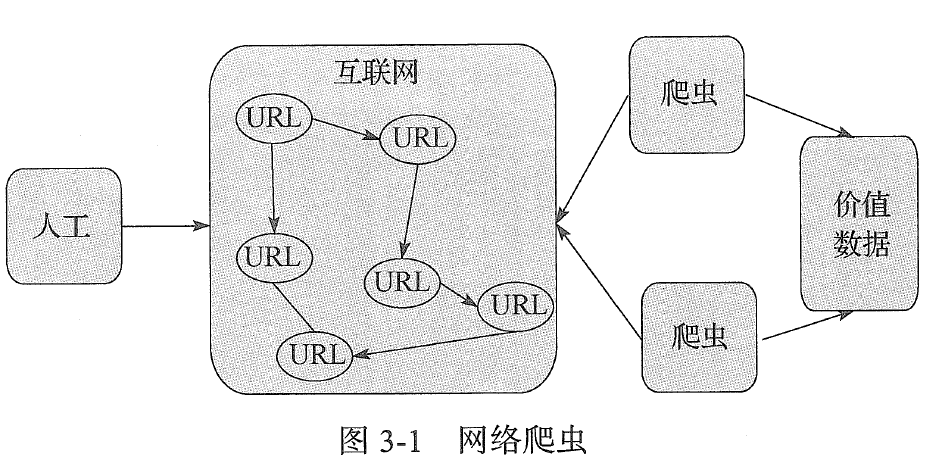
**爬虫基础介绍**

**网络爬虫：**是一种按照一定规则，自动地抓取万维网的程序或者脚本。

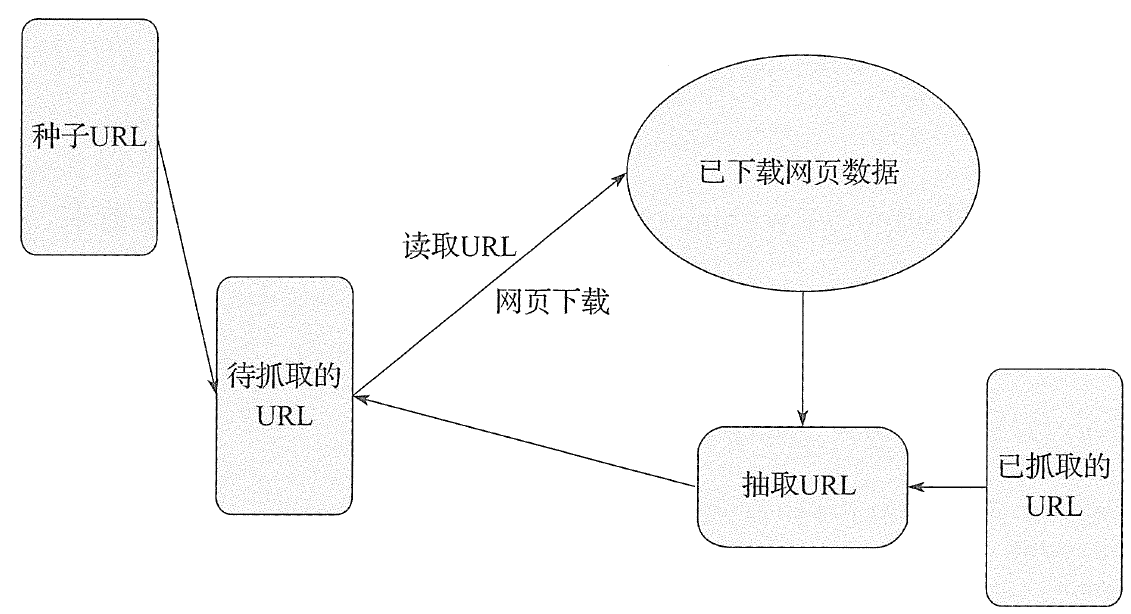
网络爬虫大致可以分为以下几种类型：通用网络爬虫、聚焦网络爬虫、增量式网络爬虫、聚焦网络爬虫、增量式网络爬虫、深层网络爬虫、实际的网络爬虫系统通常是几种爬虫技术相结合实现的



**聚焦爬虫：**一个自动下载网页的程序。它根据既定的抓取目标，有选择地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

**增量式网络爬虫**：是指对已下载网页采取增量式更新和只爬行新产生的或者已经发生变化网页的爬虫，它能够在一定程度上保证所爬行的页面是尽可能新的页面。和周期性爬行和刷新页面的网络爬虫相比，增量式爬虫只会在需要的时候爬行新产生或发生更新的页面，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬行的网页，减少时间和空间上的耗费，但是增加了爬虫算法的复杂度和实现难度。

**深层网络爬虫：**Web页面按存在方式可以分为表层网页和深层网页。表层网页是指传统搜索引擎可以索引的页面，以超链接可以到达的静态页面为主构成的Web页面。深层网络是那些大部分内容不能通过静态链接获取的、隐藏在搜索表单后的，只有用户提交一些关键词才能获得的Web页面。例如用户登录或者注册才能访问的页面



网络爬虫工作流程

1. 首先选取一部分静心挑选的种子URL
2. 将这些URL放入待抓取URL队列

3、从待抓取URL队列中读取待抓取队列的URL，解析DNS，并且得到主机的IP，并将URL对应的网页下载下来，存储进已下载网页库中。此外，将这些URL放进已经抓取URL队列

4、分析已经抓取URL队列中的URL，从已下载的网页数据中分析出其他URL，并和已抓取的URL进行比较去重，最后将去重的URL放入待抓取URL队列，从而进行下一个循环。

**爬虫基本库**

爬虫相关包

urllib、urllib2、requests、BS4、scrapy、pyspider

重点requests ，BS4

requests负责连接拓扑，出来http协议

bs4负责将网页变成结构化数据，方便爬取

re正则表达式

#!/busr/bin/env python

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

#简单爬虫示例

import urllib

import urllib.request as Req

weburl = 'http://www.douban.com'

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

req = Req.Request(weburl,headers=headers)

webPage = Req.urlopen(req)

data = webPage.read()

data = data.decode('utf-8')

print(data) #网页数据

print(type(webPage)) #类型

print(webPage.geturl()) #请求URL

print(webPage.info()) #请求头部

print(webPage.getcode()) #服务器返回状态码

基础HTTP库：urllib,urllib2,httplib2,requests,treq

在Python2版本中，有urllib和urllib2两个库可以用来实现request的发送。在Python3中，已经不存在urllib2这个库了，统一为urllib

urllib中包括了四个模块

urllib.request 用来发送request和获取request的结果

urllib.error 包含了urllib.request产生的异常

urllib.parse 用来解析和处理URL

urllib.robotparser 用来解析页面的robots.txt文件

其中模拟请求使用的最主要的库便是urllib.request，异常处理用urllib.error库

使用urllib.request发送请求

urllib.request.urlopen()基本使用

urllib.request模块提供了最基本的构造HTTP请求的方法，利用它可以模拟浏览器的一个请求发起过程，同时它还带有处理autherntication(授权验证)，redirections(重定向)，cookies(浏览器Cookies)以及其他内容

urllib.request.urlopen(url, data=None, [timeout, ]\*, cafile=None, capath=None, cadefault=False, context=None)

data参数是可选的，如果要添加data，它要是字节流编码格式的内容，即bytes累死，通过bytes()函数可以进行转化，如果你传递了这个data参数，它的请求方式就不再是GET方式请求，而是POTS

基本方法

urllib.request.urlopen(url, data=None, [timeout, ]\*, cafile=None, capath=None, cadefault=False, context=None)

url：需要打开的网站

data：Post提交的数据

timeout：设置网站的访问超时时间

直接使用urllib.request模块的urlopen()获取也没，page的数据格式为bytes类型，需要decode()解码，转换成str类型

示例

#-\*- coding: UTF-8 -\*-

import urllib

import urllib.request

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

data = urllib.request.urlopen(url).read()

data = data.decode('UTF-8')

print(data)

urlopen返回对象提供方法：

read(),readline(),readlines(),fileno(),close：对HTTPResponse类型数据进行操作

info()：返回HTTPMessage对象，表示远程服务器返回的头信心

getcode()：返回HTTP状态码。如果是HTTP请求，200请求成功完成，404网址未找到

geturl()：返回请求url

使用Request

urllib.request.Request(url, data=None, headers={}, method=None)

使用request（）来包装请求，再通过urlopen（）获取页面。

url = r'http://www.lagou.com/zhaopin/Python/?labelWords=label'

headers = {

'User-Agent': r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) '

r'Chrome/45.0.2454.85 Safari/537.36 115Browser/6.0.3',

'Referer': r'http://www.lagou.com/zhaopin/Python/?labelWords=label',

'Connection': 'keep-alive'

}

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

page = request.urlopen(req).read()

page = page.decode('utf-8')

用来包装头部的数据：

User-Agent ：这个头部可以携带如下几条信息：浏览器名和版本号、操作系统名和版本号、默认语言

Referer：可以用来防止盗链，有一些网站图片显示来源http://\*\*\*.com，就是检查Referer来鉴定的

Connection：表示连接状态，记录Session的状态。

Post数据

urllib.request.urlopen(url, data=None, [timeout, ]\*, cafile=None, capath=None, cadefault=False, context=None)

urlopen（）的data参数默认为None，当data参数不为空的时候，urlopen（）提交方式为Post

示例

from urllib import request, parse

url = r'http://www.lagou.com/jobs/positionAjax.json?'

headers = {

'User-Agent': r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) '

r'Chrome/45.0.2454.85 Safari/537.36 115Browser/6.0.3',

'Referer': r'http://www.lagou.com/zhaopin/Python/?labelWords=label',

'Connection': 'keep-alive'

}

data = {

'first': 'true',

'pn': 1,

'kd': 'Python'

}

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

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

page = request.urlopen(req).read()

page = page.decode('utf-8')

urllib.parse.urlencode(query, doseq=False, safe='', encoding=None, errors=None)

urlencode（）主要作用就是将url附上要提交的数据。

经过urlencode（）转换后的data数据为?first=true?pn=1?kd=Python，最后提交的url为

http://www.lagou.com/jobs/positionAjax.json?first=true?pn=1?kd=Python

Post的数据必须是bytes或者iterable of bytes，不能是str，因此需要进行encode（）编码

page = request.urlopen(req, data=data).read()

当然，也可以把data的数据封装在urlopen（）参数中

异常处理

def get\_page(url):

headers = {

'User-Agent': r'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) '

r'Chrome/45.0.2454.85 Safari/537.36 115Browser/6.0.3',

'Referer': r'http://www.lagou.com/zhaopin/Python/?labelWords=label',

'Connection': 'keep-alive'

}

data = {

'first': 'true',

'pn': 1,

'kd': 'Python'

}

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

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

try:

page = request.urlopen(req, data=data).read()

page = page.decode('utf-8')

except error.HTTPError as e:

print(e.code())

print(e.read().decode('utf-8'))

return page

使用代理

urllib.request.ProxyHandler(proxies=None)、

当需要抓取的网站设置了访问限制，这时就需要用到代理来抓取数据

data = {

'first': 'true',

'pn': 1,

'kd': 'Python'

}

proxy = request.ProxyHandler({'http': '5.22.195.215:80'}) # 设置proxy

opener = request.build\_opener(proxy) # 挂载opener

request.install\_opener(opener) # 安装opener

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

page = opener.open(url, data).read()

page = page.decode('utf-8')

return page

示例

#/usr/bin/env python

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

import urllib.request as Req

import urllib.parse

url = 'http://httpbin.org/post'

data = bytes(urllib.parse.urlencode({'word':'hello'}),encoding='utf8')

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

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

WebPage=Req.urlopen(req,data=data)

print(WebPage.read())

在这里我们传递了一个参数 word ，值是 hello 。它需要被转码成 bytes （字

节流）类型。其中转字节流采用了 bytes() 方法，第一个参数需要是 str (字符

串)类型，需要用 urllib.parse.urlencode() 方法来将参数字典转化为字符串。

第二个参数指定编码格式，在这里指定为 utf8 。

class urllib.request.Request(url, data=None, headers={}, origin\_

req\_host=None, unverifiable=False, method=None)

urllib.request高级特性

Handler工具，用于处理登录验证、Cookies处理等功能

urllib.request.BaseHandler,它是所有其他Handler的父类，它提供了最基本的Handler的方法

HTTPDefaultErrorHandler 用于处理HTTP响应错误，错误抛出HTTPError类型异常

HTTPRedirectHandler 用于处理重定向

HTTPCookieProcessor 用于处理Cookie

ProxyHandler 用于设置代理，默认代理为空

HTTPPasswordMgr 用于管理密码，它维护了用户名密码的表

HTTPBasicAuthHandler 用于管理认证，如果一个链接打开时需要认证，那么可以用它来解决认证问题。另外还有其他的Handler，可以参考官方文档

另外一个比较重要的就是 OpenerDirector ，我们可以称之为 Opener ，我们之前用过 urllib.request.urlopen() 这个方法，实际上它就是一个 Opener

之前我们使用的 Request 、 urlopen() 相当于类库为你封装好了极其常用的请求方法，利用它们两个我们就可以完成基本的请求。

但是现在不一样了，我们需要实现更高级的功能，所以我们需要深入一层，使用更上层的实例来完成我们的操作。所以，在这里我们就用到了比调用 urlopen() 的对象的更普遍的对象，也就是 Opener

Opener 可以使用 open() 方法，返回的类型和 urlopen() 如出一辙。那么它和 Handler 有什么关系？简而言之，就是利用 Handler 来构建 Opener 。

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

import urllib.request as Req

url = 'https://www.rjb777.com/Member/Login.aspx'

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

auth\_handler = Req.HTTPBasicAuthHandler()

auth\_handler.add\_password(None,url,'18079902120','rjb2016')

opener = Req.build\_opener(auth\_handler)

Req.install\_opener(opener)

req = Req.Request(url,headers=headers,method='POST')

Webpage = Req.urlopen(req)

data = Webpage.decode('utf-8')

print(data)

首先实例化了一个 HTTPBasicAuthHandler 对象，然后利用 add\_password() 添加进去用户名和密码，相当于建立了一个处理认证的处理器。

接下来利用 urllib.request.build\_opener() 方法来利用这个处理器构建一个 Opener ，那么这个 Opener 在发送请求的时候就具备了认证功能了。接下来利用 Opener 的 open() 方法打开链接，就可以完成认证了。

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

import urllib.request as Req

import http.cookiejar as Coo

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

url = 'https://www.rjb777.com/Member/Login.aspx'

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

cookie = Coo.CookieJar()

handler = Req.HTTPCookieProcessor(cookie)

opener = Req.build\_opener(handler)

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

response = opener.open(req)

#print(type(cookie))

for item in cookie:

# print(item)

# print(item.name)

# print(type(item))

print(item.name+"="+item.value)

首先我们必须声明一个 CookieJar 对象，接下来我们就需要利用 HTTPCookieProcessor 来构建一个 handler ，最后利用 build\_opener 方法构建出 opener ，执行 open() 即可

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

#Cookie保存

import urllib.request as Req

import http.cookiejar as Coo

url = 'https://www.rjb777.com/Member/Login.aspx'

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

filenmae = 'cookie.txt'

cookie = Coo.MozillaCookieJar(filenmae)

handler = Req.HTTPCookieProcessor(cookie)

opener = Req.build\_opener(handler)

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

response = opener.open(req)

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

这时的 CookieJar 就需要换成 MozillaCookieJar ，生成文件时需要用到它，它是 CookieJar 的子类，可以用来处理 Cookie 和文件相关的事件，读取和保存 Cookie ，它可以将 Cookie 保存成 Mozilla 型的格式

另外还有一个 LWPCookieJar ，同样可以读取和保存 Cookie ，但是保存的格式和 MozillaCookieJar 的不一样，它会保存成与libwww-perl的Set-Cookie3文件格式的 Cookie

cookie = http.cookiejar.LWPCookieJar(filename)

读取cookie发出请求

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

#使用保存的Cookie

import urllib.request as Req

import http.cookiejar as Coo

url = 'https://www.rjb777.com/Member/Login.aspx'

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

filenmae = 'cookie.txt'

cookie = Coo.LWPCookieJar()

cookie.load(filenmae,ignore\_discard=True,ignore\_expires=True)

handler = Req.HTTPCookieProcessor(cookie)

opener = Req.build\_opener(handler)

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

response = opener.open(req).read()

print(response.decode('utf-8'))

前提是我们首先利用上面的方式生成了 LWPCookieJar 格式的 Cookie ，然后利用 load() 方法，传入文件名称，后面同样的方法构建 handler 和 opener 即可。

运行结果正常输出百度网页的源代码。

使用urllib.error处理异常

urllib.error 模块定义了由 urllib.request 产生的异常。如果出现了问题， urllib.request 便会抛出 urllib.error

urllib.error.URLError

它继承自 OSError ，是 urllib.error 异常类的基类，由 urllib.request 产生的异常都可以通过捕获这个类来处理。

req = Req.Request(weburl,headers=headers)

try:

webPage = Req.urlopen(req)

except Error.URLError as e:

print(e.reason)

urllib.error.URLError是URLError的子类，专门用来处理HTTP请求错误，比如认证请求失败等等

它有三个属性

code， 返回状态码

reason， 同父类一样，返回错误的原因

headers，返回HTTP响应头部

因为URLError是HTTPError的父类，所以我们可以先选择捕获子类的错误，再去捕获父类的错误，更好的写法如下

try:

webPage = Req.urlopen(req)

except Error.HTTPError as e:

print(e.code,e.headers,e.reason)

except Error.URLError as e:

print(e.reason)

else:

print("Request Successfully")

这样我们就可以做到先捕获 HTTPError ，获取它的错误码，错误原因，服务器响应头等详细信息。如果非 HTTPError ，再捕获 URLError 错误，输出错误原因。最后用 else 来处理正常的逻辑。

使用urllib.parse解析链接

这个模块定义了处理 URL 的标准接口，例如实现 URL 各部分的抽取，合并以及链接转换。它支持如下类型的链接处理：

file 、 ftp 、 gopher 、 hdl 、 http 、 https 、 imap 、 mailto

、mms 、 news 、 nntp 、 prospero 、 rsync 、 rtsp 、 rtspu 、 sftp 、

shttp 、sip 、 sips 、 snews 、 svn 、 svn+ssh 、 telnet 、 wais

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

#链接解析

import urllib.request as Req

import urllib.parse as Par

weburl = 'image.baidu.com/search/index?tn=baiduimage&ct=201326592&lm=-1&cl=2&ie=gbk&word=%C3%C0%C5%AE&fr=ala&ala=1&alatpl=cover&pos=0&hs=2&xthttps=111111'

result = Par.urlparse(weburl,scheme='https')

print(result)

http://www.baidu.com/index.html;user?id=5#comment

urlparse() 方法将其拆分成了六部分，大体观察可以发现，解析时有特定的分隔符，比如 :// 前面的就是 scheme ，第一个 / 前面便是 netloc ，分号 ; 前面是 params 等等。

所以可以得出一个标准的链接格式如下：

scheme://netloc/path;parameters?query#fragment

除了这种最基本的解析方式， urlopen() 方法还有其他配置吗？接下来看一下它的 API 。

urllib.parse.urlparse(urlstring, scheme='', allow\_fragments=True)

它有三个参数：

第一个参数 urlstring 是必填项，即待解析的 URL 。

第二个参数 scheme 是默认的协议（比如 http 、 https 等），假如这个链接没有带协议信息，会将这个作为默认的协议。

urllib.parse.urlunparse()

有了 urlparse() 那相应地就有了它的对立方法 urlunparse() 。接受的参数是一个可迭代对象，但是它的长度必须是6，否则会抛出参数数量不足或者过多的问题。

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

#制作URL

import urllib.request as Req

import urllib.parse as Par

data = ['http','www.baidu.com','index.php','user','a=6','comment']

print(Par.urlunparse(data))

参数用了 list 类型，当然你也可以用其他的类型如 tuple 或者特定的数据结构。

urllib.parse.urlsplit()

这个和 urlparse() 方法非常相似，只不过它不会单独解析 parameters 这一部分，只返回五个结果。上面例子中的 parameters 会合并到 path 中。

urllib.parse.urlunsplit()

与 urlunparse() 类似，也是将链接的各个部分组合成完整链接的方法，传入的也是一个可迭代对象。 例如 list 、 tuple 等等，唯一的区别是，长度必须为5

urllib.parse.urljoin()

有了 urlunparse() 和 urlunsplit() 方法，我们可以完成链接的合并，不过前提必须要有特定长度的对象，链接的每一部分都要清晰分开。

生成链接还有另一个方法，利用 urljoin() 方法我们可以提供一个 base\_url （基础链接），新的链接作为第二个参数，方法会分析 base\_url 的 scheme 、 netloc 、 path 这三个内容对新链接缺失的部分进行补充，作为结果返回。

# coding=utf-8

from urllib.parse import urljoin

print(urljoin('http://www.baidu.com', 'FAQ.html'))

print(urljoin('http://www.baidu.com', 'https://cuiqingcai.com/FAQ.html'))

print(urljoin('http://www.baidu.com/about.html', 'https://cuiqingcai.com/FAQ.html'))

print(urljoin('http://www.baidu.com/about.html', 'https://cuiqingcai.com/FAQ.html?question=2'))

print(urljoin('http://www.baidu.com?wd=abc', 'https://cuiqingcai.com/index.php'))

print(urljoin('http://www.baidu.com', '?category=2#comment'))

print(urljoin('www.baidu.com', '?category=2#comment'))

print(urljoin('www.baidu.com#comment', '?category=2'))

运行结果

http://www.baidu.com/FAQ.html

https://cuiqingcai.com/FAQ.html

https://cuiqingcai.com/FAQ.html

https://cuiqingcai.com/FAQ.html?question=2

https://cuiqingcai.com/index.php

http://www.baidu.com?category=2#comment

www.baidu.com?category=2#comment

www.baidu.com?category=2

base\_url 提供了三项内容， scheme 、 netloc 、 path ，如果这三项在新的链接里面不存在，那么就予以补充，如果新的链接存在，那么就使用新的链接的部分。 base\_url 中的 parameters 、 query 、 fragments 是不起作用的。

使用urllib.robotparser分析robots协议

当一个搜索蜘蛛访问一个站点时，它首先会检查下这个站点根目录下是否存在 robots.txt 文件，如果存在，搜索蜘蛛会根据其中定义的爬取范围来爬取。如果没有找到这个文件，那么搜索蜘蛛便会访问所有可直接访问的页面。

User-agent: \*

Disallow: /

Allow: /public/

#-\*- coding=utf-8 -\*-

import urllib.robotparser as Rob

rp = Rob.RobotFileParser()

rp.set\_url('http://www.jianshu.com/robots.txt')

rp.read()

print(rp.can\_fetch('\*','http://www.jianshu.com/p/b67554025d7d'))

print(rp.can\_fetch('\*','http://www.jianshu.com/search?q=python&page=1&type=collections'))

使用requests

在使用了 urllib 之后，我们发现其中确实有不便捷的地方。比如处理网页验证、处理Cookies等等。那么在这里就有了更为强大的库 requests ，有了它，Cookies，登录验证，代理设置等等的操作都不是事儿。

所以如果我们请求一个 https 站点，但是证书验证错误的页面时，就会报这样的错误，那么如何避免这个错误呢？很简单，把 verify 这个参数设置为 False 即可。

response = requests.get('https://www.12306.cn',headers=headers,verify=False)

也可以指定证书

import requests

response = requests.get('https://www.12306.cn', cert=('/path/server.crt', '/path/key'))

print(response.status\_code)

**requests模块**

发送请求

r = request.get("http://www.baidu.com")

r = request.post("http://www.baidu.com")

传递URL参数

http://httpbin.org/get?age=22&name=alan

data = {'name':'alan','age':22}

r = requests.get('http://httpbin.org/get',params=data)

响应内容

r = request.get("http://www.baidu.com")

r.text

二进制响应内容

r.content

JSON响应内容

r.json()

原始响应内容

r = request.get("http://www.baidu.com"，stream=True)

r.raw

将文本保存到文件

with open(filename,‘wb’) as fd:

for chunk in r.iter\_content(chun\_size)：

fd.write(chunk)

使用 Response.iter\_content 将会处理大量你直接使用 Response.raw 不得不处理的。 当流下载时，上面是优先推荐的获取内容方式

定制请求头

headers = {

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

data = Req.get(url,headers=headers)

POST请求

data = {'name':'alan','age':22}

r = requests.post('http://httpbin.org/get',data=data)

很多时候你想要发送的数据并非编码为表单形式的。如果你传递一个 string 而不是一个 dict，那么数据会被直接发布出去。

POST上传文件

import requests

files = {'file':open('favicon.ico','rb')}

r = requests.post("http://httpbin.org/post",files=files)

print(r.text)

响应状态码

r.status\_code

响应头

r.headers

Cookie

r.cookie

import requests

headers = {

"Cookie":"username = kobe;\_\_jsluid = f7eff1b852bb806ff8f236fecc765f48;UM\_distinctid = 15b196a757276 - 0c125d61293ed8 - 44504130 - 1fa400 - 15b196a7573373; CNZZDATA1256179677 = 300334533 - 1490779055 - % 7C1492156578;CNZZDATA1254790094 = 1457296691 - 1490774904 - % 7C1492157709; Hm\_lvt\_f155537cd96f930920ef61528f9b57db = 1490780059, 1492054812, 1492138306, 1492158691;SECheck = 344209d118cb09c0b458e75060ac2e77ca6fef09; ASP.NET\_SessionId = xhddqhndaiitorrtqzvwqsue;SRV = b0937cd0ac78e79801;Hm\_lpvt\_f155537cd96f930920ef61528f9b57db=1492158691",

"Accept": "\*/\*",

"Accept-Language": "zh-CN,zh;q=0.8,en-US;q=0.5,en;q=0.3",

"Connection": "keep-alive",

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

}

r =requests.get('http://www.rjb777.com/Member/MemberCenter.aspx?SecondAnniversary=0&BillAlert=0',headers=headers))

print(r.text)

请求历史

r.url

重定向

data = Req.get(url,headers=headers,allow\_redirects=True)

超时

高级用法

会话对象

import requests

s = requests.Session()

s.get("http://httpbin.org/cookies/set/number/123456789")

r = s.get('http://httpbin.org/cookies')

print(r.text)

返回结果

{

"cookies": {

"number": "123456789"

}

}

请求与响应对象

任何时候调用 requests.\*() 你都在做两件主要的事情。其一，你在构建一个 Request 对象， 该对象将被发送到某个服务器请求或查询一些资源。其二，一旦 requests 得到一个从 服务器返回的响应就会产生一个 Response 对象。该响应对象包含服务器返回的所有信息， 也包含你原来创建的 Request 对象

r = requests.get('http://en.wikipedia.org/wiki/Monty\_Python')

r.headers 响应头部

r.request.headers 请求头部

准备的请求

当你从 API 或者会话调用中收到一个 Response 对象时，request 属性其实是使用了 PreparedRequest。有时在发送请求之前，你需要对 body 或者 header （或者别的什么东西）做一些额外处理，下面演示了一个简单的做法：

SSL证书验证

response = requests.get('https://www.12306.cn',headers=headers,verify=False)

也可以指定证书

import requests

response = requests.get('https://www.12306.cn', cert=('/path/server.crt', '/path/key'))

print(response.status\_code)

本地证书的私有 key 必须是解密状态。目前，Requests 不支持使用加密的 key。

CA 证书

Requests 默认附带了一套它信任的根证书，来自于 Mozilla trust store。然而它们在每次 Requests 更新时才会更新。这意味着如果你固定使用某一版本的 Requests，你的证书有可能已经 太旧了。

从 Requests 2.4.0 版之后，如果系统中装了 certifi 包，Requests 会试图使用它里边的 证书。这样用户就可以在不修改代码的情况下更新他们的可信任证书。

为了安全起见，我们建议你经常更新 certifi！

响应体内容工作流

默认情况下，当你进行网络请求后，响应体会立即被下载。你可以通过 stream 参数覆盖这个行为，推迟下载响应体直到访问 Response.content 属性

tarball\_url = 'https://github.com/kennethreitz/requests/tarball/master'

r = requests.get(tarball\_url, stream=True)

此时仅有响应头被下载下来了，连接保持打开状态，因此允许我们根据条件获取内容：

if int(r.headers['content-length']) < TOO\_LONG:

content = r.content

...

你可以进一步使用 Response.iter\_content 和 Response.iter\_lines 方法来控制工作流，或者以 Response.raw 从底层 urllib3 的 urllib3.HTTPResponse <urllib3.response.HTTPResponse 读取。

如果你在请求中把 stream 设为 True，Requests 无法将连接释放回连接池，除非你 消耗了所有的数据，或者调用了 Response.close。 这样会带来连接效率低下的问题。如果你发现你在使用 stream=True 的同时还在部分读取请求的 body（或者完全没有读取 body），那么你就应该考虑使用 contextlib.closing (文档)， 如下所示：

from contextlib import closing

with closing(requests.get('http://httpbin.org/get', stream=True)) as r:

# 在此处理响应。

保持活动状态（持久连接）

同一会话内的持久连接是完全自动处理的！同一会话内你发出的任何请求都会自动复用恰当的连接！

注意：只有所有的响应体数据被读取完毕连接才会被释放为连接池；所以确保将 stream 设置为 False 或读取 Response 对象的 content 属性。

流式上传

Requests支持流式上传，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)

警告

我们强烈建议你用二进制模式（binary mode）打开文件。这是因为 requests 可能会为你提供 header 中的 Content-Length，在这种情况下该值会被设为文件的字节数。如果你用文本模式打开文件，就可能碰到错误。

块编码请求

对于出去和进来的请求，Requests 也支持分块传输编码。要发送一个块编码的请求，仅需为你的请求体提供一个生成器（或任意没有具体长度的迭代器）：

def gen():

yield 'hi'

yield 'there'

requests.post('http://some.url/chunked', data=gen())

对于分块的编码请求，我们最好使用 Response.iter\_content() 对其数据进行迭代。在理想情况下，你的 request 会设置 stream=True，这样你就可以通过调用 iter\_content 并将分块大小参数设为 None，从而进行分块的迭代。如果你要设置分块的最大体积，你可以把分块大小参数设为任意整数。

POST 多个分块编码的文件

你可以在一个请求中发送多个文件。例如，假设你要上传多个图像文件到一个 HTML 表单，使用一个多文件 field 叫做 "images":

<input type="file" name="images" multiple="true" required="true"/>

要实现，只要把文件设到一个元组的列表中，其中元组结构为 (form\_field\_name, file\_info):

>>> url = 'http://httpbin.org/post'

>>> multiple\_files = [

('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),

('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]

>>> r = requests.post(url, files=multiple\_files)

>>> r.text

{

...

'files': {'images': 'data:image/png;base64,iVBORw ....'}

'Content-Type': 'multipart/form-data; boundary=3131623adb2043caaeb5538cc7aa0b3a',

...

}

警告

我们强烈建议你用二进制模式（binary mode）打开文件。这是因为 requests 可能会为你提供 header 中的 Content-Length，在这种情况下该值会被设为文件的字节数。如果你用文本模式打开文件，就可能碰到错误。

事件挂钩

Requests有一个钩子系统，你可以用来操控部分请求过程，或信号事件处理。

可用的钩子:

response:

从一个请求产生的响应

你可以通过传递一个 {hook\_name: callback\_function} 字典给 hooks 请求参数 为每个请求分配一个钩子函数：

hooks=dict(response=print\_url)

callback\_function 会接受一个数据块作为它的第一个参数。

def print\_url(r, \*args, \*\*kwargs):

print(r.url)

若执行你的回调函数期间发生错误，系统会给出一个警告。

若回调函数返回一个值，默认以该值替换传进来的数据。若函数未返回任何东西， 也没有什么其他的影响。

我们来在运行期间打印一些请求方法的参数：

>>> requests.get('http://httpbin.org', hooks=dict(response=print\_url))

http://httpbin.org

<Response [200]>

自定义身份验证

Requests 允许你使用自己指定的身份验证机制。

任何传递给请求方法的 auth 参数的可调用对象，在请求发出之前都有机会修改请求。

自定义的身份验证机制是作为 requests.auth.AuthBase 的子类来实现的，也非常容易定义。Requests 在 requests.auth 中提供了两种常见的的身份验证方案： HTTPBasicAuth 和 HTTPDigestAuth 。

假设我们有一个web服务，仅在 X-Pizza 头被设置为一个密码值的情况下才会有响应。虽然这不太可能，但就以它为例好了。

from requests.auth import AuthBase

class PizzaAuth(AuthBase):

"""Attaches HTTP Pizza Authentication to the given Request object."""

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

# setup any auth-related data here

self.username = username

def \_\_call\_\_(self, r):

# modify and return the request

r.headers['X-Pizza'] = self.username

return r

然后就可以使用我们的PizzaAuth来进行网络请求:

>>> requests.get('http://pizzabin.org/admin', auth=PizzaAuth('kenneth'))

<Response [200]>

流式请求

使用 requests.Response.iter\_lines() 你可以很方便地对流式 API （例如 Twitter 的流式 API ） 进行迭代。简单地设置 stream 为 True 便可以使用 iter\_lines() 对相应进行迭代：

import json

import requests

r = requests.get('http://httpbin.org/stream/20', stream=True)

for line in r.iter\_lines():

# filter out keep-alive new lines

if line:

print(json.loads(line))

警告

iter\_lines() 不保证重进入时的安全性。多次调用该方法 会导致部分收到的数据丢失。如果你要在多处调用它，就应该使用生成的迭代器对象:

lines = r.iter\_lines()

# Save the first line for later or just skip it

first\_line = next(lines)

for line in lines:

print(line)

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 = {

"http": "http://10.10.1.10:3128",

"https": "http://10.10.1.10:1080",

}

requests.get("http://example.org", proxies=proxies)

你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"

$ python

>>> import requests

>>> requests.get("http://example.org")

若你的代理需要使用HTTP Basic Auth，可以使用 http://user:password@host/ 语法：

proxies = {

"http": "http://user:pass@10.10.1.10:3128/",

}

要为某个特定的连接方式或者主机设置代理，使用 scheme://hostname 作为 key， 它会针对指定的主机和连接方式进行匹配。

proxies = {'http://10.20.1.128': 'http://10.10.1.10:5323'}

注意，代理 URL 必须包含连接方式。

SOCKS

2.10.0 新版功能.

除了基本的 HTTP 代理，Request 还支持 SOCKS 协议的代理。这是一个可选功能，若要使用， 你需要安装第三方库。

你可以用 pip 获取依赖:

$ pip install requests[socks]

安装好依赖以后，使用 SOCKS 代理和使用 HTTP 代理一样简单：

proxies = {

'http': 'socks5://user:pass@host:port',

'https': 'socks5://user:pass@host:port'

}

基本身份认证

许多要求身份认证的web服务都接受 HTTP Basic Auth。这是最简单的一种身份认证，并且 Requests 对这种认证方式的支持是直接开箱即可用。

以 HTTP Basic Auth 发送请求非常简单：

>>> from requests.auth import HTTPBasicAuth

>>> requests.get('https://api.github.com/user', auth=HTTPBasicAuth('user', 'pass'))

<Response [200]>

事实上，HTTP Basic Auth 如此常见，Requests 就提供了一种简写的使用方式：

>>> requests.get('https://api.github.com/user', auth=('user', 'pass'))

<Response [200]>

像这样在一个元组中提供认证信息与前一个 HTTPBasicAuth 例子是完全相同的。

netrc 认证

如果认证方法没有收到 auth 参数，Requests 将试图从用户的 netrc 文件中获取 URL 的 hostname 需要的认证身份。

如果找到了 hostname 对应的身份，就会以 HTTP Basic Auth 的形式发送请求。

摘要式身份认证?

另一种非常流行的 HTTP 身份认证形式是摘要式身份认证，Requests 对它的支持也是开箱即可用的：

>>> from requests.auth import HTTPDigestAuth

>>> url = 'http://httpbin.org/digest-auth/auth/user/pass'

>>> requests.get(url, auth=HTTPDigestAuth('user', 'pass'))

<Response [200]>

OAuth 1 认证

Oauth 是一种常见的 Web API 认证方式。 requests-oauthlib 库可以让 Requests 用户简单地创建 OAuth 认证的请求：

::

>>> import requests

>>> from requests\_oauthlib import OAuth1

>>> url = 'https://api.twitter.com/1.1/account/verify\_credentials.json'

>>> auth = OAuth1('YOUR\_APP\_KEY', 'YOUR\_APP\_SECRET',

'USER\_OAUTH\_TOKEN', 'USER\_OAUTH\_TOKEN\_SECRET')

>>> requests.get(url, auth=auth)

<Response [200]>

关于 OAuth 工作流程的更多信息，请参见 OAuth 官方网站。 关于 requests-oauthlib 的文档和用例，请参见 GitHub 的 requests\_oauthlib 代码库。

其他身份认证形式

Requests 的设计允许其他形式的身份认证用简易的方式插入其中。开源社区的成员时常为更复杂或不那么常用的身份认证形式编写认证处理插件。其中一些最优秀的已被收集在 Requests organization 页面中，包括:

Kerberos

NTLM

如果你想使用其中任何一种身份认证形式，直接去它们的GitHub页面，依照说明进行。

新的身份认证形式

如果你找不到所需要的身份认证形式的一个良好实现，你也可以自己实现它。Requests 非常易于添加你自己的身份认证形式。

要想自己实现，就从 requests.auth.AuthBase 继承一个子类，并实现 \_\_call\_\_() 方法：

>>> import requests

>>> class MyAuth(requests.auth.AuthBase):

... def \_\_call\_\_(self, r):

... # Implement my authentication

... return r

...

>>> url = 'http://httpbin.org/get'

>>> requests.get(url, auth=MyAuth())

<Response [200]>

当一个身份认证模块被附加到一个请求上，在设置 request 期间就会调用该模块。因此 \_\_call\_\_ 方法必须完成使得身份认证生效的所有事情。一些身份认证形式会额外地添加钩子来提供进一步的功能。

**BS4模块**

pip install beautifulsoup4

pip install lxml

pip install html5lib

beautifulsoup4是一个工具箱，通过它解析文档可以为我们十分简单的体改需要抓取的数据；它自动将我们输入的文档转换为Unicode编码，输出时转换为UTF-8编码，我们不用考虑操蛋的文本解析编码方式

html\_doc = """

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

<body>

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

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

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

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

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

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

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

"""

几个简单的流量结构化数据的方法

soup.title

soup.title.name

soup.title.string

soup.p

soup.p['class']

soup.a

soup.find\_all('a')

soup.find(id='link3')

从文档中找到所有<a>标签的链接:

for link in soup.find\_all('a'):

print(link.get('href'))

# http://example.com/elsie

# http://example.com/lacie

# http://example.com/tillie

从文档中获取所有文字内容:

print(soup.get\_text())

# The Dormouse's story

#

# The Dormouse's story

#

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

# Elsie,

# Lacie and

# Tillie;

# and they lived at the bottom of a well.

#

# ...