第3章

pyppeteer

pyppeteer是Node库puppeteer的Python版本，它提供了一个高级API，通过DevTools协议控制Chrome浏览器或Chromium浏览器。pyppeteer与浏览器有着更深的交互，这一点是selenium库无法企及的。pyppeteer的底层实现是通过封装puppeteer，对外提供了Python调用的异步接口。本章主要介绍pyppeteer的全系内容，从基础的环境部署，到常用的页面操作方法，再到一些具体场景中的运用，例如识别特征的处理、配置代理及代理的认证、拦截请求和响应等。

本章要点如下。

(1) pyppeteer的基本使用方法。

(2) pyppeteer的启动器和页面的常用操作方法。

(3) pyppeteer的Cookie操作方法和页面元素的选择方法。

(4) pyppeteer的鼠标键盘操作和页面内嵌框操作。

(5) pyppeteer的多种JavaScript的支持方法。

(6) pyppeteer的重要对象Request和Response。

(7) pyppeteer的常用启动参数和防识别处理。

(8) pyppeteer的IP代理设置及拦截器的使用方法。

3.1 pyppeteer基础

3.1.1 pyppeteer简介

如果说在Python中还有一款自动化工具能和selenium库媲美的，那么无疑是pyppeteer。puppeteer是Google开源的一个Node库，通过一系列高级接口和Chrome或Chromium浏览器在DevTools协议下交互。它具有以下特色功能。

(1) 生成页面的截图和PDF。

(2) 抓取SPA(Single Page web Application，单页应用程序)并渲染页面。

(3) 自动提交表单、UI测试、键盘输入等。

(4) 创建一个最新的自动化测试环境，使用最新的JavaScript和浏览器特性，在最新版本的Chrome浏览器中直接运行测试。

(5) 捕捉异常、跟踪堆栈来帮助诊断性能问题。

(6) 测试 Chrome 浏览器扩展程序。

(7) 其他高级功能,如 JavaScript 注入、模拟操作、异步执行、伪装等。

相比于 selenium 库,pyppeteer 具有异步加载、速度快、伪装性更强不易被识别,同时可以伪装手机平板等终端的优点,但它也有一些缺点,如接口不易理解、语义晦涩、bug 多等。

3.1.2 pyppeteer 环境安装

使用 pyppeteer 前,需要安装 pyppeteer 库和 Chromium 浏览器。如果本地未安装 Chromium 浏览器,那么在首次使用 pyppeteer 库时会自动下载 Chromium 浏览器或者在 Windows 或 Liunx 的命令界面中运行 pyppeteer-install,手动安装 Chromium 浏览器。

安装 pyppeteer 库,命令如下。

```
pip install pyppeteer
```

安装 pyppeteer 后,在 CMD 命令行中安装 Chromium,命令如下。

```
C:\Users\inlike > pyppeteer - install
[W:pyppeteer.command] chromium is already installed.
```

输入下列源码,测试是否能正常运行。

```
import asyncio
from pyppeteer import launch


async def main():
    browser = await launch({"headless": False})
    page = await browser.newPage()
    await page.goto('http://likeinlove.com')
    await page.screenshot({'path': 'example.png'})
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

上面的代码是在有界面模式下打开 Chromium 浏览器,然后在标签页中访问 http://likeinlove.com 页面,并截图保存为 example.png 的文件,最后关闭浏览器。其中 async 关键字用于声明一个异步操作,await 关键字用于声明一个耗时操作,asyncio.get_event_loop().run_until_complete(main())的作用是创建异步池,将 main 事件加入并运行。相关源码解释如下。

```
browser = await launch ({"headless": False})
# 创建一个启动器,并配置启动参数 headless,为 False 则代表需要打开浏览器界面
page = await browser.newPage()
# 新建一个页面对象,页面操作在页面对象上
await page.goto('http://example.com')
# 执行跳转功能类似于 selenium 的 driver.get()
await page.screenshot({'path': 'example.png'})
# 页面截图
await browser.close()
# 关闭浏览器对象
```

3.2 pyppeteer 的常用内部方法

3.2.1 浏览器启动器

浏览器的启动是通过初始化启动器 Launcher 来完成的，启动器在初始化时可以配置启动相关的参数，启动器初始化完成后即返回浏览器对象 Browser。

1. 启动器 Launcher

启动器可以从默认位置启动一个 Chromium 浏览器，或者连接一个已经打开的 Chromium 浏览器。启动器使用前，先从 pyppeteer 模块中导入函数 launch()，launch()用于创建一个启动器对象。

启动器在默认情况下，会使用本地的 Chromium 浏览器，但是可以配置启动项中的 executablePath 字段来指定浏览器的启动程序路径，如下面的代码是在启动 Windows 10 下默认安装的 Chrome 浏览器。

```
import asyncio
from pyppeteer import launch


async def main():
    browser = await launch({'headless': False, 'executablePath': 'C:\Program Files (x86)\
Google\Chrome\Application\chrome.exe'})
    page = await browser.newPage()
    await page.goto('http://example.com')
    await page.screenshot({'path': 'example.png'})
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

除此之外，pyppeteer 启动器还支持连接已经打开的 Chromium 浏览器，这是启动器的另一种启动方式 connect。connect 用于连接一个已经打开的浏览器，可以在程序崩溃后重连。connect()方法需要一个必选参数 browserWSEndpoint，该参数是 Browser 对象的 wsEndpoint 属性，是 pyppeteer 与 Chromium 浏览器通信的地址，它是以 ws 开头的 WebSocket 地址，形如 ws://127.0.0.1:3533/devtools/browser/6687308b-2c43-4ccb-9464-1d2c1fec7eb5 的字符串。下面的代码是先用一段代码启动一个 Chromium 浏览器，然后启动另一个代码来控制这个已启动的浏览器。

启动一个 Chromium 浏览器，并打印 wsEndpoint 属性。

```
import asyncio
import time
from pyppeteer import launch

async def main():
    browser = await launch({'headless': False})
    print(browser.wsEndpoint)
    browser.disconnect()     #断开与浏览器的连接，不是浏览器

asyncio.get_event_loop().run_until_complete(main())
# 输出
ws://127.0.0.1:56308/devtools/browser/4600b780-885c-461d-a2e8-83d4e0d0587d
```

然后使用另一段源码接管该浏览器。

```
import asyncio
import pyppeteer

async def main():
    browser = await pyppeteer.connect(
        browserWSEndpoint = "ws://127.0.0.1:56308/devtools/browser/4600b780 - 885c - 461d -
a2e8 - 83d4e0d0587d")
    page = await browser.newPage()
    await page.goto('http://example.com')
    await page.screenshot({'path': 'example.png'})
    await browser.close()

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

运行第二段代码后，Chromium 浏览器打开了指定的页面并生成了本地截图。connect() 方法通过 browserWSEndpoint 参数传入 wsEndpoint 值，这是必选参数。除此之外，还有几个可选参数。这些参数名及其作用如下。

(1) ignoreHTTPSErrors：是否忽略 HTTPS 错误，默认值为 False。

(2) slowMo：将 pyppeteer 的速度减慢指定的毫秒数。

(3) logLevel：设置日志级别，以打印日志。默认与根记录器相同，传入 int 或 str 代表的日志级别。

(4) loop：事件循环，是一个实验性参数，默认即可。

2. 浏览器对象 Browser

在启动浏览器后即返回一个 pyppeteer.browser.Browser 实例。该实例提供了与浏览器进程的交互、多个页面对象的上文管理、模拟浏览器的基础设置、创建隐身浏览器等功能。

Browser 具有下列浏览器控制的相关属性和方法，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1) browserContexts

返回所有打开的浏览器上下文的列表，包括从 Browser 创建的隐身浏览器。

2) coroutine createIncognitoBrowserContext()

创建一个新的隐身浏览器上下文对象 BrowserContext，将打开一个新浏览器操作窗口，不会与其他浏览器上下文共享 Cookie 信息和缓存。例如：

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch({"headless": False})
    content = await browser.createIncognitoBrowserContext()
    await browser.close()


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

3) coroutine disconnect()

断开浏览器，断开不等于关闭，断开后还可以通过 connect 连接。

4) coroutine newPage()

在此浏览器上创建新页面，并返回其对象。

5）coroutine pages()

获取此浏览器的所有页面。此处不会列出不可见的页面，例如 background_page，可以通过 pyppeteer. target. Target. page()查看。

6）process

返回此浏览器的进程，如果创建浏览器的是实例 pyppeteer. launcher. connect()，则返回 None。例如：

```
brwoser.process
<subprocess.Popen object at 0x02E03190>
```

7）targets()

获取浏览器中所有活动的页面列表。在多个浏览器上下文的情况下，该方法将返回包含所有浏览器上下文中的所有目标的列表。

8）coroutine userAgent()

返回浏览器的原始用户代理，页面类 page 可以通过 coroutine setUserAgent()设置代理。

9）coroutine version()

获取浏览器的版本。

10）wsEndpoint

返回 WebSocket 端点的 URL 地址。例如：

```
brwoser.wsEndpoint
'ws://127.0.0.1:4636/devtools/browser/ccb4bd48-4572-468d-8549-1f4f27da8737'
```

3. 浏览器上下文对象 BrowserContext

BrowserContext 用于创建多个独立的浏览器会话，启动浏览器时，它默认使用一个 BrowserContext。browser. newPage()将在默认浏览器上下文中创建页面，如果页面打开另一个页面，例如通过 window. open 调用，则弹出窗口也属于初始化创建的浏览器上下文。

通过 browser. createIncognitoBrowserContext()创建一个隐身浏览器进程，隐身浏览器上下文不会将任何数据写入磁盘。

BrowserContext 具有下列浏览器控制的相关属性和方法，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1）coroutine close()

关闭浏览器上下文，将关闭属于浏览器上下文的所有页面。

2）isIncognito()

返回 BrowserContext 是否隐身。

3）coroutine newPage()

在浏览器上下文中创建新页面。

4）targets()

返回浏览器上下文中所有活动目标的列表。

BrowserContext 和 Browser 都是用于管理浏览器的会话，前者是创建一个隐身浏览器会话，在创建浏览器对象之后进而创建 Page 对象，页面的主要操作都在 Page 对象上进行。

3.2.2 页面常用操作

Page 类是 pyppeteer 的核心，其价值犹如 selenium 库的 driver，具体的页面操作都在 Page 实例化对象上进行。Page 与 driver 比较具有优势的是和 JavaScript 的交互，可以修改本地 JavaScript、CSS，也可以给页面添加 JavaScript 函数，甚至添加自定义函数到浏览器的 windows 属性中，也有 JavaScript 拦截相关的设置，更有终端模拟设置，这些功能是比 driver 更为强大的功能，但是也有一些劣势，如页面超时处理方面比 driver 弱，选择器不简洁等问题。

在初始化 Browser 后，通过 Browser.newPage()方法即可创建 Page 实例化对象。Page 对象有下列常用的方法和属性，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1. coroutine addScriptTag(options, ** kwargs)

向当前页面添加 script 标签，其中需要一个必选参数 url、path 或 content，代表 script 的内容。可选参数 type 用于说明添加脚本的类型。addScriptTag 执行成功后，最后返回一个 ElementHandle 对象。如下面的源码所示，向页面中增加 JavaScript 代码。

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch({'headless': False})
    page = await browser.newPage()
    await page.goto('https://www.likeinlove.com')
    await page.addScriptTag(content = """(function () {alert("hello word")});""")
    await page.addScriptTag(path = 'test.js')
    await page.addScriptTag(url = 'https://cdn.bootcss.com/vue/2.5.16/vue.min.js ')
asyncio.get_event_loop().run_until_complete(main())
```

上面分别通过 url、path、content 向页面中添加了 script 标签，看页面中的效果如图 3-1 所示，将 script 标签及标签内容添加到页面的 head 标签中。

```
    <![endif]-->
  <script type="text/javascript">(function () {alert("hello word")});</script>
  <script type="text/javascript">(function () {
      alert("hello word")
  });//# sourceURL=test.js</script>
  <script src="https://cdn.bootcss.com/vue/2.5.16/vue.min.js "></script>
 </head>
▼<body>
  <!--top begin-->
```

图 3-1 addScriptTag 向页面的 head 标签中添加 script 标签及标签内容

2. coroutine addStyleTag(options, ** kwargs)

通过 coroutine addStyleTag()可以向当前页面添加 style 标签，其中需要一个必选参数 url、path 或 content，表示 style 的内容。如果没有 addScriptTag 的 type 参数，则返回一个 ElementHandle 对象。

3. coroutine authenticate(credentials)

通过 coroutine authenticate(credentials)可以提供 HTTP 身份验证的凭据，参数是含有 username 和 password 字段的字典，或用 username、password 作关键字的参数，该方法可以用于代理的 HTTP Bastic 认证。

4. coroutine bringToFront()

coroutine bringToFront()方法能将页面置于前面，激活当前选项卡。

5. browser

Page 对象的 browser 属性，用于获取该页面所属的浏览器对象。

6. coroutine click(selector, options, ** kwargs)

coroutine click(selector, options, ** kwargs)能实现单击匹配 selector 的元素。如果获取的 selector 元素在视图底部，将滚动到视图中，然后使用 mouse 事件单击元素的中心。如果没有匹配 selector，则该方法会引发 PageError 错误。

coroutine click 函数需要一个必选参数 selector，代表要单击的元素，还需要三个可选参数如下。

(1) button：指定单击位置，可选 left、right 或 middle，默认为 left。

(2) clickCount：单击次数，默认为 1。

(3) delay：等待时间，表示鼠标按下到弹起的时间默认为 0，以毫秒为单位。

如下源码实现的是打开百度在搜索栏中输入 python，然后单击“搜索”按钮。

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.goto("http://www.baidu.com/")
    await page.type('#kw', 'python')
    await page.click('#su', delay = 10)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

如果此方法要触发的元素，存在其他的单击事件，则容易产生混乱的判断，推荐的单击并等待的写法如下。

```
await asyncio.gather(
    page.waitForNavigation(waitOptions),
    page.click(selector, clickOptions),
)
```

7. coroutine close()

通过 coroutine close()可以关闭当前页面。

8. coroutine content()

coroutine content()是一种常用方法，返回当前页面的完整 HTML 源码。

9. coroutine emulate(options, ** kwargs)

通过 coroutine emulate(options, ** kwargs)可以模拟给定的设备信息和用户代理，此方法是调用以下两个方法的快捷方式：setUserAgent()、setViewport()。该方法需要两个关键字参数，分别是 viewport 和 userAgent。

其中 viewport 是字典值，含有模拟设备的相关属性。

(1) width：页面宽度以像素为单位。

(2) height：页面高度以像素为单位。

(3) deviceScaleFactor：指定设备比例，默认为 1。

(4) isMobile：是否使用“meta viewportt”的标记，默认为 False。

(5) hasTouch：指定 viewport 是否支持触摸事件，默认为 False。

(6) isLandscape：指定视口是否处于横向模式，默认为 False。

另一个参数 userAgent，参数值是字符串，表示用户代理信息。

使用下面的模拟参数，打开一个模拟手机的浏览器，然后在 Page 上打开百度页面。可见百度页面自适应手机端布局，相应的 userAgent 参数也变成了设置的字符串。

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    ua = 'Mozilla/5.0 (iPhone; CPU iPhone OS 10_3_1 like Mac OS X) AppleWebKit/603.1.30
(KHTML, like Gecko) Version/10.0 Mobile/14E304 Safari/602.1'
    await page.emulate(viewport = {"width": 720, "height": 1280}, userAgent = ua)
    await page.goto("http://www.baidu.com/")
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

10. coroutine focus(selector)

coroutine focus(selector)能实现聚焦匹配的元素，如果没有元素匹配，则抛出 PageError 错误，有一个必选参数 selector 代表元素选择器路径。

11. frames

frames 能完成获取此页面的所有 frame。

12. coroutine goBack(options, ** kwargs)

通过 coroutine goBack(options, ** kwargs)可以导航到历史记录中的上一页，如果不能导航到上一页，则返回 None。

13. coroutine goForward(options, ** kwargs)

coroutine goForward(options, ** kwargs)能实现导航到历史记录中的下一页，可用选项与 goto()方法相同，如果不能导航到下一页，则返回 None。

14. coroutine goto(url, options, ** kwargs)

coroutine goto(url, options, ** kwargs)是一种常用函数，可以打开指定 URL 地址，无头模式不支持导航到 PDF 文档。该方法需要一个必选参数 url，其他可选参数如下。

(1) timeout：最大导航时间以毫秒为单位，默认为 30000 毫秒；为 0 时表示禁用超时；可以使用 setDefaultNavigationTimeout()方法更改默认值。

(2) waitUntil：导航成功的标记事件，当下列可选事件触发时认为导航完成，默认为 load，即整个页面及所有依赖资源(如样式表和图片)都已完成加载时触发 load 事件。

(3) load：当 load 事件被触发时。

(4) domcontentloaded：当 DOMContentLoaded 事件被触发时完成导航。初始的 HTML 文档被完全加载并解析完成之后，DOMContentLoaded 事件被触发，无须等待样式表、图像和子框架的完成加载。

(5) networkidle0：当网络连接数不超过 0 时，至少需要 500 毫秒。

(6) networkidle2：当网络连接数不超过 2 个时，至少需要 500 毫秒。

15. coroutine hover(selector)

使用 coroutine hover(selector)时，当鼠标指针悬停在匹配元素上，如果没有元素，则抛

出 PageError。该函数的必选参数 selector，代表元素的选择器路径。

16. isClosed()

判断页面是否关闭。

17. keyboard

使用 keyboard 能获取 Keyboard 对象，提供用于管理虚拟键盘的 api。

18. mouse

通过 mouse 可以获取 Mouse 对象。

19. mainFrame

通过 mainFrame 可以获取此页面对应的 Frame 对象。

20. coroutine metrics()

使用 coroutine metrics()能获取页面属性，返回包含键值对的字典，例如 Documents（页面文档数）、JSEventListeners（页面中的事件数）、JSHeapUsedSize（使用的 JavaScript 堆大小）等信息。

21. coroutine pdf(options, ** kwargs)

coroutine pdf(options, ** kwargs)能实现生成页面的 PDF 文件，目前仅支持在无头模式下生成 PDF 文件，可选参数如下。

（1）path：生成 PDF 文件的保存路径。

（2）scale：网页渲染的比例，默认为 1。

（3）displayHeaderFooter：是否显示页眉和页脚，默认为 False。

（4）footerTemplate：页脚的 HTML 格式模板，可选标签类有 date（格式化的日期）、title（文件名）、url（文件位置）、pageNumber（当前页码）、totalPages（文档中的总页数）。需要注意的是，显示打印页脚需要设置附加选项，例如 displayHeaderFooter 显示页眉页脚，完整示例如下。

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch()          # headless = False
    page = await browser.newPage()
    await page.goto("http://www.baidu.com/")
await page.pdf(displayHeaderFooter = True, footerTemplate = '< div style = "font - size:10px;
width: 100 % ;"><span class = "date"></span> - Hello Word - <span class = "pageNumber"></span>
</div>', format = "A4",
margin = {"bottom": 70, "left": 25, "right": 35, "top": 30, },
              printBackground = True, path = '1.pdf')
await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

打印出来的页脚设置效果如图 3-2 所示，其原理是通过将模板 HTML 源码添加到页面最底部渲染，然后显示出来。

（5）headerTemplate：打印页眉的 HTML 模板，同 footerTemplate()使用方法。

（6）printBackground：是否打印背景图形，默认为 False。

图 3-2 页脚设置效果

(7) landscape：纸张方向，默认为 False。

(8) pageRanges：要打印的纸张范围，如“1-5,8,11-13”。默认为空字符串，表示所有页面。

(9) format：纸张格式，如果设置则优先于 width 或 height。默认为 Letter(8.5 英寸×11 英寸)，常用的还有 A4(8.27 英寸×11.7 英寸)。

(10) width：纸张宽度，接收标有单位的值。

(11) height：纸张高度，接收标有单位的值。

(12) margin：纸张边距，值为字典类型，默认值 None，可选参数有 top(上边距)、right(右边距)、bottom(底部边距)、left(左边距)，可用单位有 px(像素)、in(英寸)、cm(厘米)、mm(毫米)。

22. coroutine reload()

coroutine reload()方法能刷新当前页。

23. coroutine screenshot(options, ** kwargs)

通过 coroutine screenshot(options, ** kwargs)方法能实现屏幕截屏，可以使用的参数有：

(1) path：保存图像的文件路径，屏幕截图类型将从文件扩展名中推断出来。

(2) type：指定屏幕截图类型，可以是 jpeg 或 png，默认为 png。

(3) quality：图像的质量，值为 0～100，不适用于 png 图像。

(4) fullPage：是否截取完整的可滚动页面，默认为 False。

(5) clip：值为字典，截取指定区域。可选参数为 x、y、width、height，分别从 x、y 坐标处截取指定高宽的图片。

(6) omitBackground：布尔值，作用隐藏默认的白色背景，并捕获具有透明度的屏幕截图。

(7) encoding：设置图像的编码为 base64 或 binary，默认为 binary。

24. coroutine select(selector, * values)

coroutine select(selector, * values)可以选择选项并返回所选值，如果没有元素匹配 selector，则抛出 ElementHandleError 错误。

25. coroutine setBypassCSP(enabled)

coroutine setBypassCSP(enabled)能判断是否绕过页面的 Content-Security-Policy(内容安全策略)，通常在导航到指定页面之前调用它，该方法需要一个布尔值参数。Content-Security-Policy 允许站点管理者控制用户代理，能够为指定的页面加载哪些资源。

26. coroutine setCacheEnabled(enabled)

coroutine setCacheEnabled(enabled)可以实现启用/禁用缓存，默认情况为 True 启用缓存。

27. coroutine setContent(html)

通过 coroutine setContent(html)可以设置当前页面的内容，该函数有一个必选参数 html，该函数在浏览器中对应执行的 JavaScript 代码如下。

```
function(html) {
  document.open();
  document.write(html);
  document.close();
}
```

28. setDefaultNavigationTimeout(timeout)

使用 setDefaultNavigationTimeout(timeout)能实现该函数的更改默认的最大导航超时,默认为 30 秒,该函数的必选参数 timeout(单位为毫秒),设置为 0 将禁用超时。使用这个函数将覆盖 goto()、goBack()、goForward()、reload()、waitForNavigation()方法的默认 30 秒超时。

29. coroutine setExtraHTTPHeaders(headers)

coroutine setExtraHTTPHeaders(headers)可以设置额外的 HTTP 标头,将在页面启动的每个请求中发送额外的 HTTP 标头。该方法需要一个字典类型参数 headers,是要合并到请求头的字典,不保证传出请求中的标头顺序。

30. coroutine setOfflineMode(enabled)

通过 coroutine setOfflineMode(enabled)可以设置启用或禁用离线模式,该方法需要一个布尔型参数 enabled。

31. coroutine setUserAgent(userAgent)

通过 coroutine setUserAgent(userAgent)可以设置要在此页面中使用的用户代理。

32. coroutine setViewport(viewport)

coroutine setViewport(viewport)能实现设置视图,其参数同 emulate()方法的 viewport 项。

33. coroutine tap(select)

coroutine tap(select)能完成单击与之匹配的元素,类似手机上的触摸功能,该方法的必选参数 selector 表示选择器路径。

34. target

target 表示返回此页面创建的目标。

35. coroutine title()

通过 coroutine title()可以获取页面标题。

36. coroutine type(selector, text, options, ** kwargs)

coroutine type(selector, text, options, ** kwargs)方法有两个必选参数 selector、text,在指定选择器 selector 处输入文本 text,作用同 selenium 库的 keys,如果没有匹配元素,则报错 PageError 错误。

```
await page.type('#kw', '测试')
```

37. url

url 用于获取此页面的 URL 地址。

38. viewport

viewport 表示返回视图信息字典,返回信息与 setViewport()方法的字段相同。

39. coroutine waitFor(selectorOrFunctionOrTimeout, options, * args, ** kwargs)

通过 coroutine waitFor()可以实现等待页面上匹配元素出现,返回等待的对象 JSHandle。该方法参数说明如下。

(1) selectorOrFunctionOrTimeout: xpath 或函数字符串或 timeout(单位为毫秒)。

如果 selectorOrFunctionOrTimeout 是 int 或 float,则将其视为加载超时(默认为毫

秒),将返回在超时后执行的 future。如果 selectorOrFunctionOrTimeout 是一个 JavaScript 函数字符串,则此方法使用方式同 waitForFunction()。如果 selectorOrFunctionOrTimeout 是选择器字符串或 XPath 字符串,则此方法作用同 waitForSelector()或 waitForXPath()。如果字符串以//字符串开头,则将字符串视为 XPath。

(2) options:可选参数参考 waitForFunction()或 waitForSelector()方法。

(3) args:传递函数的参数。

第一个位置参数 selectorOrFunctionOrTimeout:

waitFor()方法自动检测相关标记或选择器,但可能错过检测,如果不按预期工作,可以直接使用 waitForFunction()或 waitForSelector()。例如:

```
page = await browser.newPage()
await page.goto("http://www.baidu.com/")
await page.waitFor(3000)
await page.waitFor('#kw', timout=8000)
wait page.waitFor('//*[@id="kw"]', timout=3000)
await page.type('#kw', '测试')
await page.click('#su', delay=10)
await browser.close()
```

40. coroutine waitForFunction(pageFunction, options, * args, ** kwargs)

通过 coroutine waitForFunction()可以实现等待函数执行完成并返回一个 truthy 值(真值),在 JavaScript 中指的是在布尔值上下文中,转换后的值为真值。所有值都是真值,除非它们被定义为假值(false、0、""、null、undefined 和 NaN)。

必选参数 pageFunction,是要在页面中执行的 JavaScript 字符串,该函数应设置返回值,当返回 truthy 值或达到超时时结束等待。如果 pageFunction 有参数,则以不定长参数 args 传递给 waitForFunction。

该方法还有以下可选项参数 options。

(1) polling:pageFunction 执行的间隔,默认为 raf(不断执行)。如果 polling 是数字,则将其视为执行函数的间隔(单位为毫秒),如果是 mutation,将在 DOM 树变化时执行。

(2) timeout:等待的最长时间(单位为毫秒),默认为 30 秒,如果为 0 则禁用超时。

需要注意参数传递的顺序 waitForFunction(pageFunction, options:dict, * args, ** kwargs)。例如:

```
    await page.goto("http://www.baidu.com/")
    await page.waitForFunction(("(function (){if (document.querySelector('#kw')){return
true}})()"), timeout=3000)
```

41. coroutine waitForNavigation(options, ** kwargs)

coroutine waitForNavigation(options, ** kwargs)可以实现当页面导航到新的 URL 或重新加载时,将返回响应。对于可能间接导致页面导航的代码非常有用。如果是返回上一页或下一页发生的导航,则不返回任何内容。

pyppeteer 文档提供的典型例子如下。

```
navigationPromise = asyncio.ensure_future(page.waitForNavigation())
await page.click('a.my-link')     #间接发生导航
await navigationPromise           #等待导航完成
```

或者

```
await asyncio.wait([
    page.click('a.my-link'),
    page.waitForNavigation(),
])
```

42. coroutine waitForRequest(urlOrPredicate, options, ** kwargs)

coroutine waitForRequest()能实现等待请求,请求成功则返回 Request 对象。该方法的必选参数 urlOrPredicate,可以是要等待的 URL 字符串或函数,其他可选参数有 timeout(等待超时时间)。例如:

```
firstRequest = await page.waitForRequest('http://example.com/resource')
finalRequest = await page.waitForRequest(lambda req: req.url == 'http://example.com' and
req.method == 'GET')
print(firstRequest.url)
```

43. coroutine waitForResponse(urlOrPredicate, options, ** kwargs)

coroutine waitForResponse()能实现等待响应。该方法的必选参数 urlOrPredicate,可以是要等待的 URL 字符串或函数,其他可选参数有 timeout(等待超时时间)。例如:

```
firstResponse = await page.waitForResponse('http://example.com/resource')
finalResponse = await page.waitForResponse(lambda res: res.url == 'http://example.com' and
res.status == 200)
print(finalResponse.ok)
```

44. waitForSelector(selector, options, ** kwargs)

waitForSelector()能实现等待页面上出现匹配的 selector 元素,该方法需要一个必选参数元素选择器路径 selector。返回等待的对象,该对象再将选择器字符串指定的元素添加到 DOM 时解析。

可选参数如下。

(1) visible:是否等待元素出现在 DOM 中并可见。默认为 False,即没有 display:none 或 visibility:hidden 的 CSS 属性。

(2) hidden:是否等待元素在 DOM 中不可见,默认为 False,即具有 display:none 或 visibility:hidden 的 CSS 属性。

(3) timeout:等待的最长时间(单位为毫秒)。

45. waitForXPath(xpath, options, ** kwargs)

waitForXPath()能实现等待页面上出现匹配的 XPath 元素,该方法需要一个必选参数元素 XPath 选择器路径。可选参数同 waitForSelector()方法的可选项参数。

46. workers

通过 workers 能获取当前页面所有执行的线程。web workers 概念是解决客户端 JavaScript 无法多线程运行的问题,其定义的 worker 是指代码的并行线程,不过 web worker 处于一个自包含的环境中,无法访问主线程的 window 对象和 document 对象。

3.2.3 页面 Cookie 处理

对于常用于模拟登录的 selenium 库和 pyppeteer 库而言,Cookie 的处理是获得登录状

态的重要步骤。通过获取浏览器的 Cookie，然后通过 requests 等网络请求库携带登录后的 Cookie 可以更加高效地完成请求任务。

下面介绍 pyppeteer 浏览器的 Cookie 增删改查等常用操作，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1. coroutine cookies(* urls)

获取 Cookie，如果未指定 URL，则 coroutine cookies(* urls)方法返回当前页面 URL 的 Cookie；如果指定了 URL，则仅返回指定 URL 的 Cookie。

返回的 Cookie 包含字段 name、value、url、domain、path、expires、httpOnly、secure、session、sameSite。例如：

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.goto("http://www.baidu.com/")
    cookies = await page.cookies()
    print(cookies)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
# 打印 cookie 信息
[{'name': 'PSTM', 'value': '1591714773', 'domain': '.baidu.com', 'path': '/', 'expires':
3739198420.314667, 'size': 14, 'httpOnly': False, 'secure': False, 'session': False}, … ]
```

2. coroutine deleteCookie(* cookies)

coroutine deleteCookie(* cookies)可以实现删除 Cookie，应传入包含 name、url、domain、path、secure 字段的待删除 Cookie，如果要删除当前页面的 Cookie，至少应该传入 name 和 domain 的字典。例如：

```
await page.deleteCookie({'name': 'PSTM', 'domain': '.baidu.com'})
```

3. coroutine setCookie(* cookies)

通过 coroutine setCookie(* cookies)方法实现设置 Cookie。该方法传入的参数是单条 Cookie 信息字典，应该是包含这些字段 name(必填)、value(必填)、url、domain、path、expires、httpOnly、secure、sameSite。例如：

```
await page.setCookie({'name': "test", 'value': "123456"})
```

3.2.4 页面节点选择器

在 pyppeteer 中使用 pyppeteer.element_handle.ElementHandle 对象，来表示页内 DOM 元素。ElementHandle 对象可以通过 querySelector()方法创建实现，还可以用作 querySelectorEval()和 evaluate()方法的参数。

ElementHandle 对象具有下列实用的方法和属性，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1. coroutine boundingBox()

通过 coroutine boundingBox()方法可以返回此元素的边界框，如果元素不可见，则返回

None。返回信息包括元素的 x 坐标、元素的 y 坐标、元素的宽度、元素的高度，单位为像素。

2. coroutine boxModel()

通过使用 coroutine boxModel()能返回元素框，如果元素不可见，则返回 None，框表示点列表。每个点是{x, y}的坐标形式，返回值是包含 content(内容框)、padding(填充框)、border(边框)、margin(边距框)、width(元素的宽度)、height(元素的高度)字段的字典。

3. coroutine click(selector, ** kwargs)

通过 coroutine click(selector, ** kwargs)方法能实现单击此元素的中心，如果需要，会将元素滚动到视图中。如果元素与 DOM 分离，则该方法会引发 ElementHandleError 异常。

4. coroutine contentFrame()

coroutine contentFrame()方法可以返回元素所属的内容框架，不在 iframe 中则返回 None。

5. coroutine focus()

coroutine focus()能实现聚焦该元素。

6. coroutine hover()

通过 coroutine hover()可以实现将鼠标移动元素的中心，如果需要会将元素滚动到视图中。

7. coroutine isIntersectingViewport()

coroutine isIntersectingViewport()可以判断元素在视图中是否可见，如果可见，则返回 True。

8. coroutine press(key, options, ** kwargs)

coroutine press(key, options, ** kwargs)可以向指定元素发送按键。必选参数 key 是按键名，如 ArrowLeft，有可选参数 text、delay。该函数先聚焦元素，然后使用 keyboard.down()和 keyboard.up()。

9. coroutine screenshot(options, ** kwargs)

coroutine screenshot(options, ** kwargs)可以截取元素的屏幕截图，如果该元素与 DOM 分离，则此方法会引发一个 ElementHandleError 错误，可用选项与 screenshot()方法相同。

10. coroutine tap()

单击元素的中心，如果需要，coroutine tap()方法将元素滚动到视图中。如果元素与 DOM 分离，则该方法会引发 ElementHandleError 错误。

11. coroutine type(text, options, ** kwargs)

coroutine type(text, options, ** kwargs)能聚焦元素，然后键入文本，用法同 Keyboard.type()方法。

12. coroutine uploadFile(* filePaths)

coroutine uploadFile(* filePaths)可以实现上传文件。

13. coroutine querySelector(selector)

coroutine querySelector(selector)可以实现返回在指定元素下匹配的第一个元素 selector，如果没有匹配元素，则返回 None。

14. coroutine querySelectorAll(selector)

coroutine querySelectorAll(selector)能实现返回此元素下匹配的所有 selector 元素，如果没有元素匹配，则返回空列表。

15. coroutine xpath(expression)

通过使用 coroutine xpath(expression)方法查找满足 XPath 表达式的元素，如果没有这样的元素，则返回一个空列表。

16. coroutine querySelectorAllEval(selector, pageFunction, * args)

通过使用 coroutine querySelectorAllEval(selector, pageFunction, * args)对满足条件的所有元素执行函数，并将其作为第一个参数传递给 pageFunction。如果没有元素匹配 selector，则该方法会引发 ElementHandleError 错误。

如下源码演示了 querySelectorAllEval()方法的用法。

```
<div class = "feed">
    <div class = "tweet">Hello!</div>
    <div class = "tweet">Hi!</div>
</div>
```

提取出指定文本节点的内容并判断。

```
feedHandle = await page.J('.feed')
assert (await feedHandle.JJeval('.tweet', '(nodes => nodes.map(n => n.innerText))')) ==
['Hello!', 'Hi!']
```

assert 断言用于判断一个表达式，在表达式条件为 false 的时候触发异常。箭头函数是在 es6 中添加的一种规范，x => x * x 相当于 function(x){return x * x}。

17. coroutine querySelectorEval(selector, pageFunction, * args)

coroutine querySelectorEval(selector, pageFunction, * args)可以对匹配的第一个元素执行 pageFunction()函数，将其作为第一个参数传递给 pageFunction()，如果没有匹配，则引发 ElementHandleError 错误。

```
tweetHandle = await page.querySelector('.tweet')
assert (await tweetHandle.querySelectorEval('.like', 'node => node.innerText')) == 100
assert (await tweetHandle.Jeval('.retweets', 'node => node.innerText')) == 10
```

Page 实例化对象提供了 5 个与 ElementHandle 对象相关的方法，这些方法用于在页面对象中获取单个 ElementHandle 对象或多个 ElementHandle 对象的列表，以及提供了操作 ElementHandle 对象的重要方法。这 5 个方法分别是：J 别名 querySelector()、JJ 别名 querySelectorAll()、JJeval 别名 querySelectorAllEval()、Jeval 别名 querySelectorEval()、Jx 别名 xpath()。下面详细介绍这五个方法。

1) coroutine querySelector(selector)

通过使用 coroutine querySelector(selector)方法可以获取匹配选择器 selector 路径的 ElementHandle 对象。该方法需要搜索元素的选择器字符串作参数，返回 ElementHandle 对象或者 None。例如：

```
await page.goto("http://www.baidu.com/")
select = await page.querySelector('#kw')
```

2) coroutine querySelectorAll(selector)

使用 coroutine querySelectorAll(selector)可以获取所有匹配选择器 selector 路径的 ElementHandle 对象列表。该方法需要搜索元素的选择器字符串做参数,返回 ElementHandle 列表或者返回空列表。

3) coroutine querySelectorAllEval(selector, pageFunction, * args)

coroutine querySelectorAllEval(selector, pageFunction, * args)能实现对所有匹配元素执行 JavaScript 代码。该方法需要两个参数,第一个参数是选择器 selector,第二个参数是要在浏览器上运行的 JavaScript 函数的字符串 pageFunction,pageFunction()函数将匹配元素的数组作为第二个参数。通过 args 传递其他参数给 pageFunction()。例如:

```
await page.goto("http://www.baidu.com/")
js = """
    function f(select, a, b) {
    console.log(select);
    console.log(a + b)
}
    """
select = await page.querySelectorAllEval('#kw', js, 1, 2)
```

在浏览器的开发者调试模式下的 Console 选项卡中,如图 3-3 所示结果。

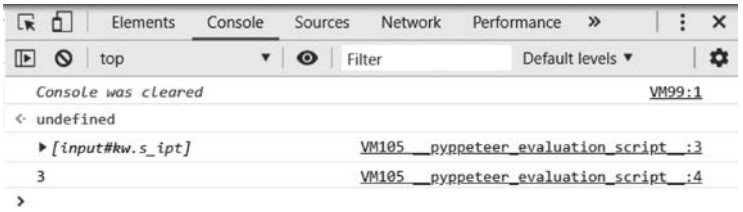

图 3-3 querySelectorAllEval 执行 JavaScript 代码

4) coroutine querySelectorEval(selector, pageFunction, * args)

通过 coroutine querySelectorEval(selector, pageFunction, * args)可以对匹配的第一个元素执行 JavaScript()函数,参数及作用与 querySelectorAllEval()相同,不同之处在于 pageFunction()函数第一个参数不是数组,而是匹配对象。

5) coroutine xpath(expression)

coroutine xpath(expression)用于获取符合 XPath 表达式的 ElementHandle 对象。该方法需要一个必选参数 XPath 表达式 expression,此方法将返回所有符合条件的 ElementHandle 列表。

3.2.5 键盘和鼠标操作

1. 键盘事件

Keyboard 负责处理键盘事件,其位于 pyppeteer.input.Keyboard 路径下。在页面中调用 Keyboard 对象,可以通过访问 Page.keyboard 实现。Keyboard 类提供了用于管理虚拟键盘的 api,其中 type()是一个输入的高级接口,它接收原始字符并在页面上生成正确的 keydown、keypress、input、keyup 等事件。如果需要加强控制,可以通过 down()、up()和 sendCharacter()等手动触发事件来模拟真实的键盘操作。

Keyboard 核心事件主要有以下几个，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1) coroutine down(key, options, ** kwargs)

coroutine down(key, options, ** kwargs)可发送按下键盘 key 键的指令。如果 key 是修饰键，例如 Shift、Meta 或 Alt，则后续按键将与该修饰符一起发送，要释放修饰键使用 up()。修饰键会影响 down 事件，按住 Shift 键将在大写中键入文本。该方法需要一个必选参数 key，key 代表要按的键的名称。可选参数 text，代表要输入的文本内容。

2) coroutine up(key)

coroutine up(key)用于释放按下的键。该方法需要一个必选参数 key，即要释放的键名。

3) coroutine press(key, options, ** kwargs)

通过 coroutine press(key, options, ** kwargs)能实现向下按一次 key 键。已按下的修饰键会影响 press()方法，如按住 Shift 键将在大写中键入文本。该方法需要一个必选参数 key，key 代表要按的键的名称。可选参数有 text 和 delay，text 是指要输入的文本内容；delay 是按下和弹起之间等待的时间，默认为 0。

4) coroutine sendCharacter(char)

将字符发送到页面，coroutine sendCharacter(char)方法会调用一个 keypress 和 input 事件，但不会发送 keydown 或 keyup 事件。修饰键对 sendCharacter 不起作用，按住 Shift 键不会以大写形式输入文本。该方法需要一个必选参数 char，即要发送的字符。

5) coroutine type(text, option, ** kwargs)

通过 coroutine type(text, option, ** kwargs)可以将字符输入鼠标焦点所在的元素，如文本框。text 是该方法的必选参数，表示键入聚焦元素的文本。可选参数 delay，是按键之间等待的时间，单位为毫秒。

下面是一些键盘事件的示例。

```
await page.keyboard.type('Hello, World!')
await page.keyboard.press('ArrowLeft')
await page.keyboard.down('Shift')
for i in 'World':
    await page.keyboard.press('ArrowLeft')
await page.keyboard.up('Shift')
await page.keyboard.press('Backspace')
# 最后显示的结果是 Hello!
```

按下大写 A 键的示例。

```
await page.keyboard.down('Shift')
await page.keyboard.press('KeyA')
await page.keyboard.up('Shift')
```

2. 鼠标事件

Mouse 负责处理鼠标事件，其位于 pyppeteer.input.Mouse 路径下。在页面中调用 Mouse 对象，可通过访问 Page.mouse 实现。Mouse 核心事件主要有以下几个，其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1) coroutine click(x, y, options, ** kwargs)

coroutine click(x, y, options, ** kwargs)表示单击 x、y 处的坐标。该函数接收必选参数 x、y 坐标,可选参数 button(键击类型,可选 left、right、middle,默认为 left)、clickCount(单击次数,默认为 1)、delay(按下和弹起的时间)。

2) coroutine down(option, ** kwargs)

coroutine down(option, ** kwargs)表示发送鼠标按下事件。可选参数 button(键击类型,可选 left、right、middle,默认为 left)、clickCount (单击次数,默认为 1)。

3) coroutine move(x, y, options, ** kwargs)

coroutine move(x, y, options, ** kwargs)方法可以实现发送鼠标移动事件。可选参数 steps(步长),如果指定 steps,则通过中间事件 mousemove 发送,默认步长为 1。

4) coroutine up(option, ** kwargs)

使用 coroutine up(option, ** kwargs)可以释放按下按钮,可选参数 button(键击类型可选 left、right、middle,默认为 left)、clickCount (单击次数,默认为 1)。

3.2.6 内嵌框处理

pyppeteer 提供了内嵌框类 pyppeteer.frame_manager.Frame,在 Page 类中通过 Page.mainFrame 可以访问当前 Page 对应的 Frame 对象,通过 Page.frames 属性可以获取当前页面的所有 frame 列表。

Frame 对象具有与 Page 对象大部分相同的属性和方法,如表 3-1 所示。其中以 coroutine 开头代表的是协程方法,使用时需要加 await 关键字。

表 3-1 Frame 对象与 Page 对象使用方法一致的属性和方法

方法或属性	说 明
coroutine J(selector)	别名 querySelector()
coroutineJJ(selector)	别名 querySelectorAll()
coroutineJJeval(selector, pageFunction, * args)	别名 querySelectorAllEval()
coroutineJeval(selector, PageFunction, * args)	别名 querySelectorEval()
coroutineJx(expression)	别名 xpath()
coroutineaddScriptTag(option)	注入 JavaScript 文件
coroutineaddStyleTag(options)	注入 CSS 文件
coroutineclick(selector, options, ** kwargs)	单击匹配的元素
coroutinecontent()	获取页面全部的 HTML 内容
coroutineevaluate(pageFunction, * args, force_expr)	框架下执行 pageFunction
coroutineevaluateHandle(pageFunction, * args)	框架下执行 pageFunction
coroutineexecutionContext()	返回此框架的上下文对象
coroutine focus(selector)	聚焦匹配的元素
coroutine hover(selector)	鼠标指针悬停匹配元素上
coroutineselect(selector, * values)	返回匹配选择器元素
coroutinesetContent(html)	将 html 设为当前内容
coroutine tap(selector)	单击匹配 selector 的元素
coroutinetitle()	获取框架的标题
coroutinetype(selector, text, options, ** kwargs)	在匹配的元素上输入文本

续表

方法或属性	说　明
url	获取框架的 URL 地址
waitFor(selectorOrFunctionOrTimeout, options, * args, ** kwargs)	根据 selectorOrFunctionOrTimeout 等待页面加载完成
waitForFunction(pageFunction, options, * args, ** kwargs)	等待函数完成
waitForSelector(selector, options, ** kwargs)	等待匹配元素
waitForXPath(xpath, options, ** kwargs)	等待匹配 XPath 的元素

除与 Page 对象相同的属性和方法外，Frame 对象还有如下部分关于内嵌框处理的独立方法和属性。

(1) childFrames：获取子框架。

(2) isDetached()：如果此框架已分离则返回 True，否则返回 False。

(3) parentFrame：获取当前框架的父框架，如果当前框架是主框架或分离框架，则返回 None。

(4) name：获取 frame 的 name。

如下源码是打开本地内嵌了百度网页的 iframe.html 文件(该文件在本章对应资源文件中)，然后通过 pyppeteer 的 Frame 对象来操作 iframe 标签内的百度网页，执行一个简单的搜索任务。

```
import asyncio
import os

from pyppeteer import launch


async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.goto(os.getcwd() + r'\iframe.html')
    frame = page.mainFrame.childFrames[0]
    title = await frame.title()
    print(title)
    print(frame.name)
    print(frame.url)
    await frame.type('#kw', '测试')
    await frame.click('#su', delay = 10)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

执行上述 代码后，在浏览器的搜索框中正确输入搜索关键字，如图 3-4 所示。

3.2.7 JavaScript 操作

在 pyppeteer 中，pyppeteer.execution_context.JSHandle 类实例化得到 JSHandle 对象，表示网页内的 JavaScript 对象可以通过 evaluateHandle()方法创建 JSHandle。

相对 selenium 库而言，pyppeteer 库在与浏览器的 JavaScript 交互上更加出色。selenium 库只提供了两个功能有限的 JavaScript 执行函数，而 pyppeteer 库除了 Page 对象和

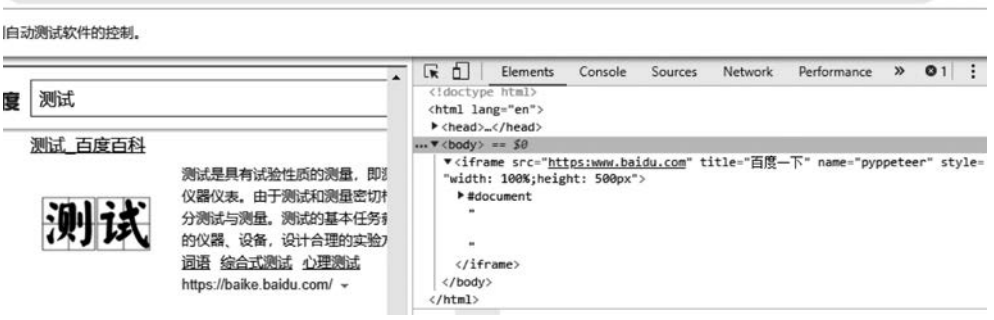

图 3-4 pyppeteer 操作内嵌框

ElementHandle 对象都具有的 JJeval()、Jeval()方法，以及 Page 对象单独具有的 addScriptTag()、waitForFunction()方法之外，Page 实例化对象还提供了下列方法用于 JavaScript 的操作。其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

1. coroutine evaluate(pageFunction, *args, force_expr)

通过 coroutine evaluate(pageFunction, *args, force_expr)可以实现在浏览器上执行 JavaScript 函数或 JavaScript 表达式，并获取结果。pageFunction 参数是要在浏览器上执行的函数或表达式的字符串；关键字参数 force_expr 为 True，则将 pageFunction 当表达式计算，如果为 False(默认值)，自动检测是函数或表达式；args 是不定长参数，可以传递 JavaScript()函数所需要的参数。例如：

```
test = await page.evaluate('function test(a, b){return a + b}', 1, 2, force_expr = False)
print(test)     #打印结果为 3
```

2. coroutine evaluateHandle(pageFunction, *args)

coroutine evaluateHandle(pageFunction, *args)可以实现在当前页面执行 JavaScript 函数，参数 pageFunction 是要执行的 JavaScript 函数字符串。evaluateHandle()和 evaluate()之间的区别是 evaluateHandle 返回的 JSHandle 对象不是数值。例如：

```
test = await page.evaluateHandle('function test(a, b){return a + b}', 1, 2)
print(test) #打印< pyppeteer.execution_context.JSHandle object at 0x000001C163B20898 >
```

3. coroutine evaluateOnNewDocument(pageFunction, *args)

通过使用 coroutine evaluateOnNewDocument(pageFunction, *args)可以实现在当前页面添加 JavaScript 函数。当发生页面导航、页面内嵌框架导航的时候，附加的 pageFunction 代码会自动执行。

4. coroutine exposeFunction(name, pyppeteerFunction)

coroutine exposeFunction(name, pyppeteerFunction)可以将 Python 函数添加到浏览器的 window 对象中，可以从浏览器进程中调用已注册的 Python 函数，是一个非常强大的功能。参数 name 是 window 对象上函数的名称；参数 pyppeteerFunction 是将在 python 进程上调用的非异步函数。

如下案例将通过 JavaScript 函数获取 html 源码，然后调用 Python 函数解析出源码中的标题。

```
import asyncio
from lxml import etree
from pyppeteer import launch

ping_script = """
    setInterval(function () {
        var html = document.documentElement.outerHTML;
        pyrsp(html);
        console.log('ping!');
    }, 1000);
"""
def pyrsp(html):
    """
    解析 JavaScript 回调返回的 html 源码
    :param html:
    :return:
    """
    xp = etree.HTML(html)
    title = xp.xpath('//title/text()')
    print(title)
async def main():
    browser = await launch(headless = False, devtools = True)
    page = await browser.newPage()
    await page.goto('https://www.baidu.com/')
    await page.exposeFunction("pyrsp", pyrsp)
    await page.evaluate(ping_script)
if __name__ == "__main__":
    loop = asyncio.new_event_loop()
    loop.create_task(main())
    loop.run_forever()
```

5. coroutine queryObjects(prototypeHandle)

迭代 JavaScript 堆并找到所有具有句柄的对象，参数 prototypeHandle(JSHandle)是原型对象的 JSHandle。

6. coroutine setJavaScriptEnabled(enabled)

coroutine setJavaScriptEnabled(enabled)具备启用或禁用 JavaScript 功能。

3.2.8 Request 和 Response

pyppeteer 提供了请求类 pyppeteer.network_manager.Request 和响应类 pyppeteer.network_manager.Response，是网络请求中常用的类。只要页面发送请求(例如请求网络资源)，pyppeteer 会发生以下事件：当页面请求发出时发生 request 事件，当页面收到请求的响应时发生 response 事件，请求完成并下载响应正文时发生 requestfinished 事件。

如果请求在某个时间点失败，那么将发出 requestfailed 事件，而不是 requestfinished 事件。如果请求得到重定向响应，则请求将视作完成 requestFinished 事件，并向重定向地址发出新的请求。

对于常用的 Request 对象和 Response 对象，在开启请求拦截后可以作为绑定的回调函数的默认参数。要获取 Request 和 Response 对象，也可以通过在 Page 对象上调用 waitForRequest()、waitForResponse()函数，返回指定 URL 地址的请求或响应对象。

1. Request 对象

Request 具有下列属性和方法，是操作请求的重要接口。其中以 coroutine 开头代表的

是协程方法，使用时需要加 await 关键字。

1）coroutine abort(errorCode)

中止请求，要使用 coroutine abort(errorCode)之前启用请求拦截方法 Page.setRequestInterception()。参数 errorCode 是可选的错误代码字符串，默认为 failed，errorCode 可选错误代码字符串如表 3-2 所示。

表 3-2　errorCode 可选错误代码字符串

errorCode 错误代码	含　义
aborted	用户终止操作
accessdenied	无访问网络以外资源的权限
addressunreachable	IP 地址无法访问
blockedbyclient	客户端阻止请求
blockedbyresponse	请求失败，因为该请求不满足设定条件(如 X-Frame-Options 和 Content Security Policy 安全检查)。X-Frame-Options HTTP 响应头是用来给浏览器指示允许一个页面能否在 <frame>、<iframe>、<embed>或者<object>中展现的标记。Content-Security-Policy 响应标头允许网站管理员控制允许用户代理为给定页面加载的资源
connectionaborted	由于未收到发送数据的 ACK 而导致的连接超时
connectionclosed	连接已关闭(对应于 TCP FIN)
connectionfailed	连接尝试失败
connectionrefused	连接尝试被拒绝
connectionreset	重置连接(对应于 TCP RST)
internetdisconnected	Internet 连接已丢失
namenotresolved	无法解析主机名
timedout	操作超时
failed	发生了一般故障

2）coroutine continue_(overrides)

使用可选的请求覆盖当前请求，要使用 coroutine continue_(overrides)方法前开启拦截器。参数 overrides 是可以包含以下字段的字典的。

(1) url：如果设置，请求 URL 将被更改。

(2) method：如果设置，则更改请求方法(例如 GET)。

(3) postData：如果设置，则更改请求正文。

(4) headers：如果设置，则更改请求 HTTP 标头。

3）failure()

使用 failure()方法可以返回错误文本，如果 requestfailed 事件未执行失败，则返回 None。当请求失败时，该方法返回具有 errorText 字段的字典，该字段包含可读的错误消息，例如 net::ERR_RAILED。

4）frame

frame 属性指返回匹配的 frame 对象，如果导航到错误页面，则返回 None。

5）headers

headers 属性可以返回此请求的 HTTP 标头字典，所有标题名称都是小写。

6）isNavigationRequest()

isNavigationRequest()请求是否让框架发生导航。

7）method

method 属性是指返回此请求的方法(GET、POST 等)。

8）postData

postData 可以实现返回此请求的正文内容。

9）redirectChain

通过 rodirestChain 可以获取重定向连接，如果没有重定向并且请求成功，则请求链接将为空，如果服务器至少响应一个重定向，则返回包含重定向的所有请求，redirectChain 在同一个请求链的所有请求之间共享。

10）resourceType

resourceType 属性可以渲染引擎分类的此请求的资源类型，类型有 document、stylesheet、image、media、font、script、texttrack、xhr、fetch、eventsource、websocket、manifest、other。

11）coroutine respond(Response)

coroutine respond(Response)可以设定请求的响应内容，参数 response 是一个字典，可包含 status(默认为 200)、headers、contentType、body。

12）response

response 可以返回匹配的 Response 对象，如果未收到响应则返回 None。

13）url

url 代表请求的 URL。

下面的案例是使用 waitForRequest()方法获得一个 Request 对象，在 waitForRequest 堵塞时通过手动刷新以获得指定 URL 地址的请求对象，源码如下。

```
import asyncio
from pyppeteer import launch
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.goto("http://example.com/resource")
    rsp = await page.waitForRequest("http://example.com/resource")  #堵塞,手动刷新页面
    #打印 http://example.com/resource
    print(rsp.url)
    #打印 GET
    print(rsp.method)
    #打印 Headers 字典
    print(rsp.headers)
    #打印 None
    print(rsp.response)
    await browser.close()
asyncio.get_event_loop().run_until_complete(main())
```

2. Response 对象

Response 对象具有下列常用的属性和方法，是处理响应对象的重要接口。其中以 coroutine 开头代表的是协程方法，使用时需要加 await 关键字。

(1) buffer()：返回等待的结果，buffer()方法解析为带有响应主体的字节。

(2) fromCache：如果是本地缓存提供的响应，则返回 True。

(3) fromServiceWorker：如果响应是由服务器返回的，则返回 True。

(4) headers：返回响应头字典。

(5) coroutine json()：通过 coroutine json()方法可以获取响应体的 JSON 表示。

(6) ok：ok 属性表示响应状态码在 200～299，则返回 True。

(7) request：使用 request 可以得到匹配的 Request 对象。

(8) securityDetails：securityDetails 可以返回与此响应关联的安全详细信息。

(9) status：返回响应的状态代码。

(10) coroutine text()：使用 coroutine text()可以获取响应体的文本表示。

(11) url：url 属性指响应的 URL。

如下示例代码是使用 waitForResponse()方法获得一个 Response 对象，注意当代码运行到 waitForResponse()时会堵塞，需要手动刷新浏览器的页面。

```
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.goto("http://example.com/resource")
    rsp = await page.waitForResponse("http://example.com/resource")   # 堵塞，手动刷新页面
    print(rsp.url)                          # 打印 http://example.com/resource
    print(rsp.headers)                      # 打印响应头字典
    print(rsp.request) # 打印< pyppeteer.network_manager.Request object at 0x000002C212CAF748 >
    print(rsp.status)                       # 打印 404
    print(rsp.ok)                           # 打印 False
    print(rsp.buffer())
    await browser.close()
```

3.3 pyppeteer 常用操作

3.3.1 启动项参数设置

在初始化启动器时设置{'headless': False, 'executablePath': 'C:\Program Files (x86)\Google\Chrome\Application\chrome.exe'}参数，其作用是不打开无头模式，指定启动浏览器的路径。这些参数是 pyppeteer 的启动项参数，像这样的启动项参数在 pyppeteer 中有数十项。通过对启动项参数的配置可以实现一些高级功能，例如无头模式、忽略 HTTPS 错误、打开开发者工具、指定使用 Chrome 浏览器或 Chromium 浏览器、代理配置等。

1. 常用配置项参数

启动器支持启动项参数设置，一些参数是非常有必要的，例如防识别参数、浏览器有头模式或者无头模式、设置浏览器数据目录等，下面是常用的配置参数。

(1) ignoreHTTPSErrors：是否忽略 HTTPS 错误，默认为 False。

(2) headless：在无头模式下运行浏览器，默认为 True。如果 appmode 或 devtools 选项为 True，则 headless 默认值为 False。

(3) executablePath：指定 chromium 或 chrome 可执行文件的路径。

(4) slowMo：设置 pyppeteer 操作延迟的毫秒数。

(5) args：要传递给浏览器的附加参数列表，通过该参数传递更多配置参数。

(6) ignoreDefaultArgs：移除指定的 pyppeteer 内置默认参数。

(7) handleSIGINT：使用 Ctrl+C 快捷键关闭浏览器进程，默认为 True。

(8) handleSIGTERM：关闭 SIGTERM 上的浏览器进程，默认为 True。

(9) handleSIGHUP：在 SIGHUP 时关闭浏览器进程，默认为 True。

(10) dumpio：是否将浏览器进程 stdout 和 stderr 导入 process.stdout 和 process.stderr 中，默认为 False。

(11) userDataDir：指定用户数据目录的路径。

(12) env：指定浏览器可用的环境变量值字典，默认与 python 进程相同。

(13) devtools：为每个选项卡自动打开 devtools 面板，默认为 False。如果此选项为 True，将设置 headless 选项为 False。

(14) logLevel：设置日志级别，默认使用根记录器。

(15) autoClose：脚本完成时自动关闭浏览器进程，默认为 True。

(16) loop(asyncio.AbstractEventLoop)：事件循环(实验参数)。

2. 常用附加参数

pyppeteer 库可用的附加参数与 selenium 库基本一致，这些参数来自 Chrome 浏览器命令行启动参数，因此有某一方面特别的功能可以参考官网文档网址是 https://peter.sh/experiments/chromium-command-line-switches/，各种参数项超过 500 余项。通过附加参数设置代理、浏览器尺寸及运行权限等配置。

部分常用的附加启动项参数如表 3-3 所示。

表 3-3 常用附加启动项参数

附加参数	说明
-user-agent=UA	设置 User-Agent
blink-settings=imagesEnabled=false	不加载图片
--disable-gpu	禁用 GPU 硬件加速
--hide-scrollbars	隐藏滚动条
--no-sandbox	禁用所有使用沙盒化的进程的沙盒
--disable-setuid-sandbox	禁用 setuid 沙盒(仅限 Linux)
--start-maximized	启动窗口最大化
--disable-popup-blocking	禁用弹出窗口拦截
--proxy-server=127.0.0.1:1080	设置 127.0.0.1:1080 为请求代理
--load-extension=chrome_extension_path	添加插件，chrome_extension_path 是插件解压后路径
--disable-extensions-except = chrome _ extension _path	允许指定的插件运行，通常与一项配置一起使用，chrome_extension_path 是插件解压后路径

3.3.2 识别特征处理

自动化浏览工具不管是 pyppeteer 库还是 selenium 库，它们控制下的浏览器有一个典型的特征，那就是浏览器中 window.navigator.webdriver 值为 true，标志该浏览器是自动化

控制的。在正常浏览器的开发者工具中，Console 面板输出该值应该是 undefined，这也是网站识别自动化控制浏览器的主要特征。

在 pyppeteer 包下的 launcher.py 文件中，定义了 DEFAULT_ARGS 列表，该列表中--enable-automation 参数的作用是 Enable indication that browser is controlled by automation，翻译成中文的意思就是，启用浏览器由自动化控制的指示。因此在初始化启动器时，移除该参数即可达到修改 webdriver 特征的目的，如下源码是移除该特征的示例。

```
import asyncio
from pyppeteer import launch, launcher

launcher.DEFAULT_ARGS.remove("--enable-automation")      #移除


async def main():
    browser = await launch(headless=False)
    page = await browser.newPage()
    await page.goto("http://www.baidu.com/")
    await page.mouse.move()
    await page.type('#kw', '测试')
    await page.click('#su', delay=10)
    await browser.close()


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

或者在启动项参数中设置 ignoreDefaultArgs，即要移除的启动参数。例如：

```
browser = await launch(headless=False, ignoreDefaultArgs=["--enable-automation"])
```

运行脚本启动 Chromuin，在浏览器的开发者工具下的 Console 选项卡中输入 window.navigator.webdriver，即可见如图 3-5 所示的无特征结果。

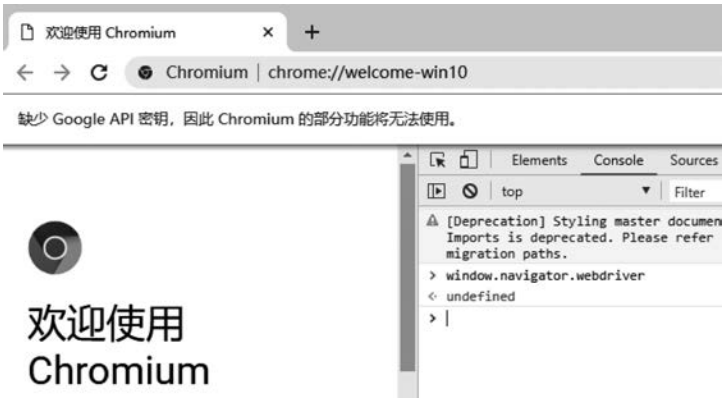

图 3-5　移除 webdriver 特征值

3.3.3　配置代理及认证

pyppeteer 设置代理的流程比较简单，通过附加启动参数--proxy-server 即可设置代理服务器地址。如果连接到代理服务器还需要认证，则通过 Page.authenticate()方法设置认证代理的用户名和密码，源码如下。

```
async def main():
    browser = await launch(headless = False, ignoreDefaultArgs = ["--enable-automation", '
--proxy-server = 127.0.0.1:1080'])
    page = await browser.newPage()
    await page.authenticate({'username': 'user', 'password': 'pwd'})
    await page.goto("http://www.baidu.com/")
    await page.mouse.move()
    await page.type('#kw', '测试')
    await page.click('#su', delay = 10)
    await browser.close()
```

3.3.4 拦截请求和响应

拦截器是 pyppeteer 的强大功能之一，这也是比 selenium 库更突出的一个功能。pyppeteer 库从内部提供了拦截器来拦截相应的事件，而 selenium 库只能通过第三方的工具才能进行拦截。pyppeteer 拦截器作用于单个 Page 对象，拦截器通过给指定事件添加回调函数来实现其拦截功能。一旦请求拦截启用，除非继续、响应或中止，否则每个请求都将暂停，直到超时。

拦截器的功能强大，应用场景广泛，例如，通过拦截器拦截响应获取 Ajax 加载的数据，通过拦截请求实现代理切换，通过拦截器过滤无效请求或敏感请求等。

使用 pyppeteer 拦截器有三个步骤。第一步是通过 Page.setRequestInterception(True)设置拦截器；第二步是通过 Page.on()绑定拦截的事件和相应的回调函数，常用的绑定事件有 request(请求事件，回调函数默认参数是 Request 对象)、response(响应事件，回调函数默认参数是 Response 对象)；第三步是使用 Page.goto()请求页面。这样请求页面过程中产生的所有拦截目标事件，都会经过回调函数的处理。

拦截请求事件和响应事件的回调函数，它们的默认参数都是请求对象(Request)和响应对象(Response)，也就意味着可以在回调函数内使用这两个对象的相关属性和方法。

如下案例将拦截 likeinlove.com 地址的请求事件和响应事件，将请求目标重定向到 www.baidu.com 地址，并且在响应事件中打印响应 HTML 文档的标题。

```
from pyppeteer import launch
from lxml import etree
import asyncio


async def request_call(request):
    if request.url == 'http://likeinlove.com/':
        await request.continue_({"url": 'https://www.baidu.com'})
    else:
        await request.continue_()                    #释放其他请求


async def resposne_call(response):
    if response.url == 'http://likeinlove.com/':
        content = await response.text()              #获取响应体正文
        xp = etree.HTML(content)
        print(xp.xpath('//title/text()'))            #打印['百度一下，你就知道']
```

```
async def main():
    browser = await launch({"headless": False})
    page = await browser.newPage()
    await page.setRequestInterception(True)
    page.on('request', lambda req: asyncio.ensure_future(request_call(req))
                                                          #绑定请求事件
    page.on('response', lambda req: asyncio.ensure_future(response_call(req))
                                                          #绑定响应事件
    await page.goto('http://likeinlove.com')
    await browser.close()


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

运行上述代码后，本来应该是 likeinlove.com 的内容被替换成了 baidu.com 的内容，响应事件的回调函数打印 baidu.com 的标题。通过上面的案例可以体会到请求拦截和响应拦截的主要处理对象还是 Request 和 Response，关于这两个对象详细的介绍详见 3.2.8 节。


视频讲解

3.4 案例：pyppeteer 动态代理的切换

一般情况下，pyppeteer 动态代理切换有两种方式。一种方式是在启动时设置代理参数--proxy-server，在每次需要切换代理时重新启动新浏览器；另一种方式是设置隧道代理，代理服务器在后台随机分配请求代理，这种方式的代理不可控。本案例是利用 pyppeteer 的拦截器功能来实现自由的代理切换，其原理是通过拦截器对关键的 request 事件进行拦截获取请求对象 Request，然后通过其他请求库获得响应内容，最后通过 Request 对象的 respond()方法设置响应信息返回浏览器。

正常的业务流程是使用 pyppeteer 的启动器打开 Chromium 浏览器，导航到百度首页，输入查询关键字 IP 单击搜索，在搜索结果中有一条结果就是当前 IP 地址。通过设置拦截器，将返回搜索结果的请求拦截，然后通过 requests 设置代理请求，将 requests 返回结果返回浏览器。创建 switch_ip.py 文件，写入如下源码。

```
import asyncio
import requests
from pyppeteer import launch, launcher

launcher.DEFAULT_ARGS.remove("--enable-automation")


async def ip(request):
    if request.url == 'https://www.baidu.com/s?ie=UTF-8&wd=ip':
        url = request.url
        headers = request.headers
        proxies = {'https': 'http://58.218.92.198:2219'}        #代理地址
        r = requests.get(url, headers=headers, proxies=proxies)
        r.encoding = 'utf-8'
        rsp = {"body": r.text, "headers": r.headers, "status": r.status_code}
        await request.respond(rsp)                              #构造该请求的响应对象
        return
    else:
        await request.continue_()                               #释放其他请求
```

```
async def main():
    browser = await launch(headless = False)
    page = await browser.newPage()
    await page.setRequestInterception(True)
    page.on('request', lambda req: asyncio.ensure_future(ip(req)))
    await page.goto("https://www.baidu.com/s?ie = UTF - 8&wd = ip")
    await browser.close()


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

测试代理最好使用正规 IP 代理商提供的免费有限测试 IP，一般网站发布的免费 IP 被泛滥使用，实际的效果并不理想。当设置好可用代理后，运行上述源码搜索出来的 IP 归属地就是代理 IP 对应的地址。

pyppeteer 还有更丰富的场景。例如，目标网站存在某一关键的加密参数，在加密逻辑复杂的情况下，可以直接使用 pyppeteer 在正常业务流程下通过拦截器、Hook 代码来获取该参数的加密结果。这得益于 pyppeteer 提供了与 Chromium 或 Chrome 浏览器更深的交互功能，它基本实现了浏览器中的开发者工具的大部分核心功能。

但是 pyppeteer 也有部分缺点。一方面是 puppteer 快速迭代，而 pyppeteer 几乎没有更新存在的 bug；另一方面是使用异步框架开发，同时也不是标准的 Python 库，对新手并不友好，并且在 Debug 模式下代码执行结果并不直观展示，不利于分析问题。