# 请求库

## urllib

urllib库是Python内置的HTTP请求库。包含如下4个模块。

* request：它是最基本的HTTP请求模块，可以用来模拟发送请求。就像在浏览器里输入网址然后回车一样，只需要给库方法传入URL以及额外的参数，就可以模拟实现这个过程了。
* error：异常处理模块，如果出现请求错误，我们可以捕获这些异常，然后进行重试或其他操作以保证程序不会意外终止。
* parse：一个工具模块，提供了许多URL处理方法，比如拆分、解析、合并等。
* robotparser：主要是用来识别网站的robots.txt文件，然后判断哪些网站可以爬，哪些网站不可以爬，用得比较少。

### 发送请求

#### User Agent和代理IP隐藏身份

User Agent的中文名为用户代理，简称UA。User Agent的中文名为用户代理，简称UA。

常见的User Agent

1.Android

Mozilla/5.0 (Linux; Android 4.1.1; Nexus 7 Build/JRO03D) AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Safari/535.19

2.Firefox

Mozilla/5.0 (Windows NT 6.2; WOW64; rv:21.0) Gecko/20100101 Firefox/21.0

3.Google Chrome

Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/27.0.1453.94 Safari/537.36

4.iOS

Mozilla/5.0 (iPad; CPU OS 5\_0 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Version/5.1 Mobile/9A334 Safari/7534.48.3

#### request模块

**urllib.request模块**

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

**提供了最基本的构造HTTP请求的方法**

模拟浏览器的一个请求发起过程，同时它还带有处理授权验证（authenticaton）、重定向（redirection)、浏览器Cookies以及其他内容。

from urllib import request  
if \_\_name\_\_ == "\_\_main\_\_":  
 # 向指定的url发送请求，并返回服务器响应的类文件对象  
 response = request.urlopen("https://my.alipay.com/portal/i.htm")  
 html = response.read()  
 print(html.decode('utf-8'))

urlopen()返回的对象为HTTPResposne类型，可以使用read()进行读取，调用status属性可以得到返回结果的状态码

同样也可以使用geturl()方法、info()方法、getcode()方法。

geturl()返回的是一个url的字符串；

info()返回的是一些meta标记的元信息，包括一些服务器的信息；

getcode()返回的是HTTP的状态码，如果返回200表示请求成功。

#### Request

请求中需要加入Headers等信息，就可以利用更强大的Request类来构建

**urllib.request.Request(url, data=None, headers={}, origin\_req\_host=None, unverifiable=False, method=None)**

* 第一个参数url用于请求URL，这是必传参数，其他都是可选参数。
* 第二个参数data如果要传，必须传bytes（字节流）类型的。如果它是字典，可以先用urllib.parse模块里的urlencode()编码。
* 第三个参数headers是一个字典，它就是请求头，我们可以在构造请求时通过headers参数直接构造，也可以通过调用请求实例的add\_header()方法添加。
* 第四个参数origin\_req\_host指的是请求方的host名称或者IP地址。
* 第五个参数unverifiable表示这个请求是否是无法验证的，默认是False，意思就是说用户没有足够权限来选择接收这个请求的结果。例如，我们请求一个HTML文档中的图片，但是我们没有自动抓取图像的权限，这时unverifiable的值就是True`。
* 第六个参数method是一个字符串，用来指示请求使用的方法，比如GET、POST和PUT等。

*from* urllib *import* request, parse  
url = 'http://httpbin.org/post'  
headers = {  
 'User-Agent': 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)',  
 'Host': 'httpbin.org'  
}  
dict = {  
 'name': 'Germey'  
}  
data = bytes(parse.urlencode(dict), encoding='utf8')  
req = request.Request(url=url, data=data, headers=headers, method='POST')  
response = request.urlopen(req)  
print(response.read().decode('utf-8'))

headers也可以用add\_header()方法来添加

req = request.Request(url=url, data=data, method='POST')  
req.add\_header('User-Agent', 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)')

#### Handler

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

* HTTPDefaultErrorHandler：用于处理HTTP响应错误，错误都会抛出HTTPError类型的异常。
* HTTPRedirectHandler：用于处理重定向。
* HTTPCookieProcessor：用于处理Cookies。
* ProxyHandler：用于设置代理，默认代理为空。
* HTTPPasswordMgr：用于管理密码，它维护了用户名和密码的表。
* 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。

#### OpenerDirector

称为Opener，urlopen()实际上就是urllib提供的一个Opener，Request和urlopen()相当于类库为你封装好了极其常用的请求方法，利用它们可以完成基本的请求，需要实现更高级的功能需要深入一层进行配置，使用更底层的实例来完成操作，就用到了Opener，Handler来构建Opener

##### 代理设置

利用这个Handler及build\_opener()方法构造一个Opener，之后发送请求即可

*from* urllib.error *import* URLError  
*from* urllib.request *import* ProxyHandler, build\_opener  
  
proxy\_handler = ProxyHandler({  
 'http': 'http://127.0.0.1:9743',  
 'https': 'https://127.0.0.1:9743'  
})  
opener = build\_opener(proxy\_handler)  
*try*:  
 response = opener.open('https://www.baidu.com')  
 print(response.read().decode('utf-8'))  
*except* URLError *as* e:  
 print(e.reason)

##### Cookies

**将网站的Cookies获取下来**

声明一个CookieJar对象，利用HTTPCookieProcessor来构建一个Handler，最后利用build\_opener()方法构建出Opener，执行open()函数

1. 变量方式保存

*import* http.cookiejar, urllib.request  
  
cookie = http.cookiejar.CookieJar()  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')

1. 文本形式保存

CookieJar需要换成MozillaCookieJar，它在生成文件时会用到，是CookieJar的子类，可以用来处理Cookies和文件相关的事件，将Cookies保存成Mozilla型浏览器的Cookies格式

filename = 'cookies.txt'  
cookie = http.cookiejar.MozillaCookieJar(filename)  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')  
cookie.save(ignore\_discard=*True*, ignore\_expires=*True*)

**读取cookies**

cookie = http.cookiejar. MozillaCookieJar ()  
cookie.load('cookies.txt', ignore\_discard=*True*, ignore\_expires=*True*)  
handler = urllib.request.HTTPCookieProcessor(cookie)  
opener = urllib.request.build\_opener(handler)  
response = opener.open('http://www.baidu.com')  
print(response.read().decode('utf-8'))

### 处理异常

urllib的error模块定义了由request模块产生的异常。如果出现了问题，request模块便会抛出error模块中定义的异常

urllib.error有两个方法，URLError和HTTPError。

URLError是OSError的一个子类，HTTPError是URLError的一个子类，服务器上HTTP的响应会返回一个状态码，根据这个HTTP状态码，我们可以知道我们的访问是否成功。

#### URLError

由request模块生的异常都可以通过捕获这个类来处理，属性reason，返回错误的原因

*from* urllib *import* request, error  
*try*:  
 response = request.urlopen('http://cuiqingcai.com/index.htm')  
*except* error.URLError *as* e:  
 print(e.reason)

#### HTTPError

专门用来处理HTTP请求错误，比如认证请求失败等，有如下3个属性

* code：返回HTTP状态码，比如404表示网页不存在，500表示服务器内部错误等。
* reason：同父类一样，用于返回错误的原因。
* headers：返回请求头。

*from* urllib *import* request,error  
*try*:  
 response = request.urlopen('http://cuiqingcai.com/index.htm')  
*except* error.HTTPError *as* e:  
 print(e.reason, e.code, e.headers, sep='\n')

#### URLError和HTTPError混合使用

# URLError  
url = 'http://www.iloveyou.com/' # 一个不存在的连接  
# HTTPError  
url = 'http://www.douyu.com/Jack.html' # 请求的资源没有在服务器上找 到,www.douyu.com这个服务器是存在的，但是我们要查找的Jack.html资源是没有的  
req = request.Request(url)  
*try*:  
 response = request.urlopen(req)  
*except* error.URLError *as* e:  
 print(e.reason)  
*except* error.HTTPError *as* e:  
 print(e.code)

如果想用HTTPError和URLError一起捕获异常，那么需要将HTTPError放在URLError的前面，因为HTTPError是URLError的一个子类。如果URLError放在前面，出现HTTP异常会先响应URLError，这样HTTPError就捕获不到错误信息了。

也可以使用hasattr函数判断URLError含有的属性，如果含有reason属性表明是URLError，如果含有code属性表明是HTTPError

### 解析链接

urllib库里还提供parse模块，定义了处理URL的标准接口

#### urlparse()

实现URL的识别和分段，根据分隔符将URL拆分

result = urlparse('http://www.baidu.com/index.html#comment', allow\_fragments=False)

print(result)

运行结果如下：

ParseResult(scheme='http', netloc='www.baidu.com', path='/index.html#comment', params='', query='', fragment='')

#### urlunparse()

实现URL的构造，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

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

#### urlsplit()

和urlparse()方法非常相似，只不过它不再单独解析params这一部分，只返回5个结果。

#### urljoin()

生成链接，提供一个base\_url（基础链接）作为第一个参数，将新的链接作为第二个参数，该方法会分析base\_url的scheme、netloc和path这3个内容并对新链接缺失的部分进行补充，最后返回结果

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

#### urlencode()

params = {  
 'name': 'germey',  
 'age': 22  
}  
base\_url = 'http://www.baidu.com?'  
url = base\_url + urlencode(params)

声明一个字典来将参数表示出来，然后调用urlencode()方法将其序列化为GET请求参数

#### parse\_qs()

GET请求参数，利用parse\_qs()方法，就可以将它转回字典

#### parse\_qsl()

将参数转化为元组组成的列表

query = 'name=germey&age=22'

print(parse\_qsl(query))

运行结果如下：

[('name', 'germey'), ('age', '22')]

#### quote()

URL中带有中文参数时，有时可能会导致乱码的问题，此时用这个方法可以将中文字符转化为URL编码

keyword = '壁纸'

url = 'https://www.baidu.com/s?wd=' + quote(keyword)

print(url)

运行结果如下：

<https://www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8>

#### unquote()

进行URL解码

url = 'https://www.baidu.com/s?wd=%E5%A3%81%E7%BA%B8'

print(unquote(url))

### Robots协议

Robots协议也称作爬虫协议、机器人协议，它的全名叫作网络爬虫排除标准（Robots Exclusion Protocol），用来告诉爬虫和搜索引擎哪些页面可以抓取，哪些不可以抓取。它通常是一个叫作robots.txt的文本文件，一般放在网站的根目录下。

当搜索爬虫访问一个站点时，它首先会检查这个站点根目录下是否存在robots.txt文件，如果存在，搜索爬虫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，搜索爬虫便会访问所有可直接访问的页面。

User-agent: \*

Disallow: /

Allow: /public/

User-agent描述了搜索爬虫的名称，这里将其设置为\*则代表该协议对任何爬取爬虫有效。

Disallow指定了不允许抓取的目录，比如上例子中设置为/则代表不允许抓取所有页面。若全部允许，设置为空

Allow一般和Disallow一起使用，一般不会单独使用，用来排除某些限制。设置为/public/，则表示所有页面不允许抓取，但可以抓取public目录。

#### robotparser

该模块提供了一个类RobotFileParser，它可以根据某网站的robots.txt文件来判断一个爬取爬虫是否有权限来爬取这个网页。

**urllib.robotparser.RobotFileParser(url='')**

也可以在声明时不传入，默认为空，最后再使用set\_url()方法设置

* set\_url()：用来设置robots.txt文件的链接。如果在创建RobotFileParser对象时传入了链接，那么就不需要再使用这个方法设置了。
* read()：读取robots.txt文件并进行分析。注意，这个方法执行一个读取和分析操作，如果不调用这个方法，接下来的判断都会为False，所以一定记得调用这个方法。这个方法不会返回任何内容，但是执行了读取操作。
* parse()：用来解析robots.txt文件，传入的参数是robots.txt某些行的内容，它会按照robots.txt的语法规则来分析这些内容。
* can\_fetch()：该方法传入两个参数，第一个是User-agent，第二个是要抓取的URL。返回的内容是该搜索引擎是否可以抓取这个URL，返回结果是True或False。
* mtime()：返回的是上次抓取和分析robots.txt的时间，这对于长时间分析和抓取的搜索爬虫是很有必要的，你可能需要定期检查来抓取最新的robots.txt。
* modified()：它同样对长时间分析和抓取的搜索爬虫很有帮助，将当前时间设置为上次抓取和分析robots.txt的时间。

*from* urllib.robotparser *import* RobotFileParser  
  
rp = 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"))

## requests



### get（）



1. 修改headers字段、cookie、代理

cookies={}

#自动保持  
s = requests.session()

#打开cookie文件  
*with* open('E:\example\豆瓣读书爬虫\cookie.txt')*as* file:  
 raw\_cookies=file.read()  
 *for* line *in* raw\_cookies.split(';'):  
 key,value=line.split('=',1)  
 cookies[key]=value  
proxie = {'http': 'http://14.118.254.31:6666'}

headers={'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0'}  
res=s.get(url=*url*,cookies=cookies,headers=headers,proxies=proxie)

构造合理HTTP请求头

if \_\_name\_\_ == '\_\_main\_\_':  
 url='https://www.baidu.com/'  
 header= {'Upgrade-Insecure-Requests':'1',  
 'User-Agent':'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36',  
 'Accept':'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8',  
 'Accept-Encoding':'gzip, deflate, sdch, br',  
 'Accept-Language':'zh-CN,zh;q=0.8',  
 }  
 s=requests.Session()  
 req=s.get(url=url,headers=header)  
 print(s.cookies)

1. 修改params参数提交关键词

百度的关键词接口：http://www.baidu.com/s?wd=keyword

360的关键词接口：<http://www.so.com/s?q=keyword>

url="http://www.baidu.com/s"   
kv={'wd':'Python'}   
r=requests.get(url,params=kv)

1. 获取cookie

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

1. 发送cookie

url = 'http://httpbin.org/cookies'  
cookies = {'testCookies\_1': 'Hello\_Python3', 'testCookies\_2': 'Hello\_Requests'}  
# 在Cookie Version 0中规定空格、方括号、圆括号、等于号、逗号、双引号、斜杠、问号、@，冒号，分号等特殊符号都不能作为Cookie的内容。  
r = requests.get(url, cookies=cookies)

1. 证书验证

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

1. 超时设置

*from* requests.exceptions *import* ReadTimeout  
*try*:  
 response = requests.get("http://httpbin.org/get", timeout = 0.5)  
 print(response.status\_code)  
*except* ReadTimeout:  
 print('Timeout')

1. 身份认证

*import* requests  
*from* requests.auth *import* HTTPBasicAuth  
r = requests.get('http://localhost:5000', auth=HTTPBasicAuth('username', 'password'))

如果用户名和密码正确的话，请求时就会自动认证成功，会返回200状态码，如果认证失败，则返回401状态码。requests提供了一个更简单的写法，可以直接传一个元组，它会默认使用HTTPBasicAuth这个类来认证。

r = requests.get('http://localhost:5000', auth=('username', 'password'))

1. 重定向与历史信息

r = requests.get('https://www.12306.cn', allow\_redirects=*True* '))

allow\_redirects为true，允许重定向，false不允许

如果允许重定向，可通过r.history查看历史信息，即访问成功前所有请求跳转信息。

**requests库的response对象：**

response对象包含服务器返回的所有信息，也包含请求的request信息。



**抓取网页**

如果请求普通的网页,调用.text属性，获得相应的内容

如果返回的是json格式，调用json()方法变为字典格式

**抓取二进制数据**

图片、音频、视频这些文件本质上都是由二进制码组成的，想要抓取它们，就要调用.content拿到它们的二进制码，为bytes类型

*import* requests  
  
r = requests.get("https://github.com/favicon.ico")  
*with* open('favicon.ico', 'wb') *as* f:  
 f.write(r.content)

open()方法，第一个参数是文件名称，第二个参数代表以二进制写的形式打开，可以向文件里写入二进制数据。

音频和视频文件也可以用这种方法获取

### post（）

数据通过post提交

*import* requests  
  
data = {'name': 'germey', 'age': '22'}  
r = requests.post("http://httpbin.org/post", data=data)  
print(r.text)

### 文件上传

*import* requests  
files = {'file': open('favicon.ico', 'rb')}  
r = requests.post("http://httpbin.org/post", files=files)  
print(r.text)

文件需要和当前脚本在同一目录下，文件上传部分会单独有一个files字段来标识。

### 会话维持

requests中，如果直接利用get()或post()等方法的确可以做到模拟网页的请求，但是这实际上是相当于不同的会话，也就是说相当于你用了两个浏览器打开了不同的页面。

第一个请求利用post()方法登录了某个网站，第二次想获取成功登录后的自己的个人信息，你又用了一次get()方法去请求个人信息页面。实际上，这相当于打开了两个浏览器，是两个完全不相关的会话。

解决这个问题的主要方法就是维持同一个会话，也就是相当于打开一个新的浏览器选项卡而不是新开一个浏览器

Session对象可以方便地维护一个会话，而且不用担心cookies的问题

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

利用Session，可以做到模拟同一个会话而不用担心Cookies的问题。它通常用于模拟登录成功之后再进行下一步的操作。

Prepared Request

*from* requests *import* Request, Session  
url = 'http://httpbin.org/post'  
data = {  
 'name': 'germey'  
}  
headers = {  
 'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.116 Safari/537.36'  
}  
s = Session()  
req = Request('POST', url, data=data, headers=headers)  
prepped = s.prepare\_request(req)  
r = s.send(prepped)  
print(r.text)

引入Request，用url、data和headers参数构造了一个Request对象，再调用Session的prepare\_request()方法将其转换为一个Prepared Request对象，然后调用send()方法发送

## selenium

自动化测试工具。它支持各种浏览器。如果用Selenium来驱动浏览器加载网页的话，就可以直接拿到JavaScript渲染的结果了

### webdriver

#模拟提交搜索  
browser = webdriver.Firefox()  
#打开请求的URL，webdriver会等到页面全部加载完毕后返回  
browser.get('http://www.baidu.com/')  
assert '百度一下，你就知道' in browser.title  
#find\_element\_by\_\*方法寻找网页元素  
elem=browser.find\_element\_by\_name('wd')  
#模拟键盘输入  
elem.send\_keys('python')  
#模拟回车  
elem.send\_keys(Keys.RETURN)  
#打印提交后页面信息  
print(browser.page\_source)

#### 声明浏览器对象

的初始化并将其赋值为browser对象

from selenium import webdriver

browser = webdriver.Chrome()

browser = webdriver.Firefox()

browser = webdriver.Edge()

browser = webdriver.PhantomJS()

browser = webdriver.Safari()

#### 设置user-agent

针对phantomJS修改

dcap=dict(DesiredCapabilities.PHANTOMJS)  
dcap['phantomjs .page.settings.userAgent']=(  
 'Mozilla/5.0(Linux;Android 5.1.1;Nexus 6 Build/LY228E)nAppleWebKit/537.36(KHTML,like Gecko)Chrome/48.0.2564.23 Mobile Safari/537.36')  
driver=webdriver .PhantomJS() # desired\_capabilzties=dcap)  
driver .get('http://www .google .com')  
driver .quit()

#### 访问页面

get()方法来请求网页，参数传入链接URL即可。

browser.get('https://www.taobao.com')

browser.page\_source page\_source输出网页源码

#### 添加cookie

driver.add\_cookie({'name':'51zxw','value':'www.51zxw.net'})

browser = webdriver.Chrome()  
browser.get('https://www.zhihu.com/explore')  
browser.get\_cookies()  
browser.add\_cookie({'name': 'name', 'domain': 'www.zhihu.com', 'value': 'germey'})  
browser.delete\_all\_cookies()

添加一个Cookie，这里传入一个字典，有name、domain和value等内容

接下来，再次获取所有的Cookies。可以发现，结果就多了这一项新加的Cookie。

#### 获取cookie

加载完成后，浏览器实际上已经生成Cookies了。接着，调用get\_cookies()方法获取所有的Cookies。

**cookie=driver.get\_cookies()**

* get\_cookies() 获得所有cookie 信息
* get\_cookie(name) 返回特定name 有cookie 信息
* add\_cookie(cookie\_dict) 添加cookie，必须有name 和value 值
* delete\_cookie(name) 删除特定(部分)的cookie 信息
* delete\_all\_cookies() 删除所有cookie 信息
* print(cookie) 打印全部cookile信息
* print(cookie[0]) 打印cookie第一组信息

#### 选项卡管理

browser = webdriver.Chrome()  
browser.get('https://www.baidu.com')  
browser.execute\_script('window.open()')  
print(browser.window\_handles)  
browser.switch\_to\_window(browser.window\_handles[1])

调用execute\_script()方法，传入window.open()这个JavaScript语句新开启一个选项卡。

window\_handles属性获取当前开启的所有选项卡，返回的是选项卡的代号列表

切换选项卡，调用switch\_to\_window()方法即可，其中参数是选项卡的代号

#### 弹窗处理

处理页面中触发事件跳出弹框，使用switch\_to\_alert获取弹框对象，进行关闭和获取信息操作

alert=driver. switch\_to\_alert()

alert.dismiss()

#### 延时等待

get()方法会在网页框架加载结束后结束执行，此时如果获取page\_source，可能并不是浏览器完全加载完成的页面

1. 隐式等待 implicitly\_wait()

当查找节点而节点并没有立即出现的时候，隐式等待将等待一段时间再查找DOM，默认的时间是0，超出设定时间后，则抛出找不到节点的异常

browser = webdriver.Chrome()  
browser.implicitly\_wait(10)  
browser.get('https://www.zhihu.com/explore')  
input = browser.find\_element\_by\_class\_name('zu-top-add-question')

1. 显式等待

它指定要查找的节点，然后指定一个最长等待时间。如果在规定时间内加载出来了这个节点，就返回查找的节点；如果到了规定时间依然没有加载出该节点，则抛出超时异常

*from* selenium *import* webdriver  
*from* selenium.webdriver.common.by *import* By  
*from* selenium.webdriver.support.ui *import* WebDriverWait  
*from* selenium.webdriver.support *import* expected\_conditions *as* EC  
browser = webdriver.Chrome()  
browser.get('https://www.taobao.com/')  
wait = WebDriverWait(browser, 10)  
input = wait.until(EC.presence\_of\_element\_located((By.ID, 'q')))  
button = wait.until(EC.element\_to\_be\_clickable((By.CSS\_SELECTOR, '.btn-search')))  
print(input, button)

引入WebDriverWait这个对象，指定最长等待时间

调用它的until()方法，传入要等待条件expected\_conditions

presence\_of\_element\_located这个条件，代表节点出现的意思，其参数是节点的定位元组，也就是ID为q的节点搜索框

效果就是，在10秒内如果ID为q的节点（即搜索框）成功加载出来，就返回该节点；如果超过10秒还没有加载出来，就抛出TimeoutException异常。

| **等待条件** | **含义** |
| --- | --- |
| title\_is | 标题是某内容 |
| title\_contains | 标题包含某内容 |
| presence\_of\_element\_located | 节点加载出来，传入定位元组，如(By.ID, 'p') |
| visibility\_of\_element\_located | 节点可见，传入定位元组 |
| visibility\_of | 可见，传入节点对象 |
| presence\_of\_all\_elements\_located | 所有节点加载出来 |
| text\_to\_be\_present\_in\_element | 某个节点文本包含某文字 |
| text\_to\_be\_present\_in\_element\_value | 某个节点值包含某文字 |
| frame\_to\_be\_available\_and\_switch\_to\_it | 加载并切换 |
| invisibility\_of\_element\_located | 节点不可见 |
| element\_to\_be\_clickable | 节点可点击 |
| staleness\_of | 判断一个节点是否仍在DOM，可判断页面是否已经刷新 |
| element\_to\_be\_selected | 节点可选择，传节点对象 |
| element\_located\_to\_be\_selected | 节点可选择，传入定位元组 |
| element\_selection\_state\_to\_be | 传入节点对象以及状态，相等返回True，否则返回False |
| element\_located\_selection\_state\_to\_be | 传入定位元组以及状态，相等返回True，否则返回False |
| alert\_is\_present | 是否出现警告 |

#### 历史记录--前进和后退

back()方法后退， forward()方法前进

#### 异常处理

可以使用try except语句来捕获各种异常, 一旦出现错误，就进行异常处理

browser = webdriver.Chrome()  
*try*:  
 browser.get('https://www.baidu.com')  
*except* TimeoutException:  
 pass

#### 动作链

一些交互动作都是针对某个节点执行的。比如，对于输入框，我们就调用它的输入文字和清空文字方法。

一些操作没有特定的执行对象，比如鼠标拖曳、键盘按键等，这些动作用动作链来执行

*from* selenium *import* webdriver  
*from* selenium.webdriver *import* ActionChains  
  
browser = webdriver.Chrome()  
url = 'http://www.runoob.com/try/try.php?filename=jqueryui-api-droppable'  
browser.get(url)  
browser.switch\_to.frame('iframeResult')  
source = browser.find\_element\_by\_css\_selector('#draggable')  
target = browser.find\_element\_by\_css\_selector('#droppable')  
actions = ActionChains(browser)  
actions.drag\_and\_drop(source, target)  
actions.perform()

首先，打开网页中的一个拖曳实例，然后依次选中要拖曳的节点和拖曳到的目标节点，接着声明ActionChains对象并将其赋值为actions变量，然后通过调用actions变量的drag\_and\_drop()方法，再调用perform()方法执行动作，此时就完成了拖曳操作

#### 切换Frame

网页中有一种节点叫作iframe，也就是子Frame，相当于页面的子页面，它的结构和外部网页的结构完全一致

Selenium打开页面后，它默认是在父级Frame里面操作，而此时如果页面中还有子Frame，它是不能获取到子Frame里面的节点的。使用switch\_to.frame()方法来切换Frame

browser.get(url)

browser.switch\_to.frame('iframeResult')

#### 执行JavaScript

browser = webdriver.Chrome()  
browser.get('https://www.zhihu.com/explore')  
browser.execute\_script('window.scrollTo(0,document.body.scrollHeight)')  
browser.execute\_script('alert("To Bottom")')

execute\_script()方法将进度条下拉到最底部，然后弹出alert提示框

将窗口滑动到page这个位置

page = driver.find\_elements\_by\_xpath("//div[@class='page']")

#拖动到可见的元素去  
driver.execute\_script('arguments[0].scrollIntoView();', page[-1])

### Xpath（XML Path）

元素查找方式，几乎可以定位到页面上的任意元素

xpath路径以//开头时，则表示让xpath引擎从文档的任意符合的元素节点开始进行解析，/出现在xpath路径中时，则表示寻找父节点的直接子节点，当//出现在xpath路径中时，表示寻找父节点下任意符合条件的子节点，用\*号省略具体的标签名称，但元素的层级关系必须体现出来

#### 查找节点

1. 获取单个节点

* find\_element\_by\_id
* find\_element\_by\_name
* find\_element\_by\_xpath
* find\_element\_by\_link\_text
* find\_element\_by\_partial\_link\_text
* find\_element\_by\_tag\_name
* find\_element\_by\_class\_name
* find\_element\_by\_css\_selector CSS选择器

通用方法find\_element()，它需要传入两个参数：查找方式By和值

find\_element\_by\_id(id)就等价于find\_element(By.ID, id)

1. 获取多个节点

如果有多个节点，用find\_element()方法查找，就只能得到第一个节点

* find\_elements\_by\_id
* find\_elements\_by\_name
* find\_elements\_by\_xpath
* find\_elements\_by\_link\_text
* find\_elements\_by\_partial\_link\_text
* find\_elements\_by\_tag\_name
* find\_elements\_by\_class\_name
* find\_elements\_by\_css\_selector

#### 获取节点信息

通过page\_source属性可以获取网页的源代码，接着就可以使用解析库（如正则表达式、Beautiful Soup、pyquery等）来提取信息

Selenium返回的是WebElement类型，也有相关的方法和属性来直接提取节点信息

1. 获取属性

选中这个节点，**get\_attribute()**方法来获取节点的属性，通过get\_attribute()方法，传入想要获取的属性名，就可以得到它的值了。

logo = browser.find\_element\_by\_id('zh-top-link-logo')

print(logo.get\_attribute('class'))

1. 获取文本值

每个WebElement节点都有text属性，直接调用这个属性就可以得到节点内部的文本信息，这相当于Beautiful Soup的get\_text()方法、pyquery的text()方法

browser.get(url)

input = browser.find\_element\_by\_class\_name('zu-top-add-question')

print(input.text)

1. 获取id、位置、标签名和大小

id属性可以获取节点id，location属性可以获取该节点在页面中的相对位置，tag\_name属性可以获取标签名称，size属性可以获取节点的大小，也就是宽高

input = browser.find\_element\_by\_class\_name('zu-top-add-question')

print(input.id)

print(input.location)

print(input.tag\_name)

print(input.size)

#### 节点交互

输入文字时用send\_keys()，清空文字时用clear()，点击按钮时用click()。

通过元素选取，能够找到元素的位置，根据这个元素的位置进行相应的事件操作，例如输入文本框内容、鼠标单击、填充表单、元素拖拽等等

elem = driver.find\_element\_by\_xpath("//a[@data-fun='next']")  
elem.click()

点击这个按键之前，窗口最好移动到那里，因为如果这个按键被其他元素遮挡，click()就触发异常。

### PhantomJS

使用 Chrome 或 Firefox 进行网页抓取的话，每次抓取的时候，都会弹出一个浏览器，比较影响使用。

PhantomJS是一个基于WebKit的服务器端JavaScript API。它全面支持Web而无需浏览器支持，运行快，原生支持各种Web标准:DOM处理、CSS选择器、JSON, Canvas,和SVG。

PhantomJS可以用于页面自动化、网络监测、网页截屏，以及无界面测试等。它会把网站加载到内存并执行页面上的 JavaScript，但不会向用户展示网页的图形界面。可以看做一个没有界面的浏览器，既有Firefox、chome的功能，又因为没有界面而更加快速，占更小的内存

driver=webdriver.PhantomJS(executable\_path='D:/下载/phantomjs\_xpgod/phantomjs-2.1.1-windows/bin/phantomjs.exe')  
driver.get(url)  
driver.implicitly\_wait(1)  
print(driver.get\_cookies())

执行js文件：phantomjs hello.js

## Splash

Splash是一个Scrapy中支持JavaScript渲染的工具，是一个带有HTTP API的轻量级浏览器，同时它对接了Python中的Twisted和QT库，可以实现动态渲染页面的抓取

* 异步方式处理多个网页渲染过程；
* 获取渲染后的页面的源代码或截图；
* 通过关闭图片渲染或者使用Adblock规则来加快页面渲染速度；
* 可执行特定的JavaScript脚本；
* 可通过Lua脚本来控制页面渲染过程；
* 获取渲染的详细过程并通过HAR（HTTP Archive）格式呈现。

1. **安装、启动**

**docker run -p 8050:8050 scrapinghub/splash**

-d 代表容器以守护态运行，中断远程服务器连接后不会终止 Splash 服务的运行。

1. **打开**[**http://localhost:8050/**](http://localhost:8050/)



脚本是用Lua语言写的。首先调用go()方法去加载页面，然后调用wait()方法等待了一定时间，最后返回了页面的源码、截图和HAR信息。

Splash是通过Lua脚本来控制了页面的加载过程的，加载过程完全模拟浏览器，最后可返回各种格式的结果，如网页源码和截图等。

输入网址后Render me按钮开始渲染

## aiohttp

Requests库是一个阻塞式HTTP请求库，当我们发出一个请求后，程序会一直等待服务器响应，直到得到响应后，程序才会进行下一步处理

aiohttp是一个提供异步Web服务的库，使用异步请求库进行数据抓取时，会大大提高效率

# 解析库

## LXML

LXML全称XML Path Language，即XML路径语言， 是 Python 的一个解析库，支持 HTML 和 XML 的解析，支持 XPath 解析方式，而且解析效率非常高

### XPath常用规则

* nodename 选取此节点的所有子节点
* / 从当前节点选取直接子节点
* // 从当前节点选取子孙节点
* . 选取当前节点
* .. 选取当前节点的父节点
* @ 选取属性

*from* lxml *import* etree  
  
html = etree.parse('./test.html', etree.HTMLParser())  
result = etree.tostring(html)  
print(result.decode('utf-8'))

导入lxml库的etree模块, 读取文本文件，调用HTML类进行初始化，这样就成功构造了一个XPath解析对象. etree模块可以自动修正HTML文本

调用tostring()方法即可输出修正后的HTML代码，但是结果是bytes类型。这里利用decode()方法将其转成str类型

### 属性匹配

用@符号进行属性过滤，返回列表

result = html.xpath('//li[@class="item-0"]') class为item-0的li节点

### 文本获取

用XPath中的text()方法获取节点中的文本

result = html.xpath('//li[@class="item-0"]/text()')

### 属性获取

用@符号，以列表形式返回

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

### 属性多值匹配

某些节点的某个属性可能有多个值，用contains()函数

result = html.xpath('//li[contains(@class, "li")]/a/text()')

通过contains()方法，第一个参数传入属性名称，第二个参数传入属性值，只要此属性包含所传入的属性值，就可以完成匹配，返回列表

### 多属性匹配

根据多个属性确定一个节点，这时就需要同时匹配多个属性。使用运算符and来连接, 相连之后置于中括号内进行条件筛选

r=html.xpath('//li[contains(@class,"li")and@name="item"]/a/text()')

### 按序选择

中括号传入索引的方法获取特定次序的节点

result = html.xpath('//li[1]/a/text()') 第一个li节点

result = html.xpath('//li[last()]/a/text()') 最后一个li节点

result = html.xpath('//li[position()<3]/a/text()') 位置小于3的li节点

result = html.xpath('//li[last()-2]/a/text()') 倒数第三个li节点

## BeautifulSoup

BeautifulSoup 的 HTML 和 XML 解析器是依赖于 LXML 库的

BeautifulSoup自动将输入文档转换为Unicode编码，输出文档转换为UTF-8编码。

初始化BeautifulSoup自动更正格式, 调用prettify()方法可以把要解析的字符串以标准的缩进格式输出。

soup = BeautifulSoup(html, 'lxml')  
print(soup.prettify())  
print(soup.title.string)

****

### Beautiful Soup对象

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

Beautiful Soup四大对象种类

* Tag
* NavigableString
* BeautifulSoup
* Comment

#### tag

**Beautiful Soup获取 Tags**

它查找的是在所有内容中的第一个符合要求的标签

soup = BeautifulSoup(html, 'lxml')

soup.标签名.name/attrs

Tag有两个重要的属性：name和attrs：

1. **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** 标签的所有属性打印输出，得到的类型是一个字典。

print soup.p.attrs

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

如果需要**单独获得**某个属性可以使用字典的打印方法或者get(‘key’)

print soup.p['class']

print soup.p.get('class')

可以对这些属性和内容等等进行修改 soup.p['class']="newClass"

对这个属性进行删除: del soup.p['class']

#### NavigableString

获取标签内部的文字 soup.标签名.string

通过Unicode（）方法可以直接将NavigableString对象转换为Unicode字符串

#### BeautifulSoup

表示的是一个文档的全部内容，大部分情况下，可以当做特殊的tag对象，没有name和attr属性。

soup = BeautifulSoup(html, 'lxml')

#### Comment

特殊类型的NavigableString对象，文档的注释部分。

首先判断的类型是否为 Comment 类型，然后再进行其他操作，如打印输出

if type(soup.li.string) == element.Comment:

print(soup.li.string)

### 遍历文档树

#### 直接子节点(不包含孙节点)

1. .contents

tag的content属性可以将tag的子节点以列表的方式输出：soup.标签名.contents

可以用列表索引来获取它的某一个元素: soup.标签名.contents[1]

1. .children

children是一个 list 生成器对象,返回的不是一个 list, 可以通过遍历获取所有子节点

for child in soup. 标签名.children:

print(child)

#### 所有子孙节点

.descendants 属性

.contents 和 .children 属性仅包含tag的直接子节点，.descendants 属性可以对所有tag的子孙节点进行递归循环，和 children类似，我们也需要遍历获取其中的内容。

for child in soup.descendants:

print child

#### 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（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。）
* 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

1. 多个内容

.strings .stripped\_strings 属性

for string in soup.stripped\_strings:

print(repr(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

1. 父节点
2. .parent 属性

content = soup.head.title.string

print content.parent.name

1. .parents 属性

递归得到元素的所有父辈节点

content = soup.head.title.string

for parent in content.parents:

print parent.name

1. 兄弟节点
2. .next\_sibling .previous\_sibling 属性

.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

1. .next\_siblings .previous\_siblings 属性

对当前节点的兄弟节点迭代输出

for sibling in soup.a.next\_siblings:

print(repr(sibling))

1. 前后节点
2. .next\_element .previous\_element 属性

所有节点，不分层次

比如 head 节点为

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

print soup.head.next\_element

那么它的下一个节点便是 title，它是不分层次关系的

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

1. .next\_elements .previous\_elements 属性

可以向前或向后访问文档的解析内容

for element in last\_a\_tag.next\_elements:

print(repr(element))

### 搜索文档树

#### find\_all()

**find\_all(name, attrs, recursive, text, limit, \*\*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

查找所有名字为name的tag,字符串对象会被自动忽略掉

* 1. 传递字符：soup.find\_all('b')

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

* 1. 传递正则表达式：通过正则表达式的 match() 来匹配内容

for tag in soup.find\_all(re.compile("^b")):

print(tag.name)

# body

# b

* 1. 传递列表：将与列表中任一元素匹配的内容返回

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>]

* 1. 传递True：可以匹配任何值，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

* 1. 传递方法：可以定义一个方法,方法只接受一个元素参数,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

def has\_class\_but\_no\_id(tag): 包含 class 属性却不包含 id 属性

    return tag.has\_attr('class') and not tag.has\_attr('id')

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>]

1. attrs参数：定义一个字典参数来搜索包含特殊属性的tag

soup.find\_all (attrs={"data-foo": "value"})

1. 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。

soup.html.find\_all("title", recursive=False)

1. text参数：搜索文档中的字符串，接受字符串，正则表达式 ，列表，True

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"]

1. limit参数：find\_all() 方法返回全部的搜索结构，使用 limit 参数限制返回结果的数量

soup.find\_all("a", limit=2)

1. kwargs参数：kwargs 接收字符串，正则表达式

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

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')

soup.find\_all("a", class\_="sister") 使用class 过滤，加个下划线

#### 更多find方式

1. **find( name , attrs , recursive , text , \*\*kwargs )**

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

1. **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

1. **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

1. **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

1. **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

1. **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

#### css选择器

**soup.select()，返回类型是 list**

1. 通过标签名查找 soup.select('title')
2. 通过类名查找 soup.select('.sister')
3. 通过 id 名查找 soup.select('#link1')
4. 组合查找 soup.select('p #link1') p 标签中，id 等于 link1的内容

空格分开

soup.select("head > title") 直接子标签查找

1. 属性查找 soup.select('a[class="sister"]')

属性和标签属于同一节点，中间不能加空格

属性仍然可以与上述查找方式组合，不在同一节点的空格隔开，同一节点的不加空格

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()

1. 获取属性

获取属性还可以用原来的方法

soup = BeautifulSoup(html, 'lxml')  
*for* ul *in* soup.select('ul'):  
 print(ul['id'])  
 print(ul.attrs['id'])

1. 获取文本

可以用string或者get\_text()，二者的效果完全一致

## PyQuery

提供了和 jQuery 类似的语法来解析 HTML 文档，支持 CSS 选择器

## Tesserocr

OCR，即 Optical Character Recognition，光学字符识别。是指通过扫描字符，然后通过其形状将其翻译成电子文本的过程。大多数验证码还是图形验证码，这时候我们可以直接用 OCR 来识别。

Tesserocr 是 Python 的一个 OCR 识别库，但其实是对 Tesseract 做的一层 Python API 封装，所以它的核心是 Tesseract

# web库

## Flask

Flask 是一个轻量级的 Web 服务程序，主要用它来做一些 API 服务

## Tornado

Tornado 是一个支持异步的Web框架，通过使用非阻塞 I/O 流，它可以支撑成千上万的开放连接，效率非常高

# APP库

APP 一些简单的接口通过 Charles 或 MitmProxy 分析找出规律就可以直接用程序模拟来抓取

如果遇到更复杂的接口就需要利用 MitmDump 对接Python来对抓取到的请求和响应进行实时处理和保存

要做规模采集就需要自动化 APP 的操作而不是人工去采集，Appium，它可以像 Selenium 一样对 APP 进行自动化控制，如自动化模拟APP的点击、下拉等操作。

## Charles

1. [证书配置](https://germey.gitbooks.io/python3webspider/content/1.7.1-Charles%E7%9A%84%E5%AE%89%E8%A3%85.html)

## MitmProxy

MitmProxy 是一个支持 HTTP 和 HTTPS 的抓包程序，类似 Fiddler、Charles 的功能，只不过它是一个控制台的形式操作。

同时 MitmProxy 还有两个关联组件，一个是 MitmDump，它是 MitmProxy 的命令行接口，利用它我们可以对接 Python 脚本，用 Python 实现监听后的处理。另一个是 MitmWeb，它是一个 Web 程序，通过它我们可以清楚地观察到 MitmProxy 捕获的请求。

* 拦截HTTP和HTTPS请求和响应。
* 保存HTTP会话并进行分析。
* 模拟客户端发起请求，模拟服务端返回响应。
* 利用反向代理将流量转发给指定的服务器。
* 支持Mac和Linux上的透明代理。
* 利用Python对HTTP请求和响应进行实时处理。

1. 安装

pip3 install mitmproxy

1. 产生CA证书（5个），并启动mitmdump：

mitmdump # 在C:\Users\Administrator\.mitmproxy里面找到CA证书

1. 保存证书，不需要设置密码，所有存入受信任的根证书颁发机构

注：

|  |  |
| --- | --- |
| mitmproxy-ca.pem | PEM格式的证书私钥 |
| mitmproxy-ca-cert.pem | 适用于大多数非Windows平台 |
| mitmproxy-ca-cert.p12 | 适用于Windows平台 |
| mitmproxy-ca-cert.cer | 适用于部分Android平台 |
| mitmproxy-dhparam.pem | PEM格式的秘钥文件，增强SSL安全性 |

1. mitmproxy-ca-cert.pem文件发送到手机，打开安装
2. 设置代理

运行mitmproxy或者mitmdump会在8080端口上运行一个代理服务

## Appium

Appium 是移动端的自动化测试工具，可以驱动 Android、iOS 等设备完成自动化测试

Appium 负责驱动移动端来完成一系列操作，对 iOS 设备来说，它使用苹果的 UIAutomation 来实现驱动，对于 Android 来说，它使用 UiAutomator 和 Selendroid 来实现驱动。

# 存储数据

用解析器解析出数据之后，接下来就是存储数据了。保存的形式可以多种多样，最简单的形式是直接保存为文本文件，如TXT、JSON、CSV等。另外，还可以保存到数据库中，如关系型数据库MySQL，非关系型数据库MongoDB、Redis等。

## 文件存储

### TXT文本

file = open('explore.txt', 'a', encoding='utf-8')  
file.write('\n'.join([question, author, answer]))  
file.write('\n' + '=' \* 50 + '\n')  
file.close()

open()方法打开一个文本文件，获取一个文件操作对象。利用file对象的write()方法将提取的内容写入文件，最后调用close()方法将其关闭。

open()方法的第一个参数即要保存的目标文件名称，第二个参数为a，代表以追加方式写入到文本。指定了文件的编码为utf-8

**打开方式**

* + r：以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
  + rb：以二进制只读方式打开一个文件。文件指针将会放在文件的开头。
  + r+：以读写方式打开一个文件。文件指针将会放在文件的开头。
  + rb+：以二进制读写方式打开一个文件。文件指针将会放在文件的开头。
  + w：以写入方式打开一个文件。如果该文件已存在，则将其覆盖。如果该文件不存在，则创建新文件。
  + wb：以二进制写入方式打开一个文件。如果该文件已存在，则将其覆盖。如果该文件不存在，则创建新文件。
  + w+：以读写方式打开一个文件。如果该文件已存在，则将其覆盖。如果该文件不存在，则创建新文件。
  + wb+：以二进制读写格式打开一个文件。如果该文件已存在，则将其覆盖。如果该文件不存在，则创建新文件。
  + a：以追加方式打开一个文件。如果该文件已存在，文件指针将会放在文件结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，则创建新文件来写入。
  + ab：以二进制追加方式打开一个文件。如果该文件已存在，则文件指针将会放在文件结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，则创建新文件来写入。
  + a+：以读写方式打开一个文件。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，则创建新文件来读写。
  + ab+：以二进制追加方式打开一个文件。如果该文件已存在，则文件指针将会放在文件结尾。如果该文件不存在，则创建新文件用于读写。

**简化写法**

文件操作中可能会出现IO异常，推荐使用with语句

with as语法。在with控制块结束时，文件会自动关闭，所以就不需要再调用close()方法了。

with open('explore.txt', 'a', encoding='utf-8') as file:  
 file.write('\n'.join([question, author, answer]))  
 file.write('\n' + '=' \* 50 + '\n')

### JSON文件存储

Python对JSON文件的操作分为编码和解码，通过JSON模块来实现。

* **对象**：它在JavaScript中是使用花括号{}包裹起来的内容，数据结构为{key1：value1, key2：value2, ...}的键值对结构。在面向对象的语言中，key为对象的属性，value为对应的值。键名可以使用整数和字符串来表示。值的类型可以是任意类型。
* **数组**：数组在JavaScript中是方括号[]包裹起来的内容，数据结构为["java", "javascript", "vb", ...]的索引结构。在JavaScript中，数组是一种比较特殊的数据类型，它也可以像对象那样使用键值对，但还是索引用得多。同样，值的类型可以是任意类型。

1. **读取JSON**

解码过程是把json对象转换成python对象的一个过程，常用的两个函数是load和loads

函数

函数原型如下:

1. loads(s，encoding=None，cls=None，object\_hook=None，parse\_float=None，parse\_int=None，parse\_constant=None，object\_pairs\_ hook=None，\*\*kw)
2. load(fp，encoding=None，cls=None，object\_hook=None，parse\_float=None，parse\_int=None，parse\_constant=None，object\_pairs\_ hook=None，\*\*kw)

常用参数分析:

* encoding:指定编码格式。
* parse-fl oat:如果指定，将把每一个JSON字符串按照float解码调用。默认情况下，这相当于float(num\_str)
* parse\_ it:如果指定，将把每一个JSON字符串按照int解码调用。默认情况下，这相当于int(num\_str)

调用库的loads()方法将JSON文本字符串转为JSON对象

import json  
str = '''  
[{  
 "name": "Bob",  
 "gender": "male",  
 "birthday": "1992-10-18"  
}, {  
 "name": "Selina",  
 "gender": "female",  
 "birthday": "1995-10-18"  
}]  
'''  
data = json.loads(str)

可以用索引来获取对应的内容

通过中括号加0索引，可以得到第一个字典元素，然后再调用其键名即可得到相应的键值

获取键值时有两种方式，一种是中括号加键名，另一种是通过get()方法传入键名

get()方法，如果键名不存在，不会报错，会返回None

get()方法还可以传入第二个参数（即默认值）

data[0]['name']

data[0].get('name')

1. **输出JSON**

编码过程是把Python对象转换成JSON对象的一个过程，常用的两个函数是dumps和dump函数。

两个函数的唯一区别就是dump把Python对象转换成JSON对象，并将JSON对象通过fp文件流写入文件中，而 dumps则是生成了一个字符串。

dumps和dump的函数原型:

1. dumps(obj，skipkeys=False，ensure\_ascii=True，check-circular=True，allow\_ nan=True，cls=None，indent=None，separators=None，encoding=’utf-8’，default=None，sort\_keys=False，\*\*kw)
2. dump(obj，fp，skipkeys=False，ensure\_ascii=True，check-circular--True，allow\_ nan=True，cls=None，indent=None，separators=None，encoding=’utf-8’，default=None，sort\_keys=False，\*\*kw)

常用参数分析:

* Skipkeys：默认值是False。如果diet的keys内的数据不是python的基本类型(str,unicode, int, long, float, pool, None)，设置为False时，就会报TypeError错误。此时设置成True，则会跳过这类key。
* ensure\_ ascii：默认值True。如果dict内含有非ASCII的字符，则会以类似“\uXXXX”的格式显示数据，设置成False后，就能正常显示。
* indent：应该是一个非负的整型，如果是0，或者为空，则一行显示数据，否则会换行且按照 indent的数量显示前面的空白，将JSON内容进行格式化显示。
* separators：分隔符，实际上是(item\_separator, diet\_separator)的一个元组，默认的就是（‘，’，‘：’），这表示dictionary内keys之间用“，”隔开，而key和value之间用“:”隔开。
* encoding：默认是UTF-8。设置JSON数据的编码方式，在处理中文时一定要注意。
* sor\_keys：将数据根据keys的值进行排序。

dumps()方法将JSON对象转化为字符串

with open('data.json', 'w') as file:

file.write(json.dumps(data))

如果想保存JSON的格式，可以再加一个参数indent，代表缩进字符个数。

with open('data.json', 'w') as file:

file.write(json.dumps(data, indent=2))

JSON中包含中文字符

需要指定参数ensure\_ascii为False，另外还要规定文件输出的编码

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

file.write(json.dumps(data, indent=2, ensure\_ascii=False))



### CSV文件存储

CSV，全称为Comma-Separated Values，中文可以叫作逗号分隔值或字符分隔值，其文件以纯文本形式存储表格数据。

所有记录都有完全相同的字段序列，相当于一个结构化表的纯文本形式。

1. **写入**

import csv  
with open('data.csv', 'w') as csvfile:  
 fieldnames = ['id', 'name', 'age']  
 writer = csv.DictWriter(csvfile, fieldnames=fieldnames)  
 writer.writeheader()  
 writer.writerow({'id': '10001', 'name': 'Mike', 'age': 20})  
 writer.writerow({'id': '10002', 'name': 'Bob', 'age': 22})  
 writer.writerow({'id': '10003', 'name': 'Jordan', 'age': 21})

一般情况下，爬虫爬取的都是结构化数据，我们一般会用字典来表示。

1. 打开data.csv文件，然后指定打开的模式为w（即写入）
2. 定义3个字段，用fieldnames表示
3. 获得文件句柄，调用csv库的DictWriter ()初始化一个字典写入对象，传入该句柄
4. 调用writeheader()方法先写入头信息，再调用writerow()方法传入相应字典

运行结束后，会生成一个名为data.csv的文件，此时数据就成功写入了。直接以文本形式打开的话，写入的文本默认以逗号分隔

* 如果想修改列与列之间的分隔符，可以传入delimiter参数

writer = csv.writer(csvfile, delimiter=' ')

* 调用writerows()方法同时写入多行，此时参数就需要为二维列表,格式可以是列表或者字典

writer.writerow(['id', 'name', 'age'])

writer.writerows([['10001', 'Mike', 20], ['10002', 'Bob', 22], ['10003', 'Jordan', 21]])

* 要写入中文内容的话，可能会遇到字符编码的问题，此时需要给open()参数指定编码格式

with open('data.csv', 'a', encoding='utf-8') as csvfile:

1. **读取**

import csv  
with open('data.csv', 'r', encoding='utf-8') as csvfile:  
 reader = csv.reader(csvfile)  
 for row in reader:  
 print(row)

构造Reader对象，通过遍历输出了每行的内容，每一行都是一个列表形式

如果CSV文件中包含中文的话，还需要指定文件编码。

可以利用read\_csv()方法将数据从CSV中读取出来

import pandas as pd  
df = pd.read\_csv('data.csv')  
print(df)

* 在存储CSV文件时，需要统一存储数据的类型。代码中使用ecoe('utf 8’)作用就是将title, real title,hr ef, date变量类型统一为str
* Beautifulsoup如果使用lxml作为解析库，会发现解析出来的HTML内容缺失，这是由于Beautifulsoup为不同的解析器提供了相同的接口，但解析器本身是有区别的，同一篇文档被不同的解析器解析后可能会生成不同结构的树型文档。因此如果遇到缺失的情况，Beautifulsoup可以使用html.parser作为解析器，或者单独使用lxml进行解析即可。

## 多媒体文件抽取

存储媒体文件主要有两种方式:只获取文件的URL链接，或者直接将媒体文件下载到本

地。

urlretrieve()方法直接将远程数据下载到本地，函数原型如下:

urlretrieve(url, filename=None, reporthook=None, data=None)

* 参数filename指定了存储的本地路径(如果参数未指定，urllib会生成一个临时文件保存数据。)
* 参数reporthook是一个回调函数。当连接上服务器以及相应的数据块传输完毕时会触发该回调函数，我们可以利用这个回调函数来显示当前的下载进度。
* 参数data指post到服务器的数据，该方法返回一个包含两个元素的(filename, headers )元组，filename表示保存到本地的路径，header表示服务器的响应头。

*import* os  
*import* urllib.request  
*def* cbk(*a*, *b*, *c*):  
 '''''回调函数  
 @a:已经下载的数据块  
 @b:数据块的大小  
 @c:远程文件的大小  
 '''  
 per = 100.0 \* *a* \* *b* / *c  
 if* per > 100:  
 per = 100  
 print('%.2f%%' % per)  
url = 'http://www.baidu.com'  
dir = os.path.abspath('.')  
work\_path = os.path.join(dir, 'baidu.html')  
urllib.request.urlretrieve(url, work\_path, cbk)

## MySQL

关系型数据库基于关系模型的数据库，而关系模型是通过二维表来保存的，所以它的存储方式就是行列组成的表，每一列是一个字段，每一行是一条记录。

关系型数据库有多种，如 SQLite、MySQL、Oracle、SQL Server、DB2等等

### 启动数据库

启动：net start mysql

停止：net stop mysql

进入mysql：mysql –uroot -p

### 连接数据库

connection = pymysql.connect(host='localhost',  
 port=3306,  
 user='root',  
 passwd='zkyr1006',  
 db='python',  
 charset='utf8')  
cursor = connection.cursor()

sql = "USE python;"  
cursor.execute(sql)  
connection.commit()

调用cursor()方法获得MySQL的操作游标，利用游标来执行SQL语句。

### 操作命令

1. **创建数据库、表**

创建数据库

cursor.execute("CREATE DATABASE 数据库名DEFAULT CHARACTER SET utf8")

创建表  
sql = 'CREATE TABLE IF NOT EXISTS 表名(id VARCHAR(255) NOT NULL, name VARCHAR(255) NOT NULL, age INT NOT NULL, PRIMARY KEY (id))'

cursor.execute(sql)

1. **插入数据**

sql = 'INSERT INTO students(id, name, age) values(%s, %s, %s)'  
*try*:  
 cursor.execute(sql, (id, user, age))  
 db.commit()  
*except*:  
 db.rollback()  
db.close()

需要执行db对象的commit()方法才可实现数据插入，这个方法才是真正将语句提交到数据库执行的方法。如果执行失败，则调用rollback()执行数据回滚，相当于什么都没有发生过。

**动态插入数据**

做成一个通用方法，只需要传入一个动态变化的字典，SQL语句会根据字典动态构造，元组也动态构造

data = {  
 'id': '20120001',  
 'name': 'Bob',  
 'age': 20  
}  
table = 'students'  
keys = ', '.join(data.keys())  
values = ', '.join(['%s'] \* len(data))  
sql = 'INSERT INTO {table}({keys}) VALUES ({values})'.format(table=table, keys=keys, values=values)  
*try*:  
 *if* cursor.execute(sql, tuple(data.values())):  
 print('Successful')  
 db.commit()  
*except*:  
 print('Failed')  
 db.rollback()  
db.close()

1. **更新数据**

重复数据，如果数据存在，则更新数据；如果数据不存在，则插入数据。

sql = 'INSERT INTO {table}({keys}) VALUES ({values}) ON DUPLICATE KEY UPDATE'.format(table=table, keys=keys, values=values)  
update = ','.join([" {key} = %s".format(key=key) *for* key *in* data])  
sql += update  
*try*:  
 *if* cursor.execute(sql, tuple(data.values())\*2):  
 print('Successful')  
 db.commit()  
*except*:  
 print('Failed')  
 db.rollback()  
db.close()

ON DUPLICATE KEY UPDATE 如果主键已经存在，就执行更新操作。

1. **删除数据**

DELETE语句，指定要删除的目标表名和删除条件，使用db的commit()方法生效。

删除条件有多种多样，运算符有大于、小于、等于、LIKE等，条件连接符有AND、OR等

sql = 'DELETE FROM {table} WHERE {condition}'.format(table=table, condition=condition)

## MongoDB

MongoDB是由C++语言编写的非关系型数据库，是一个基于分布式文件存储的开源数据库系统，它的字段值可以包含其他文档、数组及文档数组，在高负载的情况下，添加更多的节点，可以保证服务器性能，适合爬虫开发存储大规模数据

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。



有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库。

* admin： "root"数据库。将一个用户添加到这个数据库，自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。
* local: 这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合
* config: 当Mongo用于分片设置时，config数据库在内部使用，用于保存分片的相关信息。



**文档**是一组键值(key-value)对(即BSON)。MongoDB 的文档不需要设置相同的字段，并且相同的字段不需要相同的数据类型，这与关系型数据库有很大的区别。BSON 字符串都是 UTF-8 编码。

**集合**就是 MongoDB 文档组，类似于 RDBMS （关系数据库管理系统：Relational Database Management System)中的表格。

集合存在于数据库中，没有固定的结构，可以插入不同格式和类型的数据

[安装](https://germey.gitbooks.io/python3webspider/content/1.4.2-MongoDB%E7%9A%84%E5%AE%89%E8%A3%85.html)

* + 1. **启动MongoDB**

启动：net start MongoDB service mongod start

停止：net stop MongoDB service mongod stop

移除：mongod.exe –remove

运行 MongoDB：mongod --port 27017 --dbpath /data/db

进入 MongoDB 命令行：mongo --port 27017

后台运行：/usr/bin/mongod --dbpath /var/lib/mongodb/ --logpath /var/log/mongodb/mongodb.log --logappend &

加入自启动，vi /etc/rc.d/rc.local，最后加入后台运行命令

* + 1. **MongoDB设置远程访问**
  1. 配置用户名和密码

mongo --port 27017

use admin

db.createUser({user: 'admin', pwd: 'zkyr1006', roles: [{role: 'root', db: 'admin'}]})

创建了一个用户名为 admin，密码为 zkyr1006 的用户，赋予最高权限。

* 1. 修改 MongoDB 的配置文件

sudo vi /etc/mongod.conf

将mongod.conf中bind\_ip设置为0.0.0.0

添加security : authorization: enabled

重启MongoDB服务

service mongodb restart

### 连接MongoDB

使用 PyMongo 库里面的 MongoClient，一般来说传入IP 及端口即可，第一个参数为地址 host，第二个参数为端口 port，端口如果不传默认是 27017。

*import* pymongo  
client=pymongo.MongoClient(host='localhost',port=27017)

# client = MongoClient('mongodb://localhost:27017/')

### 操作命令

1. **创建和切换数据库**

**use 数据库名**

如果数据库不存在，则创建数据库，否则切换到指定数据库。

刚创建的数据库并不在数据库的列表中，要显示它，需要向数据库插入数据。

db = client.python #指定数据库  
collection=db.python #指定集合

1. **插入数据**
2. **insert方法**

python = {  
 'id': '20170101',  
 'name': 'Jordan',  
 'age': 20,  
 'gender': 'male'  
}  
result=collection.insert(python)  
print(result)

调用collection的insert()方法插入数据，每条数据都有一个\_id属性来唯一标识。如果没有显式指明该属性，MongoDB会自动产生一个ObjectId类型的\_id属性。insert()方法会在执行后返回\_id值。

也可以插入多条数据，列表形式传递

result=collection.insert([python1,python2…])

返回对应\_id集合

1. **insert\_one()和insert\_many()**

result=collection.insert\_one(python)  
print(result)  
print(result.inserted\_id)

返回的是InsertOneResult对象，我调用其inserted\_id属性获取\_id

result = collection.insert\_many([python1, python2])  
print(result)  
print(result.inserted\_ids)

返回的类型是InsertManyResult，调用inserted\_ids属性可以获取插入数据的\_id列表。

1. **查询**

**find\_one()**或**find()**方法进行查询，其中find\_one()查询得到的是单个结果，find()则返回一个生成器对象。

1. 单个数据

result=collection.find\_one({'name':'Jordan'})  
print(type(result))  
print(result)

查询结果返回字典，包括插入的数据和自动生成的\_id属性

也可以使用ObjectId来查询，需要使用bson库里面的objectid

*from* bson.objectid *import* ObjectId  
result=collection.find\_one({'\_id': ObjectId('5b177a5df728ae9074b49a67')})

1. 多条数据

results=collection.find({'age':20})  
print(results)  
*for* result *in* results:  
 print(result)

查询结果返回字典， Cursor类型，不存在返回none，需要遍历获取所有结果，可以通过比较符号和正则表达式筛选查询





1. **计数**

调用count()方法统计查询结果有多少条数据，或者某个符合条件的数据

count=collection.find().count()

count2=collection.find({'age':20}).count()

1. **排序**

调用sort()方法，并在其中传入排序的字段及升降序标志

**pymongo.ASCENDING升序**

**pymongo.DESCENDING降序**

results=collection.find().sort('\_id',pymongo.ASCENDING)  
*for* result *in* results:  
 print(result['\_id'])

1. **偏移**

只取某几个元素时可以利用skip()方法偏移几个位置，比如偏移2，就忽略前两个元素，得到第三个及以后的元素，limit()方法指定要取的结果个数

results=collection.find().sort('name',pymongo.DESCENDING).skip(2).limit(1)  
*for* result *in* results:  
 print(result['name'])

数据库数量非常庞大的时候，如千万、亿级别，最好不要使用大的偏移量来查询数据，很可能导致内存溢出。可以使用筛选查询

1. **更新**
2. update（）

update()方法，指定更新的条件和更新后的数据，首先指定查询条件，然后将数据查询出来，修改年龄后调用update()方法将原条件和修改后的数据传入。

condition={'python':'malei'}  
newdata=collection.find\_one(condition)  
newdata['id']= '20170109'  
result=collection.update(condition,newdata)

也可以使用$set操作符对数据进行更新

result=collection.update(condition,{'$set':newdata})

1. **update\_one()方法和update\_many()**
2. update\_one()

condition={'python':'malei'}  
newdata=collection.find\_one(condition)

result=collection.update\_one(condition,{'$set':newdata})  
print(result.matched\_count, result.modified\_count)

update\_one()第二个参数不能再直接传入修改后的字典，需要使用{'$set': newdata }这样的形式，其返回结果是UpdateResult类型。然后分别调用matched\_count和modified\_count属性，可以获得匹配的数据条数和影响的数据条数。

condition={'age': {'$gte': 20}}  
result=collection.update\_one(condition,{'$inc': {'age': 1}})

查询条件为年龄大于等于20，更新条件为{'$inc': {'age': 1}}，也就是年龄加1

1. update\_many()

调用update\_many()方法，则会将所有符合条件的数据都更新

result=collection.update\_many(condition,{'$inc': {'age': 1}})

1. **删除**
2. **remove**

collection.remove({'python': 'malei'})

1. **delete\_one()和delete\_many()**
2. delete\_one()

result=collection.delete\_one({'name':'Jordan'})  
print(result.deleted\_count)

delete\_one()即删除第一条符合条件的数据，返回DeleteResult类型，调用deleted\_count属性获取删除的数据条数。

1. delete\_many()

result=collection.delete\_many({'name':'Jordan'})

delete\_many()删除所有符合条件的数据。返回DeleteResult类型

1. **其他操作**

PyMongo还提供了一些组合方法，如find\_one\_and\_delete()、find\_one\_and\_replace()和find\_one\_and\_update()，它们是查找后删除、替换和更新操作，其用法与上述方法基本一致。

PyMongo官方文档：<http://api.mongodb.com/python/current/api/pymongo/collection.html>

### MongoDB副本集

在分布式爬虫爬取过程中，MonogoDB存储服务器宕机了，那整个爬虫系统将立刻陷入瘫痪。

为了避免这种情况，可以使用多个MonogoDB存储节点，当主节点挂掉，从节点可以立刻补

充进来，保持系统的稳定运行.



副本集特征：

* N 个节点的集群
* 任何节点可作为主节点
* 所有写入操作都在主节点上
* 自动故障转移
* 自动恢复



通常建议将复制集成员数量设置为奇数，从上表可以看出3个节点和4个节点的复制集都只能容忍1个节点失效，从『服务可用性』的角度看，其效果是一样的。（但无疑4个节点能提供更可靠的数据存储）

**步骤**

1. 启动MongoDB

启动一个MongoDB实例，若要在本机开启主从模式，修改端口和存储文件位置即可，复制集名称不变

**mongod --port "PORT" --dbpath "数据存储目录" --replSet 复制集名称**



1. 初始化副本集

启动MongoDB实例后，打开cmd使用**mongo ip:port**连接服务，使用**rs.initiate()**初始化一个新的副本集。如果不初始化，接下来命令会出错



1. 副本集添加成员

使用**rs.add(“HOST\_NAME:PORT”)**方法来添加。MongoDB中只能通过主节点将Mongo服务添加到副本集中， 判断当前运行的Mongo服务是否为主节点可以使用命令**db.isMaster()** 。



1. 查看副本集状态使用 **rs.status()** 命令



1. 使用**rs.conf()**来查看副本集的配置



1. 测试是否同步

主节点连接MongoDB，选定数据库，插入数据

**use python**

**db.python.insert({"name":"无敌最俊美"})**

中断主节点，连接从节点查询（默认主节点读写数据，从节点不允许）

**db.getMongo().setSlaveOk(); // 设置从节点可读**

**db.python.find();**



1. 强制关闭主节点，经过投票选择从节点为主节点



1. python访问副本集

client=MongoClient("mongodb://127.0.0.1:27017,127.0.0.1:27021,127.0.0.1:27022",replicaset='rs0')  
print(client.python.python.find\_one())



### MongoDB分片

分片（sharding）是MongoDB用来将大型集合分割到不同服务器（或者说一个集群）上所采用的方法。高数据量和吞吐量的数据库应用会对单机的性能造成较大压力,大的查询量会将单机的CPU耗尽,大的数据量对单机的存储压力较大,最终会耗尽系统的内存而将压力转移到磁盘IO上。

垂直扩展：增加更多的CPU和存储资源来扩展容量。

水平扩展：将数据集分布在多个服务器上。水平扩展即分片。

**分片机制提供了如下三种优势：**

1. 对集群进行抽象，让集群“不可见”

　　MongoDB自带了一个叫做mongos的专有路由进程。mongos就是掌握统一路口的路由器，其会将客户端发来的请求准确无误的路由到集群中的一个或者一组服务器上，同时会把接收到的响应拼装起来发回到客户端。

1. 保证集群总是可读写

　　MongoDB通过多种途径来确保集群的可用性和可靠性。将MongoDB的分片和复制功能结合使用，在确保数据分片到多台服务器的同时，也确保了每分数据都有相应的备份，这样就可以确保有服务器换掉时，其他的从库可以立即接替坏掉的部分继续工作。

1. 使集群易于扩展

　　当系统需要更多的空间和资源的时候，MongoDB使我们可以按需方便的扩充系统容量。

**分片集群架构**

Config Server

存储集群所有节点、分片数据路由信息。默认需要配置3个Config Server节点。

Mongos

提供对外应用访问，所有操作均通过mongos执行。一般有多个mongos节点。数据迁移和数据自动平衡。

Mongod

存储应用数据记录。一般有多个Mongod节点，达到数据分片目的。

要构建一个MongoDB分片集群，需要三个角色：

* shard server  即存储实际数据的分片，每个shard 可以是一个Mongod实例，也可以是一组mongod实例构成得Replica Set（复制集）。为了实现每个shard内部的auto-failover，MongoDB官方建议每个shard 为一组Replica set。
* Config Server  为了将一个特定的collection存储在多个shard中，需要为该collection指定一个shard key,例如｛age:1｝,shard key可以决定该条记录属于哪个chunk。Config Servers就是用来存储所有shard节点的配置信息，每个chunk的shard key范围，chunk在各shard的分布情况 、该集群中所有DB和collection的sharding 配置信息
* Route Process 前端路由，客户端由此接入，然后询问config servers需要到哪个shard上查询或把保存记录，在连接相应的shard进行操作，最后将结果返回客户端。客户端只需要将原本发给mongod的查询或更新请求原封不动地发给rounting processl.而不必关心所操作的记录存储在哪个shard上。让整个集群看上去像单一数据库



## Redis

Redis是一个基于内存的高效的键值型非关系型数据库，存取效率极高，而且支持多种存储数据结构。支持数据持久化，可以将内存中数据保存在磁盘中，提供list/set/sorted set/hash等数据结构的存储，因此也被称为数据结构服务器。

### 启动Redis

linux端：

启动：sudo /etc/init.d/redis-server start

停止： sudo /etc/init.d/redis-server stop

连接：redis-cli

win端：

redis作为windows服务启动方式  
redis-server --service-install redis.windows.conf

连接Redis：redis-cli -h 127.0.0.1 -p 6379 –a password 如果在本机运行且无密码，可以直接输入：redis-cli  
启动服务：redis-server --service-start  
停止服务：redis-server --service-stop

### redis数据类型

redis支持五种数据类型： string(字符串)、hash(哈希)、list(列表)、set(集合)及sorted set(有序集合)。

1. string类型

string是Redis最基本的类型，一个key对应一个value。string类型可以包含任何数据，比如jpg图片或者序列化的对象。从内部实现来看，其实string可以看作byte数组，是二进制安全的，一个键最大能存储512MB。

1. hash类型

Redis的hash类型是一个string类型的field和value的映射表，特别适合用于存储对象。相较于将对象的每个字段存成单个string类型，将一个对象存储在hash类型中会占用更少的内存，并且可以更方便地存取整个对象。每个hash可以存储个键对，大约40多亿条。

1. list类型

list类型是一个双向键表，其每个子元素都是string类型，最多可存储个元素，大约40多亿，可以使用push, pop操作从链表的头部或者尾部添加删除元素，key可以理解为链表的名字。

1. set类型

set是string类型的无序集合，最大可以包含个元素，约40多亿。对集合可以添加删除元素，也可以对多个集合求交并差，key可以理解为集合的名字。set通过hash table实现，添加、删除和查找的复杂度都是O(1)。hash table会随着添加或者删除自动调整大小。重复数据会被忽略，即使用set类型进行去重操作。

1. sorted set类型

和set一样，sorted set也是string类型元素的集合，不允许重复的成员。sorted set算是set的升级版本，它在set的基础上增加了一个顺序属性，会关联一个double类型的score。这一属性在添加和修改元素的时候可以指定，每次指定后，sorted set会自动重新按新的值调整顺序。sorted set成员是唯一的，但score却可以重复。

### python操作redis

#### 连接redis

*import* redis  
r=redis.Redis(host='127.0.0.1',port=6379)

用连接池管理redis连接，避免每次建立、释放连接的开销

pool=redis.ConnectionPool(host='127.0.0.1',port=6379)  
r=redis.Redis(connection\_pool=pool)

#### 操作string类型

1. **set(name, value, ex=None,px=None,nx=False, xx=False)**

设置键值对

name：键

value：值

ex：过期时间（秒） 时间过后name对应值为None

px：过期时间（毫秒）

nx：True——name不存在时执行当前set操作

xx：True——name存在时执行当前set操作

1. **setnx(name, value)**

当name不存在时才能执行

1. setex(name,value,time)

设置键值对，time可以是timedelta对象或者秒，过期后name为None

1. psetex(name,time\_ms,value)

设置键值对，time可以是timedelta对象或者秒，过期后name为None

1. **mset(\*args, \*\*kwargs)**批量设置键值对 r.mset(age=20,country=’china’)

keys：多个键

1. **mget(keys, \*args)**批量获取键值 r.mset(‘age’,’country’)
2. **getset(name, value)**设置新值并获取原来的值
3. **getrange(key, start, end)**根据字节获取字符串  
   参数：  
   start，起始位置（字节）  
   end，结束位置（字节）
4. **setrange(name, offset, value)**修改字符串内容，从指定字符串索引开始向后替换（新值太长时，则向后添加）  
   offset，字符串的索引，字节

value，要设置的值

1. **setbit(name, offset, value)**对name对应值的二进制形式的位进行操作  
   offset，位的索引（将值变换成二进制后再进行索引）  
   value，值只能是 1 或 0
2. **getbit(name, offset)**获取name对应的值的二进制形式中的某位的值 （0或1）
3. **bitcount(key, start=None, end=None)**获取name对应的值的二进制表示中 1 的个数  
   start 字节起始位置  
   end，字节结束位置
4. **bitop(operation, dest, \*keys)**获取多个值，并将值做位运算，将最后的结果保存至新的name对应的值

operation,AND（并） 、 OR（或） 、 NOT（非） 、 XOR（异或）  
dest, 新的Redis的name  
\*keys,要查找的Redis的name

1. **strlen(name)**返回name对应值的字节长度（一个汉字3个字节）
2. **incr(self, name, amount=1)**自增 name对应的值，当name不存在时，则创建name＝amount，否则，则自增。  
   amount,自增数（必须是整数）
3. **incrbyfloat(self, name, amount=1.0)**自增 name对应的值，当name不存在时，则创建name＝amount，否则，则自增。  
   amount,自增数（浮点型）
4. **decr(self, name, amount=1)**自减 name对应的值，当name不存在时，则创建name＝amount，否则，则自减。  
   amount,自减数（整数)
5. **append(key, value)**name对应的值后面追加内容

#### 操作hash类型

1. **hset(name, key, value)**

设置name对应的hash中的一个键值对（不存在，则创建；否则，修改）  
name，hash的name  
key，name对应的hash中的key  
value，name对应的hash中的value  
注：  
hsetnx(name, key, value)

当name对应的hash中不存在当前key时则创建（相当于添加）

1. **hmset(name, mapping)**在name对应的hash中批量设置键值对  
   name，hash的name  
   mapping，字典
2. **hget(name)**获取name对应hash的键值
3. **hlen(name)**获取name对应的hash中键值对的个数
4. **hmget(name,keys,\*args)**

批量获取name对应的hash中多个key的值

\*args：要获取的key

#### 操作set类型

#### 操作sorted set类型

## Email

爬虫在运行过程中遇到异常或者服务器遇到问题，可以通过Email及时向自己报告。

发送邮件的协议是STMP, Python内置对SMTP的支持，可以发送纯文本邮件、HTML邮件以及带附件的邮件。Python对SMTP支持有smtplib和email两个模块，email负责构造邮件，smtplib负责发送邮件。

邮箱需要开启STMP服务，获取授权码后才能使用

*from* email.header *import* Header  
*from* email.mime.text *import* MIMEText  
*from* email.utils *import* parseaddr, formataddr  
*import* smtplib  
  
*def* \_format\_addr(*s*):  
 name, addr = parseaddr(*s*)  
 *return* formataddr((Header(name, 'utf-8').encode(), addr))  
  
from\_addr = '826908021@qq.com'  
password = 'jtomjlibaioqbdch'  
to\_addr = '799960740@qq.com'  
smtp\_server = 'smtp.qq.com'  
  
msg = MIMEText('hello, send by Python...', 'plain', 'utf-8')  
msg['From'] = \_format\_addr('Python爱好者 <%s>' % from\_addr)  
msg['To'] = \_format\_addr('管理员 <%s>' % to\_addr)  
msg['Subject'] = Header('来自SMTP的问候……', 'utf-8').encode()  
  
server = smtplib.SMTP(smtp\_server, 587)  
server.set\_debuglevel(1)  
server.login(from\_addr, password)  
server.sendmail(from\_addr, [to\_addr], msg.as\_string())  
server.quit()

# 爬虫框架——[Scrapy](http://scrapy-chs.readthedocs.io/zh_CN/0.24/intro/tutorial.html)

Scrapy是一个用Python写的Crawler Framework，使用Twisted异步网络库来处理网络通信，架构清晰，并且包含了各种中间件接口。

Scrapy中的各大组件及其功能:

* Scrapy引擎(Engine)。控制数据流在系统的所有组件中流动，并在相应动作发生时触发事件。
* 调度器(Scheduler )。从引擎接收Request并将它们人队，以便之后引擎请求request时提供给引擎。
* 下载器(Downloader )。获取页面数据并提供给引擎，而后提供给Spider,
* Spider, Scrapy用户编写用于分析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。
* Item Pipeline,处理被Spider提取出来的Item。典型的处理有清理验证及持久化(例如存储到数据库中)。



* 下载器中间件(Downloader midlewares)。引擎及下载器之间的特定钩子(specific hook)，处理Downloader传递给引擎的Response。其提供了一个简便的机制，通过插人自定义代码来扩展Scrapy功能。
* Spider中间件(Spider middlewares )。引擎及Spider之间的特定钩子(specific hook)，处理Spider的输人(response)和输出(Items及Requests)。其提供了一个简便的机制，通过插入自定义代码来扩展Scrapy功能。

工作流程：

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

2)引擎从Spider中获取到第一个要爬取的URL并通过调度器(Scheduler)以Request进行调度。

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的Request,引擎关闭该网站

## 创建项目

命令行：**scrapy startproject 项目名**

│ scrapy.cfg #项目的配置文件，存放的目录被认为是项目的根目录

└─csdn\_scrapy #该项目的python模块。将在此加入代码。

│ items.py

│ middlewares.py

│ pipelines.py

│ settings.py

│ \_\_init\_\_.py

└─spiders #放置spider代码的目录.

csdn.py

\_\_init\_\_.py

## 命令行工具

以无参数的方式启动Scrapy工具。会给出一些使用帮助以及可用的命令

scrapy startproject myproject 创建项目

cd myproject

scrapy genspider mydomain mydomain.com 控制项目

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)。

查看所有可用的命令: scrapy –h

获取关于每个命令的详细内容: scrapy <command> -h

1. 全局命令:

* startproject scrapy startproject <project\_name>

在project\_name文件夹下创建一个名为project\_name的Scrapy项目。

* settings scrapy settings [options]

获取Scrapy的设定

* runspider scrapy runspider <spider\_file.py>

未创建项目的情况下，运行一个编写在Python文件中的spider。

* shell scrapy shell [url]

以给定的URL(如果给出)或者空(没有给出URL)启动Scrapy shell。

* fetch scrapy fetch <url>

使用Scrapy下载器下载给定的URL，并将获取到的内容送到标准输出。

* view scrapy view <url>

在浏览器中打开给定的URL，并以Scrapy spider获取到的形式展现。

* version scrapy version [-v]

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。

1. 项目(Project-only)命令:

* crawl scrapy crawl <spider>

使用spider进行爬取。

* check scrapy check [-l] <spider>

运行contract检查。

* list scrapy list

列出当前项目中所有可用的spider。每行输出一个spider。

* edit scrapy edit <spider>

使用 EDITOR 中设定的编辑器编辑给定的spider

* parse scrapy parse <url> [options]

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

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
* genspider scrapy genspider [-t template] <name> <domain>

在当前项目中创建spider。

* deploy scrapy deploy [ <target:project> | -l <target> | -L ]

将项目部署到Scrapyd服务

* bench scrapy bench

运行benchmark测试。

1. shell分析

scrapy shell "url"

暂停、恢复和停止

engine.pause() engine.unpause() engine.stop()

## Item

从非结构性的数据源提取结构性数据，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。

### 声明Item

Item使用class定义语法以及 Field 对象来声明

*import* scrapy  
*class* CsdnScrapyItem(scrapy.Item):  
 # define the fields for your item here like:  
 # name = scrapy.Field()  
 # 文章名  
 title = scrapy.Field()  
 # 文章链接  
 link\_url = scrapy.Field()

### Item字段(Item Fields)

Field 对象指明了每个字段的元数据(metadata)。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。

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。

用来声明item的 Field 对象并没有被赋值为class的属性。不过可以通过 Item.fields 属性进行访问。

### 提取Item

Scrapy使用了一种基于 XPath 和 CSS 表达式机制: Scrapy Selectors。

xpath：

* /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

Selector：

* 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。

## Item Pipeline

Item传递到Item Pipeline，被定义的Item Pipeline组件会顺次调用，完成一连串的处理过程，比如数据清洗、存储等。

Item Pipeline的主要功能

* 清理HTML数据。
* 验证爬取数据，检查爬取字段。
* 查重并丢弃重复内容。
* 将爬取结果保存到数据库。

可以自定义Item Pipeline，只需要实现指定的方法，其中必须要实现的一个方法是： **process\_item(item, spider)**

另外还有如下几个比较实用的方法

* open\_spider(spider)
* close\_spider(spider)
* from\_crawler(cls, crawler)

**1. process\_item(item, spider)**

Item Pipeline会默认调用这个方法对Item进行处理。必须返回Item类型的值或者抛出一个DropItem异常。

**参数**

* item，是Item对象，即被处理的Item。
* spider，是Spider对象，即生成该Item的Spider。

**返回类型**

如果它返回的是Item对象，那么此Item会被低优先级的Item Pipeline的process\_item()方法处理，直到所有的方法被调用完毕。

如果它抛出的是DropItem异常，那么此Item会被丢弃，不再进行处理。

**2. open\_spider(self, spider)**

open\_spider()方法是在Spider开启的时候被自动调用的。在这里我们可以做一些初始化操作，如开启数据库连接等。其中，参数spider就是被开启的Spider对象。

**3. close\_spider(spider)**

close\_spider()方法是在Spider关闭的时候自动调用的。在这里我们可以做一些收尾工作，如关闭数据库连接等。其中，参数spider就是被关闭的Spider对象。

**4. from\_crawler(cls, crawler)**

from\_crawler()方法是一个类方法，用@classmethod标识，是一种依赖注入的方式。它的参数是crawler，通过crawler对象，我们可以拿到Scrapy的所有核心组件，如全局配置的每个信息，然后创建一个Pipeline实例。参数cls就是Class，最后返回一个Class实例。

*class* CsdnScrapyPipeline(object):  
 *def \_\_init\_\_*(self,*mongo\_url*,*mongo\_db*):  
 self.mongo\_url=*mongo\_url* self.mongo\_db=*mongo\_db* @classmethod  
 *def* from\_crawler(cls,*crawler*):  
 *return* cls(  
 mongo\_url=*crawler*.settings.get("MONGO\_URL"),  
 mongo\_db=*crawler*.settings.get("MONGODB\_DATABASE")  
 )  
 *def* open\_spider(self,spider):  
 self.client=pymongo.MongoClient(self.mongo\_url)  
 self.db=self.client[self.mongo\_db]  
 *def* process\_item(self, *item*, spider):  
 self.db['csdn'].insert(dict(*item*))  
 *return item  
 def* close\_spider(self,spider):  
 self.client.close()

## Middleware

### Downloader Middleware

Downloader Middleware即下载中间件，它是处于Scrapy的Request和Response之间的处理模块

Downloader Middleware在整个架构中起作用的位置是以下两个：

* 在Scheduler调度出队列的Request发送给Downloader下载之前，也就是我们可以在Request执行下载之前对其进行修改。
* 在下载后生成的Response发送给Spider之前，也就是我们可以在生成Resposne被Spider解析之前对其进行修改。

Scrapy已经提供了许多Downloader Middleware，被DOWNLOADER\_MIDDLEWARES\_BASE变量所定义。数字小的Downloader Middleware会被优先调用。

{

'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,

…

}

如果自己定义的Downloader Middleware要添加到项目里，DOWNLOADER\_MIDDLEWARES\_BASE变量不能直接修改。Scrapy提供了DOWNLOADER\_MIDDLEWARES，修改这个变量添加自己定义的Downloader Middleware，以及禁用DOWNLOADER\_MIDDLEWARES\_BASE里面定义的Downloader Middleware。

**核心方法**

只需要实现至少一个方法，就可以定义一个Downloader Middleware

* process\_request(request, spider)。
* process\_response(request, response, spider)。
* process\_exception(request, exception, spider)。

**1. process\_request(request, spider)**

Request被Scrapy引擎调度给Downloader之前，process\_request()方法就会被调用，可以用process\_request()方法对Request进行处理。

方法的返回值必须为None、Response对象、Request对象之一，或者抛出IgnoreRequest异常。

**参数**

* **request**，是Request对象，即被处理的Request。
* **spider**，是Spdier对象，即此Request对应的Spider。

返回类型不同，产生的效果也不同。下面归纳一下不同的返回情况。

* 当返回是None时，Scrapy将继续处理该Request，接着执行其他Downloader Middleware的process\_request()方法，一直到Downloader把Request执行后得到Response。
* 当返回为Response对象时，更低优先级的Downloader Middleware的process\_request()和process\_exception()方法就不会被继续调用，每个Downloader Middleware的process\_response()方法转而被依次调用。调用完毕之后，直接将Response对象发送给Spider来处理。
* 当返回为Request对象时，更低优先级的Downloader Middleware的process\_request()方法会停止执行。这个Request会重新放到调度队列里，其实它就是一个全新的Request，等待被调度。如果被Scheduler调度了，那么所有的Downloader Middleware的process\_request()方法会被重新按照顺序执行。
* 如果IgnoreRequest异常抛出，则所有的Downloader Middleware的process\_exception()方法会依次执行。如果没有一个方法处理这个异常，那么Request的errorback()方法就会回调。如果该异常还没有被处理，那么它便会被忽略。

**2. process\_response(request, response, spider)**

Downloader执行Request下载之后，会得到对应的Response。Scrapy引擎便会将Response发送给Spider进行解析。在发送之前，我们都可以用process\_response()方法来对Response进行处理。方法的返回值必须为Request对象、Response对象之一，或者抛出IgnoreRequest异常。

**参数**

* request，是Request对象，即此Response对应的Request。
* response，是Response对象，即此被处理的Response。
* spider，是Spider对象，即此Response对应的Spider。

下面归纳一下不同的返回情况。

* 当返回为Request对象时，更低优先级的Downloader Middleware的process\_response()方法不会继续调用。该Request对象会重新放到调度队列里等待被调度，它相当于一个全新的Request。然后，该Request会被process\_request()方法顺次处理。
* 当返回为Response对象时，更低优先级的Downloader Middleware的process\_response()方法会继续调用，继续对该Response对象进行处理。
* 如果IgnoreRequest异常抛出，则Request的errorback()方法会回调。如果该异常还没有被处理，那么它便会被忽略。

**3. process\_exception(request, exception, spider)**

当Downloader或process\_request()方法抛出异常时，例如抛出IgnoreRequest异常，process\_exception()方法就会被调用。方法的返回值必须为None、Response对象、Request对象之一。

**参数**

* request，是Request对象，即产生异常的Request。
* exception，是Exception对象，即抛出的异常。
* spdier，是Spider对象，即Request对应的Spider。

下面归纳一下不同的返回值。

* 当返回为None时，更低优先级的Downloader Middleware的process\_exception()会被继续顺次调用，直到所有的方法都被调度完毕。
* 当返回为Response对象时，更低优先级的Downloader Middleware的process\_exception()方法不再被继续调用，每个Downloader Middleware的process\_response()方法转而被依次调用。
* 当返回为Request对象时，更低优先级的Downloader Middleware的process\_exception()也不再被继续调用，该Request对象会重新放到调度队列里面等待被调度，它相当于一个全新的Request。然后，该Request又会被process\_request()方法顺次处理。

### Spider Middleware

Spider Middleware是介入到Scrapy的Spider处理机制的钩子框架。

Spider Middleware有如下三个作用。

* 在Downloader生成的Response发送给Spider之前对Response进行处理。
* 在Spider生成的Request发送给Scheduler之前对Request进行处理。
* 在Spider生成的Item发送给Item Pipeline之前对Item进行处理。

Scrapy已经提供了许多Spider Middleware，被SPIDER\_MIDDLEWARES\_BASE变量定义。

Spider Middleware首先加入到SPIDER\_MIDDLEWARES设置中，该设置会和Scrapy中SPIDER\_MIDDLEWARES\_BASE定义的Spider Middleware合并。然后根据键值的数字优先级排序，得到一个有序列表。

**核心方法**

只需要实现其中一个方法就可以定义一个 Spider Middleware

* process\_spider\_input(response, spider)。
* process\_spider\_output(response, result, spider)。
* process\_spider\_exception(response, exception, spider)。
* process\_start\_requests(start\_requests, spider)。

**1. process\_spider\_input(response, spider)**

当Response被Spider Middleware处理时，process\_spider\_input()方法被调用。

**参数**

* response，是Response对象，即被处理的Response。
* spider，是Spider对象，即该Response对应的Spider。
* process\_spider\_input()应该返回None或者抛出一个异常。

如果它返回None，Scrapy将会继续处理该Response，调用所有其他的Spider Middleware，直到Spider处理该Response。

如果它抛出一个异常，Scrapy将不会调用任何其他Spider Middleware的process\_spider\_input()方法，而调用Request的errback()方法。errback的输出将会被重新输入到中间件中，使用process\_spider\_output()方法来处理，当其抛出异常时则调用process\_spider\_exception()来处理。

**2. process\_spider\_output(response, result, spider)**

当Spider处理Response返回结果时，process\_spider\_output()方法被调用。

**参数**

response，是Response对象，即生成该输出的Response。

result，包含Request或Item对象的可迭代对象，即Spider返回的结果。

spider，是Spider对象，即其结果对应的Spider。

process\_spider\_output()必须返回包含Request或Item对象的可迭代对象。

**3. process\_spider\_exception(response, exception, spider)**

当Spider或Spider Middleware的process\_spider\_input()方法抛出异常时，process\_spider\_exception()方法被调用。

**参数**

* response，是Response对象，即异常被抛出时被处理的Response。
* exception，是Exception对象，即被抛出的异常。
* spider，是Spider对象，即抛出该异常的Spider。

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。

如果它返回None，Scrapy将继续处理该异常，调用其他Spider Middleware中的process\_spider\_exception()方法，直到所有Spider Middleware都被调用。

如果它返回一个可迭代对象，则其他Spider Middleware的process\_spider\_output()方法被调用，其他的process\_spider\_exception()不会被调用。

**4. process\_start\_requests(start\_requests, spider)**

process\_start\_requests()方法以Spider启动的Request为参数被调用，执行的过程类似于process\_spider\_output()，只不过它没有相关联的Response，并且必须返回Request。

**参数**

start\_requests，是包含Request的可迭代对象，即Start Requests。

spider，是Spider对象，即Start Requests所属的Spider。

process\_start\_requests()必须返回另一个包含Request对象的可迭代对象。

## spiders

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)。

*class* ZhihuComSpider(scrapy.Spider):

爬取的循环:

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。

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 。

*def* start\_requests(self):  
 *yield* scrapy.Request(url=self.start\_urls[0],callback=self.parse)

1. 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回调函数(函数可相同，可以说parse()或者其他函数)。
2. 在回调函数内，可以使用 选择器(Selectors) (也可以使用BeautifulSoup, lxml 或其他)来分析网页内容，并根据分析的数据生成item。
3. 最后，由spider返回的item可以由 Item Pipeline 处理被存到数据库或使用 Feed exports 存入到文件中。

### scrapy.Spider

*class* comicspider(scrapy.Spider):

通过继承scrapy.Spider类来实现Spider 模块是一种最简单的spider。

每个spider必须继承自scrapy.Spider类(包括Scrapy自带的spider以及自定义的spider ),Spider并没有提供什么特殊的功能，仅仅提供了start requests()的默认实现，读取并请求spider属性中的start urls，并根据返回的response调用spider的parse方法。

**Spider类常用属性：**

* **name**。定义spider名字，Scrapy使用spider的名字来定位和初始化spider,所以它必须是唯一的。不过可以生成多个相同的spider实例，这没有任何限制。name是spider最重要的属性，而且是必需的。一个常见的做法是以该网站的域名来命名spider。例如，如果spider爬取cnblogs.com，该spider通常会被命名为cnblogs
* **allowed\_domains**。可选。包含了spider允许爬取的域名列表，防止爬虫跑飞。当OffsiteMidleware组件启用时，域名不在列表中的URL不会被跟进。
* **start\_urls**为URL列表。当没有使用start\_requests( )方法配置Requests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URL将会从获取到的数据中提取。
* **custom\_settings**。该设置是一个dict。当启动spider时，该设置将会覆盖项目级的设置。由于设置必须在初始化前被更新，所以该属性必须定义为class属性。
* **crawler**。该属性在初始化class后，由类方法from\_crawler( )设置，并且链接了本spider实例对应的Crawler对象。Crawler包含了很多项目中的组件，作为单一的入口点(例如插件、中间件、信号管理器等)。
* **settings**。它是一个Settings对象，利用它我们可以直接获取项目的全局设置变量。

**Spider类常用方法：**

**start\_requests()**。回一个可迭代对象，包含了spider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。如果想要修改最初爬取某个网站的Request对象，可以重写该方法。例如在进行深层次爬取时，在启动阶段需要POST登录某个网站，获取用户权限

*class* MySpider(scrapy.Spider):  
 name ='myspider'  
 *def* start\_requests(self):  
 *return*[scrapy.FormRequest("http://www.example.com/login",  
 formdata = {'user':'john','pass':  
 'secret'},callback=self.login)]

*def* login(self,*response*):  
 *pass*

* 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
* parse(response)。response参数即用于分析的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跟进的URL，该方法及其他的Request回调函数必须返回一个包含Request, dict或Item的可迭代的对象。
* closed(reason)。当spider关闭时，该函数被调用。可以用来在spider关闭时释放占用的资源。

**Spider参数**

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。

### scrapy.spiders.CrawlSpider

CrawlSpider是Scrapy提供的一个通用Spider。在Spider里可以指定一些爬取规则来实现页面的提取，这些爬取规则由一个专门的数据结构Rule表示。Rule里包含提取和跟进页面的配置，Spider会根据Rule来确定当前页面中的哪些链接需要继续爬取、哪些页面的爬取结果需要用哪个方法解析等。

CrawlSpider继承自Spider类。除了Spider类的所有方法和属性，它还提供了一个非常重要的属性和方法。

* rules，它是爬取规则属性，是包含一个或多个Rule对象的列表。每个Rule对爬取网站的动作都做了定义，CrawlSpider会读取rules的每一个Rule并进行解析。
* parse\_start\_url()，它是一个可重写的方法。当start\_urls里对应的Request得到Response时，该方法被调用，它会分析Response并必须返回Item对象或者Request对象。

**rule参数  
class scrapy.contrib.spiders.Rule**

**(link\_extractor, callback=None, cb\_kwargs=None, follow=None, process\_links=None, process\_request=None)**

**link\_extractor**：是Link Extractor对象。通过它，Spider可以知道从爬取的页面中提取哪些链接。提取出的链接会自动生成Request。它又是一个数据结构，一般常用LxmlLinkExtractor对象作为参数，其定义和参数如下所示：

**class scrapy.linkextractors.lxmlhtml.LxmlLinkExtractor(allow=(), deny=(), allow\_domains=(), deny\_domains=(), deny\_extensions=None, restrict\_xpaths=(), restrict\_css=(), tags=('a', 'area'), attrs=('href', ), canonicalize=False, unique=True, process\_value=None, strip=True)**

* allow是一个正则表达式或正则表达式列表，它定义了从当前页面提取出的链接哪些是符合要求的，只有符合要求的链接才会被跟进。deny则相反。
* allow\_domains定义了符合要求的域名，只有此域名的链接才会被跟进生成新的Request，它相当于域名白名单。deny\_domains则相反。
* restrict\_xpaths定义了从当前页面中XPath匹配的区域提取链接，其值是XPath表达式或XPath表达式列表。
* restrict\_css定义了从当前页面中CSS选择器匹配的区域提取链接，其值是CSS选择器或CSS选择器列表。
* 还有一些其他参数代表了提取链接的标签、是否去重、链接的处理等内容，使用的频率不高。可以参考文档的参数说明：<http://scrapy.readthedocs.io/en/latest/topics/link-extractors.html#module-scrapy.linkextractors.lxmlhtml>。

**callback**：即回调函数，和之前定义Request的callback有相同的意义。每次从link\_extractor中获取到链接时，该函数将会调用。该回调函数接收一个response作为其第一个参数，并返回一个包含Item或Request对象的列表。注意，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parse()方法覆盖了，CrawlSpider将会运行失败。

**cb\_kwargs**：字典，它包含传递给回调函数的参数。

**follow**：布尔值，即True或False，它指定根据该规则从response提取的链接是否需要跟进。如果callback参数为None，follow默认设置为True，否则默认为False。

**process\_links**：指定处理函数，从link\_extractor中获取到链接列表时，该函数将会调用，它主要用于过滤。

**process\_request**：同样是指定处理函数，根据该Rule提取到每个Request时，该函数都会调用，对Request进行处理。该函数必须返回Request或者None。

## scrapy分布式实现

### 搭建redis服务器

分布式部署，多台主机需要共享爬取队列和去重集合，这两部分内容都存于Redis数据库中，需要[搭建一个可公网访问的Redis服务器](https://germey.gitbooks.io/python3webspider/content/1.4.3-Redis%E7%9A%84%E5%AE%89%E8%A3%85.html)。

### scrapy-redis组件配置

scrapy实现分布式需要redis作为消息队列

 

将scrapy项目添加scrapy-redis组件，需要修改settings.py。



* 1. 修改scrapy核心调度器，替换为scrapy-redis调度器

SCHEDULER = "scrapy\_redis.scheduler.Scheduler"

* 1. 替换去重class

DUPEFILTER\_CLASS = "scrapy\_redis.dupefilter.RFPDupeFilter"

* 1. 添加redis连接信息

REDIS\_URL = 'redis://:password@ip:port' 需要用户名和密码，优先使用url

REDIS\_HOST='127.0.0.1' 不需要用户名和密码  
REDIS\_PORT=6379

* 1. 添加ITEM\_PIPELINES，scrapy-redis的pipelines，数据存入redis，数字和scrapy的pipelines修改为不同的，一般不使用

'scrapy\_redis.pipelines.RedisPipeline': 301

分布式存储，如果注释掉scrapy-redis的pipelines，各主机爬取的东西存储到本地主机，不会再同步至一个redis库，爬取效率会提升

### 配置存储目标

设置MongoDB存储，本地爬虫会连接localhost的MongoDB，可以在服务器搭建MongoDB服务，所有存储存放到同一个MongoDB中。

1. [安装MongoDB](https://germey.gitbooks.io/python3webspider/content/1.4.2-MongoDB%E7%9A%84%E5%AE%89%E8%A3%85.html)
2. 爬虫settings.py文件配置

MONGO\_URL = 'mongodb://154.8.171.68:27017'  
MONGODB\_DATABASE = "python"  
MONGODB\_SHEETNAME = "csdn"

1. pipelines.py文件配置

*def \_\_init\_\_*(self):  
 mongo\_url=settings["MONGO\_URL"]  
 mongo\_db=settings["MONGODB\_DATABASE"]  
 mongo\_sheet=settings["MONGODB\_SHEETNAME"]  
 client = pymongo.MongoClient(mongo\_url)  
 db = client[mongo\_db]  
 self.post = db[mongo\_sheet]  
*def* process\_item(self, *item*, spider):  
 data=dict(*item*)  
 self.post.insert(data)  
 *return item*

## 分布式爬虫部署—scrapyd

scrapyd是运行scrapy爬虫的服务程序，支持以http命令方式通过json api进行发布、删除、启动、停止爬虫程序的操作，同时可以管理多个爬虫，每个爬虫可以有多个版本，也是部署分布式爬虫的有效手段。

scrapyd在Linux的安装需要添加配置文件，修改绑定ip使外网可以访问。[教程](https://germey.gitbooks.io/python3webspider/content/1.9.2-Scrapyd%E7%9A%84%E5%AE%89%E8%A3%85.html)

### 启动（后台运行）

命令行输入scrapyd启动，默认监听6800端口，浏览器输入<http://127.0.0.1:6800/>打开scrapyd页面。

linux后台运行scrapyd可使用scrapyd > /dev/null &命令，控制台输出忽略

想记录输出日志，scrapyd > ~/scrapyd.log & 则会输出到~/scrapyd.log 文件中

### scrapyd-client部署

用于发布scrapy爬虫的工具，安装后自动在python安装目录下scripts文件夹中生成scrapyd-deploy工具，可直接使用python scrapyd-deploy方式运行。



1. 打包项目
2. 调用json接口上传到远程服务器

scrapyd-client安装完后，将scrapyd-deploy拷贝到爬虫项目目录下，与scrapy.cfg在同一级目录下。

1. 调用接口

方法一：修改scrapy.cfg文件



[deploy:100]  
url = http:// 154.8.171.68:6800/addversion.json  
project = zhihu

url为scrapyd服务器网址，project为项目名称，修改deploy为[deploy:100]，表示把爬虫发布到名为100的爬虫服务器上，一般在需要同时发布到多个目标服务器时使用。

方法二：使用python-scrapyd-api

1. 使用scrapyd-deploy命令将项目部署到主机

切换到工程目录下

爬虫发布命令：scrapyd-deploy <target> -p <project> --version<version>

target: deploy后面的名称

project: 自行定义名称，与爬虫工程名字无关

version: 自行定义版本号，不写默认为当前时间戳





发布完成后根据API发送启动爬虫命令，爬虫就可以正常工作了。

### API

scrapyd支持十种操作方式，**curl+api**

1. **启动服务器某一爬虫**：http://localhost:6800/schedule.json -d project=项目名称 -d spider=爬虫名称 POST

响应：{"status": "ok", "jobid": "6487ec79947edab326d6db28a2d86511e8247444"}

启动远程接口调起爬虫，命令使用多少次，调起多少个爬虫进程运行

1. 给工程添加版本，若不存在则创建：http://localhost:6800/addversion.json -F project=项目名 -F version=项目版本 -F [egg=@项目名称.egg](mailto:egg=@项目名称.egg) POST

响应：{"status": "ok", "spiders": 3}

本地代码通过接口部署到远程主机，打包为.egg文件上传，方式比较繁琐，可使用scrapyd-client部署

1. 获取项目列表：<http://localhost:6800/listprojects.json> GET

响应：{"status": "ok", "projects": ["项目名称1", "项目名称2"]}

1. 获取项目下已发布爬虫列表：[http://localhost:6800/listspiders.json?project=](http://localhost:6800/listspiders.json?project=myproject)项目名称 GET

响应：{"status": "ok", "spiders": ["spider1", "spider2", "spider3"]}

1. 获取爬虫运行状态：[http://localhost:6800/listjobs.json?project=](http://localhost:6800/listjobs.json?project=myproject)项目名称 GET

响应：{"status": "ok",

"pending": [{"id": "78391cc0fcaf11e1b0090800272a6d06", "spider": "spider1"}],

"running": [{"id": "422e608f9f28cef127b3d5ef93fe9399", "spider": "spider2", "start\_time": "2012-09-12 10:14:03.594664"}],

"finished": [{"id": "2f16646cfcaf11e1b0090800272a6d06", "spider": "spider3", "start\_time": "2012-09-12 10:14:03.594664", "end\_time": "2012-09-12 10:24:03.594664"}]}

1. 取消一个运行的爬虫任务：http://localhost:6800/cancel.json -d project=项目名称 -d job=任务id POST

响应：{"status": "ok", "prevstate": "running"}

jobid可以通过listjobs.json查看

1. 获取已发布爬虫版本列表：[http://localhost:6800/listversions.json?project=](http://localhost:6800/listversions.json?project=myproject)项目名称 GET

响应：{"status": "ok", "versions": ["r99", "r156"]}

1. 删除某工程，并将工程下各版本爬虫一起删除：http://localhost:6800/delproject.json -d project=项目名称 POST

响应：{"status": "ok"}

1. 获取scrapyd状态：<http://localhost:6800/daemonstatus.json> GET

响应：{ "status": "ok", "running": "0", "pending": "0", "finished": "0", "node\_name": "node-name" }

1. 删除某版本爬虫：http://localhost:6800/delversion.json -d project=项目名称 -d version=爬虫版本 POST

响应：{"status": "ok"}

### Python-scrapy-api

对api进行封装，在Python中可以直接使用

*from* scrapyd\_api *import* ScrapydAPI  
scrapyd=ScrapydAPI('http://localhost:6800/')

scrapyd.list\_jobs('zhihu')

[接口函数说明](https://github.com/MaLei666/python-scrapyd-api)

## 分布式爬虫部署—git同步方式

下载项目文件到指定文件夹

mkdir home/py

sudo git clone https://github.com/MaLei666/Spider.git -b master (使用https格式避免ssh配置)

（更新GitHub代码： git pull origin master）

## Scrapyd对接Docker

Python环境配置问题和版本冲突， 将Scrapyd直接打包成一个Docker镜像，那么在服务器上只需要执行Docker命令就可以启动Scrapyd服务。

### 新建三个文件



配置文件内容：scrapyd.conf

[scrapyd]

eggs\_dir = eggs

logs\_dir = logs

items\_dir =

jobs\_to\_keep = 5

dbs\_dir = dbs

max\_proc = 0

max\_proc\_per\_cpu = 10

fin ished\_to\_keep = 100

poll\_interval = 5.0

bind\_address = 0.0.0.0

http\_port = 6800

debug = off

runner = scrapyd.runner

application = scrapyd.app.application

launcher = scrapyd.launcher.Launcher

webroot = scrapyd.website.Root

[services]

schedule.json = scrapyd.webservice.Schedule

cancel.json = scrapyd.webservice.Cancel

addversion.json = scrapyd.webservice.AddVersion

listprojects.json = scrapyd.webservice.ListProjects

listversions.json = scrapyd.webservice.ListVersions

listspiders.json = scrapyd.webservice.ListSpiders

delproject.json = scrapyd.webservice.DeleteProject

delversion.json = scrapyd.webservice.DeleteVersion

listjobs.json = scrapyd.webservice.ListJobs

daemonstatus.json = scrapyd.webservice.DaemonStatus

* max\_proc\_per\_cpu=10，原本是4，即1核的主机最多同时只能运行4个Scrapy任务，这里设置上限为10，也可以自行设置。
* bind\_address = 0.0.0.0，原本是127.0.0.1，不能公开访问，这里修改为0.0.0.0即可解除此限制。

requirements.txt，将一些Scrapy项目常用的库都列进去  
requests

selenium

aiohttp

beautifulsoup4

pyquery

pymysql

redis

pymongo

flask

django

scrapy

scrapyd

scrapyd-client

scrapy-redis

scrapy-splash

Dockerfile

FROM python:3.6

ADD . /home/py

WORKDIR /home/py

COPY ./scrapyd.conf /etc/scrapyd/

EXPOSE 6800

RUN pip3 install -r requirements.txt

CMD scrapyd

* 第一行的FROM是指在python:3.6这个镜像上构建，也就是说在构建时就已经有了Python 3.6的环境。
* 第二行的ADD是将本地的代码放置到虚拟容器中。它有两个参数：第一个参数是. ，即代表本地当前路径；第二个参数/code代表虚拟容器中的路径，也就是将本地项目所有内容放置到虚拟容器的/code目录下。
* 第三行的WORKDIR是指定工作目录，这里将刚才添加的代码路径设成工作路径，这个路径下的目录结构和当前本地目录结构是相同的，所以在这个目录下可以直接执行库安装命令。
* 第四行的COPY是将当前目录下的scrapyd.conf文件复制到虚拟容器的/etc/scrapyd/目录下，Scrapyd在运行的时候会默认读取这个配置。
* 第五行的EXPOSE是声明运行时容器提供服务端口，注意这里只是一个声明，运行时不一定会在此端口开启服务。这个声明的作用，一是告诉使用者这个镜像服务的运行端口，以方便配置映射，二是在运行使用随机端口映射时，容器会自动随机映射EXPOSE的端口。
* 第六行的RUN是执行某些命令，一般做一些环境准备工作。由于Docker虚拟容器内只有Python 3环境，而没有Python库，所以我们运行此命令来在虚拟容器中安装相应的Python库，这样项目部署到Scrapyd中便可以正常运行。
* 第七行的CMD是容器启动命令，容器运行时，此命令会被执行。这里我们直接用scrapyd来启动Scrapyd服务。

### 构建、运行

三个文件放入文件夹，在此文件夹中执行

docker build -t scrapyd:latest . # 最后有个 . 表示当前目录

docker run -d -p 6800:6800 scrapyd  
打开本地查看scrapyd

### 推送到docker hub

docker tag scrapyd:latest malei666/scrapyd:latest

docker push malei666/scrapyd

## Scrapyd批量部署

### 镜像部署

1. 服务器配置好docker，加入docker用户组

usermod -G docker ubuntu # 给ubuntu用户设置docker附属用户组

cat /etc/group #查看是否成功

1. 从docker hub上pull下scrapyd镜像运行

docker pull malei666/scrapyd:latest

1. Scrapyd镜像启动加到开机启动脚本中

vi /etc/rc.local

添加docker run -d -p 6800:6800 malei666/scrapyd:latest

reboot重启

1. 进入腾讯云后台，点击更多选项制作镜像
2. 使用制作好的镜像创建新的主机

### 模板部署

## Gerapy分布式管理

Gerapy是一个基于Scrapyd、Scrapyd API、Django、Vue.js搭建的分布式爬虫管理框架。

1. 安装gerapy

pip3 install gerapy

1. 新建项目

gerapy init

1. 数据库初始化

gerapy migrate

生成一个SQLite数据库，数据库保存各个主机配置信息、部署版本等

1. 启动Gerapy服务

gerapy runserver

1. 打开<http://localhost:8000>
2. gerapy目录下有一个空的projects文件夹，它就是存放Scrapy目录的文件夹。如果想要部署某个Scrapy项目，只需要将该项目文件放到projects文件夹下即可。

## ScrapySplash

ScrapySplash 是一个 Scrapy 中支持 JavaScript 渲染的工具

# PySpider

PySpider 是支持 JavaScript 渲染的，而这个过程是依赖于 PhantomJS 的

pyspider all启动 PySpider

# docker

## docker镜像使用

当运行容器时，使用的镜像如果在本地中不存在，docker 就会自动从 docker 镜像仓库中下载，默认是从 Docker Hub 公共镜像源下载。[安装](http://www.runoob.com/docker/ubuntu-docker-install.html)

linux 加速器

curl -sSL https://get.daocloud.io/daotools/set\_mirror.sh | sh -s http://0613bbb6.m.daocloud.io Copy

win

http://0613bbb6.m.daocloud.io

### 列出镜像列表

**docker images**

* REPOSITORY：表示镜像的仓库源
* TAG：镜像的标签
* IMAGE ID：镜像ID
* CREATED：镜像创建时间
* SIZE：镜像大小



同一仓库源可以有多个 TAG，代表这个仓库源的不同个版本，如ubuntu仓库源里，有15.10、14.04等多个不同的版本，我们使用 REPOSITORY:TAG 来定义不同的镜像。

### 使用镜像运行容器

**docker run -t -i ubuntu:14.04 /bin/bash**

不指定一个镜像的版本标签 docker 将默认使用 ubuntu:latest 镜像。使用一个不存在的镜像时 Docker 就会自动下载

### 预先获取新的镜像

**docker pull ubuntu:13.10**

### 搜索镜像

**docker search httpd**

* NAME:镜像仓库源的名称
* DESCRIPTION:镜像的描述
* OFFICIAL:是否docker官方发布

### 创建镜像

从docker镜像仓库中下载的镜像不能满足需求时，可以对镜像进行更改：

1. 从已经创建的容器中更新镜像，并且提交这个镜像
2. 使用 Dockerfile 指令来创建一个新的镜像

### 更新镜像

使用镜像来创建一个容器



ID为e218edb10161的容器，是按需求更改的容器。可以通过docker commit提交容器副本。

**docker commit -m="has update" -a="runoob" e218edb10161 runoob/ubuntu:v2**

* -m:提交的描述信息
* -a:指定镜像作者
* e218edb10161：容器ID
* runoob/ubuntu:v2:指定要创建的目标镜像名

### 镜像打包

docker save -o filename.tar  imageID / docker save imageID > filename.tar

#打包

docker load --input filename / docker load < filename # 镜像读取

### 设置镜像标签

使用 docker tag 命令，为镜像添加一个新的标签。

**docker tag 860c279d2fec runoob/centos:dev**

### 删除

docker rmi 镜像名/ID

docker rm 容器ID

#查询所有的容器，过滤出Exited状态的容器，列出容器ID，删除这些容器

sudo docker rm `docker ps -a|grep Exited|awk '{print $1}'`

### 进入容器

docker attach ID

# DeepLearning框架

## TensorFlow

使用Python的captcha库来生成验证码

pip3 install captcha pillow

*from* captcha.image *import* ImageCaptcha  
*from* PIL *import* Image  
text = '1234'  
image = ImageCaptcha()  
captcha = image.generate(text)  
captcha\_image = Image.open(captcha)  
captcha\_image.save('test\_img.jpg')

### 卷积神经网络

**tf.nn.conv2d(input, filter, strides, padding, use\_cudnn\_on\_gpu=None, name=None)**

* input： 指需要做卷积的输入图像，它要求是一个Tensor，具有[batch, in\_height, in\_width, in\_channels]这样的shape，具体含义是[训练时一个batch的图片数量, 图片高度, 图片宽度, 图像通道数]，注意这是一个4维的Tensor，要求类型为float32和float64其中之一
* filter： 相当于CNN中的卷积核，它要求是一个Tensor，具有[filter\_height, filter\_width, in\_channels, out\_channels]这样的shape，具体含义是[卷积核的高度，卷积核的宽度，图像通道数，卷积核个数]，要求类型与参数input相同，有一个地方需要注意，第三维in\_channels，就是参数input的第四维
* strides：卷积时在图像每一维的步长，这是一个一维的向量，长度4
* padding： string类型的量，只能是”SAME”,”VALID”其中之一，这个值决定了不同的卷积方式
* use\_cudnn\_on\_gpu： bool类型，是否使用cudnn加速，默认为true

结果返回一个Tensor，这个输出，就是我们常说的feature map（特征图）