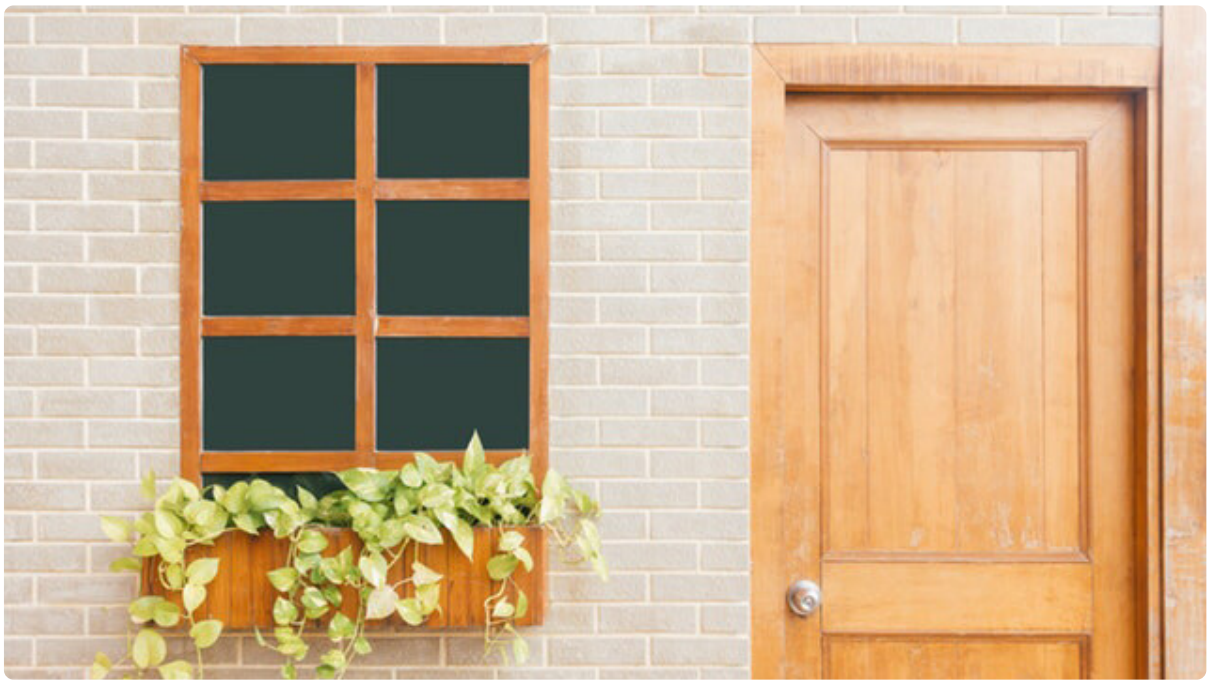
打开花瓣网的大门

更新时间：2019-06-28 17:52:37



> 对自己不满是任何真正有才能的人的根本特征之一。
>
> ——契诃夫

在上一节我们已经有一个基本的开发思路，在这一节一上来当然就是建立一个花瓣网爬虫直接动手开干！

建立基本环境

建立项目并激活 Python 虚环境

```
$ scrapy startproject huaban
$ cd huahan
$ virtualenv -p python3
$ . venv/bin/activate
```

先打开配置文件将爬虫配置为低速运行并且可以随机降速：

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/601.7.8 (KHTML, like Gecko) Version/9.1.3 Safari/537.86.7',

CONCURRENT_REQUESTS = 1
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 5
AUTOTHROTTLE_MAX_DELAY = 60
AUTOTHROTTLE_TARGET_CONCURRENCY = 1.0
AUTOTHROTTLE_DEBUG = False
TELNETCONSOLE_ENABLED = False
HTTPCACHE_ENABLED = False
```

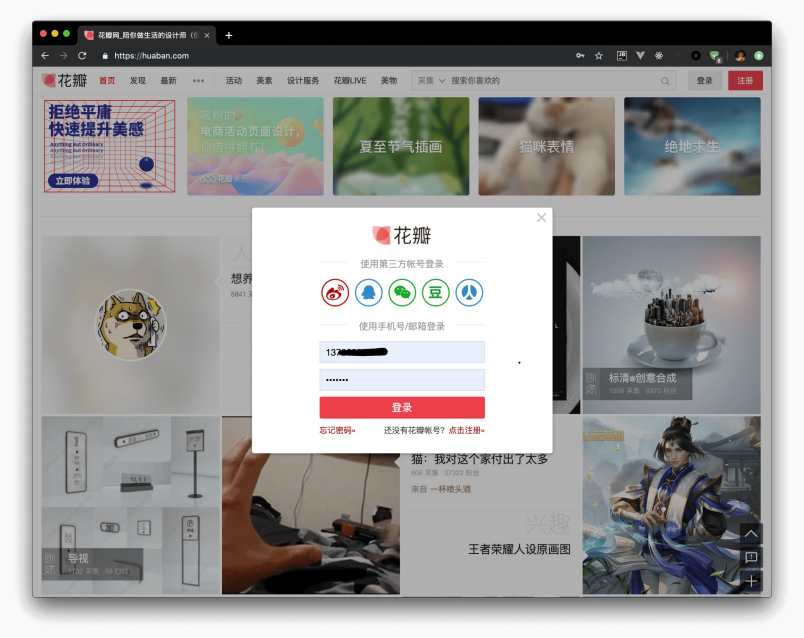
我禁用了 HTTP 缓存，因为花瓣的每个请求所返回的内容是随机的，同一样 URL 返回的内容可能不同，所以不使用缓存。

将`TELNETCONSOLE_ENABLED = False`是为了让项目起动得更快，我们有 PyCharm 的调试器根本就不需要用 Telnet 来进入到 Scrapy 的内嵌调试外壳中。

启用自动降速，并且将`CONCURRENT_REQUESTS = 1`让爬虫只能每次启用一个线程，让爬虫不要因为速度过快而暴露了非人的身份。

如何进行登录？

要开始爬取之前就必须先用已注册的身份进行登录。在浏览器中退出原有的登录然后重新打开登录对话框



打开它的网页源码可以发现这个登录模态框的代码是个标准式的登录表单:

```
<form action="/auth/" method="post" class="mail-login">
  <input type="hidden" name="_ref" value="frame" />
  <input
    type="text"
    name="email"
    placeholder="输入手机号或邮箱"
    class="clear-input"
  />
  <input
    name="password"
    type="password"
    placeholder="密码"
    class="clear-input"
  />
  <a href="#" onclick="return false;" class="btn btn18 rbtn">
    <span class="text">登录</span>
  </a>
</form>
```

如果要实现代码登录只要向`https://huaban.com/auth`发送一个`POST`请求将手机与密码发到服务端可以了。

因为我们并不确定这个想法是否靠谱，也不清楚用代码 POST 到服务器后会有什么限制，那么最佳的办法就是先写个测试来验证一下以上的想法。

先来安装一个`requests`工具来发送网络请求:

```
(venv) $ pip install requests
```

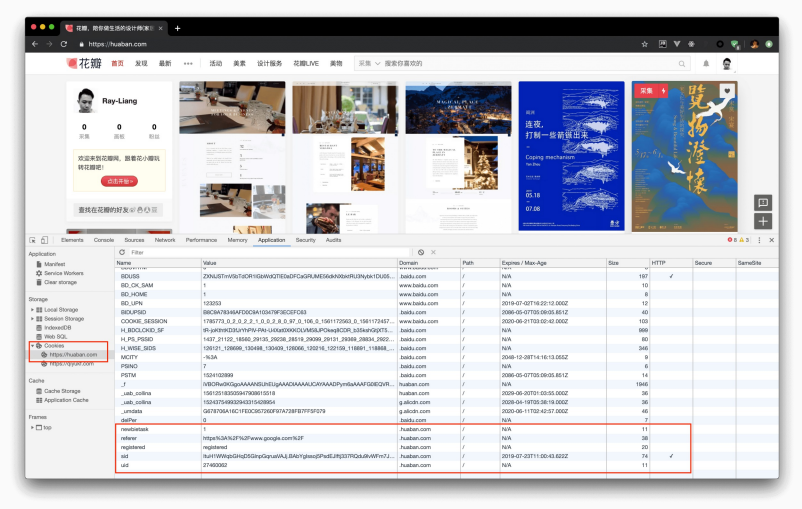
在`tests/test_login.py`文件内添加以下的代码内容:

```
from unittest import TestCase,skip
import requests

class LoginTestCase(TestCase):

    def test_login(self):
        url = 'https://huaban.com/auth/'
        values = {
            'email': '这里输入你注册的手机号',
            'password': '这里输入你注册的密码',
            '_ref': 'frame'
        }
        response = requests.post(url, data=values)
        self.assertIsNotNone(response)
        self.assertEqual(response.status_code, 200)
```

这个测试只是检查请求是否正常地返回`200`的响应代码，但这是不够的。只要仔细一想就能了解到这么一点：那服务器是凭什么记住了我的登录信息的呢？

答案很简单，那就是 Cookie。

绝大多数的网站都会采用 Cookie 来存储用户的登录后信息，花瓣也不例外。即使不写代码也能从 Chrome 的开发人员工具里面找到由花瓣网产生的 Cookie:



那只要在上述测试中加入一个对 Cookie 的断言就可以确认是否能正确地登入花瓣网了：

```
#根据Domain获取花瓣颁发的Cookie
cookies = dict(response.cookies['.huaban.com'])

self.assertIsNotNone(response)
self.assertEqual(response.status_code, 200)
self.assertIsNotNone(cookies['sid'])
self.assertIsNotNone(cookies['uid'])
```

只要在调试器或者直接打印一下`cookies`对象就可以知道花瓣登录之后会生成一个有`sid`与`uid`值对的`Cookie`，也就是这两个值"记住"了我们的登录状态。

FormRequest

接下来就是要将`requests`的方式转化为 Scrapy 的写法。在 Scrapy 中有一个`FormRequest`对象专门用于处理`Form`请求。

这是一个继承自`Request`对象并专门用于处理表单请求的对象。`FormRequest`可以从`Response`对象中自动分析响应网页结果中的表单并对表单内的输入域进行预填充。

以下是FormRequest的构造函数：

```
class scrapy.http.FormRequest(url[, formdata, ...])
```

FormRequest的使用非常简单，只要提供表单提交时的目标 URL 和表单中必备的输入域的“键-值”对的字典或元组对象即可。

将测试中的请求对象改写为：

```
req = FormRequest(
        url='https://huaban.com/auth/',
        method='POST',
        formdata={
          'email': '这里输入你注册的手机号',
          'password': '这里输入你注册的密码',
          '_ref': 'frame'},
        callback=self.after_login
    )
```

那这个请求应该在哪里发起呢？接下来我们就创建一个蜘蛛对象，在开始爬网之前就发起这个FormRequest请求。

创建一个新的蜘蛛:

```
(venv) $ scrapy genspider -t basic bee https://huaban.com
```

修改 settings.py

```
BOT_NAME = 'bee'
```

在 spaiders 目录的__init__.py文件中加以下代码，令到 Scrapy 可以发现这个蜘蛛:

```
from .bee import BeeSpider
```

打开bee.py蜘蛛文件，删除掉allowed_domains和start_urls:

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

class BeeSpider(scrapy.Spider):
    name = 'bee'
```

上文中我们第一次用basic模板来创建蜘蛛，这个模板所创建出来的蜘蛛是最简单的。Spider类提供了蜘蛛的最基本行为与特性，其他蜘蛛都必须继承自该类（包括 Scrapy 自带的蜘蛛以及用户自己编写的蜘蛛）。Spider类并没有提供太多什么特殊的功能，其仅仅请求给定的start_urls/start_requests，并根据返回的结果（resulting responses）然后调用parse方法对返回结果进行深入爬取或提取出目标数据。

借此机会，我们不妨来看看这个基类的源码，然后回顾一下我们曾经使用过的蜘蛛的一些特性，你可能会从中理解很多当时使用之时没有理解清楚的概念，以下是我截取Spider部分重要的源代码：

```
class Spider(object_ref):
    name = None # 定义蜘蛛的名称
    custom_settings = None

    def __init__(self, name=None, **kwargs):
        if name is not None:
            self.name = name
        elif not getattr(self, 'name', None):
            raise ValueError("%s must have a name" % type(self).__name__)
        self.__dict__.update(kwargs)
        if not hasattr(self, 'start_urls'):
            self.start_urls = []

    def start_requests(self):
        """
        就是这个方法将 start_urls生成种子请求
        """
        for url in self.start_urls:
            yield self.make_requests_from_url(url)

    def make_requests_from_url(self, url):
        """
        将URL转换成为Request对象
        """
        return Request(url, dont_filter=True)

    def parse(self, response):
        """
        子类必须实现此方法用于分析返回的响应对象的内容
        """
        raise NotImplementedError
```

从 Spider 类的源代码中可以得知，一旦 Spider 的子类被实例化， __init__ 中的代码就会被执行，

```
def __init__(self, name=None, **kwargs):
        if name is not None:
            self.name = name
        elif not getattr(self, 'name', None):
            raise ValueError("%s must have a name" % type(self).__name__)
        self.__dict__.update(kwargs)
        if not hasattr(self, 'start_urls'):
            self.start_urls = []
```

那么就需要设定 name 与 start_urls 两个属性，这就解释了为什么我们之的范例中所有的蜘蛛在一开始就初始化 name 与 start_urls 两个属性。然而 Spider 的子类被实例化后并不会马上执行爬网，只有 start_requests 被调用时，蜘蛛才会执行爬网的动作。我们重点看一下 start_requests 方法：

```
def start_requests(self):
    for url in self.start_urls:
        yield self.make_requests_from_url(url)
```

这个函数非常简单，它只是一个生成 make_requests_from_url 方法调用结果的枚举器，枚举的条件就是 Spider 实例化时设定的 start_urls，也就是爬虫爬网的“起点”。那么 make_requests_from_url 又干了什么呢？以下是它的代码：

```
def make_requests_from_url(self, url):
    return Request(url, dont_filter=True)
```

它只是一个工厂方法，用于生成 Request 对象。将 start_urls 、 start_requests 和 make_requests_from_url 合起来理解就是： start_requests 一旦被调用就会产生与 start_urls 相同数量的 Request 对象的枚举器。

为什么需要了解这个工作过程？因为只有了解了Request是如何产生的，我们才能在一些特殊的场合深度地定制自己的蜘蛛。例如，在很多情况下start_urls有可能不是一个常量，而是一个生成规则，这样就可以通过方法重写来重新实现start_requests。

如果想要修改最初爬取某个网站的Request对象，可以重写（override）start_requests方法。 例如每次访问花瓣网我们都需要先进行登录，那么我们就可以重写start_requests这个方法，在此进行登录操作。

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

class BeeSpider(scrapy.Spider):
    name = 'bee'

    def start_requests(self):
     req = FormRequest(
            url='https://huaban.com/auth/',
            method='POST',
            formdata={
                'email': '这里输入你注册的手机号',
                'password': '这里输入你注册的密码',
                '_ref': 'frame'},
            callback=self.after_login
        )
     yield req

    def after_login(self, response)
     pass

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

当成功登录后上述的FormRequest对象就会调用after_login并传入响应对象。after_login此时才是真正爬虫起点，它应该处理两个问题：

1. Hold 住 Cookie 并将其加载到每次新发出到花瓣的请求中
2. 执行返回后的响应对象正文中的 Javascript。

Cookie

前文我们通过 Chrome 浏览器的开发人员工具得知当成功登录后会得到两个 Cookie，一存放的是uid另一个存放的是sid虽然没有看懂这个两个 Cookie 内的值有什么具体含义，但只有它们不过期，仍然可以存在客户端则就会被花瓣网认为是已登录用户。

那么在代码之中应该如何去获取 Cookie 呢？它在response对象的哪里？

Scrapy 搭载了一个CookiesMiddleware的中间件，这个中间件提供了自动管理Cookie的功能。如果在after_login中加入一个断点观察返回的response变量在headers中有一个Set-Cookie的字段，它保存了以下的一段内容：

```
[
 b'uid=27460062; domain=huaban.com; path=/',
 b'sid=NtgJGFSG0LNxsOxfgpm1rudu9vt.d6mhETDGJbFVUToaLor0%2BV%2FSF0sZDr6eRjJH9r0hcdc; domain=huaban.com; path=/; expires=Tue, 23 Jul 2019 11:33:44 GMT; httpOnly']
```

这个字节串列表就是在成功登录花瓣网之后由服务器颁发的，只要每次发出请求时都将两个uid和sid Cookie一并放回到请求的headers中，花瓣网就会认为我们是已经通过身份验证了。CookieMiddleware所谓的自动管理Cookie功能就是如此了，一定要先从服务器端获取一次Cookie然后将其放到CookieMiddleware内部的一个CookieJar对象中（你可以理解为一个专门存放Cookie的容器）当发送一下个请求时CookieMiddleware会自动地从CookieJar中重新拿出Cookie写入到的请求头中。

此时就可以在`after_login`方法中生成通过身份验证后的起始网络请求。至于怎么能让我们的Scrapy在获取网页内容后执行上面的javascript我会在下一节中进行详细的讲解。

Request 与 Response 中的 meta

在本节结束之前还有一个预备知识，那就是如何在Scrapy的`Request`与`Response`之间传输变量。

有些时候我们需要在发出请求的时候产生一些数据在得到该请求的响应之后重新从响应对象中取出来使用，这时就需要使用到`meta`这个变量了，假如在上述的例子中，如果我需要将用户名与密码在得到服务器响应后重新如出就可以像以下代码这样做：

```
def start_requests(self):
 req = FormRequest(
        url='https://huaban.com/auth/',
        method='POST',
        formdata={
            'email': '这里输入你注册的手机号',
            'password': '这里输入你注册的密码',
            '_ref': 'frame'},
        callback=self.after_login,
        meta = {
         'email': '这里输入你注册的手机号',
          'password': '这里输入你注册的密码'
        }
    )
 yield req

def after_login(self, response)
 print response.meta['email']
 print response.meta['password']
```

这个小技巧非常的实用，而且这个`meta`属性也是非常有用，里面还有不少保留关键字，在此卖个关子，有兴趣的可以深入了解一下`meta`。

小结

至此，我们已经敲开了花瓣网的大门，只有成功走出这一步才可能更进一步地做到拟人式的爬网。

花瓣网爬虫的分析和设计

用Splash服务处理花瓣网JS网页——高速方案