在前面我们学习了 Selenium 的基本用法，它功能的确非常强大，但很多时候我们会发现 Selenium 有一些不太方便的地方，比如环境的配置，得安装好相关浏览器，比如 Chrome、Firefox 等等，然后还要到官方网站去下载对应的驱动，最重要的还需要安装对应的 Python Selenium 库，而且版本也得好好看看是否对应，确实不是很方便，另外如果要做大规模部署的话，环境配置的一些问题也是个头疼的事情。

那么本课时我们就介绍另一个类似的替代品，叫作 Pyppeteer。注意，是叫作 Pyppeteer，而不是 Puppeteer。

Pyppeteer 介绍

Puppeteer 是 Google 基于 Node.js 开发的一个工具，有了它我们可以通过 JavaScript 来控制 Chrome 浏览器的一些操作，当然也可以用作网络爬虫上，其 API 极其完善，功能非常强大，Selenium 当然同样可以做到。

而 Pyppeteer 又是什么呢？它实际上是 Puppeteer 的 Python 版本的实现，但它不是 Google 开发的，是一位来自于日本的工程师依据 Puppeteer 的一些功能开发出来的非官方版本。

在 Pyppetter 中，实际上它背后也是有一个类似 Chrome 浏览器的 Chromium 浏览器在执行一些动作进行网页渲染，首先说下 Chrome 浏览器和 Chromium 浏览器的渊源。

> Chromium 是谷歌为了研发 Chrome 而启动的项目，是完全开源的。二者基于相同的源代码构建，Chrome 所有的新功能都会先在 Chromium 上实现，待验证稳定后才会移植，因此 Chromium 的版本更新频率更高，也会包含很多新的功能，但作为一款独立的浏览器，Chromium 的用户群体要小众得多。两款浏览器“同根同源”，它们有着同样的 Logo，但配色不同，Chrome 由蓝红绿黄四种颜色组成，而 Chromium 由不同深度的蓝色构成。

总的来说，两款浏览器的内核是一样的，实现方式也是一样的，可以认为是开发版和正式版的区别，功能上基本是没有太大区别的。

Pyppeteer 就是依赖于 Chromium 这个浏览器来运行的。那么有了 Pyppeteer 之后，我们就可以免去那些烦琐的环境配置等问题。如果第一次运行的时候，Chromium 浏览器没有安装，那么程序会帮我们自动安装和配置，就免去了烦琐的环境配置等工作。另外 Pyppeteer 是基于 Python 的新特性 async 实现的，所以它的一些执行也支持异步操作，效率相对于 Selenium 来说也提高了。

那么下面就让我们来一起了解下 Pyppeteer 的相关用法吧。

安装

首先就是安装问题了，由于 Pyppeteer 采用了 Python 的 async 机制，所以其运行要求的 Python 版本为 3.5 及以上。

安装方式非常简单：

```
pip3 install pyppeteer
```

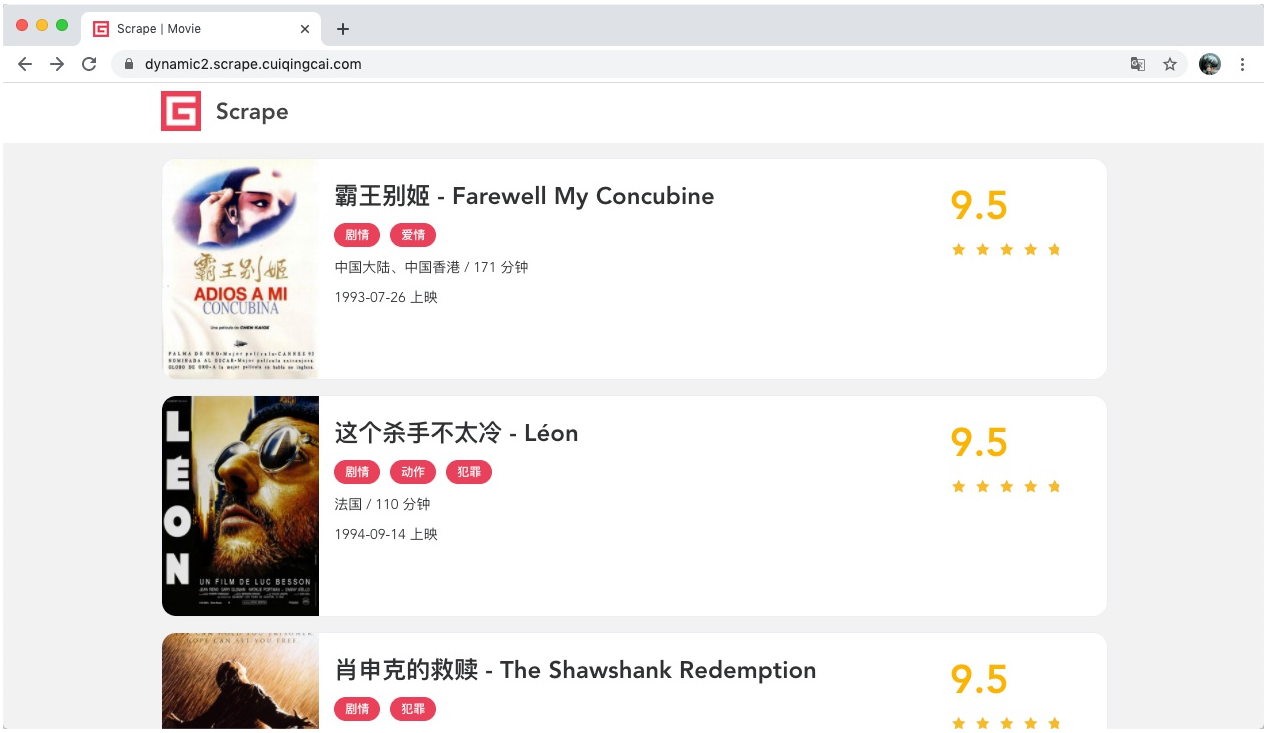
好了，安装完成之后我们在命令行下测试：

```
>>> import pyppeteer
```

如果没有报错，那么就证明安装成功了。

快速上手

接下来我们测试基本的页面渲染操作，这里我们选用的网址为：https://dynamic2.scrape.cuiqingcai.com/，如图所示。



这个网站我们在之前的 Selenium 爬取实战课时中已经分析过了，整个页面是用 JavaScript 渲染出来的，同时一些 Ajax 接口还带有加密参数，所以这个网站的页面我们无法直接使用 requests 来抓取看到的数据，同时我们也不太好直接模拟 Ajax 来获取数据。

所以前面一课时我们介绍了使用 Selenium 爬取的方式，其原理就是模拟浏览器的操作，直接用浏览器把页面渲染出来，然后再直接获取渲染后的结果。同样的原理，用 Pyppeteer 也可以做到。

下面我们用 Pyppeteer 来试试，代码就可以写为如下形式：

```
import asyncio
from pyppeteer import launch
```

```
from pyquery import PyQuery as pq
async def main():
    browser = await launch()
    page = await browser.newPage()
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await page.waitForSelector('.item .name')
    doc = pq(await page.content())
    names = [item.text() for item in doc('.item .name').items()]
    print('Names:', names)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

运行结果：

```
Names: ['霸王别姬 - Farewell My Concubine', '这个杀手不太冷 - Léon', '肖申克的救赎 - The Shawshank Redemption', '泰坦尼克号 - Titanic', '罗马假日 - Roman Holiday', '唐伯虎点秋香 - Flirting Scholar
```

先初步看下代码，大体意思是访问了这个网站，然后等待 .item .name 的节点加载出来，随后通过 pyquery 从网页源码中提取了电影的名称并输出，最后关闭 Pyppeteer。

看运行结果，和之前的 Selenium 一样，我们成功模拟加载出来了页面，然后提取到了首页所有电影的名称。

那么这里面的具体过程发生了什么？我们来逐行看下。

- launch 方法会新建一个 Browser 对象，其执行后最终会得到一个 Browser 对象，然后赋值给 browser。这一步就相当于启动了浏览器。
- 然后 browser 调用 newPage 方法相当于浏览器中新建了一个选项卡，同时新建了一个 Page 对象，这时候新启动了一个选项卡，但是还未访问任何页面，浏览器依然是空白。
- 随后 Page 对象调用了 goto 方法就相当于在浏览器中输入了这个 URL，浏览器跳转到了对应的页面进行加载。
- Page 对象调用 waitForSelector 方法，传入选择器，那么页面就会等待选择器所对应的节点信息加载出来，如果加载出来了，立即返回，否则会持续等待直到超时。此时如果顺利的话，页面会成功加载出来。
- 页面加载完成之后再调用 content 方法，可以获得当前浏览器页面的源代码，这就是 JavaScript 渲染后的结果。
- 然后进一步的，我们用 pyquery 进行解析并提取页面的电影名称，就得到最终结果了。

另外其他的一些方法如调用 asyncio 的 get_event_loop 等方法的相关操作则属于 Python 异步 async 相关的内容了，你如果不熟悉可以了解下前面所讲的异步相关知识。

好，通过上面的代码，我们同样也可以完成 JavaScript 渲染页面的爬取了。怎么样？代码相比 Selenium 是不是更简洁易读，而且环境配置更加方便。在这个过程中，我们没有配置 Chrome 浏览器，也没有配置浏览器驱动，免去了一些烦琐的步骤，同样达到了 Selenium 的效果，还实现了异步抓取。

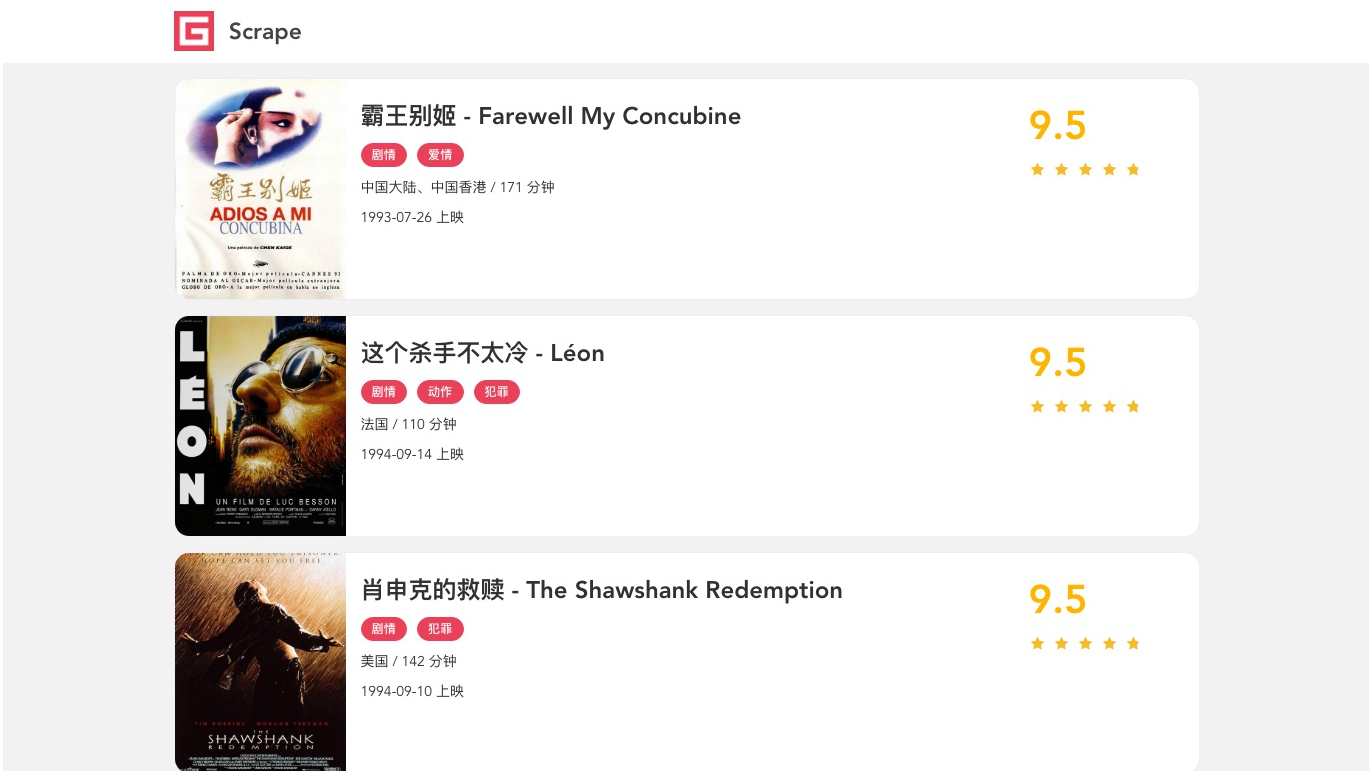
接下来我们再看看另外一个例子，这个例子设定了浏览器窗口大小，然后模拟了网页截图，另外还可以执行自定义的 JavaScript 获得特定的内容，代码如下：

```
import asyncio
from pyppeteer import launch
width, height = 1366, 768
async def main():
    browser = await launch()
    page = await browser.newPage()
    await page.setViewport({'width': width, 'height': height})
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await page.waitForSelector('.item .name')
    await asyncio.sleep(2)
    await page.screenshot(path='example.png')
    dimensions = await page.evaluate('''() => {
        return {
            width: document.documentElement.clientWidth,
            height: document.documentElement.clientHeight,
            deviceScaleFactor: window.devicePixelRatio,
        }
    }''')

    print(dimensions)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

这里我们又用到了几个新的 API，完成了页面窗口大小设置、网页截图保存、执行 JavaScript 并返回对应数据。

首先 screenshot 方法可以传入保存的图片路径，另外还可以指定保存格式 type、清晰度 quality、是否全屏 fullPage、裁切 clip 等各个参数实现截图。

截图的样例如下：



可以看到它返回的就是 JavaScript 渲染后的页面，和我们在浏览器中看到的结果是一模一样的。

最后我们又调用了 evaluate 方法执行了一些 JavaScript，JavaScript 传入的是一个函数，使用 return 方法返回了网页的宽高、像素大小比率三个值，最后得到的是一个 JSON 格式的对象，内容如下：

```
{'width': 1366, 'height': 768, 'deviceScaleFactor': 1}
```

OK，实例就先感受到这里，还有太多太多的功能还没提及。

总之利用 Pyppeteer 我们可以控制浏览器执行几乎所有动作，想要的操作和功能基本都可以实现，用它来自由地控制爬虫当然就不在话下了。

详细用法

了解了基本的实例之后，我们再来梳理一下 Pyppeteer 的一些基本和常用操作。Pyppeteer 的几乎所有功能都能在其官方文档的 API Reference 里面找到，链接为：https://miyakogi.github.io/pyppeteer/reference.html，用到哪个方法就来这里查询就好了，参数不必死记硬背，即用即查就好。

launch

使用 Pyppeteer 的第一步便是启动浏览器，首先我们看下怎样启动一个浏览器，其实就相当于我们点击桌面上的浏览器图标一样，把它运行起来。用 Pyppeteer 完成同样的操作，只需要调用 launch 方法即可。

我们先看下 launch 方法的 API，链接为：https://miyakogi.github.io/pyppeteer/reference.html#pyppeteer.launcher.launch，其方法定义如下：

```
pyppeteer.launcher.launch(options: dict = None, **kwargs) → pyppeteer.browser.Browser
```

可以看到它处于 launcher 模块中，参数没有在声明中特别指定，返回类型是 browser 模块中的 Browser 对象，另外观察源码发现这是一个 async 修饰的方法，所以调用它的时候需要使用 await。

接下来看看它的参数：

- ignoreHTTPSErrors (bool)：是否要忽略 HTTPS 的错误，默认是 False。
- headless (bool)：是否启用 Headless 模式，即无界面模式，如果 devtools 这个参数是 True 的话，那么该参数就会被设置为 False，否则为 True，即默认是开启无界面模式的。
- executablePath (str)：可执行文件的路径，如果指定之后就不需要使用默认的 Chromium 了，可以指定为已有的 Chrome 或 Chromium。
- slowMo (int|float)：通过传入指定的时间，可以减缓 Pyppeteer 的一些模拟操作。
- args (List[str])：在执行过程中可以传入的额外参数。
- ignoreDefaultArgs (bool)：不使用 Pyppeteer 的默认参数，如果使用了这个参数，那么最好通过 args 参数来设定一些参数，否则可能会出现一些意想不到的问题。这个参数相对比较危险，慎用。
- handleSIGINT (bool)：是否响应 SIGINT 信号，也就是可以使用 Ctrl＋C 来终止浏览器程序，默认是 True。
- handleSIGTERM (bool)：是否响应 SIGTERM 信号，一般是 kill 命令，默认是 True。
- handleSIGHUP (bool)：是否响应 SIGHUP 信号，即挂起信号，比如终端退出操作，默认是 True。
- dumpio (bool)：是否将 Pyppeteer 的输出内容传给 process.stdout 和 process.stderr 对象，默认是 False。
- userDataDir (str)：即用户数据文件夹，即可以保留一些个性化配置和操作记录。
- env (dict)：环境变量，可以通过字典形式传入。
- devtools (bool)：是否为每一个页面自动开启调试工具，默认是 False。如果这个参数设置为 True，那么 headless 参数就会无效，会被强制设置为 False。
- logLevel (int|str)：日志级别，默认和 root logger 对象的级别相同。
- autoClose (bool)：当一些命令执行完之后，是否自动关闭浏览器，默认是 True。
- loop (asyncio.AbstractEventLoop)：事件循环对象。

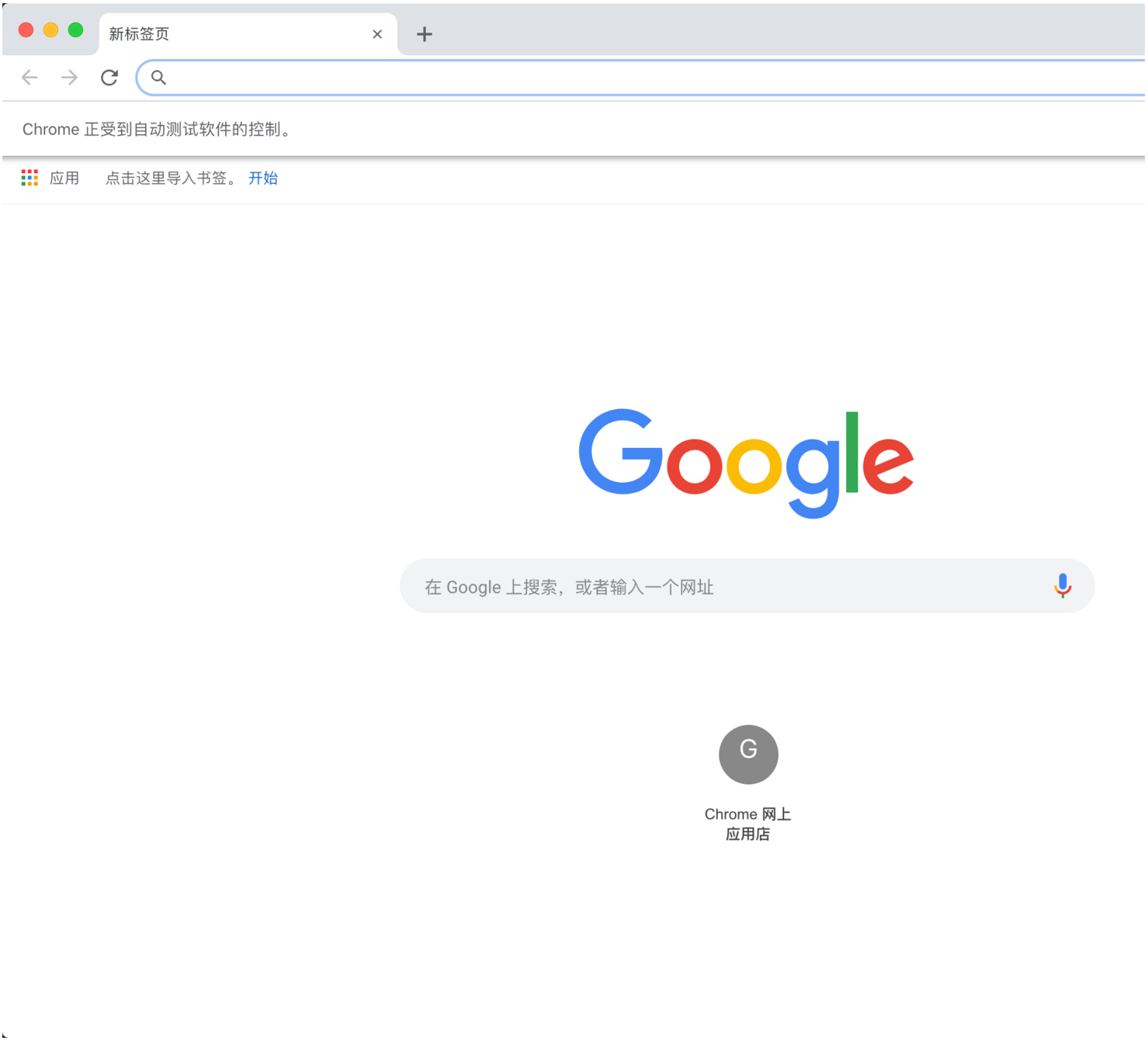
好了，知道这些参数之后，我们可以先试试看。

- 无头模式

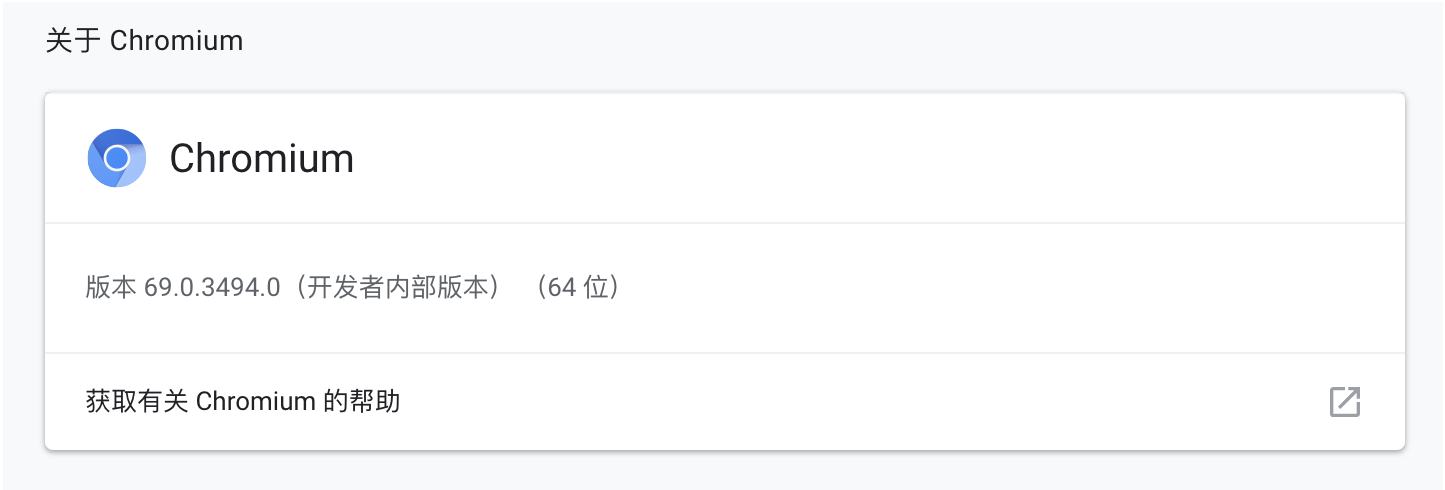
首先可以试用下最常用的参数 headless，如果我们将它设置为 True 或者默认不设置它，在启动的时候我们是看不到任何界面的，如果把它设置为 False，那么在启动的时候就可以看到界面了，一般我们在调试的时候会把它设置为 False，在生产环境上就可以设置为 True，我们先尝试一下关闭 headless 模式：

```
import asyncio
from pyppeteer import launch
async def main():
    await launch(headless=False)
    await asyncio.sleep(100)
asyncio.get_event_loop().run_until_complete(main())
```

运行之后看不到任何控制台输出，但是这时候就会出现一个空白的 Chromium 界面了：



但是可以看到这就是一个光秃秃的浏览器而已，看一下相关信息：



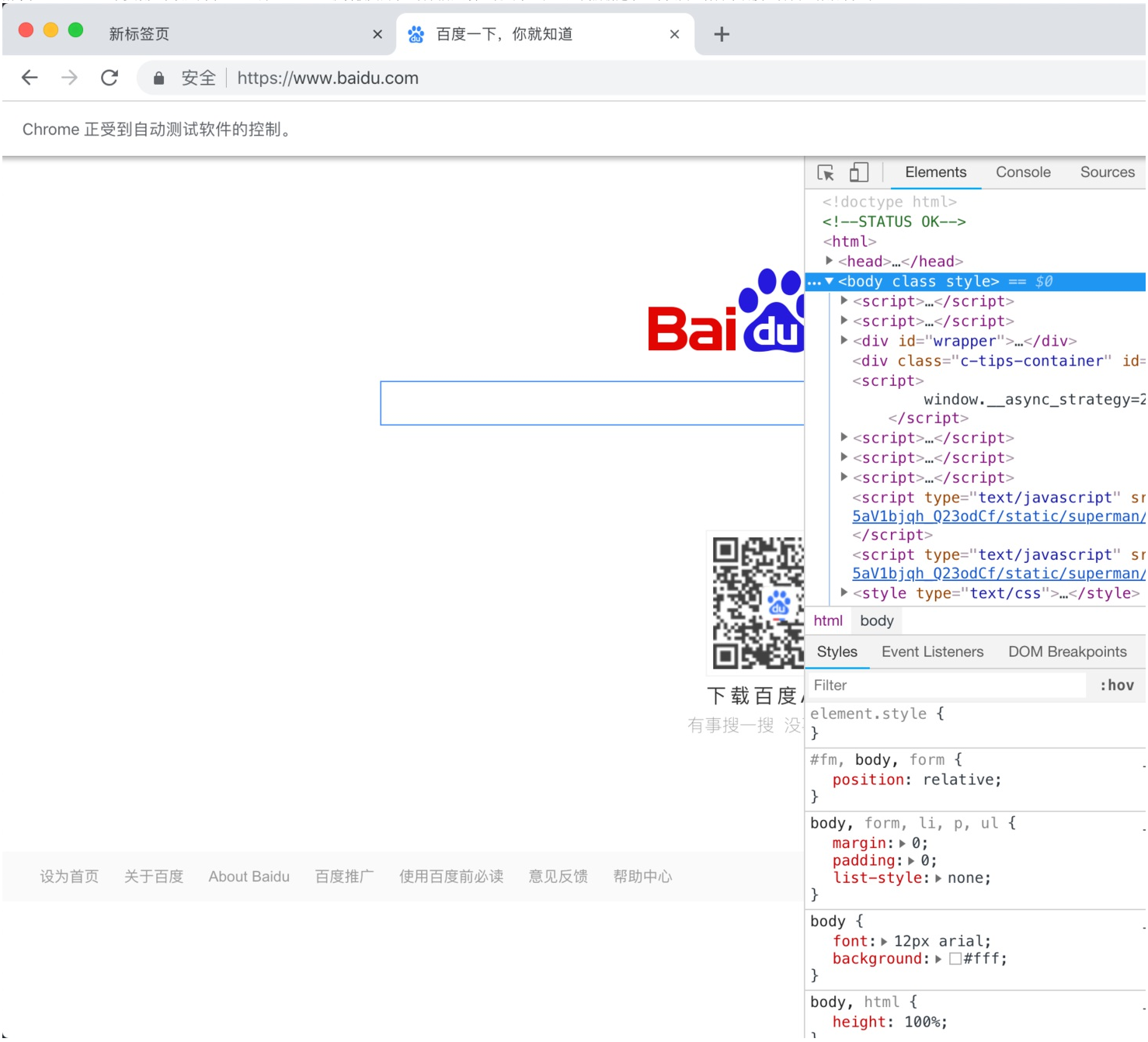
看到了，这就是 Chromium，上面还写了开发者内部版本，你可以认为是开发版的 Chrome 浏览器就好。

- 调试模式

另外我们还可以开启调试模式，比如在写爬虫的时候会经常需要分析网页结构还有网络请求，所以开启调试工具还是很有必要的，我们可以将 devtools 参数设置为 True，这样每开启一个界面就会弹出一个调试窗口，非常方便，示例如下：

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch(devtools=True)
    page = await browser.newPage()
    await page.goto('https://www.baidu.com')
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

刚才说过 devtools 这个参数如果设置为了 True，那么 headless 就会被关闭了，界面始终会显现出来。在这里我们新建了一个页面，打开了百度，界面运行效果如下：



- 禁用提示条

这时候我们可以看到上面的一条提示："Chrome 正受到自动测试软件的控制"，这个提示条有点烦，那该怎样关闭呢？这时候就需要用到 args 参数了，禁用操作如下：

```
browser = await launch(headless=False, args=['--disable-infobars'])
```

这里就不再写完整代码了，就是在 launch 方法中，args 参数通过 list 形式传入即可，这里使用的是 --disable-infobars 的参数。

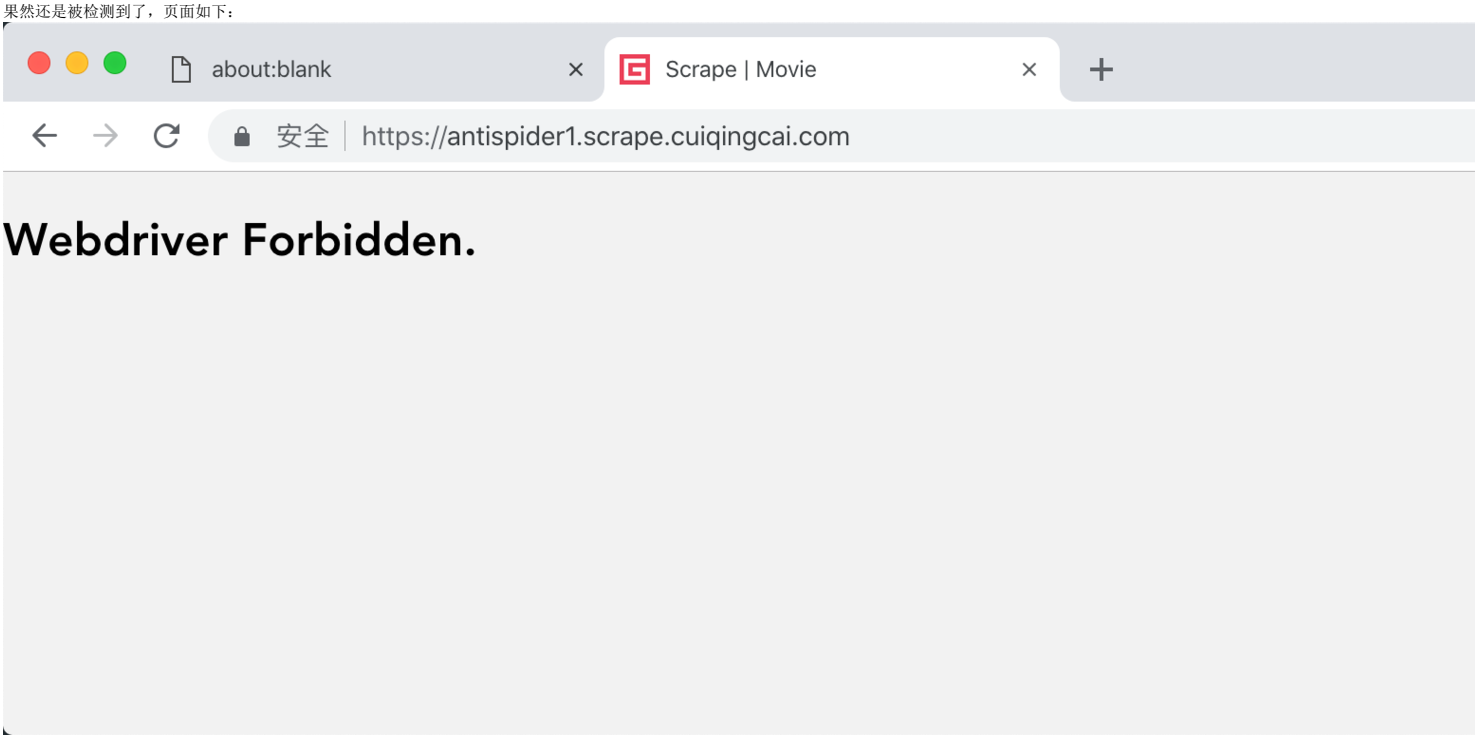
- 防止检测

你可能会说，如果你只是把提示关闭了，有些网站还是会检测到是 WebDriver 吧，比如拿之前的检测 WebDriver 的案例 https://antispider1.scrape.cuiqingcai.com/ 来验证下，我们可以试试：

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch(headless=False, args=['--disable-infobars'])
    page = await browser.newPage()
    await page.goto('https://antispider1.scrape.cuiqingcai.com/')
    await asyncio.sleep(100)
```

```
asyncio.get_event_loop().run_until_complete(main())
```

果然还是被检测到了，页面如下：

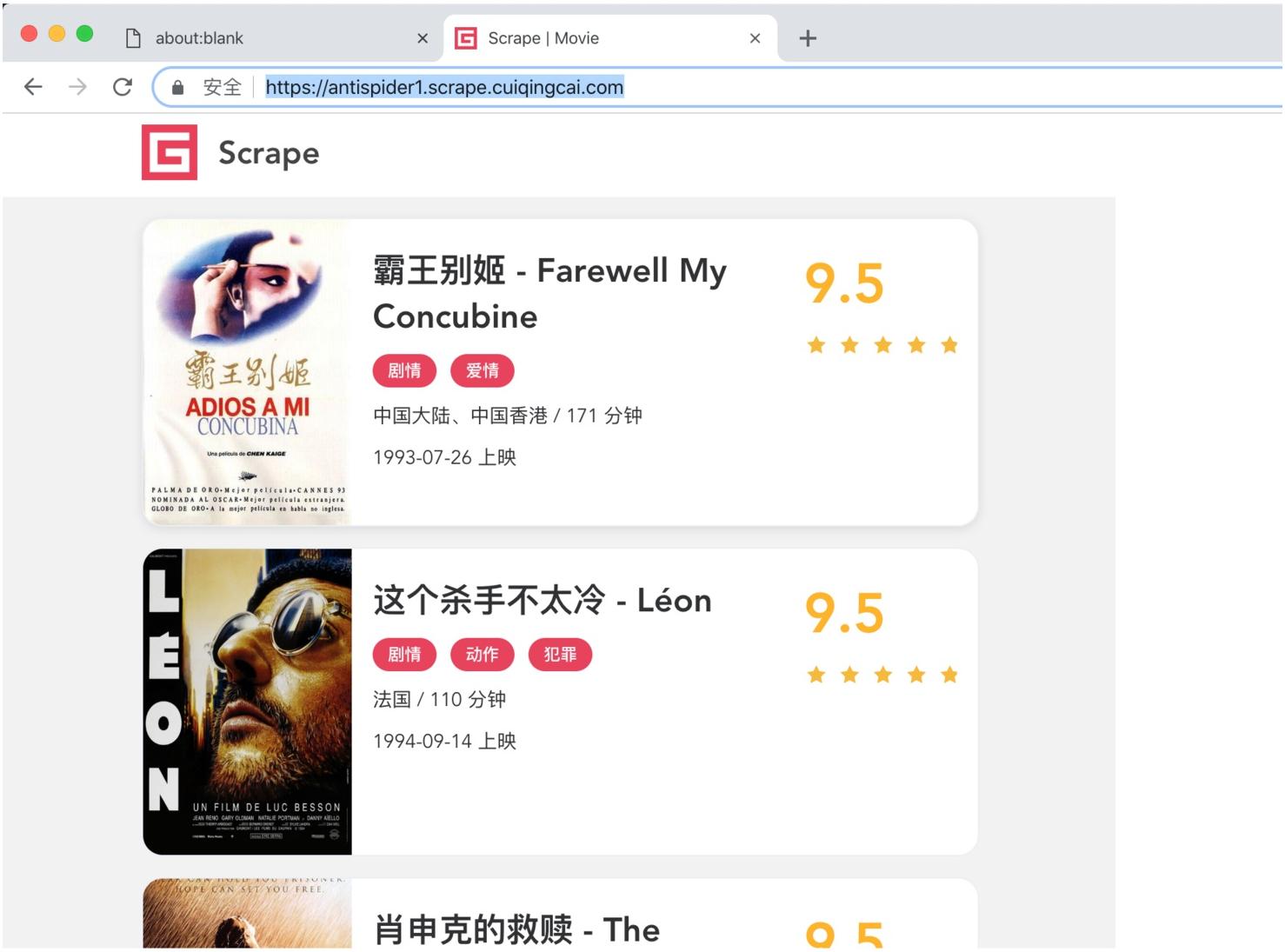


这说明 Pyppeteer 开启 Chromium 照样还是能被检测到 WebDriver 的存在。

那么此时如何规避呢？Pyppeteer 的 Page 对象有一个方法叫作 evaluateOnNewDocument，意思就是在每次加载网页的时候执行某个语句，所以这里我们可以执行一下将 WebDriver 隐藏的命令，改写如下：

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch(headless=False, args=['--disable-infobars'])
    page = await browser.newPage()
    await page.evaluateOnNewDocument('Object.defineProperty(navigator, "webdriver", {get: () => undefined})')
    await page.goto('https://antispider1.scrape.cuiqingcai.com/')
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

这里我们可以看到整个页面就可以成功加载出来了，如图所示。



我们发现页面就成功加载出来了，绕过了 WebDriver 的检测。

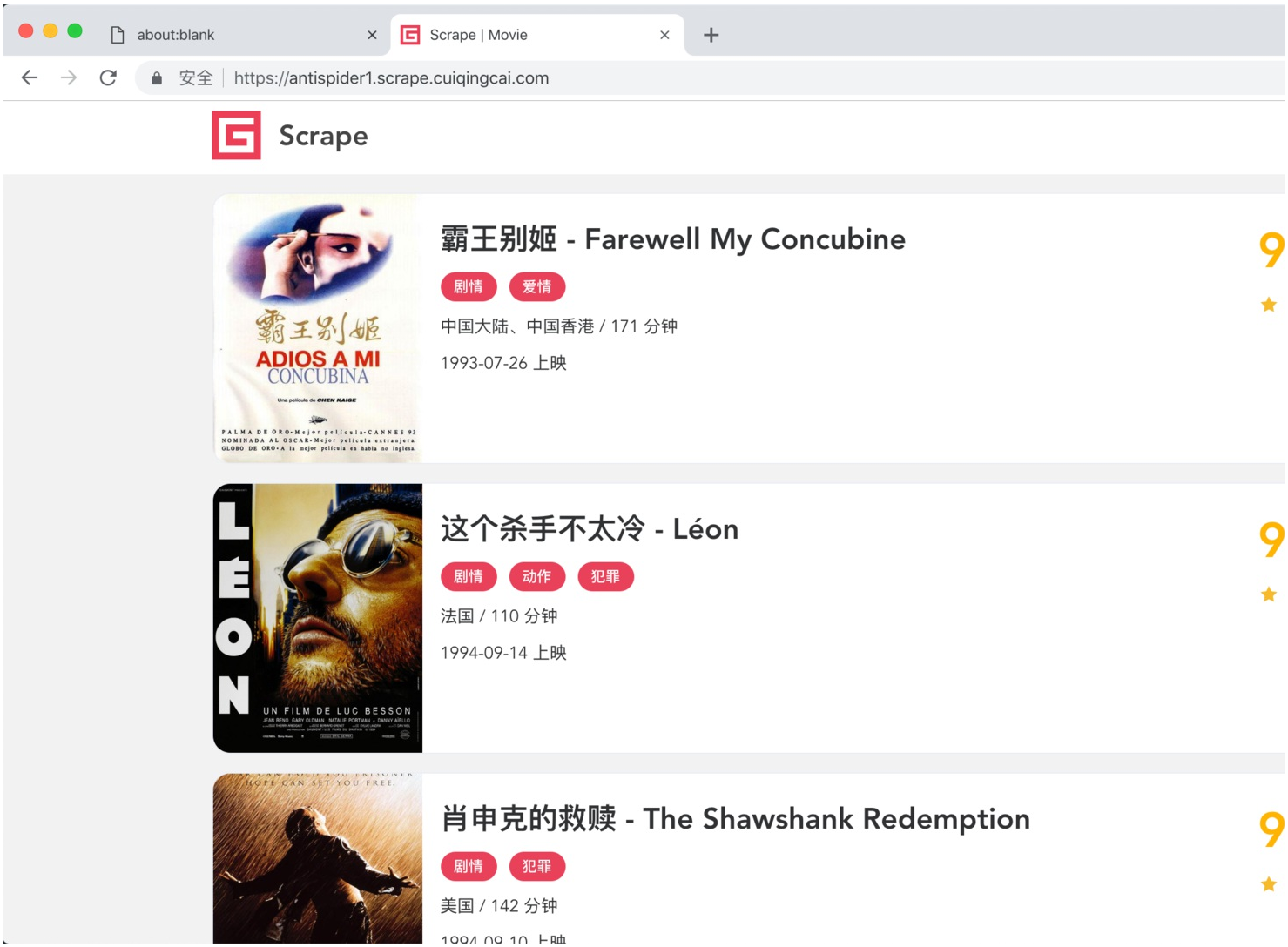
- 页面大小设置

在上面的例子中，我们还发现了页面的显示 bug，整个浏览器窗口比显示的内容窗口要大，这个是某些页面会出现的情况。

对于这种情况，我们通过设置窗口大小就可以解决，可以通过 Page 的 setViewport 方法设置，代码如下：

```
import asyncio
from pyppeteer import launch

width, height = 1366, 768

async def main():
    browser = await launch(headless=False, args=['--disable-infobars', f'--window-size={width},{height}'])
    page = await browser.newPage()
    await page.setViewport({'width': width, 'height': height})
    await page.evaluateOnNewDocument('Object.defineProperty(navigator, "webdriver", {get: () => undefined})')
    await page.goto('https://antispider1.scrape.cuiqingcai.com/')
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

这里我们同时设置了浏览器窗口的宽高以及显示区域的宽高，使得二者一致，最后发现显示就正常了，如图所示。



- 用户数据持久化

刚才我们可以看到，每次我们打开 Pyppeteer 的时候都是一个新的空白的浏览器。而且如果遇到了需要登录的网页之后，如果我们这次登录上了，下一次再启动又是空白了，又得登录一次，这的确是一个问题。

比如以淘宝举例，平时我们逛淘宝的时候，在很多情况下关闭了浏览器再打开，淘宝依然还是登录状态。这是因为淘宝的一些关键 Cookies 已经保存到本地了，下次登录的时候可以直接读取并保持登录状态。

那么这些信息保存在哪里了呢？其实就是保存在用户目录下了，里面不仅包含了浏览器的基本配置信息，还有一些 Cache、Cookies 等各种信息都在里面，如果我们能在浏览器启动的时候读取这些信息，那么启动的时候就可以恢复一些历史记录甚至一些登录状态信息了。

这也就解决了一个问题：很多时候你在每次启动 Selenium 或 Pyppeteer 的时候总是一个全新的浏览器，那这究其原因就是没有设置用户目录，如果设置了它，每次打开就不再是一个全新的浏览器了，它可以恢复之前的历史记录，也可以恢复很多网站的登录信息。

那么这个怎么来做呢？很简单，在启动的时候设置 userDataDir 就好了，示例如下：

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch(headless=False, userDataDir='./userdata', args=['--disable-infobars'])
    page = await browser.newPage()
    await page.goto('https://www.taobao.com')
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

好，这里就是加了一个 userDataDir 的属性，值为 userdata，即当前目录的 userdata 文件夹。我们可以首先运行一下，然后登录一次淘宝，这时候我们同时可以观察到在当前运行目录下又多了一个 userdata 的文件夹，里面的结构是这样子的：

具体的介绍可以看官方的一些说明，如： https://chromium.googlesource.com/chromium/src/+/master/docs/user_data_dir.md，这里面介绍了 userdatadir 的相关内容。

再次运行上面的代码，这时候可以发现现在就已经是登录状态了，不需要再次登录了，这样就成功跳过了登录的流程。当然可能时间太久了，Cookies 都过期了，那还是需要登录的。

以上便是 launch 方法及其对应的参数的配置。

Browser

上面我们了解了 launch 方法，其返回的就是一个 Browser 对象，即浏览器对象，我们会通常将其赋值给 browser 变量，其实它就是 Browser 类的一个实例。

下面我们来看看 Browser 类的定义：

```
class pyppeteer.browser.Browser(connection: pyppeteer.connection.Connection, contextIds: List[str], ignoreHTTPSErrors: bool, setDefaultViewport: bool, process: Optional[subprocess.Pope
```

这里我们可以看到其构造方法有很多参数，但其实多数情况下我们直接使用 launch 方法或 connect 方法创建即可。

browser 作为一个对象，其自然有很多用于操作浏览器本身的方法，下面我们来选取一些比较有用的介绍下。

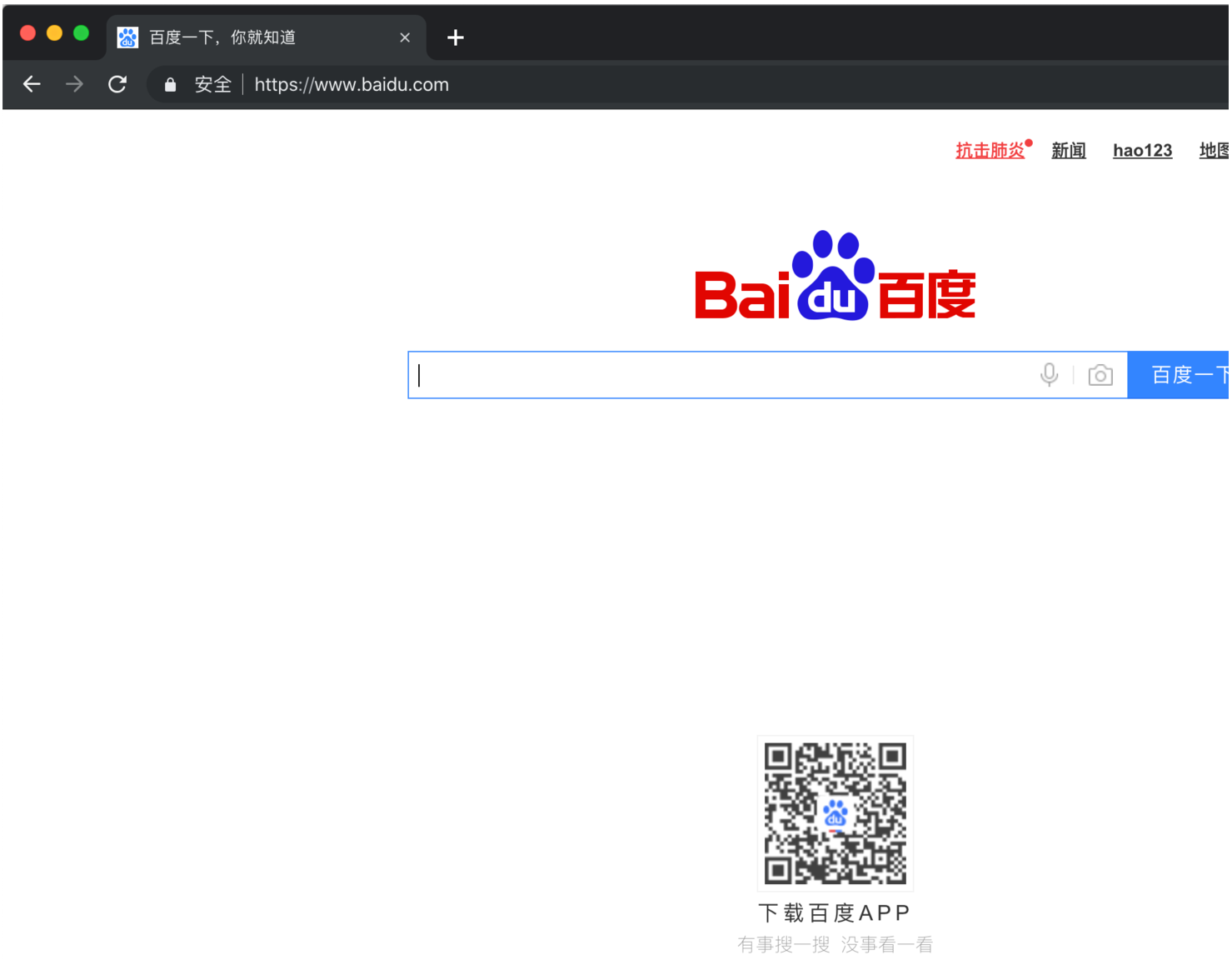
- 开启无痕模式

我们知道 Chrome 浏览器是有一个无痕模式的，它的好处就是环境比较干净，不与其他的浏览器示例共享 Cache、Cookies 等内容，其开启方式可以通过 createIncognitoBrowserContext 方法，示例如下：

```
import asyncio
from pyppeteer import launch

width, height = 1200, 768

async def main():
    browser = await launch(headless=False,
                           args=['--disable-infobars', f'--window-size={width},{height}'])
    context = await browser.createIncognitoBrowserContext()
    page = await context.newPage()
    await page.setViewport({'width': width, 'height': height})
    await page.goto('https://www.baidu.com')
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

这里关键的调用就是 createIncognitoBrowserContext 方法，其返回一个 context 对象，然后利用 context 对象我们可以新建选项卡。

运行之后，我们发现浏览器就进入了无痕模式，界面如下：



- 关闭
  怎样关闭自不用多说了，就是 close 方法，但很多时候我们可能忘记了关闭而造成额外开销，所以要记得在使用完毕之后调用一下 close 方法，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch()
    page = await browser.newPage()
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

Page

Page 即页面，就对应一个网页，一个选项卡。在前面我们已经演示了几个 Page 方法的操作了，这里我们再详细看下它的一些常用用法。

- 选择器

Page 对象内置了一些用于选取节点的选择器方法，如 J 方法传入一个选择器 Selector，则能返回对应匹配的第一个节点，等价于 querySelector。如 JJ 方法则是返回符合 Selector 的列表，类似于 querySelectorAll。

下面我们来看下其用法和运行结果，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch()
    page = await browser.newPage()
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await page.waitForSelector('.item .name')
    j_result1 = await page.J('.item .name')
    j_result2 = await page.querySelector('.item .name')
    jj_result1 = await page.JJ('.item .name')
    jj_result2 = await page.querySelectorAll('.item .name')
    print('J Result1:', j_result1)
    print('J Result2:', j_result2)
    print('JJ Result1:', jj_result1)
    print('JJ Result2:', jj_result2)
```

```
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

在这里我们分别调用了 J、querySelector、JJ、querySelectorAll 四个方法，观察下其运行效果和返回结果的类型，运行结果：

```
J Result1: <pyppeteer.element_handle.ElementHandle object at 0x1166f7dd0>
J Result2: <pyppeteer.element_handle.ElementHandle object at 0x1166f07d0>
JJ Result1: [<pyppeteer.element_handle.ElementHandle object at 0x11677df50>, <pyppeteer.element_handle.ElementHandle object at 0x1167857d0>, <pyppeteer.element_handle.ElementHandle obj
...
<pyppeteer.element_handle.ElementHandle object at 0x11679db10>, <pyppeteer.element_handle.ElementHandle object at 0x11679dbd0>]
JJ Result2: [<pyppeteer.element_handle.ElementHandle object at 0x116794f10>, <pyppeteer.element_handle.ElementHandle object at 0x116794d10>, <pyppeteer.element_handle.ElementHandle obj
...
<pyppeteer.element_handle.ElementHandle object at 0x11679f690>, <pyppeteer.element_handle.ElementHandle object at 0x11679f750>]
```

在这里我们可以看到，J、querySelector 一样，返回了单个匹配到的节点，返回类型为 ElementHandle 对象。JJ、querySelectorAll 则返回了节点列表，是 ElementHandle 的列表。

- 选项卡操作

前面我们已经演示了多次新建选项卡的操作了，也就是 newPage 方法，那新建了之后怎样获取和切换呢，下面我们来看一个例子：

```
import asyncio
from pyppeteer import launch

async def main():
    browser = await launch(headless=False)
    page = await browser.newPage()
    await page.goto('https://www.baidu.com')
    page = await browser.newPage()
    await page.goto('https://www.bing.com')
    pages = await browser.pages()
    print('Pages:', pages)
    page1 = pages[1]
    await page1.bringToFront()
    await asyncio.sleep(100)

asyncio.get_event_loop().run_until_complete(main())
```

在这里我们启动了 Pyppeteer，然后调用了 newPage 方法新建了两个选项卡并访问了两个网站。那么如果我们要切换选项卡的话，只需要调用 pages 方法即可获取所有的页面，然后选一个页面调用其 bringToFront 方法即可切换到该页面对应的选项卡。

- 常见操作

作为一个页面，我们一定要有对应的方法来控制，如加载、前进、后退、关闭、保存等，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch(headless=False)
    page = await browser.newPage()
    await page.goto('https://dynamic1.scrape.cuiqingcai.com/')
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    # 后退
    await page.goBack()
    # 前进
    await page.goForward()
    # 刷新
    await page.reload()
    # 保存 PDF
    await page.pdf()
    # 截图
    await page.screenshot()
    # 设置页面 HTML
    await page.setContent('<h2>Hello World</h2>')
    # 设置 User-Agent
    await page.setUserAgent('Python')
    # 设置 Headers
    await page.setExtraHTTPHeaders(headers={})
    # 关闭
    await page.close()
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

这里我们介绍了一些常用方法，除了一些常用的操作，这里还介绍了设置 User-Agent、Headers 等功能。

- 点击

Pyppeteer 同样可以模拟点击，调用其 click 方法即可。比如我们这里以 https://dynamic2.scrape.cuiqingcai.com/ 为例，等待节点加载出来之后，模拟右键点击一下，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch(headless=False)
    page = await browser.newPage()
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await page.waitForSelector('.item .name')
    await page.click('.item .name', options={
        'button': 'right',
        'clickCount': 1,  # 1 or 2
        'delay': 3000,  # 毫秒
    })
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

这里 click 方法第一个参数就是选择器，即在哪里操作。第二个参数是几项配置：

- button：鼠标按钮，分为 left、middle、right。
- clickCount：点击次数，如双击、单击等。
- delay：延迟点击。
- 输入文本。

对于文本的输入，Pyppeteer 也不在话下，使用 type 方法即可，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch(headless=False)
```

```
    page = await browser.newPage()
    await page.goto('https://www.taobao.com')
    # 后退
    await page.type('#q', 'iPad')
    # 关闭
    await asyncio.sleep(10)
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

这里我们打开淘宝网，使用 type 方法第一个参数传入选择器，第二个参数传入输入的内容，Pyppeteer 便可以帮我们完成输入了。

- 获取信息

Page 获取源代码用 content 方法即可，Cookies 则可以用 cookies 方法获取，示例如下：

```
import asyncio
from pyppeteer import launch
from pyquery import PyQuery as pq

async def main():
    browser = await launch(headless=False)
    page = await browser.newPage()
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    print('HTML:', await page.content())
    print('Cookies:', await page.cookies())
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

- 执行

Pyppeteer 可以支持 JavaScript 执行，使用 evaluate 方法即可，看之前的例子：

```
import asyncio
from pyppeteer import launch

width, height = 1366, 768

async def main():
    browser = await launch()
    page = await browser.newPage()
    await page.setViewport({'width': width, 'height': height})
    await page.goto('https://dynamic2.scrape.cuiqingcai.com/')
    await page.waitForSelector('.item .name')
    await asyncio.sleep(2)
    await page.screenshot(path='example.png')
    dimensions = await page.evaluate('''() => {
        return {
            width: document.documentElement.clientWidth,
            height: document.documentElement.clientHeight,
            deviceScaleFactor: window.devicePixelRatio,
        }
    }''')

    print(dimensions)
    await browser.close()

asyncio.get_event_loop().run_until_complete(main())
```

这里我们通过 evaluate 方法执行了 JavaScript，并获取到了对应的结果。另外其还有 exposeFunction、evaluateOnNewDocument、evaluateHandle 方法可以做了解。

- 延时等待

在本课时最开头的地方我们演示了 waitForSelector 的用法，它可以让页面等待某些符合条件的节点加载出来再返回。

在这里 waitForSelector 就是传入一个 CSS 选择器，如果找到了，立马返回结果，否则等待直到超时。

除了 waitForSelector 方法，还有很多其他的等待方法，介绍如下。

- waitForFunction：等待某个 JavaScript 方法执行完毕或返回结果。
- waitForNavigation：等待页面跳转，如果没加载出来就会报错。
- waitForRequest：等待某个特定的请求被发出。
- waitForResponse：等待某个特定的请求收到了回应。
- waitFor：通用的等待方法。
- waitForSelector：等待符合选择器的节点加载出来。
- waitForXPath：等待符合 XPath 的节点加载出来。

通过等待条件，我们就可以控制页面加载的情况了。

更多

另外 Pyppeteer 还有很多功能，如键盘事件、鼠标事件、对话框事件等等，在这里就不再一一赘述了。更多的内容可以参考官方文档的案例说明：https://miyakogi.github.io/pyppeteer/reference.html。

以上，我们就通过一些小的案例介绍了 Pyppeteer 的基本用法，下一课时，我们来使用 Pyppeteer 完成一个实战案例爬取。

本节代码：https://github.com/Python3WebSpider/PyppeteerTest。