用Chrome无头浏览器处理JS网页——简单方案

更新时间：2019-07-03 15:11:55

> 人的差异在于业余时间。
>
> ——爱因斯坦

如何使用 chrome 更好地处理重深度集成 javascript 框架的网站。

可能不少读者会觉得采用Splash的爬取方式会比较难学，而且原理上也比较绕但毕竟那是在GitHub上获得超过2K star的一个出名的库，这个数字也证明了从众之多必然有其优势。

为了可以更加让更多读者可以更简单地在scrapy正确地渲染重度应用javascript的网站，本节将会介绍另一种更简单也更传统的做法——Chrome无头浏览器(Chrome Headless Browser)

Seleiumn 与过去的 PhantomJS

Selenium是一个基于Java开发的自动化浏览器处理器，从另一个角度来说,它更像是一个浏览器驱动的代理。它提供了一套访问浏览器内核的编程接口，程序可以调用Selenium的编程接口以控制浏览器的行为，甚至操控DOM。它自身是没有配备浏览器的,因此需要配合本机上安装的浏览器驱动一同使用，例如，Firefox、Chrome、Safari等。由于自动化测试与持续集成的大面积推广，使得Selenium被广泛地应用于各种语言平台。NodeJS、Python、Ruby、Java等主流Web开发语言工具链中必然会发现它的身影。

Selenium支持各种主流浏览器，更重要的是，它的接口是统一的，并不需要因为更换浏览器而改动任何的编码。因此我们也可以使用它来作为爬虫渲染动态网页的工具。

Selenium多用于界面测试的开发场景，由于Selenium能完全以编程方式模仿真人浏览网页的特性也非常符合我们用来开发各种高匿爬虫。

Selenium多年来更多是配合着PhantomJS这个无头浏览器一并使用，PhantomJS的内核是采用Google V8引擎进行开发的，它可以属于一个Chrome的第三方开发的无头浏览器版本。

新一代的 Chrome 无头浏览器

因为界面测试的普及，界业对无头浏览器的需求也在激增。所以在新一代的Chrome驱动中就附带了Chrome的无头浏览器版本。它与普通的Chrome不同的地方是它是不会显示界面的，所有的功能都需要通过编程实现，这样会比引导一个完全版的Chrome来访问网页会省下更多的内存消耗，继而会得到更高的运行效率。

安装Chrome无头浏览器

Ubuntu

第一步：更新必要的安装包

```
$ sudo apt-get update
$ sudo apt-get install -y unzip xvfb libxi6 libgconf-2-4
```

如果你的机器上没有安装 JDK还需要执行以下的步骤：

```
$ sudo apt-get install default-jdk
```

第二步：安装Chrome浏览器

```
$ sudo curl -sS -o - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add
$ sudo echo "deb [arch=amd64]  http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sources.list.d/google-chrome.list
$ sudo apt-get -y update
$ sudo apt-get -y install google-chrome-stable
```

第三步：安装ChromeDriver

```
$ wget https://chromedriver.storage.googleapis.com/2.41/chromedriver_linux64.zip
$ unzip chromedriver_linux64.zip
$ sudo mv chromedriver /usr/bin/chromedriver
$ sudo chown root:root /usr/bin/chromedriver
$ sudo chmod +x /usr/bin/chromedriver
```

macOS

对于macOS而言这个过程就变得相当简单了，用Homebrew安装只需要执行以下几步

```
$ brew tap homebrew/cask
$ brew update
$ brew cask install chromedriver
```

Windows

windows的安装请移步到ChromeDriver - WebDriver for Chrome上下载吧。

> **建议**：我强烈建议不要在windows环境下进行python开发，windows只适用于办公，用于做一些高级的服务端开发只会产生各种各样的坑，毕竟windows的编程生态只适合于 .net 生存，其它的语言还是回归到 linux的怀抱会更省心省力，生命如此短暂可不是用来为windows填坑的。

Chrome无头浏览器的实例化

在使用selenium + Chrome无头浏览器来改写我们的花瓣蜘蛛之前，我们可以先来写个测试对Chrome无头浏览器的使用有一个简单的认识。

接下来要在scrapy的虚拟环境中安装selenium：

```
$ pip install selenium
```

建立一个`test_chromeheadless.py`的测试文件，然后在这个测试文件中先将在上一章我们在lua代码中进行模拟登录的代码用python来进行复现。

首先导入必要的库

```
# -*- coding: utf-8 -*-
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.keys import Keys
```

然后构造一个浏览器实例:

```
def test_should_login_huaban(self):
 # 以无头方式使用 Chrome 而无需再采用PhantomJS的方式
 chrome_options = Options()
 chrome_options.add_argument("--headless") # 指定采用无头方式

 browser = webdriver.Chrome(executable_path="/usr/local/Caskroom/chromedriver/75.0.3770.90/chromedriver",
 options=chrome_options)
```

上述的代码中`add_argument("--headless)"`Chrome新增的无头浏览器功能，在过去的版本中如果没有加上这个参数的话selenium会直接起动一个Chrome的浏览器并在窗口中打开，只要你去观察内存的消耗就会发现一个这样的实例一起动就得消耗掉200M以上的内存，打开一个网页这个消耗量还会不停地增加，随之而来的就是速度的下降。

对于测试程序和爬虫项目而言这个我们只需要Chrome的功能而不需要它的界面，无头浏览器功能的加入就完美地解决了这一问题。

隐式等待与显式等待

下一步就如同在Splash中一样直接打开一个网页：

```
browser.get("https://huaban.com")
browser.implicitly_wait(1)
```

与Splash中一样的是这里也需要调用一个等待方法`implicitly_wait`先阻塞住线程，等待浏览器将网页加载完了再进入下步。

两者相比的话Chrome会更好用！在Chrome的中`implicitly_wait`被称为**隐式等待**，也就是这种等待方式并不无明确地知道页面是否真的加载完成，只是给一个大约数字而已（包括Splash也是采用这种模式）。

Chrome更一步的是可以进行**显式等待**，如果我们希望花瓣网的页面加载完最后的页脚能完成加载那代码就可以这样写：

```
try:
    element = WebDriverWait(driver,10).until(
    EC.presence_of_element_located((By.ID,"#index_footer"))
    )
finally:
    # 加载完成进行下一步的操作
```

这段代码会等待10秒，如果10秒内找到元素则立即返回，否则抛出 TimeoutException 异常。 EC 是 ExpectedCondition 的缩写，意思就是期待条件/等待条件。 WebDriverWait默认每500毫秒调用一次 ExpectedCondition，直到它返回成功为止。 ExpectedCondition 的类型是布尔型的，成功的返回值就是 true，其他类型的 ExpectedCondition 成功的返回值就是 not null 。

selenium提供了一系列的这种等待条件的判断，以下是我整理的一个表格，便于你以后使用的时候进行查阅：

条件	说明
title_is	判断当前页面的title是否精确等于预期
title_contains	判断当前页面的title是否包含预期字符串
presence_of_element_located	判断某个元素是否被加入DOM树中，并不代表该元素一定可见
visibility_of_element_located	判断某个元素是否可见。可见代表元素非隐藏，并且元素的宽和高都不等于0
visibility_of	跟上面的方法做一样的事情，只是上面的方法要传入locator，这个方法直接传定位到element就好了
presence_of_all_elements_located	判断是否至少有1个元素存在于DOM树中。举个例子，如果页面上有n个元素的class都是’column-md-3’，那么只要有1个元素存在，这个方法就返回True
text_to_be_present_in_element	判断某个元素中的text是否包含了预期的字符串
text_to_be_present_in_element_value	判断某个元素中的 value 属性是否包含了预期的字符串
frame_to_be_available_and_switch_to_it	判断该frame是否可以“switch”进去，如果可以，则返回True并且“switch”进去，否则返回False
invisibility_of_element_located	判断某个元素中是否不存在于DOM树或不可见
element_to_be_clickable	判断某个元素中是否可见并且是enable的，这样才叫clickable
staleness_of	等某个元素从dom树中移除，注意，这个方法也返回True或False
element_to_be_selected	判断某个元素是否被选中了，一般用在下拉列表中
element_located_to_be_selected	跟上面的方法作用一样，只是上面的方法传入定位到的element，而这个方法传入locator
element_selection_state_to_be	判断某个元素的选中状态是否符合预期
element_located_selection_state_to_be	跟上面的方法作用一样，只是上面的方法传入定位到的element，而这个方法传入locator
alert_is_present	判断页面上是否存在alert

以下是隐式等待与显式等待的一些区别：

- 显示等待: 明确的行为表现，在本地的selenium中运行（选择的编程语言）。可以在任何能想到的条件下工作，返回成功或者超时；可以定义元素的缺失为条件；可以定制重试间隔；可以忽略某些异常。
- 隐式等待: 不明确的行为表现，同一个问题在不同的操作系统、不同的浏览器、不同的selenium版本下会有各种不同的表现。在远程selenium上运行（控制浏览器的那部分），只能在寻找元素的函数上工作，返回找到元素或者（在超时以后）没有找到 。如果检查元素缺失，那么总是会等待到超时，除了时间什么都不能指定。

模拟登录

接下来就是模拟人工操作：点击网页上的"登录"按钮，弹出登录框后填写用户名与密码进行登录：

```
login_button = browser.find_element_by_css_selector('.login.btn')
login_button.click()
form = browser.find_element_by_css_selector('form.mail-login')
email_input = form.find_element_by_name('email')
password_input = form.find_element_by_name('password')
email_input.send_keys('<你的用户名>')
password_input.send_keys('<你的密码>')
form.submit()
```

selenium定位元素的方式很丰富，都是以`find_element_by_xxxx`提供，更多的方法可以参考selenium文档的locating elements的具体说明。

这里要注意的是填充`input`元素时用的并不是复制语句，而是使用`send_keys(值)`。

完成登录之后就可以从`browser`中获取Cookie了：

```
cookies = browser.get_cookies()
```

编写 ChromeMiddleware 中间件

如果使用selenium + chrome方式来访问网页那么就不能将这个访问写到蜘蛛中了，因为这样做就会失去Scrapy的并发能力。

原因是`browser.get(url)`方法是由浏览器自己发出请求并获得响应结果，这样就会完全打断了scrapy的处理链条。蜘蛛将会一次性处理所有发出的请求与返回请求，如果用这样的思路来写编写的话可能不用scrapy比用scrapy还省事。因为只是写一个能作循环处理的简单脚本即可了。

所以，在使用这个技术之前我们得做一些设计，要基于scrapy的机制在其运行环节中增加一个中间件来截获由scrapy引擎传递的请求，在中间件中将请求的处理转发给chrome然后直接返回一个标准scrapy 响应对象。这样一来我们又可以像之前的方式来设计与编写蜘蛛了。

```
# -*- coding: utf-8 -*-
from selenium import webdriver
from selenium.common.exceptions import TimeoutException
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from scrapy.http import HtmlResponse
from logging import getLogger


class ChromeMiddleware():
    """
    Chrome 无头浏览器中间件。
    """

    @classmethod
    def from_crawler(cls, crawler):
        return cls(timeout=crawler.settings.get('SELENIUM_TIMEOUT'),
                   exec_path=crawler.settings.get('CHROMEDRIVER'))

    def __init__(self, timeout=None, exec_path=''):
        self.logger = getLogger(__name__) # 打开日志
        self.timeout = timeout
        options = webdriver.ChromeOptions()
        options.add_argument('headless') # 采用无头浏览器
        self.browser = webdriver.Chrome(executable_path=exec_path,
                options=options)

        self.browser.set_window_size(1400, 700) # 设置浏览窗口
        self.browser.set_page_load_timeout(self.timeout) # 设置浏览器加载网页的超时时间

    def __del__(self):
        self.browser.close() # 释构时关闭浏览器实例


    def login(self):
        login_button = self.browser.find_element_by_css_selector('.login.btn')
        login_button.click()
        form = self.browser.find_element_by_css_selector('form.mail-login')
        email_input = form.find_element_by_name('email')
        password_input = form.find_element_by_name('password')
        email_input.send_keys('13725260426')
```

```
        password_input.send_keys('Lruikun')
        form.submit()
        self.browser.implicitly_wait(3)

    def process_request(self, request, spider):
        """
        用Chrome抓取页面
        :param request: Request对象
        :param spider: Spider对象
        :return: HtmlResponse
        """
        self.logger.debug(u'启动Chrome...')

        try:
            self.browser.get(request.url)
            # 等待页脚被渲染完成
            self.browser.implicitly_wait(3)

            # 检查浏览器中是否已经存有登录的Cookie
            cookies = self.browser.get_cookies()
            is_login = False
            for cookie in cookies:
                if cookie['name'] == 'sid':
                    is_login = True
                    break

            if not is_login:
                self.login()
                self.browser.get(request.url)
                self.browser.implicitly_wait(3)

            return HtmlResponse(url=request.url,
                                body=self.browser.page_source,
                                request=request,
                                encoding='utf-8',
                                status=200)

        except TimeoutException:
            # 超时抛出异常
            return HtmlResponse(url=request.url, status=500, request=request)
```

蜘蛛

当我们将Chrome集成到中间件时蜘蛛的逻辑又将回归到原有的路上，与本专栏开始介绍的蜘蛛的编写方式完全一致：

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

from scrapy.spiders import Spider
from huaban.items import HuabanItem


def _gen_start_urls():
    for i in range(1, 200):
        yield 'https://huaban.com/search/?q=app&type=pins&page=%s&per_page=20' % i

class Bee(Spider):
    name = 'bee'
    start_urls = _gen_start_urls()

    def parse(self, response):
        item = HuabanItem()
        pinEles = response.css("#waterfall div.pin")
        for pinEle in pinEles:
            item['img_url'] = 'https:%s' % pinEle.css('a.img>img::attr(src)').get()
            item['link'] = pinEle.css('a::attr(href)').get()
            item['desc'] = pinEle.css('p.description::text').get()
            yield item
```

配置

以下是整个爬虫项目的配置：

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

BOT_NAME = 'bee'

SPIDER_MODULES = ['huaban.spiders']
NEWSPIDER_MODULE = 'huaban.spiders'
ROBOTSTXT_OBEY = False

SELENIUM_TIMEOUT =10
CHROMEDRIVER = "/usr/local/Caskroom/chromedriver/75.0.3770.90/chromedriver"

# 加入Chrome中间件
DOWNLOADER_MIDDLEWARES = {
  'huaban.middlewares.ChromeMiddleware': 1,
}

CONCURRENT_REQUESTS = 2
COOKIES_ENABLED = False
TELNETCONSOLE_ENABLED = False
AUTOTHROTTLE_ENABLED = True
AUTOTHROTTLE_START_DELAY = 5
AUTOTHROTTLE_MAX_DELAY = 60
AUTOTHROTTLE_TARGET_CONCURRENCY = 1.0

HTTPCACHE_ENABLED = True
```

这里要注意的地方，首先是要将Chrome的中间件加入到`DOWNLOADER_MIDDLEWARES`配置中，并且将优级调节到最高，其次就是要打开自动降速配置`AUTOTHROTTLE_ENABLED = True`，如果不打开自动降速配置的话蜘蛛就会以一种**”非人“**的速度来爬网了，这样就对方一定会发现这是爬虫行为，而失去了隐秘性了。

小结

至此我们已经能够成功地爬取花瓣网这种重度使用Javascript的网站内容了，这种蜘蛛的特性就是“慢”与“高匿”，它的速度只是比我们人工访问网站的速度快上一些，但决不能与之前所介绍的所有爬虫相比，这一点在本章第一节中已经很详细地提及。

本章书的爬网方法不单单能应用于爬虫，还可以应用在软件的界面测试中。更确切地说这种渲染Javascript网页的爬虫方式就是借用python界面测试的方法而来的。

用Splash服务处理花瓣网JS网页—高速方案

将采集到的图片存储于阿里云oss