ok]
    if not image_paths :
        raise DropItem("Item contains no images")
    item[' image_paths']= image_paths
    return item
```

图 8-14 item_completed

默认情况下， item_completed() 方法返回项目。

8.5.6 定制图片管道的例子

下面是一个图片管道的完整例子，其方法如上所示，代码如图 8-15 所示。

```
import scrapy
from scrapy.contrib.pipeline.images impor t ImagesPipeline
from scrapy.exceptionsimp ortDropItem
class IxiaochouyuImagesPipeline(ImagesPipeline) :
    def get_media_requests(self,item,info) :
        for image_ur1 in item[' image_urls' ]:
            yield scrapy.Request(image_ur1)
    def item completed(self,results,item,info):
        image_paths= [x[' path'] for ok,x in results if ok]
        if not image_paths :
            raise DropItem("Not have images" )
            item[' image_paths']= image_paths
            return item
```

Code 8-15 定制图片管道的例子

第 9 章 中间件

本章工作任务

- 任务 1：中间件的作用
- 任务 2：下载器中间件(Downloader Middleware)
- 任务 3：Spider 中间件(Middleware)

本章技能目标及重难点

编号	技能点描述	级别
1	中间件的作用	★★
2	下载器中间件(Downloader Middleware)	★★★
3	Spider 中间件(Middleware)	★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 爬虫的中间件机制，学完同学们就可以定制下载器中间件和 Spider 中间件。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

9.1 中间件的作用

在我们之前的课程中已经提及过中间件，我们来回顾一下这些中间件：

- 下载器中间件
- Spider 中间件
- 调度器中间件

如图 9-1 所示，通过 scrapy 结构可以发现中间件（Middlewares），是介于 Scrapy 的 Spiders 或 Downloader 或 Scheduler 处理的钩子框架，是一个一个轻量、底层的系统。接下来，我们来一一进行讲解。

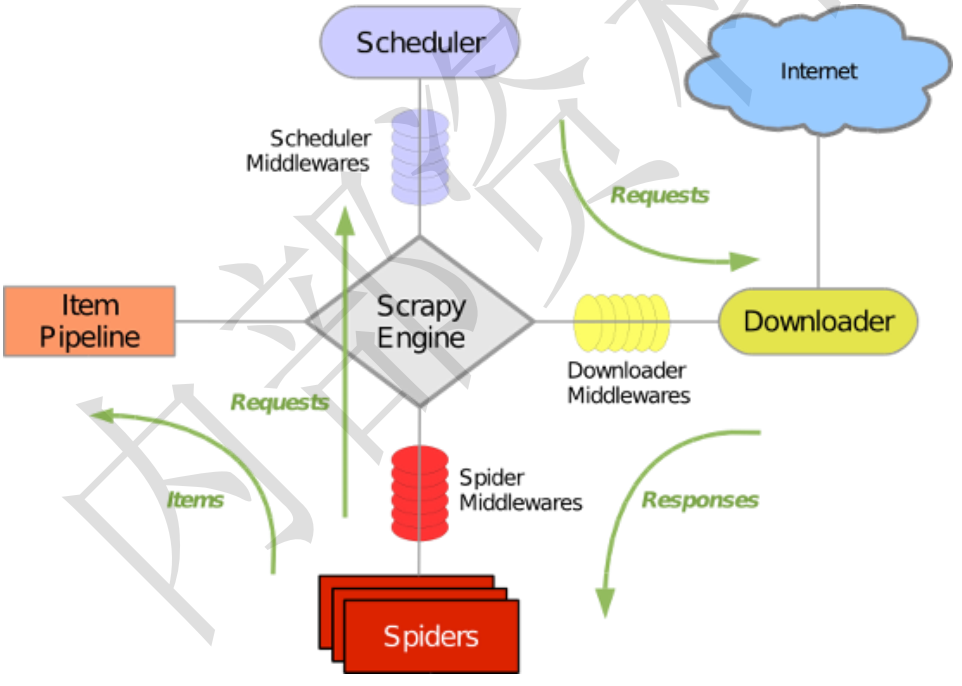

图 9-1 scrapy 结构

9.2 下载器中间件(Downloader Middleware)

下载器中间件是介于 Scrapy 的 request/response 处理的钩子框架。 是用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。

9.2.1 激活下载器中间件

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)。

我们来写一个例子，setting 配置文件中的配置，如 Code 9-2 所示。

```
DOWNLOADER_MIDDLEWARES={
        'myproject.middlewares.CustomDownloaderMiddleware' :543,
}
```

Code 9-2 downloader_middlewares

DOWNLOADER_MIDDLEWARES 设置会与 Scrapy 定义的 DOWNLOADER_MIDDLEWARES_BASE 设置合并(但不是覆盖)，而后根据顺序(order)进行排序，最后得到启用中间件的有序列表: 第一个中间件是最靠近引擎的，最后一个中间件是最靠近下载器的。

关于如何分配中间件的顺序请查看 DOWNLOADER_MIDDLEWARES_BASE 设置，而后根据您想要放置中间件的位置选择一个值。由于每个中间件执行不同的动作，您的中间件可能会依赖于之前(或者之后)执行的中间件，因此顺序是很重要的。

如果您想禁止内置的(在 DOWNLOADER_MIDDLEWARES_BASE 中设置并默认启用的)中间件，您必须在项目的 DOWNLOADER_MIDDLEWARES 设置中定义该中间件，并将其值赋为 None。例如，如果您想要关闭 user-agent 中间件，如 Code 9-3 所示。

```
DOWNLOADER_MIDDLEWARES={
        'myproject.middlewares.CustomDownloaderMiddleware' :543,
        'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware' :None
}
```

Code 9-3 downloader_middlewares

最后，请注意，有些中间件需要通过特定的设置来启用。

9.2.2 私人订制下载器中间件

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

(1) process_request(request, spider)

当每个 request 通过下载中间件时，该方法被调用。

process_request() 必须返回其中之一： 返回 None 、返回一个 Response 对象、返回一个 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，则安装的下载中间件的 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

（2）process_response(request, response, spider)

process_request() 必须返回以下之一： 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

（3）process_exception(request, exception, spider)

当下载处理器(download handler)或 process_request()(下载中间件)抛出异常(包括 IgnoreRequest 异常)时，Scrapy 调用 process_exception() 。

process_exception()应该返回以下之一：返回 None、一个 Response 对象、或者一个 Request 对象。

如果其返回 None，Scrapy 将会继续处理该异常，接着调用已安装的其他中间件的 process_exception()方法，直到所有中间件都被调用完毕，则调用默认的异常处理。

如果其返回一个 Response 对象，则已安装的中间件链的 process_response()方法被调用。Scrapy 将不会调用任何其他中间件的 process_exception() 方法。

如果其返回一个 Request 对象， 则返回的 request 将会被重新调用下载。这将停止中间件的 process_exception()方法执行，就如返回一个 response 的那样。

参数：

- request (是 Request 对象) – 产生异常的 request
- exception (Exception 对象) – 抛出的异常
- spider (Spider 对象) – request 对应的 spider

9.2.3 内置下载中间件

本页面介绍了 Scrapy 自带的所有下载中间件。

（1）CookiesMiddleware

class scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware

该中间件使得爬取需要 cookie(例如使用 session)的网站成为了可能。其追踪了 web server 发送的 cookie，并在之后的 request 中发送回去，就如浏览器所做的那样。

以下设置可以用来配置 cookie 中间件：

- COOKIES_ENABLED
- COOKIES_DEBUG

单 spider 多 cookie session

Scrapy 通过使用 cookiejar Request meta key 来支持单 spider 追踪多 cookie session。默认情况下其使用一个 cookie jar(session)，不过您可以传递一个标示符来使用多个。代码如 Code 9-4 所示。

```
def parse_item (self,response):
    for i,url in enumerate(urls):
        yield scrapy.Request( ‘http://www.example.com’ ,meta={ ‘cookoejar’ :i},
            callback=self.parse_page)
```

```
def parse_page(self,response):
    #do some processing
    return scrapy.Request( 'http://www.example.com/otherpage' ,
        meta={ 'cookiejar' :response.meta[ 'cookiejar' ]},
        callback=self.parse_other_page)
```

Code 9-4 downloader_middlewares

需要注意的是 cookiejar meta key 不是" 黏性的(sticky)" 。您需要在之后的 request 请求中接着传递。

COOKIES_ENABLED

默认： True

是否启用 cookies middleware。如果关闭，cookies 将不会发送给 web server。

COOKIES_DEBUG

默认： False

如果启用，Scrapy 将记录所有在 request(Cookie 请求头)发送的 cookies 及 response 接收到的 cookies(Set-Cookie 接收头)。

（2）DefaultHeadersMiddleware

class scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddleware

该中间件设置 DEFAULT_REQUEST_HEADERS 指定的默认 request header。

（3）DownloadTimeoutMiddleware

class scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddleware

该中间件设置 DOWNLOAD_TIMEOUT 或 spider 的 download_timeout 属性指定的 request 下载超时时间。

您也可以使用 download_timeout Request.meta key 来对每个请求设置下载超时时间。这种方式在 DownloadTimeoutMiddleware 被关闭时仍然有效。

（4）HttpAuthMiddleware

class scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware

该中间件完成某些使用 Basic access authentication (或者叫 HTTP 认证)的 spider 生成的请求的认证过程。

在 spider 中启用 HTTP 认证，请设置 spider 的 http_user 及 http_pass 属性，如 Code 9-5 所示。

```
Class SomeIntranetSiteSpider(CrawlSpider):
    http_user=’ someuser’
    http_pass=’ somepass’
    name=’ intranet.example.com’
```

Code 9-5 downloader_middlewares

（5）HttpCacheMiddleware

class scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware

该中间件为所有 HTTP request 及 response 提供了底层(low-level)缓存支持。其由 cache 存储后端及 cache 策略组成。

Scrapy 提供了两种 HTTP 缓存存储后端：

- Filesystem storage backend (默认值)
- DBM storage backend

您可以使用 HTTPCACHE_STORAGE 设定来修改 HTTP 缓存存储后端。您也可以实现您自己的存储后端。

Scrapy 提供了两种了缓存策略：

- RFC2616 策略
- Dummy 策略(默认值)

您可以使用 HTTPCACHE_POLICY 设定来修改 HTTP 缓存存储后端。您也可以实现您自己的存储策略。

Dummy 策略(默认值)

该策略不考虑任何 HTTP Cache-Control 指令。每个 request 及其对应的 response 都被缓存。 当相同的 request 发生时，其不发送任何数据，直接返回 response。

Dummpy 策略对于测试 spider 十分有用。其能使 spider 运行更快(不需要每次等待下载完成)， 同时在没有网络连接时也能测试。其目的是为了能够回放 spider 的运行过程，使之与之前的运行过程一模一样。

使用这个策略请设置：

HTTPCACHE_POLICY 为 scrapy.contrib.httpcache.DummyPolicy

RFC2616 策略

该策略提供了符合 RFC2616 的 HTTP 缓存，例如符合 HTTP Cache-Control，针对生产环境并且应用在持续性运行环境所设置。该策略能避免下载未修改的数据(来节省带宽 ,提高爬取速度)。

实现了：

- 当 no-store cache-control 指令设置时不存储 response/request。
- 当 no-cache cache-control 指定设置时不从 cache 中提取 response，即使 response 为最新。
- 根据 max-age cache-control 指令中计算保存时间(freshness lifetime)。
- 根据 Expires 指令来计算保存时间(freshness lifetime)。
- 根据 response 包头的 Last-Modified 指令来计算保存时间(freshness lifetime)。(Firefox 使用的启发式算法)。
- 根据 response 包头的 Age 计算当前年龄(current age)。
- 根据 Date 计算当前年龄(current age)。
- 根据 response 包头的 Last-Modified 验证老旧的 response。
- 根据 response 包头的 ETag 验证老旧的 response。
- 为接收到的 response 设置缺失的 Date 字段。

目前仍然缺失：

- Pragma: no-cache 支持 http://www.mnot.net/cache_docs/#PRAGMA
- Vary 字段支持 http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.6
- 当 update 或 delete 之后失效相应的 response
- ... 以及其他可能缺失的特性 ..

使用这个策略，设置：

HTTPCACHE_POLICY 为 scrapy.contrib.httpcache.RFC2616Policy

Filesystem storage backend (默认值)

文件系统存储后端可以用于 HTTP 缓存中间件。

使用该存储端，设置：

HTTPCACHE_STORAGE 为 scrapy.contrib.httpcache.FilesystemCacheStorage

每个 request/response 组存储在不同的目录中，包含下列文件：

- request_body - the plain request body
- request_headers - the request headers (原始 HTTP 格式)
- response_body - the plain response body

- response_headers - the request headers (原始 HTTP 格式)
- meta - 以 Python repr()格式(grep-friendly 格式)存储的该缓存资源的一些元数据。
- pickled_meta - 与 meta 相同的元数据，不过使用 pickle 来获得更高效的反序列化性能。

目录的名称与 request 的指纹(参考 scrapy.utils.request.fingerprint)有关，而二级目录是为了避免在同一文件夹下有太多文件 (这在很多文件系统中是十分低效的)。目录的例子：

```
/path/to/cache/dir/example.com/72/72811f648e718090f041317756c03adb0ada46c7
```

DBM storage backend

同时也有 DBM 存储后端可以用于 HTTP 缓存中间件。

默认情况下，其采用 anydbm 模块，不过您也可以通过 HTTPCACHE_DBM_MODULE 设置进行修改。

使用该存储端，设置：

HTTPCACHE_STORAGE 为 scrapy.contrib.httpcache.DbmCacheStorage

LevelDB storage backend

LevelDB 存储后端也可用于 HTTP 缓存中间件。

这是不推荐用于后端开发，因为只能一个进程进行同时访问 LevelDB 数据库，所以你不能运行一个爬行并同时打开 Scrapy 框架内相同的 Spider。

为了使用这个存储后端：

- 设置 HTTPCACHE_STORAGE 到 scrapy.contrib.httpcache.LeveldbCacheStorage
- 安装 LevelDB python bindings ，例如 pip install leveldb

HTTPCache 中间件设置

HttpCacheMiddleware 可以通过以下设置进行配置：

HTTPCACHE_ENABLED

默认： False

HTTP 缓存是否开启。

在 0.11 版更改: 在 0.11 版本前，是使用 HTTPCACHE_DIR 来开启缓存。

HTTPCACHE_EXPIRATION_SECS

默认： 0

缓存的 request 的超时时间，单位秒。

超过这个时间的缓存 request 将会被重新下载。如果为 0，则缓存的 request 将永远不会超时。

在 0.11 版更改： 在 0.11 版本前，0 的意义是缓存的 request 永远超时。

HTTPCACHE_DIR

默认： 'httpcache'

存储(底层的)HTTP 缓存的目录。如果为空，则 HTTP 缓存将会被关闭。 如果为相对目录，则相对于项目数据目录(project data dir)。更多内容请参考默认的 Scrapy 项目结构。

HTTPCACHE_IGNORE_HTTP_CODES

默认： []

不缓存设置中的 HTTP 返回值(code)的 request。

HTTPCACHE_IGNORE_MISSING

默认： False

如果启用，在缓存中没找到的 request 将会被忽略，不下载。

HTTPCACHE_IGNORE_SCHEMES

默认： ['file']

不缓存这些 URI 标准(scheme)的 response。

HTTPCACHE_STORAGE

默认： 'scrapy.contrib.httpcache.FilesystemCacheStorage'

实现缓存存储后端的类。

HTTPCACHE_DBM_MODULE

默认： 'anydbm'

在 DBM 存储后端的数据库模块。 该设定针对 DBM 后端。

HTTPCACHE_POLICY

默认： 'scrapy.contrib.httpcache.DummyPolicy'

实现缓存策略的类。

9.3 Spider 中间件(Middleware)

Spider 中间件(Middleware) 下载器中间件是介入到 Scrapy 的 spider 处理机制的钩子框架，您可以添加代码来处理发送给 Spiders 的 response 及 spider 产生的 item 和 request。

9.3.1 激活 Spider 中间件

要启用 spider 中间件，您可以将其加入到 SPIDER_MIDDLEWARES 设置中。该设置是一个字典，键位中间件的路径，值为中间件的顺序(order)。Setting 文件设置，代码如 Code 9-6。

```
SPIDER_MIDDLEWARES={
    'myproject.middlewares.CustomSpiderMiddleware' :543,
}
```

Code9-6 SPIDER_MIDDLEWRES

SPIDER_MIDDLEWARES 设置会与 Scrapy 定义的 SPIDER_MIDDLEWARES_BASE 设置合并(但不是覆盖)， 而后根据顺序(order)进行排序，最后得到启用中间件的有序列表: 第一个中间件是最靠近引擎的，最后一个中间件是最靠近 spider 的。

关于如何分配中间件的顺序，请同学们查看 SPIDER_MIDDLEWARES_BASE 设置，而后根据您想要放置中间件的位置选择一个值。由于每个中间件执行不同的动作，您的中间件可能会依赖于之前(或者之后)执行的中间件，因此顺序是很重要的。

如果您想禁止内置的(在 SPIDER_MIDDLEWARES_BASE 中设置并默认启用的)中间件，您必须在项目的 SPIDER_MIDDLEWARES 设置中定义该中间件，并将其值赋为 None。setting 文件设置方式如下载器中间件一致。如果您想要关闭 off-site 中间件，设置代码如 Code9-7 所示。

```
SPIDER_MIDDLEWARES={
    'myproject.middlewares.CustomSpiderMiddleware' :543,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware' :None,
}
```

Code 9-7 SPIDER_MIDDLEWRES

9.3.2 私人订制 Spider 中间件

编写 spider 中间件十分简单，设置方式与下载器中间件类似。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.spidermiddleware.SpiderMiddleware

process_spider_input(response, spider)

当 response 通过 spider 中间件时，该方法被调用，处理该 response。

process_spider_input()应该返回 None 或者抛出一个异常。

如果其返回 None，Scrapy 将会继续处理该 response，调用所有其他的中间件直到 spider 处理该 response。

如果其跑出一个异常(exception)，Scrapy 将不会调用任何其他中间件的 process_spider_input()方法，并调用 request 的 errback。 errback 的输出将会以另一个方向被重新输入到中间件链中，使用 process_spider_output()方法来处理，当其抛出异常时则带调用 process_spider_exception()。

参数:

- response (Response 对象) – 被处理的 response
- spider (Spider 对象) – 该 response 对应的 spider。

process_spider_output(response, result, spider)

与 process_spider_input(response, spider)协同使用的方法。当 Spider 处理 response 返回 result 时，该方法被调用。

process_spider_output()必须返回包含 Request 或 Item 对象的可迭代对象(iterable)。

参数:

- response (Response 对象) – 生成该输出的 response
- result (包含 Request 或 Item 对象的可迭代对象(iterable)) – spider 返回的 result
- spider (Spider 对象) – 其结果被处理的 spider

process_spider_exception(response, exception, spider)

异常处理机制，当 spider 或(其他 spider 中间件的) process_spider_input()跑出异常时，该方法被调用。

process_spider_exception()必须要么返回 None，要么返回一个包含 Response 或 Item 对象的可迭代对象(iterable)。

如果其返回 None，Scrapy 将继续处理该异常，调用中间件链中的其他中间件的 process_spider_exception()方法，直到所有中间件都被调用，该异常到达引擎(异常将被记录并被忽略)。

如果其返回一个可迭代对象，则中间件链的 process_spider_output()方法被调用，其他的 process_spider_exception()将不会被调用。

参数:

- response (Response 对象) – 异常被抛出时被处理的 response

- exception (Exception 对象) – 被跑出的异常
- spider (Spider 对象) – 抛出该异常的 spider

9.3.3 内置 Spider 中间件

本小节介绍了 Scrapy 自带的所有 spider 中间件。

DepthMiddleware

class scrapy.contrib.spidermiddleware.depth.DepthMiddleware

DepthMiddleware 是一个用于追踪每个 Request 在被爬取的网站的深度的中间件。 其可以用来限制爬取深度的最大深度或类似的事情。

DepthMiddleware 可以通过下列设置进行配置(更多内容请参考设置文档):

- DEPTH_LIMIT - 爬取所允许的最大深度，如果为 0，则没有限制。
- DEPTH_STATS - 是否收集爬取状态。
- DEPTH_PRIORITY - 是否根据其深度对 requet 安排优先级

HttpErrorMiddleware

class scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware

过滤出所有失败(错误)的 HTTP response，因此 spider 不需要处理这些 request。 处理这些 request 意味着消耗更多资源，并且使得 spider 逻辑更为复杂。

根据 HTTP 标准，返回值为 200-300 之间的值为成功的 resonse。

如果您想处理在这个范围之外的 response，您可以通过 spider 的 handle_httpstatus_list 属性或 HTTPERROR_ALLOWED_CODES 设置来指定 spider 能处理的 response 返回值。

例如，如果您想要处理返回值为 404 的 response 您可以这么做，如 Code 9-8 所示。

```
from scrapy.spider import CrawlSpider
class MySpider(CrawlSpider):
    handle_httpstatus_list=[404]
```

Code 9-8 HttpErrorMiddleware

Request.meta 中的 handle_httpstatus_list 键也可以用来指定每个 request 所允许的 response code。

不过请记住，除非您知道您在做什么，否则处理非 200 返回一般来说是个糟糕的决定。

- HttpErrorMiddleware settings
  - HTTPERROR_ALLOWED_CODES

默认:[]

忽略该列表中所有非 200 状态码的 response。

- HTTPERROR_ALLOW_ALL

  默认:False

  忽略所有 response，不管其状态值。

OffsiteMiddleware

class scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware

过滤出所有 URL 不由该 spider 负责的 Request。

该中间件过滤出所有主机名不在 spider 属性 allowed_domains 的 request。

当 spide 返回一个主机名不属于该 spider 的 request 时， 该中间件将会做一个类似于下面的记录:

DEBUG: Filtered offsite request to 'www.othersite.com': <GET http://www.othersite.com/some/page.html>

为了避免记录太多无用信息，其将对每个新发现的网站记录一次。因此，例如，如果过滤出另一个 www.othersite.com 请求 ,将不会有新的记录 但如果过滤出 someothersite.com 请求，仍然会有记录信息(仅针对第一次)。

如果 spider 没有定义 allowed_domains 属性，或该属性为空，则 offsite 中间件将会允许所有 request。

如果 request 设置了 dont_filter 属性，即使该 request 的网站不在允许列表里，则 offsite 中间件将会允许该 request。

第 10 章 爬虫小技巧

本章工作任务

- 任务 1：了解掌握代理配置
- 任务 2：熟练使用 Item Loaders
- 任务 3：了解掌握

本章技能目标及重难点

编号	技能点描述	级别
1	了解掌握代理配置	★★
2	熟练使用 Item Loaders	★★
3		★★

注： "★"理解级别 "★★"掌握级别 "★★★"应用级别

本章学习目标

本章开始学习 Scrapy 爬虫一些小技巧，需要同学们学会配置代理模块。

本章学习建议

本章适合有 Python 爬虫基础的学员学习。

本章内容（学习活动）

10.1 代理模块

本节内容主要是教各位怎么初期配置一下 scrapy 的代理。

10.1.1 简述

在 scrapy 中专门有一个模块 Downloader Middleware 来实现 scrapy 爬虫中请求和相应的某些通用功能，比如我们这次要用到的代理功能，就是通过其子模块 HttpProxyMiddleware 来实现的，至于 Downloader Middleware 的其他子模块，我以后有空会一一介绍的，现在大家只要知道这个模块的基本含义就行了。

10.1.2 代理配置

1.创建代理模块

首先我们现在爬虫目录下面新建一个名称为 middlewares.py 的文件，代码如下 Code 10-1 所示。

```
import base64
class ProxyMiddleware(object):
    def process_request(self,request,spider):
        request.meta[‘proxy’]=’ http://IP 地址：端口号’
        proxy_user_pass=”用户名：账号”
        encoded_user_pass=base64.encodestring(proxy_user_pass)
        request.headers[‘Proxy-Authorization’]=’ Basic’ +encoded_user_pass
```

Code 10-1 middlewares

其中

1. process_request 这个方法是自带的，是在每个请求之前都会自动执行一下，借此来修改参数
2. request.meta['proxy']直接通过这个属性直接设置代理，填的是 IP 地址和端口号，如 http://218.200.66.196:8080,一般你搞来的代理都会有着两个东西，直接填上去就好~
3. 如果你搞来的代理还需要账号和密码的话，我们就需要给 proxy_user_pass 赋值，形如 root:12345。
4. 为了让你的代理帐号密码能良好运行，我们还需要 base64.encodestring 这个函数来帮一下

忙，对你的代理帐号密码进行编码。

5. 最后将编码过后的函数赋值给 http 头中的 Proxy-Authorization 参数即可。

6. 注意:以上 3~5 步，如果代理没有帐号密码的话就不要折腾了。

建立完毕之后文件目录结构如下:

```
|---- tutorial
| |---- tutorial
|    |---- __init__.py
|    |---- items.py            #用来存储爬下来的数据结构（字典形式）
|    |---- middlewares.py   #用来配置代理
|    |---- pipelines.py      #用来对爬出来的 item 进行后续处理，如存入数据库等
|    |---- settings.py      #爬虫配置文件
|    |---- spiders            #此目录用来存放创建的新爬虫文件（爬虫主体）
|       |---- __init__.py
| |---- scrapy.cfg            #项目配置文件
```

2.修改配置文件

代理文件创建完毕之后，需要修改配置文件 settings.py,并在添加如下内容：

```
DOWNLOADER_MIDDLEWARES = {
    'njupt.middlewares.ProxyMiddleware': 100,
}
```

就是添加一下大的配置模块 Downloader Middleware，外加配置一个子模块 HttpProxyMiddleware。

10.2 Item Loaders

Item Loaders 提供了一种便捷的方式填充抓取到的:Items。虽然 Items 可以使用自带的类字典形式 API 填充，但是 Items Loaders 提供了更便捷的 API，可以分析原始数据并对 Item 进行赋值。

从另一方面来说，Items 提供保存抓取数据的容器，而 Item Loaders 提供的是 填充 容器的机制。

Item Loaders 提供的是一种灵活，高效的机制，可以更方便的被 spider 或 source format (HTML，XML，etc)扩展，并 override 更易于维护的、不同的内容分析规则。

10.2.1 运用 Item Loaders 于本地的 items

要使用 Item Loader, 你必须先将它实例化。你可以使用类似字典的对象(例如: Item or dict)来进行实例化，或者不使用对象也可以，当不用对象进行实例化的时候，Item 会自动使用

ItemLoader.default_item_class 属性中指定的 Item 类在 Item Loader constructor 中实例化。

然后,你开始收集数值到 Item Loader 时，通常使用 Selectors。你可以在同一个 item field 里面添加多个数值；Item Loader 将知道如何用合适的处理函数来“添加”这些数值。

下面是在 Spider 中典型的 Item Loader 的用法，使用 Items chapter 中声明的 Product item，如 Code 10-2 所示。

```
from scrapy.contrib.loader im port ItemLoader
from myproject.items import Product
def parse(self,response):
    1= ItemLoader(item=Product(),response=response)
    1.add_xpath('name' ， '//div[@class="product_name"]' )
    1.add_xpath('name','//div[@class="product_title"]' )
    1.add_xpath('price','//p[@id="price"]' )
    1.add_css('stock' ，p#stock]' )
    1.add_value('last_updated','today' )#youcanalsouseliteralvalues
    return 1.load_item()
```

Code 10-2 Item loader

快速查看这些代码之后,我们可以看到发现 name 字段被从页面中两个不同的 XPath 位置提取:

1. //div[@class="product_name"]
2. //div[@class="product_title"]

换言之,数据通过用 add_xpath()的方法,把从两个不同的 XPath 位置提取的数据收集起来. 这是将在以后分配给 name 字段中的数据。

之后,类似的请求被用于 price 和 stock 字段 (后者使用 CSS selector 和 add_css() 方法)， 最后使用不同的方法 add_value()对 last_update 填充文本值(today)。

最终，当所有数据被收集起来之后，调用 ItemLoader.load_item() 方法，实际上填充并且返回了之前通过调用 add_xpath()，add_css()，and add_value()所提取和收集到的数据的 Item。

10.2.2 输入输出加工

Item Loader 在每个(Item)字段中都包含了一个输入处理器和一个输出处理器。输入处理器收到数据时立刻提取数据 (通过 add_xpath()，add_css()或者 add_value()方法)之后输入处理器的结果被收集起来并且保存在 ItemLoader 内. 收集到所有的数据后，调用 ItemLoader.load_item()方法来填充,并

得到填充后的 Item 对象。这是当输出处理器被和之前收集到的数据(和用输入处理器处理的)被调用.输出处理器的结果是被分配到 Item 的最终值。

让我们看一个例子来说明如何输入和输出处理器被一个特定的字段调用(同样适用于其他 field)，如下 Code 10-3 所示。

```
from scrapy.contrib.loader import ItemLoader
from myproject.items imprt Product
def parse(self,response) :
    1= ItemLoader(item=Product(),response=response)
    1.add_xpath('name','//div[@class="product_name"]')
    1.add_xpath('name','//div[@class="product_title"]')
    1.add_xpath('price','//p[@id="price"]')
    1.add_css('stock','p#stock]')
    1.add_value('last_updated','today'  )#youcanalsouseliteralvalues
    return 1.load_item()
```

Code 10-3 Item loader

发生了这些事情:

1. 从 xpath1 提取出的数据,传递给 输入处理器 的 name 字段。输入处理器的结果被收集和保存在 Item Loader 中(但尚未分配给该 Item)。
2. 从 xpath2 提取出来的数据,传递给(1)中使用的相同的 输入处理器。输入处理器的结果被附加到在(1)中收集的数据(如果有的话)。
3. 和之前相似，只不过这里的数据是通过 css CSS selector 抽取，之后传输到在(1)和(2)使用 的 input processor 中。最终输入处理器的结果被附加到在(1)和(2)中收集的数据之后 (如果存在数据的话)。
4. 这里的处理方式也和之前相似，但是此处的值是通过 add_value 直接赋予的， 而不是利用 XPath 表达式或 CSS selector 获取。得到的值仍然是被传送到输入处理器。 在这里例程中，因为得到的值并非可迭代，所以在传输到输入处理器之前需要将其 转化为可迭代的单个元素，这才是它所接受的形式。
5. 在之前步骤中所收集到的数据被传送到 output processor 的 name field 中。输出处理器的结果就是赋到 item 中 name field 的值。

需要注意的是，输入和输出处理器都是可调用对象，调用时传入需要被分析的数据， 处理后返回分析得到的值。因此你可以使用任意函数作为输入、输出处理器。唯一需注意的是它们必须接收一个（并且只是一个）迭代器性质的 positional 参数。

10.2.3 自定义 Item Loader

使用类定义语法，下面是一个例子，如 Code 10-4 所示。

```
from scrapy.loader importItemLoader
from scrapy.loader.processors importTakeFirst,MapCompose,Join
class ProductLoader(ItemLoader):
    default_output_processor=TakeFirst()
    name_in=MapCompose(Unicode.title)
    name_out=Join()
    price_in=MapCompose(Unicode.strip)
```

Code 10-4 自定义 Item loader

通过 _in 和 _out 后缀来定义输入和输出处理器，并且还可以定义默认的 ItemLoader.default_input_processor 和 ItemLoader.default_input_processor.

10.3 Settings

Scrapy 设定(settings)提供了定制 Scrapy 组件的方法。您可以控制包括核心(core)，插件(extension)，pipeline 及 spider 组件。

设定为代码提供了提取以 key-value 映射的配置值的的全局命名空间(namespace)。设定可以通过下面介绍的多种机制进行设置。

设定(settings)同时也是选择当前激活的 Scrapy 项目的方法(如果您有多个的话)。

10.3.1 指定设定(Designating the settings)

当您使用 Scrapy 时，您需要声明您所使用的设定。这可以通过使用环境变量: SCRAPY_SETTINGS_MODULE 来完成。

SCRAPY_SETTINGS_MODULE 必须以 Python 路径语法编写, 如 myproject.settings 。

10.3.2 获取设定值(Populating the settings)

设定可以通过多种方式设置，每个方式具有不同的优先级。 下面以优先级降序的方式给出方式列表:

1. 命令行选项(Command line Options)(最高优先级)
2. 项目设定模块(Project settings module)
3. 命令默认设定模块(Default settings per-command)
4. 全局默认设定(Default global settings) (最低优先级)

这些设定(settings)由 scrapy 内部很好的进行了处理，不过您仍可以使用 API 调用来手动处理。 详情请参考 设置(Settings) API.

这些机制将在下面详细介绍。

1. 命令行选项(Command line options)

命令行传入的参数具有最高的优先级。 您可以使用 command line 选项 -s (或 --set) 来覆盖一个(或更多)选项。

样例:

scrapy crawl myspider -s LOG_FILE=scrapy.log

2. 项目设定模块(Project settings module)

项目设定模块是您 Scrapy 项目的标准配置文件。 其是获取大多数设定的方法。例如:: myproject.settings 。

3. 命令默认设定(Default settings per-command)

每个 Scrapy tool 命令拥有其默认设定，并覆盖了全局默认的设定。 这些设定在命令的类的 default_settings 属性中指定。

4. 默认全局设定(Default global settings)

全局默认设定存储在 scrapy.settings.default_settings 模块，并在 内置设定参考手册 部分有所记录。

10.3.3 内置设定参考手册

这里以字母序给出了所有可用的 Scrapy 设定及其默认值和应用范围。

如果给出可用范围，并绑定了特定的组件，则说明了该设定使用的地方。 这种情况下将给出该组件的模块，通常来说是插件、中间件或 pipeline。 同时也意味着为了使设定生效，该组件必须被启用。

BOT_NAME

默认: 'scrapybot'

Scrapy 项目实现的 bot 的名字(即项目名称)。 这将用来构造默认 User-Agent，同时也用来

log。

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。

CONCURRENT_REQUESTS_PER_DOMAIN

默认: 8

对单个网站进行并发请求的最大值。

CONCURRENT_REQUESTS_PER_IP

默认: 0

对单个 IP 进行并发请求的最大值。如果非 0，则忽略 CONCURRENT_REQUESTS_PER_DOMAIN 设定，使用该设定。也就是说，并发限制将针对 IP，而不是网站。

该设定也影响 DOWNLOAD_DELAY: 如果 CONCURRENT_REQUESTS_PER_IP 非 0，下载延迟应用在 IP 而不是网站上。

DEFAULT_ITEM_CLASS

默认: 'scrapy.item.Item'

Scrapy shell 中实例化 item 使用的默认类。

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。

DEPTH_PRIORITY

默认: 0

整数值。用于根据深度调整 request 优先级。

如果为 0，则不根据深度进行优先级调整。

DEPTH_STATS

默认: True

是否收集最大深度数据。

DEPTH_STATS_VERBOSE

默认: False

是否收集详细的深度数据。如果启用，每个深度的请求数将会被收集在数据中。

DNSCACHE_ENABLED

默认: True

是否启用 DNS 内存缓存(DNS in-memory cache)。

DOWNLOADER

默认: 'scrapy.core.downloader.Downloader'

用于 crawl 的 downloader.

DOWNLOADER_MIDDLEWARES

默认:: {}

保存项目中启用的下载中间件及其顺序的字典。

DOWNLOADER_MIDDLEWARES_BASE

包含 Scrapy 默认启用的下载中间件的字典。永远不要在项目中修改该设定，而是修改 DOWNLOADER_MIDDLEWARES 。

DOWNLOADER_STATS

默认: True

是否收集下载器数据。

DOWNLOAD_DELAY

默认: 0

下载器在下载同一个网站下一个页面前需要等待的时间。该选项可以用来限制爬取速度， 减轻服务器压力。同时也支持小数:

```
DOWNLOAD_DELAY = 0.25    # 250 ms of delay
```

该设定影响(默认启用的) RANDOMIZE_DOWNLOAD_DELAY 设定。 默认情况下，Scrapy 在两个请求间不等待一个固定的值， 而是使用 0.5 到 1.5 之间的一个随机值 * DOWNLOAD_DELAY 的结果作为等待间隔。

当 CONCURRENT_REQUESTS_PER_IP 非 0 时，延迟针对的是每个 ip 而不是网站。

另外您可以通过 spider 的 download_delay 属性为每个 spider 设置该设定。

DOWNLOAD_HANDLERS

默认: {}

保存项目中启用的下载处理器(request downloader handler)的字典。

DOWNLOAD_HANDLERS_BASE

保存项目中默认启用的下载处理器(request downloader handler)的字典。 永远不要在项目中修改该设定，而是修改 DOWNLOADER_HANDLERS 。

如果需要关闭上面的下载处理器，您必须在项目中的 DOWNLOAD_HANDLERS 设定中设置该处理器，并为其赋值为 None 。

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)习惯设定在 0-1000 范围内。

为了兼容性，ITEM_PIPELINES 支持列表。

ITEM_PIPELINES_BASE

默认: {}

保存项目中默认启用的 pipeline 的字典。 永远不要在项目中修改该设定，而是修改 ITEM_PIPELINES 。

LOG_ENABLED

默认: True

是否启用 logging。

LOG_ENCODING

默认: 'utf-8'

logging 使用的编码。

LOG_FILE

默认: None

logging 输出的文件名。如果为 None，则使用标准错误输出(standard error)。

LOG_LEVEL

默认: 'DEBUG'

log 的最低级别。可选的级别有: CRITICAL、 ERROR、WARNING、INFO、DEBUG。

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。

MEMDEBUG_ENABLED

默认: False

是否启用内存调试(memory debugging)。

MEMDEBUG_NOTIFY

默认: []

如果该设置不为空，当启用内存调试时将会发送一份内存报告到指定的地址,否则该报告将写到 log 中。

例如：

MEMUSAGE_NOTIFY = ['user@example.com']

MEMUSAGE_ENABLED

默认: False

是否启用内存使用插件。当 Scrapy 进程占用的内存超出限制时 ,该插件将会关闭 Scrapy 进程，同时发送 email 进行通知。

MEMUSAGE_LIMIT_MB

默认: 0

在关闭 Scrapy 之前所允许的最大内存数(单位: MB)(如果 MEMUSAGE_ENABLED 为 True)。如果为 0，将不做限制。

MEMUSAGE_NOTIFY_MAIL

默认: False

达到内存限制时通知的 email 列表。

例如：

MEMUSAGE_NOTIFY_MAIL = ['user@example.com']

MEMUSAGE_REPORT

默认: False

每个 spider 被关闭时是否发送内存使用报告。

查看 内存使用扩展(Memory usage extension).

MEMUSAGE_WARNING_MB

默认: 0

在发送警告 email 前所允许的最大内存数(单位: MB)(如果 MEMUSAGE_ENABLED 为 True)。如果为 0，将不发送警告。

REDIRECT_MAX_TIMES

默认: 20

定义 request 允许重定向的最大次数。超过该限制后该 request 直接返回获取到的结果。 对某些任务我们使用 Firefox 默认值。

REDIRECT_PRIORITY_ADJUST

默认: +2

修改重定向请求相对于原始请求的优先级。 负数意味着更多优先级。

SPIDER_MIDDLEWARES

默认:: {}

保存项目中启用的下载中间件及其顺序的字典。 更多内容请参考 激活 spider 中间件 。

SPIDER_MIDDLEWARES_BASE

保存项目中默认启用的 spider 中间件的字典。 永远不要在项目中修改该设定，而是修改 SPIDER_MIDDLEWARES 。