第 5 章

Selenium WebDriver 基础

本章先带大家熟悉 Selenium WebDriver API 基础内容，读者如果是初学者，则建议反复学习本章内容，因为笔者在后期对 Selenium WebDriver 进行二次封装也是建立在本章基础上进行的。


7min

5.1 Selenium 简介

在学习 Selenium 如何使用之前，读者应该简单了解一下 Selenium 自动化测试工具的原理，并根据自己的需求使用 Selenium 相应的工具去达到 UI 自动化测试的目的。

5.1.1 Selenium 测试准备

要想在 Python 中使用 Selenium，必须先安装 Selenium 包，再下载浏览器 Driver。

1. 安装 Selenium 包

在 Python 中 Selenium 是一个第三方模块包，所以读者需要先使用 pip 命令对其进行安装，命令如下：

```
pip install selenium -i https://pypi.douban.io/simple
```

安装好 Selenium 之后，读者可以使用 pip 命令查看是否安装成功，命令如下：

```
>pip list
Package          Version
----------       ----------
selenium         3.141.0
```

2. 下载浏览器 Driver

通过上述命令安装了 Selenium 之后，读者还是不能使用 Selenium 对浏览器进行操作，因为想对浏览器进行操作必须有对应浏览器的 Driver。浏览器的 Driver 是区分版本的，接下来笔者以 Chrome 浏览器为例，为读者介绍浏览器 Driver 的下载，步骤如下。

1）查看浏览器版本

打开 Chrome 浏览器，单击浏览器右上角“…”→“帮助”→“关于 Google Chrome”按钮，

此处可以看到浏览器版本信息，如图 5-1 所示。

图 5-1　Chrome 版本

2）下载对应版本的 ChromeDriver

例如笔者安装的 Chrome 版本是 102，所以只需选择下载相近版本的 Driver。下载网址如下：

```
http://chromedriver.storage.googleapis.com/index.html
```

进入下载页面后，笔者选择最接近的版本 102.0.5005.61 进行下载，如图 5-2 所示。

图 5-2　ChromeDriver 版本

3）ChromeDriver 配置环境变量

由于 Python 安装时已经配置过环境变量，所以笔者将 ChromeDriver 放到 Python 目录下，相当于添加了环境变量。

4）测试代码

安装完 Selenium 且下载好对应版本的 ChromeDriver 后，读者可以使用最简单的 Selenium 代码打开百度首页。如果代码能通过 Chrome 浏览器打开百度首页，则表示 Selenium 准备成功，代码如下：

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
```

5.1.2　Selenium 工具介绍

笔者使用的是 Selenium 3 版本，Selenium 3 中包含 WebDriver、SeleniumIDE、Grid。每个工具有不同的功能，具体如下。

（1）Selenium WebDriver：提供一套 API，可以通过浏览器 Driver 控制浏览器，从而达到模拟用户操作的目的。WebDriver 是 Selenium 自动化测试中最常用的工具，本书也是针对 WebDriver 进行详细讲解、封装。

(2) SeleniumIDE 是一个图形化工具,对于初学者来讲非常容易上手,但在实际开发过程中一般不会使用 SeleniumIDE,所以笔者后边也不会讲解此工具。

(3) Selenium Grid:使用该工具可以在不同平台和不同机器上分布式运行测试用例。笔者会在后面的章节中做一个简单的介绍。

5.1.3 Selenium WebDriver 原理

WebDriver 采用 Server-Client 设计模式,Server 指的是使用脚本启动的浏览器,Client 指的是我们编写的测试脚本。当读者使用测试脚本启动浏览器后,该浏览器就是一个 Remote Server,它的职责就是监听 Client 发送请求,并做出相应的响应。Client 测试脚本实现的所有操作会以 HTTP 请求的方式发给 Server 端,Server 使用浏览器 Driver 来操作浏览器,操作浏览器的结果 Server 再使用 HTTP 请求返回,如图 5-3 所示。

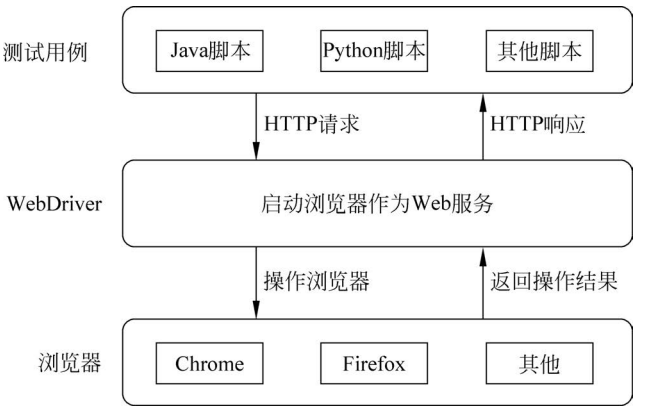

图 5-3 WebDriver 工作流程

5.1.4 Selenium Grid 原理

Grid 的作用就是分布式执行测试用例,Grid 的分布式测试由 Hub 主节点和若干 Node 节点组成,Hub 节点是用来管理 Node 节点的注册和状态信息,当 Hub 节点接收远程客户端代码请求调用时会将请求的命令转发给 Node 节点来执行,如图 5-4 所示。

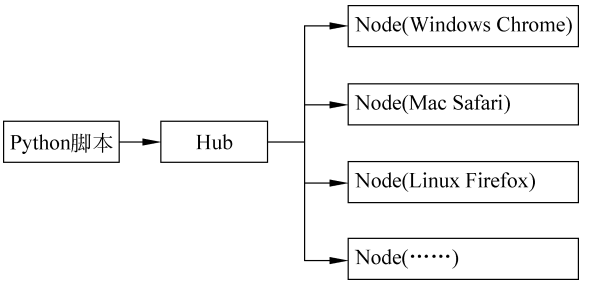

图 5-4 Grid 简介

例如当自动化测试用例较多又需要在 Chrome 和 Firefox 两个浏览器上执行时，一般会部署两台机器分别执行测试用例。有了 Grid 之后只需启动 Hub 节点并注册两个需要的 Node 节点便可以在不同的 Node 节点中同时执行用例，即分布式执行用例。

5.2 WebDriver 浏览器操作

Selenium UI 自动化测试是针对浏览器进行的，所以进行 UI 自动化测试的第 1 步就应该是打开浏览器，然后才能根据自己的需要跳转到某个网站，再去定位某个元素、操作某个元素等。

5.2.1 启动浏览器

启动浏览器的代码很简单，只需实例化一个浏览器 Driver。笔者实例化了一个 Chrome 浏览器 Driver，用于启动 Chrome 浏览器，代码如下：

```
from selenium import webdriver
#打开浏览器
driver = webdriver.Chrome()
```

示例中，虽然只有一行实例化 Chrome 浏览器的代码，但当笔者执行该代码后就会打开一个 Chrome 浏览器。此时的浏览器中没有任何内容，因为笔者并未指定浏览器跳转到哪个具体的网站，如图 5-5 所示。

图 5-5 启动浏览器

通过前面的学习得知，使用 Selenium 操作浏览器需要下载对应浏览器的 Driver，并将其设置到环境变量中才可以正确地操作浏览器。那么如果浏览器的 Driver 版本不能正确地执行代码，则会报什么错误？如果浏览器 Driver 版本正确但没有将 Driver 设置到环境变量中，则又会报什么错误？带着这两个疑问，笔者将进行针对性测试。

1. 浏览器 Driver 版本错误

例如在实际工作中,Chrome 浏览器的版本进行了自动升级,但笔者并没有下载及升级相应的 Chrome Driver,此时执行原有的代码 Selenium 的报错如下:

```
#浏览器 Driver 版本错误
selenium.common.exceptions.SessionNotCreatedException: Message: session not created: This
version of ChromeDriver only supports Chrome version 102
Current browser version is 108.0.5359.72 with binary path
C:\Users\test\AppData\Local\Google\Chrome\Application\chrome.exe
```

当 Chrome Driver 版本不正确时,Selenium 报错为 SessionNotCreatedException。提示目前的 Chrome Driver 只能支持 102 版本的 Chrome 浏览器,当前 Chrome 浏览器的版本是 108,所以读者如果在工作中遇到这种报错,则应该下载对应版本的 Chrome Driver 进行替换。

2. 浏览器 Driver 没有设置环境变量

如果读者在搭建 Selenium 环境时忘记了将 Chrome Driver 配置到环境变量中,则此时执行代码 Selenium 的报错如下:

```
#浏览器 Driver 没有设置环境变量
selenium.common.exceptions.WebDriverException: Message: 'chromedriver' executable needs to be
in PATH. Please see
https://sites.google.com/a/chromium.org/chromedriver/home
```

当 Chrome Driver 没有被配置到环境变量中时,Selenium 报错为 WebDriverException。提示 Chrome Driver 需要被配置到环境变量 PATH 中。

5.2.2 导航到网页

以上面的例子打开浏览器后,浏览器中显示的内容为空白,原因是笔者没有指定跳转到哪个具体的网址,接下来笔者将使用自研项目对页面跳转进行测试。

笔者在本地启动自研的测试项目,地址为 http://localhost:8080/,其中 localhost 表示网站的 IP 地址是本机,8080 表示网站的端口。浏览器页面跳转到指定网站的代码很简单,只需调用 WebDriver 的 get()方法,代码如下:

```
#导航到网页
driver.get('http://localhost:8080/Login')
```

示例中,笔者在 get()方法中传入了自研测试项目的首页地址,执行代码后可以跳转到自研测试项目首页,浏览器的效果如图 5-6 所示。

5.2.3 最大化浏览器

当笔者启动浏览器并跳转到需要测试的网址后发现了一个问题,即浏览器没有在计算机上全屏显示。如果想要浏览器最大化,则只需调用 WebDriver 的 maximize_window()方法,该方法不需要传入任何参数即可实现浏览器最大化,代码如下:

```
#最大化浏览器
driver.maximize_window()
```

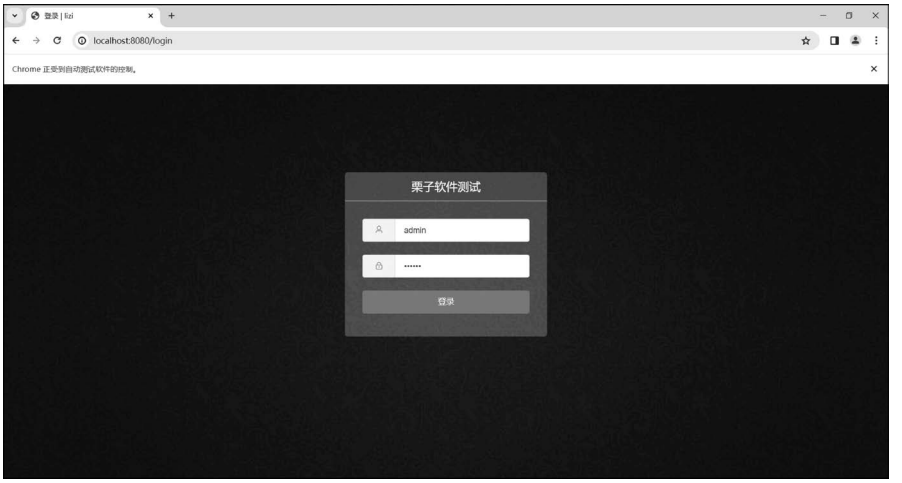

图 5-6　跳转到测试网址

5.2.4　关闭浏览器

假设笔者的自动化测试用例执行完毕，此时就需要关闭脚本打开的浏览器。Selenium中关闭浏览器的方法有两种，一种是使用 WebDriver 的 close()方法，另一种是使用 WebDriver 的 quit()方法。接下来笔者将介绍这两种方法的不同。

1. 关闭当前窗口

假设笔者在自动化测试过程中打开了两个窗口，如图 5-7 所示。

图 5-7　浏览器打开两个窗口

此时当前窗口是第 2 个窗口。当笔者只想关闭第 2 个窗口时，可以调用 WebDriver 的 close()方法，该方法不需要传任何参数，代码如下：

```
#关闭当前窗口
driver.close()
```

2. 退出驱动并关闭所有窗口

在上述例子中，当笔者想关闭所有窗口时，需要调用 WebDriver 的 quit()方法，也不需要传任何参数即可关闭所有窗口，当关闭所有窗口同时 Driver 也就跟着退出了，代码如下：

```
#退出驱动,关闭所有窗口
driver.quit()
```

5.2.5 总结

以上代码虽然没有涉及使用 Selenium 操作元素,但每个测试用例在开始和结束时都会执行这些代码,所以笔者对代码进行了总结,后期还会对这些代码进行二次封装,代码如下:

```
//第 5 章/new_selenium/my_sel_0.py
from selenium import webdriver
#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#关闭所有浏览器
driver.quit()
```

5.3 WebDriver 元素定位

通过前面的学习,读者已经可以进入需要测试的网站。接下来读者可以思考一下手工测试会做哪些操作,然后根据手工测试思路进行自动化测试学习。以登录为例,手工测试会先找到用户名和密码输入框,输入用户名和密码,然后单击登录。自动化测试的步骤跟手工测试一样,读者也需要先定位用户名、密码、登录按钮这 3 个元素。

5.3.1 开发者工具

定位元素的工具有很多,前端开发经常会打开浏览器的开发者工具进行定位。打开开发者工具的方式有两种,一种是按 F12 快捷键;另一种是在网页上右击,选择“检查”。开发者工具打开后如图 5-8 所示,右边有代码的部分就是开发者工具。

在图 5-8 中,读者在开发者工具中单击最左边的箭头按钮,然后单击用户名输入框,这样就可以定位到用户名输入框的前端代码,代码如下:

```
< input class = " el - input _ _ inner" id = " username" type = " text" autocomplete = " off"
placeholder = "username">
```

示例中,用户名输入框是一个 input 标签,并且 type 属性为 text。input 标签的属性包含 placeholder 属性、class 属性、id 属性等,这些属性在实际工作中都可以用来帮助定位元素。

5.3.2 id 属性定位

以用户名输入框为例,笔者首先尝试使用 id 属性定位元素,原因是 id 属性在开发过程

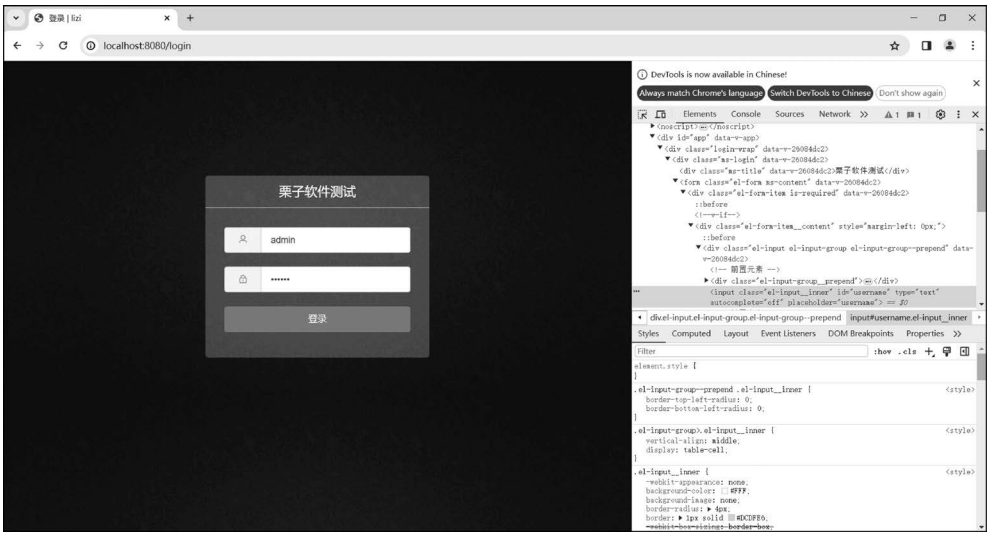

图 5-8　开发者工具

中是唯一的，所以使用 id 属性定位元素一定是唯一的。笔者再次观察用户名输入框的 id 属性，代码如下：

```
< input class = " el - input _ _ inner" id = " username" type = " text" autocomplete = " off"
placeholder = "username">
```

示例中，用户名输入框 id 属性值为 username，使用 id 定位元素只需调用 WebDriver 的 find_element_by_id()方法，参数需要传入用户名输入框 id 属性的值，代码如下：

```
//第 5 章/new_selenium/my_sel_1.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#id 定位
element = driver.find_element_by_id("username")
print(type(element))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中，笔者使用 WebDriver 的 find_element_by_id()方法定位用户名输入框，并将方法返回值赋值给 element 变量，接着打印 element 变量的类型。从执行结果可以看出，element 变量的类型是 WebElement，表示定位成功。

现在读者可以使用 id 属性定位到元素,那么读者应该考虑一个问题,如果定位的元素不存在,则代码会有什么提示呢? 接下来笔者使用 find_element_by_id()方法传入错误的参数值,演示元素不存在时代码的报错,代码如下:

```
//第 5 章/new_selenium/my_sel_1.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#id 定位
element = driver.find_element_by_id("username2")
print(type(element))
print(element)

#执行结果
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"css selector","selector":"[id = "username2"]"}
```

示例中,笔者调用 find_element_by_id()方法时将参数故意错误地写成 username2,由于在前端代码中 id=username,所以 id=username2 元素是不存在的。此时执行代码读者可以看到执行结果中报错 NoSuchElementException,表示元素不存在,并且指明了 id="username2"部分发生了错误,这些报错信息足以让笔者找到问题所在。

5.3.3 name 属性定位

为了演示 name 属性定位元素,笔者特意在密码的 input 标签中添加了 name 属性,并将 name 属性的值设置为 password,前端代码如下:

```
<input class = "el - input__inner" name = "password" type = "password" autocomplete = "off"
placeholder = "password">
```

相信读者根据前面 id 属性定位所学知识,可以猜到 name 属性定位元素只需调用不同的方法。这里笔者就调用 WebDriver 的 find_element_by_name()方法,传入元素 name 属性值实现 name 属性元素定位,代码如下:

```
//第 5 章/new_selenium/my_sel_2.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
```

```
driver.maximize_window()
#name 定位
element = driver.find_element_by_name("password")
print(type(element))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中，笔者调用 WebDriver 的 find_element_by_name()方法并传入了密码输入框 name 属性值 password。从执行结果可以看出，密码输入框也是一个 WebElement，表示定位成功。

5.3.4　class 属性定位

HTML 中标签的 class 属性对应的是标签的样式，由于样式并非 UI 自动化测试的重点内容，所以笔者在前面的章节中并没有进行重点介绍，读者可以简单地将样式理解为元素长什么样子、什么颜色等。

以登录按钮为例，笔者使用开发者工具查看登录按钮的前端代码，代码如下：

```
<div class = "login - btn" data - v - 26084dc2 = "">
<button class = "el - button el - button -- primary" type = "button" data - v - 26084dc2 = "">
<span >登录</span ></button >
</div >
```

在前端代码中，button 标签有 class 属性，button 标签外层的 div 容器也有 class 属性，那么应该选择哪个 class 属性值进行定位呢？笔者这里选择的是 div 标签的 class 属性值，因为 div 标签的 class 属性值更加明确地表示了是一个登录按钮，定位代码如下：

```
//第 5 章/new_selenium/my_sel_3.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#class 定位
element = driver.find_element_by_class_name("login - btn")
print(type(element))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中，笔者使用 WebDriver 的 find_element_by_class()方法，传入 div 标签的 class 属性值 login-btn，从执行结果中可以看出 class 属性定位成功。由于 div 标签相当于容器，

它包含了 button 标签,所以定位到 div 标签也就相当于定位到了 button 标签。读者可以根据 HTML 结构尝试理解标签之间的关系,这样有助于对后面元素定位的学习。

5.3.5 CSS 选择器定位

CSS(Cascading Style Sheets)是一种语言,该语言用来描述 HTML 和 XML 的元素显示样式。CSS 语言中有 CSS 选择器,在 Selenium 中也可以使用这些选择器来对元素进行定位操作。笔者在此简单地总结了 CSS 选择器的一些基本用法,见表 5-1。关于 CSS 选择器的具体细节读者可以翻阅相关书籍进行详细学习,这里不进行过多讲解。

表 5-1 CSS 选择器

选 择 器	格 式	备 注
点(.)	.xx	class 选择器
井号(#)	#xx	id 选择器
[attribute=value]	[name=password]	属性选择器

表 5-1 中笔者只介绍了 3 个选择器,即 class 选择器、id 选择器和属性选择器,使用上面这 3 种选择器就可以把前面学到的 id 属性定位、name 属性定位和 class 属性定位使用 CSS 选择器进行实现。

使用 CSS 选择器定位需要用到 WebDriver 的 find_element_by_css_selector()方法,该方法的参数可以使用 id 选择器、class 选择器、属性选择器。笔者还是以登录为例,分别用这 3 种不同的选择器定位用户名输入框、密码输入框和登录按钮,代码如下:

```
//第 5 章/new_selenium/my_sel_4.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#用户名定位:id 选择器
element_username = driver.find_element_by_css_selector("#username")
print(type(element_username))
#密码定位:属性选择器
element_password = driver.find_element_by_css_selector("[name = 'password']")
print(type(element_password))
#登录按钮定位:class 选择器
element_login_btn = driver.find_element_by_css_selector(".login-btn")
print(type(element_login_btn))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中，笔者统一使用 find_element_by_css_selector()方法定位登录页面的 3 个元素，定位 3 个元素时的区别在于传入的参数不同。定位用户名输入框时使用“＃id 属性值”格式，表示使用的是 CSS 的 id 选择器；定位密码输入框时在方括号中输入标签“name 属性＝属性值”，表示使用的是 CSS 的属性选择器；定位登录按钮时使用“. class 属性值”格式，表示使用的是 CSS 的 class 选择器。从执行结果可以看出，使用 CSS 选择器定位 3 个元素均可以成功。

5.3.6　link text 定位

link text 定位从字面意思上就可以猜测出，这是使用 a 标签的文本来定位的。笔者以百度首页左上角“新闻”链接为例，先按 F12 键打开开发者工具，然后抓取前端代码，代码如下：

```
<a href = "http://news.baidu.com" target = "_blank" class = "mnav c - font - normal c - color - t">
新闻</a>
```

从上述代码可以看出，a 标签的文本是“新闻”二字，只要读者稍加思索就可以想到，应该调用 WebDriver 的 find_element_by_link_text()方法定位该元素，代码如下：

```
//第 5 章/new_selenium/my_sel_5.py
from selenium import webdriver

＃打开浏览器
driver = webdriver.Chrome()
＃导航到网页
driver.get('https://www.baidu.com')
＃最大化浏览器
driver.maximize_window()
＃link text 定位
element = driver.find_element_by_link_text("新闻")
print(type(element))

＃执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中，读者需要注意两个问题。第 1 个问题，笔者使用百度首页作为测试页面，所以在调用 WebDriver 的 get()方法时，需要传入的参数应该改为百度首页地址；第 2 个问题，使用 WebDriver 的 find_element_by_link_text()方法定位 a 标签元素时，需要传入 a 标签的文字“新闻”。从执行结果可以看出定位元素成功。

5.3.7　partial link text 定位

partial link text 定位与 link text 定位方式基本相同，partial link text 的英文意思是使用部分文本对 a 标签元素进行定位，从字面意思可以看出，定位 a 标签元素时只需传入标签部分文字便可以定位成功。定位时使用 WebDriver 的 find_element_by_partial_link_text()

方法,代码如下:

```
//第 5 章/new_selenium/my_sel_6.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('https://www.baidu.com')
#最大化浏览器
driver.maximize_window()
#partial link text 定位
element = driver.find_element_by_partial_link_text("闻")
print(type(element))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中,笔者还是使用百度首页进行测试,但在调用 find_element_by_partial_link_text()方法时只传入了新闻的“闻”字,从执行结果可以看出,只传入 a 标签的部分内容也是可以定位成功的。

5.3.8 tag name 定位

tag name 定位的意思就是通过标签的名字进行定位,但由于一般在同一个页面中同名标签会有多个,所以使用标签名定位并不一定能定位到唯一的标签。笔者还是以登录页面为例,演示如何定位 input 标签元素,代码如下:

```
//第 5 章/new_selenium/my_sel_7.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#link text 定位
element = driver.find_element_by_tag_name("input")
print(type(element))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中,笔者调用 WebDriver 的 find_element_by_tag_name()方法定位 input 标签元素,所以传入的参数就是 input。从执行结果可以看出定位成功,但读者需要注意的是,当页面包含多个 input 标签时定位到的是第 1 个 input 标签。例如登录页面包含用户名和密码

两个输入框，使用 find_element_by_tag_name()方法定位到的是用户名输入框。

5.3.9 xpath 表达式定位

xpath 是 XML path 的简称，它是一种用来确定 XML 文档中查找信息的语言，HTML 可以看作 XML 的一种实现。xpath 定位是笔者后续章节中一直使用的定位方式，读者需要重点关注。对于有 id 属性或 name 属性等唯一属性的元素，可以使用简单的 xpath 进行定位，如果元素没有唯一属性，则可以使用 xpath 轴进行定位。

1. xpath 定位

对于已经有唯一属性的元素来讲 xpath 定位非常简单，只需找到标签元素并使用唯一属性。笔者总结了 xpath 简单语法供读者参考，见表 5-2。

表 5-2 xpath 简单语法

语 法	格 式	备 注
/	/html/body/div	从根节点开始选取
//	//input	从 input 标签开始选取
@	//input[@id='username']	选取 input 标签的 id 属性
text()	//span[text()='登录']	文本定位
contains()	//div[contains(@class,'login-btn')]	模糊定位

接下来笔者以登录页面为例，使用 xpath 定位登录页面上的所有元素，代码如下：

```
//第 5 章/new_selenium/my_sel_8.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#用户名定位
element_username = driver.find_element_by_xpath("//input[@id='username']")
print(type(element_username))
#密码定位
element_password = driver.find_element_by_xpath("//input[@name='password']")
print(type(element_password))
#登录按钮定位
element_login_btn_1 = driver.find_element_by_xpath("//div[@class='login-btn']")
element_login_btn_2 = driver.find_element_by_xpath("//span[text()='登录']")
element_login_btn_3 = driver.find_element_by_xpath("//div[contains(@class, 'login-btn')]")
print(type(element_login_btn_1))
print(type(element_login_btn_2))
print(type(element_login_btn_3))

#执行结果
```

```
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中,笔者调用 WebDriver 的 find_element_by_xpath()方法定位元素,由于不同元素有不同的唯一属性,所以笔者分别传入了 id 属性、name 属性、class 属性进行定位。对于登录按钮笔者特意使用了多种属性进行定位,包括 class 属性、text()文本、contains()模糊定位,目的是让读者在使用 xpath 定位元素时有更多的选择,以便更容易地进行定位。


19min

2. xpath 轴定位

除了基础定位方法外 xpath 还有轴定位方法,即当所要定位的标签元素没有唯一的属性可以定位,但亲戚节点很好定位时,读者可以使用 xpath 轴先定位元素的亲戚节点,然后通过亲戚节点找到想要的元素。笔者总结了 xpath 轴定位的语法,见表 5-3。

表 5-3 xpath 轴定位的语法

轴	备 注
ancestor	选取当前节点的所有先辈(父、祖父等)
ancestor-or-self	选取当前节点的所有先辈(父、祖父等)及当前节点本身
attribute	选取当前节点的所有属性
child	选取当前节点的所有子元素
descendant	选取当前节点的所有后代元素(子、孙等)
descendant-or-self	选取当前节点的所有后代元素(子、孙等)及当前节点本身
following	选取文档中当前节点的结束标签之后的所有节点
namespace	选取当前节点的所有命名空间节点
parent	选取当前节点的父节点
preceding	选取文档中当前节点的开始标签之前的所有节点
preceding-sibling	选取当前节点之前的所有同级节点
self	选取当前节点

其实读者只要明白轴语法英文单词的意思,大概就会猜到轴是用来做什么的了,需要注意的细节就是轴语法如何使用。由于笔者将在后面小节中介绍一个 Firefox 浏览器 xpath 定位插件,有了此插件读者可以很方便地获取标签元素的 xpath,所以笔者在此只举一些简单的例子。这些例子没有实际意义,但足以让读者理解 xpath 轴语法应该如何应用,具体如下。

(1) 找到 input 节点的祖先节点 form: //input[@id='username']/ancestor::form。

(2) 找到 input 节点的父亲节点 div: //input[@id='username']/parent::div。

(3) 找到 input 节点的兄弟节点 div: //input[@id='username']/following-sibling::div。

(4) 找到 input 节点的儿子节点 div: //input[@id='username']/child::div。

(5) 找到 input 节点的后代节点 div: //input[@id='username']/descendant::div。

5.3.10 By模块定位

上述笔者所讲述的单个元素的定位方法还有另外一种写法，即通过By模块声明定位的方法。当读者使用By声明定位方法时，需要调用WebDriver的find_element()方法，该方法需要传入两个参数，第1个参数是By模块操作的属性，第2个参数是属性值。By模块定位方法见表5-4。

表5-4 By模块定位方法

方 法	备 注
find_element(By.ID, "")	id定位
find_element(By.NAME, "")	name定位
find_element(By.CLASS_NAME, "")	class name定位
find_element(By.CSS_SELECTOR, "")	css选择器定位
find_element(By.LINK_TEXT, "")	链接文本定位
find_element(By.PARTIAL_LINK_TEXT, "")	链接部分文本定位
find_element(By.TAG_NAME, "")	标签名定位
find_element(By.XPATH, "")	xpath定位

虽然笔者在后面的章节中不常使用By模块进行元素定位，但笔者还是以登录页面为例，以By模块的方式实现页面元素的定位，代码如下：

```
//第5章/new_selenium/my_sel_9.py
from selenium import webdriver
from selenium.webdriver.common.by import By

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#用户名定位
element_username = driver.find_element(By.ID, "username")
print(type(element_username))
#密码定位
element_password = driver.find_element(By.NAME, "password")
print(type(element_password))
#登录按钮定位
element_login_btn = driver.find_element(By.CLASS_NAME, "login-btn")
print(type(element_login_btn))

#执行结果
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
<class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中,笔者使用的是 find_element()方法。当笔者想通过 id 定位用户名输入框时,向第 1 个参数传入 By.ID;向第 2 个参数传入用户名输入框 id 属性的值,其他元素定位以此类推。

5.3.11 定位多个元素

除了需要定位单个元素之外,读者在工作中还可能需要定位多个元素,定位多个元素的方法是 find_elements_by_id()等。定位单个元素和定位多个元素的区别在于定位单个元素使用的是以 find_element 开头的方法,而定位多个元素使用的是以 find_elements 开头的方法。具体见表 5-5。

表 5-5 多个元素定位

方 法	备 注
find_elements_by_id()	id 定位
find_elements_by_name()	name 定位
find_elements_by_class_name()	class 属性定位
find_elements_by_css_selector()	css 选择器定位
find_elements_by_link_text()	链接文本定位
find_elements_by_partial_link_text()	链接部分文本定位
find_elements_by_tag_name()	标签名定位
find_elements_by_xpath()	xpath 定位

在登录页面中包含两个 input 标签元素,如果读者想定位这两个 input 标签,则可以使用 WebDriver 的 find_elements_by_tag_name()方法,代码如下:

```
//第 5 章/new_selenium/my_sel_10.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#定位多个元素
elements = driver.find_elements_by_tag_name("input")
print(elements)

#执行结果
[<selenium.webdriver.remote.webelement.WebElement (session="41ec4d5809f71e41a0bb57538a68cabd",
element="519ACF5ED6582097A49DEC45664A4F24_element_2")>, <selenium.webdriver.
remote.webelement.WebElement (session="41ec4d5809f71e41a0bb57538a68cabd", element=
"519ACF5ED6582097A49DEC45664A4F24_element_3")>]
```

示例中,笔者使用 find_elements_by_tag_name()方法并传入 input 作为参数,意思是定

位到页面上所有的 input 标签元素。从执行结果可以看出，定位到了两个 input 标签。

如果读者想要操作每个 input 标签元素，则可以使用 for 循环进行遍历，代码如下：

```
elements = driver.find_elements_by_tag_name("input")
for element in elements:
    print(element)
```

如果读者仅需要操作某个 input 标签元素，则可以通过下标获取指定元素，前提是读者已经知道 input 标签在 HTML 文档中的顺序。例如笔者想要操作用户名输入框，则需要执行的元素的下标为 0，代码如下：

```
element = driver.find_elements_by_tag_name("input")[0]
print(element)
```

5.3.12　XPath 插件

在前后端分离的开发过程中，前端开发工程师不可能在每个标签中都添加 id 或 name 之类的唯一属性，所以在实际测试工作中，笔者一般统一使用 xpath 方式进行定位。由于 xpath 定位有一定难度，逐级书写会影响测试进度，所以笔者将介绍一款 Firefox 浏览器插件来解决这一问题。读者可以打开附件组件管理器，搜索 Ruto-XPath Finder 然后进行安装，如图 5-9 所示。

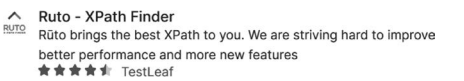

图 5-9　Ruto-XPath 插件

安装好 Ruto 插件后，Firefox 浏览器的右上角就会出现 Ruto 的图标。开启 Ruto 后单击想要进行 xpath 定位的标签元素，就可以得到元素的多个 xpath，读者可以选择其中比较容易理解的 xpath 进行使用，如图 5-10 所示。

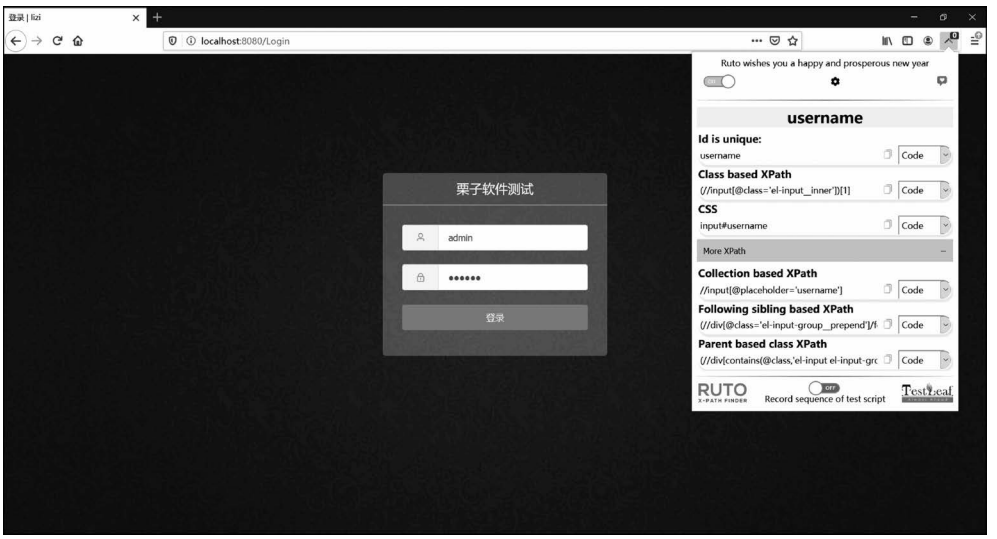

图 5-10　Ruto-XPath 插件的使用

还是以登录页面为例,笔者使用 Ruto 插件获取不同的 xpath,看一看是否能够成功定位元素,代码如下:

```
//第 5 章/new_selenium/my_sel_11.py
from selenium import webdriver

#打开浏览器
driver = webdriver.Chrome()
#导航到网页
driver.get('http://localhost:8080/Login')
#最大化浏览器
driver.maximize_window()
#用户名定位
element_username = driver.find_element_by_xpath("//input[@placeholder = 'username']")
print(type(element_username))
#密码定位
element_password = driver.find_element_by_xpath("//input[@placeholder = 'password']")
print(type(element_password))
#登录按钮定位
element_login_btn = driver.find_element_by_xpath("//div[@class = 'login - btn']//button[1]")
print(type(element_login_btn))

#执行结果
< class 'selenium.webdriver.remote.webelement.WebElement'>
< class 'selenium.webdriver.remote.webelement.WebElement'>
< class 'selenium.webdriver.remote.webelement.WebElement'>
```

示例中,3 个元素的 xpath 都是由 Ruto 插件获取的。对于用户名输入框和密码输入框,Ruto 插件使用 placeholder 属性进行定位;登录按钮定位更加精确,先使用 class 属性定位到登录按钮的父 div 标签,再在父 div 标签中找到第 1 个 button 标签。从执行结果可以看出,Ruto 插件获取的 xpath 完全可以正常使用,不会报任何错误。

5.4 WebDriver 基本操作

学习完标签元素定位后,接下来的工作就是去操作元素了。本节中,笔者会使用第 4 章 HTML 基础中的内容进行演示,后面的章节再过渡到笔者自己开发的测试项目上进行二次演示,这样既能兼顾学习的完整性,同时又能达到更好的学习效果。

5.4.1 输入操作

在 HTML 基础中笔者写了一个简单的登录页面,包括用户名输入框、密码输入框和一个登录按钮,代码如下:

```
用户名:< input type = "text" name = "username">< br >
密码:< input type = "password" name = "password">< br >
< button >登录</button >
```

1. 输入操作

要对输入框进行输入操作，需要调用 WebDriver 的定位方法获取 Webelement，然后再调用 WebElement 的 send_keys()方法，代码如下：

```
//第 5 章/new_selenium/my_sel_html_1.py
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test2.html')
driver.maximize_window()
#输入操作
driver.find_element_by_xpath("//input[@name = 'username']").send_keys('admin')
driver.find_element_by_xpath("//input[@name = 'password']").send_keys('123456')
```

示例中，笔者调用 send_keys()方法并传入用户名和密码，用户名和密码输入成功，如图 5-11 所示。

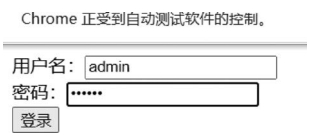

图 5-11　输入结果

2. 清空操作

当用户名和密码为空时，以上输入操作没有任何问题，但当用户名和密码有默认值时，读者会发现脚本输入的内容会被追加到原内容的后边，这样就会造成登录失败，那么如何解决这一问题呢？

笔者对登录的 HTML 代码进行了简单修改，给用户名和密码标签增加了 value 属性，这样用户名和密码就有了默认值，代码如下：

```
用户名:< input type = "text" name = "username" value = "user1">< br >
密码:< input type = "password" name = "password" value = "123456">< br >
< button >登录</button >
```

为了解决输入框默认值问题，笔者在定位完元素之后，首先调用 WebElement 的 clear()方法，然后调用该元素的 send_keys()方法，这样就能先清除默认值，然后输入新的用户名和密码，代码如下：

```
//第 5 章/new_selenium/my_sel_html_2.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test2.html')
driver.maximize_window()
time.sleep(2)
#输入操作
element_username = driver.find_element_by_xpath("//input[@name = 'username']")
element_username.clear()
element_username.send_keys('admin')
element_password = driver.find_element_by_xpath("//input[@name = 'password']")
element_password.clear()
element_password.send_keys('123456')
```

示例中,笔者先定位元素,然后依次调用 WebElement 的 clear()方法和 send_keys()方法,结果达到了先清空后输入的目的。另外,笔者在代码中使用了 time 模块的 sleep()方法,该方法是让代码等待几秒,其目的是让读者看清清空操作的过程。

5.4.2 单击操作

单击操作不仅指单击按钮,单选框、复选框也是通过单击进行操作的,接下来将一一讲解不同标签元素的单击操作。

1. 按钮单击

还是以登录按钮为例,单击“登录”按钮需要调用 WebElement 的 click()方法,代码如下:

```
//第 5 章/new_selenium/my_sel_html_3.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test2.html')
driver.maximize_window()
time.sleep(2)
#输入操作
element_username = driver.find_element_by_xpath("//input[@name = 'username']")
element_username.clear()
element_username.send_keys('admin')
element_password = driver.find_element_by_xpath("//input[@name = 'password']")
element_password.clear()
element_password.send_keys('123456')
#单击操作
element_button = driver.find_element_by_xpath("//button[text() = '登录']")
element_button.click()
```

示例中,笔者使用文本属性定位登录按钮,然后调用 click()方法对其进行单击操作。

2. 单选框单击

在操作单选框之前,先回忆下单选按钮的前端代码,代码如下:

```
<!-- 单选 -->
< input type = "radio" name = "sex" value = "male" checked = "checked">男< br >
< input type = "radio" name = "sex" value = "female">女< br >
```

示例中,单选框默认选中的选项是“男”,如果笔者想单击选择“女”选项,则需要调用 WebElement 的 click()方法,代码如下:

```
//第 5 章/new_selenium/my_sel_html_4.py
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test4.html')
```

```
driver.maximize_window()
#选择女
driver.find_element_by_xpath("//input[@value='female']").click()
```

示例中，笔者通过观察得出 input 标签元素的 value 属性可以区分两个单选框，所以笔者使用 value 属性进行定位并进行单击操作，执行结果如图 5-12 所示。

图 5-12　单选结果

3. 复选框单击

在操作复选框之前，先回忆下多选按钮的前端代码，代码如下：

```
<!-- 多选 -->
<input type="checkbox" name="vehicle" value="Bike">自行车<br>
<input type="checkbox" name="vehicle" value="Car" checked="checked">汽车
```

示例中，复选框默认选中的选项是“汽车”。接下来笔者想在选择时先判断选项是否已被选中，如果没有被选中，则进行单击操作，如果已被选中，则打印该选项已经被选中无须再选择，代码如下：

```
//第5章/new_selenium/my_sel_html_5.py
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a-lizi-workspace\\lizitest-1\\new_html\\test4.html')
driver.maximize_window()
#定位所有复选框
elements = driver.find_elements_by_xpath("//input[@name='vehicle']")
#遍历并选择
for element in elements:
    if element.is_selected():
        print("{} 选项已经被选中无须再选择!".format(element.get_attribute("value")))
        continue
    else:
        element.click()

#执行结果
Car 选项已经被选中无须再选择!
```

示例中，笔者调用 WebDriver 的 find_elements_by_xpath()方法获取所有的复选框，然后使用 for 循环对所有复选框进行遍历。调用 WebElement 的 is_selected()方法判断当前复选框是否被选中，如果已被选中，则打印提示并继续循环；如果没被选中，则调用 WebElement 的 click()方法选中该复选框。执行结果如图 5-13 所示。

图 5-13　多选结果

值得注意的是，如果想取消复选框，则只需调用 WebElement 的 click()方法对其进行单击，希望读者不要将事情复杂化，记住复选框都是单击操作即可。

5.4.3 下拉列表操作

下拉列表操作比较特殊,操作过程中涉及一个 Select 类。读者需要先实例化 Select 类,然后调用 select_by_visible_text()方法传入选择项的文本进行选择。

首先回忆下下拉列表 HTML 代码,代码如下:

```
<select name = "city">
    <option value = "beijing" >北京</option>
    <option value = "shanghai" >上海</option>
    <option value = "guangzhou" selected >广州</option>
    <option value = "shenzhen" >深圳</option>
</select>
```

代码中,下拉列表默认被选中的选项是“广州”,笔者想将被选中的选项变为“北京”,代码如下:

```
//第 5 章/new_selenium/my_sel_html_6.py
from selenium import webdriver
from selenium.webdriver.support.select import Select

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test5.html')
driver.maximize_window()
#定位元素
select_element = driver.find_element_by_xpath("//select[@name = 'city']")
#实例化 Select 对象
select = Select(select_element)
#选择北京
select.select_by_visible_text("北京")
```

示例中,笔者导入了 Select 类,当时实例化 Select 时传入的是 select 标签元素的 xpath,实例化之后就可以调用 Select 类的 select_by_visible_text()方法,传入“北京”进行选中操作。执行结果如图 5-14 所示。

图 5-14 下拉选中结果

当然,选中下拉列表中的选项有多种方法,笔者在这里做了简单总结,读者可以在需要时选择不同的方法,见表 5-6。

表 5-6 下拉选择方法

方 法	备 注
select_by_visible_text()	通过文本选中
select_by_value()	通过 value 属性选中
select_by_index()	通过下标选中

5.4.4 文件上传操作

文件上传操作在前端开发中一般使用 input 标签并将其 type 属性设置为 file。当用户选择需要上传的文件时,前端开发人员会按照需求对文件进行相应处理。接下来笔者将介绍前端在上传文件时如何处理;WebDriver 如何实现文件上传;第三方工具如何实现文件上传,目的是让读者更加深入地了解文件上传的相关内容。

1. 文件上传实现

以前后端分离系统为例,笔者参考 ElementUI 中组件的属性,总结出前端开发人员在实现文件上传时一般会考虑以下几点,具体如下。

1) 自动上传

ElementUI 中 el-upload 标签有一个 auto-upload 属性,一般单个文件上传时会将该属性设置为 True,表示自动上传文件,设置该属性后就不需要用户再单击一个上传按钮进行上传了。

2) 判断文件类型和大小

在上传文件之前,前端开发人员可以调用 beforeUpload()方法,在该方法中开发人员可以根据需求判断文件的类型或者文件的大小等。如果文件类型或文件大小不满足要求,则给出错误提示。

3) 调用后端接口

当上传文件合法时,前端开发人员可以调用 uploadHttpRequest()方法,在该方法中正式调用后端的上传接口进行文件上传。

2. 文件上传操作

明白了文件上传的原理后,读者就可以调用 WebElement 的 send_keys()方法直接输入文件路径以实现上传功能了。

首先,回忆下简单的 HTML 文件上传代码,代码如下:

```
<input type = "file" name = "upload_input">
```

示例中,input 标签元素有 id 属性,所以笔者只需使用 id 属性进行 xpath 定位,然后在 send_keys()方法中输入文件路径便可以实现文件上传,代码如下:

```
//第 5 章/new_selenium/my_sel_html_7.py
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test6.html')
driver.maximize_window()
#文件上传
driver.find_element_by_xpath("//input[@name = 'file_upload']").send_keys("D://测试.txt")
```

示例中，笔者使用 D 盘中名为“测试. txt”的文件进行上传，上传结果如图 5-15 所示。

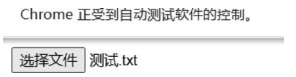

图 5-15　文件上传结果

3. 第三方工具文件上传操作

当然，如果读者找不到 input 标签，无法对其进行输入操作，就需要使用第三方工具进行自动化文件上传操作。笔者这里讲解 Selenium 如何结合 Pywinauto 完成文件上传操作。

1) HTML 标签和 Windows 窗口

在使用第三方工具 Pywinauto 之前，读者应该弄明白什么是 HTML 网页，以及什么是系统窗口，如图 5-15 所示，“选择文件”按钮是 HTML 标签，Selenium 可以操作该按钮，单击“选择文件”按钮后弹出的是系统窗口，Selenium 不可以操作弹出的系统窗口，只能使用第三方工具进行操作，如图 5-16 所示。

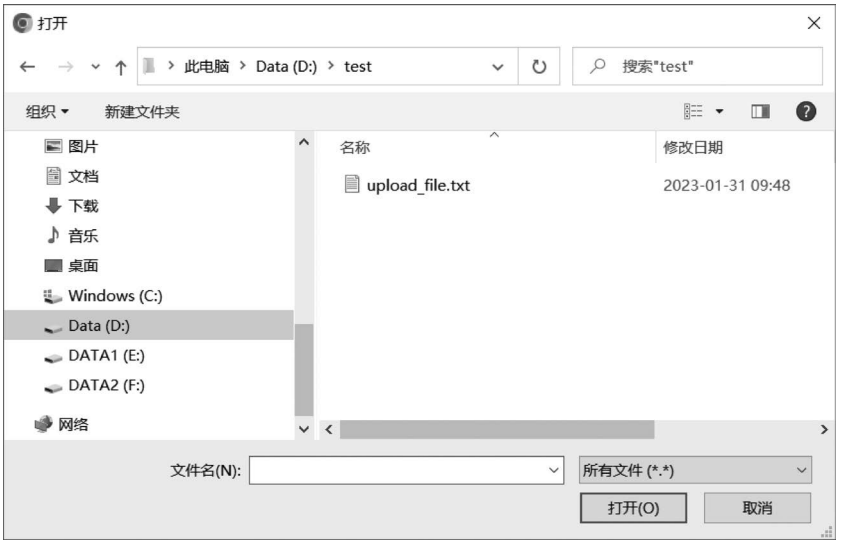

图 5-16　文件选择系统弹窗

2) Pywinauto 安装

安装 Pywinauto 非常简单，使用 pip 命令进行安装即可，代码如下：

```
pip install pywinauto -ihttps://pypi.doubanio.com/simple/
```

3) Selenium＋Pywinauto 实现文件上传

Selenium 和 Pywinauto 工具结合使用时，Selenium 只能操作 HTML 标签，即只能执行到单击“选择文件”按钮，而单击按钮打开的 Windows 窗口则由 Pywinauto 工具操作。

(1) Selenium 单击 input 标签报错。

笔者直接调用 WebElement 的 click()方法单击 input 标签元素时系统会报错，提示如下：

```
#文件上传
driver.find_element_by_xpath("//input[@name = 'file_upload']").click()

#执行结果
selenium.common.exceptions.InvalidArgumentException: Message: invalid argument
```

如果想解决以上报错问题，则需要使用 Selenium 中的 ActionChains 动作链类，ActionChains 实例化该类后可以模拟移动、单击、双击等不同的鼠标动作。

(2) ActionChains 操作。

笔者将单击 input 标签元素代码修改为先定位 input 标签元素，然后通过 Driver 实例化 ActionChains 动作链，最后调用 ActionChains 的 click()方法单击 input 标签元素，代码如下：

```
//第 5 章/new_selenium/my_sel_html_8.py
from selenium import webdriver
from selenium.webdriver import ActionChains

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test6.html')
driver.maximize_window()
#单击选择文件按钮
file_input = driver.find_element_by_xpath("//input[@name = 'file_upload']")
action = ActionChains(driver)
action.click(file_input).perform()
```

示例中，笔者除调用了 ActionChains 的 click()方法外，还调用了 perform()方法。该方法是动作链执行方法，如果不写 perform()方法，则 ActionChains 动作链是不会被执行的。

(3) Pywinauto 工具操作。

打开文件并选择系统弹窗后，接下来的工作就应该使用 Pywinauto 工具来完成了。实例化 Pywinauto 工具后可以通过 Windows 窗口的 title 找到该窗口，然后打开文件所在路径→输入文件名→单击“打开”按钮，即可完成文件上传操作，代码如下：

```
//第 5 章/new_selenium/my_sel_html_8.py
import time
import pywinauto
from pywinauto.keyboard import send_keys

#Pywinauto
app = pywinauto.Desktop()
#根据 title 找到弹出窗口
dialog = app['打开']
#在网址栏输入文件路径
dialog.window(found_index = 0, title_re = ". * 地址. * ").click()
send_keys("D:/test")          #在网址栏输入文件的路径
send_keys("{VK_RETURN}")      #按 Enter 键
time.sleep(2)
```

```
#输入文件名
dialog["Edit"].type_keys("upload_file.txt")
#单击打开按钮
dialog["Button"].click()
```

示例中,笔者已经在代码上标明了注释信息,这部分不是本书的重点内容,所以笔者不再做具体的解释,如果读者需要使用 Pywinauto 工具进行文件上传操作,则只需复制代码并更换上传文件的路径和文件名。

5.4.5 ActionChains 操作

既然在文件上传操作中提到了 ActionChains 类,接下来笔者就讲解 ActionChains 类的相关应用。ActionChains 叫作动作链,当读者使用 ActionChains 类的方法时该方法不会被立即执行,而是按照顺序存放在一个队列中,只有当读者调用 perform()方法时,队列中的方法才会依次被执行。

还是以登录为例,笔者使用 ActionChains 类重新编写元素的定位和操作,代码如下:

```
//第 5 章/new_selenium/my_sel_html_9.py
from selenium import webdriver
from selenium.webdriver import ActionChains

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test2.html')
driver.maximize_window()
#找到元素
username = driver.find_element_by_xpath("//input[@name = 'username']")
password = driver.find_element_by_xpath("//input[@name = 'password']")
login = driver.find_element_by_xpath("//button[text() = '登录']")
#实例化动作链
action = ActionChains(driver)
#编写动作
action.send_keys_to_element(username, 'admin')
action.send_keys_to_element(password, '123456')
action.click(login)
#执行动作
action.perform()
```

示例中,笔者先定位出 3 个需要操作的标签元素,然后实例化 ActionChains 类,并使用 action 对象对 3 个标签元素进行动作编写,最后执行 ActionChains 类的 perform()方法执行所有动作。

5.4.6 悬停操作

悬停的意思就是将鼠标停放在某个元素上。例如将鼠标悬停在百度的首页的右上角的“设置”上,此时就会出现多个设置功能选项,如图 5-17 所示。对这些功能选项直接进行定

位操作时会报错，所以需要先将鼠标悬停在“设置”位置，待设置功能选项出现后再进行操作。

图 5-17　百度设置

接下来笔者将演示直接操作设置功能选项和鼠标悬停后操作设置功能选项，读者可以从中理解鼠标悬停的作用。

1．直接操作

按照前面学习过的知识，如果要操作某个元素，则只需定位后操作，笔者依据这个思路对“高级设置”选项进行操作，代码如下：

```
//第 5 章/new_selenium/my_sel_html_10.py
from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
driver.maximize_window()
#鼠标不悬停，直接单击高级搜索
time.sleep(3)
driver.find_element_by_xpath("//span[text() = '高级搜索']").click()

#执行结果
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"xpath","selector":"//span[text() = '高级搜索']"}
```

示例中，笔者直接定位并操作“高级搜索”选项，从执行结果可以看出，代码报错并提示找不到元素。

2．悬停后操作

既然当直接定位并操作元素时代码会报错，那么就需要像功能测试一样，先将鼠标悬停在“设置”上，再单击“高级设置”选项，代码如下：

```
//第 5 章/new_selenium/my_sel_html_11.py
from selenium import webdriver
from selenium.webdriver import ActionChains
import time
```

```
driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
driver.maximize_window()
time.sleep(3)
#实例化动作链
action = ActionChains(driver)
#操作设置
setting = driver.find_element_by_xpath("//span[text()='设置']")
action.move_to_element(setting).perform()
#操作高级搜索
search = driver.find_element_by_xpath("//span[text()='高级搜索']")
action.click(search).perform()
```

示例中,笔者先定位“设置”按钮,然后使用 ActionChains 类中的 move_to_element()方法将鼠标移动到“设置”按钮上,然后定位“高级搜索”选项,使用 ActionChains 类中的 click()方法单击“高级搜索”选项。执行结果如图 5-18 所示。

图 5-18 百度高级搜索

值得注意的是,笔者在调用了 move_to_element()后直接调用了 perform()方法,原因是不调用 perform()方法鼠标是不会移动到“设置”按钮上的,而鼠标不移动到“设置”按钮上就不会显示“高级搜索”按钮,从而会造成定位操作“高级搜索”按钮时报错。

5.4.7 窗口切换操作

为了演示窗口切换操作,笔者对 a 标签的 HTML 代码进行修改,在 a 标签中添加一个

属性 target 并将其值设置为 blank，意思是单击 a 标签时打开一个新的窗口，代码如下：

```
<div>
    <a href = "https://www.baidu.com" target = "_blank">百度链接测试</a>
</div>
```

当读者单击 a 标签中的文字“百度链接测试”时浏览器会在新窗口打开百度首页，效果如图 5-19 所示。

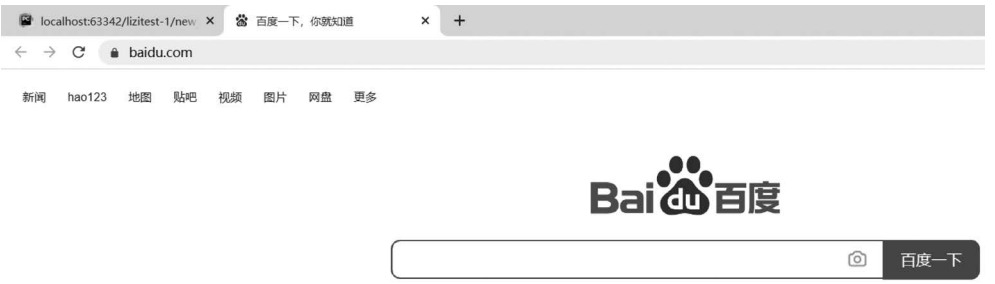

图 5-19　两个窗口

接下来笔者将演示直接操作新窗口和窗口切换后操作新窗口，读者可以从中理解窗口切换的作用。

1. 直接操作

按照前面学习过的知识，如果要操作百度窗口的输入框，则只需定位后操作，笔者依据此思路编写百度输入框操作代码，代码如下：

```
//第 5 章/new_selenium/my_sel_html_12.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test7.html')
driver.maximize_window()
#操作
time.sleep(1)
driver.find_element_by_xpath("//a[text() = '百度链接测试']").click()
time.sleep(1)
driver.find_element_by_xpath("//input[@id = 'kw']").send_keys("测试")

#执行结果
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"xpath","selector":"//input[@id = 'kw']"}
```

示例中，打开百度首页后笔者使用代码直接向百度输入框中输入内容，从执行结果可以看出，Selenium 根本找不到百度输入框这个元素。

2. 窗口切换后操作

找不到百度输入框的原因是百度输入框存在于一个新的窗口中，如果想对新窗口的标

签元素进行操作,则需要先切换到新窗口。浏览器中每个窗口都有一个窗口句柄,由于该句柄是窗口的唯一标识,所以笔者将使用窗口句柄进行窗口切换,代码如下:

```
//第 5 章/new_selenium/my_sel_html_13.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test7.html')
driver.maximize_window()
#操作
time.sleep(1)
driver.find_element_by_xpath("//a[text() = '百度链接测试']").click()
time.sleep(1)
#获取所有窗口句柄
handles = driver.window_handles
#遍历窗口句柄
for handle in handles:
    #切换窗口
    driver.switch_to.window(handle)
    #判断窗口 title 是否包含"百度一下"
    if "百度一下" in driver.title:
        time.sleep(1)
        driver.find_element_by_xpath("//input[@id = 'kw']").send_keys("测试")
        break
    else:
        continue
```

示例中,笔者先调用 WebDriver 的 window_handles()方法获取所有的窗口句柄,然后对所有窗口句柄进行遍历。遍历过程中笔者先调用 WebDriver 的 switch_to.window()方法切换窗口,然后判断窗口的 title 中是否包含“百度一下”字样,如果包含,则操作百度输入框并终止循环,如果不包含,则继续循环。代码的执行效果如图 5-20 所示。

图 5-20 多窗口操作百度输入框

5.4.8 iframe 切换操作

当前端开发人员想要在一个页面中包含另一个页面时会使用 iframe 标签,当 Selenium 自动化测试要操作 iframe 标签包含的网页中的内容时需要切换到 iframe 之内。

1. iframe 前端代码

为了让读者更加容易地理解 iframe 标签，笔者修改了之前的 iframe 前端代码，新增 1 个 input 标签，用于输入，iframe 标签的 src 属性重新赋值为必应首页，代码如下：

```
我的输入框:< input id = "my_baidu"></br ></br >
< iframe id = "my_iframe" src = "https://cn.bing.com/"
        width = "100 % " height = "500" frameborder = "0"
        allowfullscreen sandbox >
</iframe >
```

iframe 标签在浏览器中的效果如图 5-21 所示。

图 5-21　iframe 标签在浏览器中的效果

2. 不切换 iframe 操作

从浏览器的效果来看，笔者编写的输入框和必应的输入框在同一个窗口，按理应该可以直接定位操作，接下来笔者尝试直接操作 iframe 标签中的输入框，代码如下：

```
//第 5 章/new_selenium/my_sel_html_14.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test9.html')
driver.maximize_window()
# 操作
time.sleep(2)
driver.find_element_by_xpath("//input[@id = 'my_baidu']").send_keys("栗子")
driver.find_element_by_xpath("//input[@id = 'sb_form_q']").send_keys("测试")

# 执行结果
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"xpath","selector":"//input[@id = 'sb_form_q']"}
```

示例中，笔者操作自定义 input 标签没有报错，操作 iframe 标签中的输入框时提示定位不到元素，证明 iframe 标签中的标签不可以直接操作。代码的执行效果如图 5-22 所示。

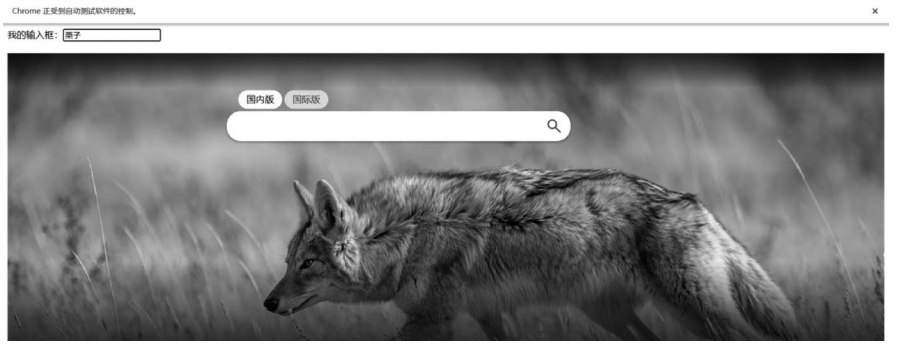

图 5-22　不切换 iframe 操作

3. 切换 iframe 操作

既然无法直接操作 iframe 中的标签，那么就应该先切换到 iframe 标签，然后对其内部的标签进行操作。切换到 iframe 标签需要先调用 WebDriver 的 switch_to()方法，通过方法返回调用 frame()方法，代码如下：

```
//第 5 章/new_selenium/my_sel_html_15.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test9.html')
driver.maximize_window()
#操作
time.sleep(2)
driver.find_element_by_xpath("//input[@id = 'my_baidu']").send_keys("栗子")
#切换 iframe
my_iframe = driver.find_element_by_xpath("//iframe[@id = 'my_iframe']")
driver.switch_to.frame(my_iframe)
driver.find_element_by_xpath("//input[@id = 'sb_form_q']").send_keys("测试")
```

示例中，笔者先定位 iframe 标签元素，然后调用 WebDriver 的 switch_to.frame()方法切换到 iframe 标签，最后操作必应输入框时不再报错。浏览器的效果如图 5-23 所示。

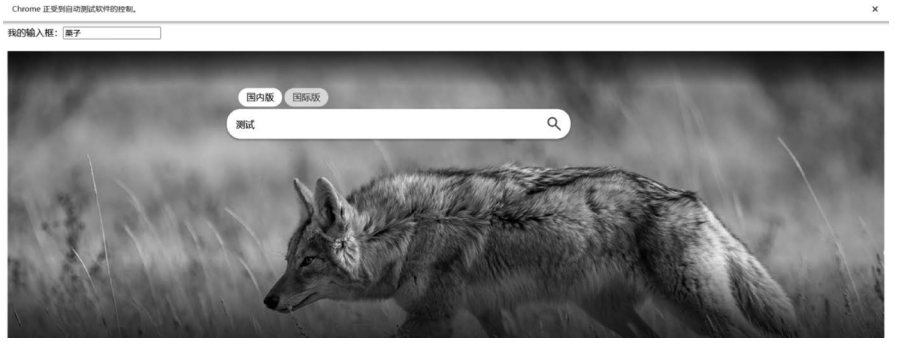

图 5-23　切换 iframe 操作

4. 退出 iframe 操作

当笔者切换到 iframe 标签操作完必应的输入框后，如果笔者想再次操作自定义输入框，就需要切换回来，不然代码还是会报错，即定位不到元素。从 iframe 标签切换回来需要调用 WebDriver 的 switch_to.default_content()方法，代码如下：

```
//第 5 章/new_selenium/my_sel_html_16.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test9.html')
driver.maximize_window()
#操作我的输入框
time.sleep(2)
driver.find_element_by_xpath("//input[@id = 'my_baidu']").send_keys("栗子")
#切换 iframe
my_iframe = driver.find_element_by_xpath("//iframe[@id = 'my_iframe']")
driver.switch_to.frame(my_iframe)
driver.find_element_by_xpath("//input[@id = 'sb_form_q']").send_keys("测试")
#切换回默认
driver.switch_to.default_content()
driver.find_element_by_xpath("//input[@id = 'my_baidu']").send_keys("666")
```

示例中，笔者先调用 WebDriver 的 switch_to.frame()方法，以此切换到 iframe 标签，再操作其中的内容，然后调用 WebDriver 的 switch_to.default_content()方法切换回默认窗口，在自定义输入框中输入 666，执行结果在浏览器中的效果如图 5-24 所示。

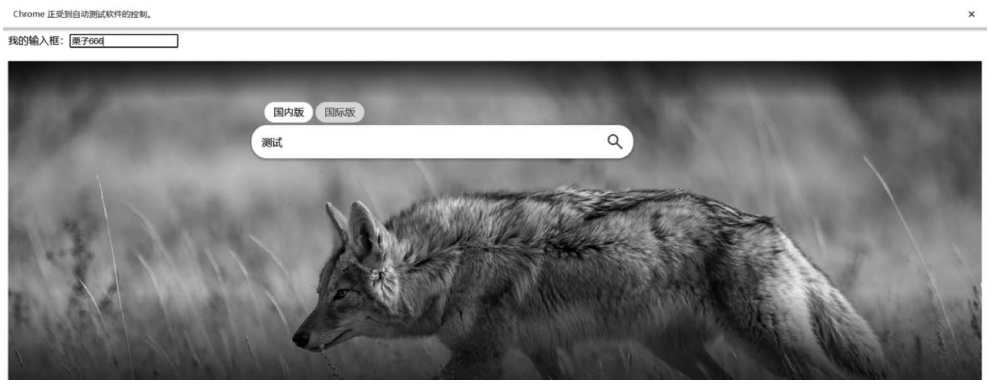

图 5-24　退出 iframe 操作

5.4.9　JavaScript 弹框操作

JavaScript 是 Web 编程语言，简称 JS。在 HTML 页面中可以使用 JavaScript 实现警告弹框或二次确认弹框。与 iframe 操作相同，如果读者想使用 Selenium 处理 JavaScript 的弹框，则需要先切换到 JavaScript 弹框，然后才能对其进行操作。

1. JavaScript 警告框

笔者写了一段简单的前端代码,代码包括一个输入框和一个手机号校验按钮,当读者单击手机号校验按钮时会弹出警告框,代码如下:

```
//第 5 章/new_selenium/test13.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>我的窗口</title>
    <script>
        function myFunction(){
            alert("请输入正确的手机号!");
        }
    </script>
</head>
<body>
    <div>
        <p>手机号输入框:<input id="phone_num"></p>
        <input type="button" onclick="myFunction()" value="手机号校验">
    </div>
</body>
</html>
```

示例中,input 标签的 onclick 属性表示单击操作,当用户单击“手机号校验”按钮时,代码会调用 myFunction()方法,该方法会弹出一个警告框,提示用户输入正确的手机号,浏览器的效果如图 5-25 所示。

图 5-25 JavaScript 警告框

2. 警告框操作

从浏览器的效果来看,JavaScript 警告框也是在窗口内部,按理可以直接单击“确定”按钮,但实际上警告框操作需要切换到警告框,然后调用 Alert 类的 accept()方法进行确认,代码如下:

```
//第 5 章/new_selenium/my_sel_html_17.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
```

```
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test13.html')
driver.maximize_window()
#单击"手机号校验"按钮
time.sleep(1)
driver.find_element_by_xpath("//input[@id = 'phone_num']").send_keys('123')
driver.find_element_by_xpath("//input[@value = '手机号校验']").click()
#处理消息提示框
time.sleep(1)
driver.switch_to.alert.accept()
```

示例中，笔者先调用 WebDriver 的 switch_to.alert 切换到警告框，然后调用 Alert 类 accept()方法单击警告框的“确定”按钮。

3. 再次操作输入框

根据 iframe 标签切换经验，从 iframe A 切换到 iframe B 完成操作后，需要再次切换到默认 iframe A 才可以继续对 iframe A 的元素进行操作。那么读者可以思考一下，切换到警告框完成操作后，是否还需要再次切换回来？答案是不需要。接下来笔者演示操作完消息提示框后直接操作输入框，代码如下：

```
//第 5 章/new_selenium/my_sel_html_18.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test13.html')
driver.maximize_window()
#单击"手机号校验"按钮
time.sleep(1)
driver.find_element_by_xpath("//input[@id = 'phone_num']").send_keys('123')
driver.find_element_by_xpath("//input[@value = '手机号校验']").click()
#操作消息提示框
time.sleep(1)
driver.switch_to.alert.accept()
#直接操作输入框
driver.find_element_by_xpath("//input[@id = 'phone_num']").send_keys('456')
```

示例中，笔者先在输入框中输入“123”，然后单击“手机号校验”按钮、单击消息提示框中的“确认”按钮，然后在没有进行任何切换操作的情况下直接在输入框内输入“456”，操作成功后代码没有报错。浏览器的效果如图 5-26 所示。

图 5-26　JavaScript 弹框无须切回到默认状态

4. JavaScript 二次确认框

JavaScript 除了可以实现警告框外，还可以实现二次确认框。笔者模拟系统中删除手机号时二次提示是否删除的功能，新建一个删除按钮，当用户单击“删除”按钮时，JavaScript 弹出二次确认提示框，询问是否删除，代码如下：

```
//第 5 章/new_selenium/test14.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>我的窗口</title>

</head>
<body>
    <div>
        <p id="my_number">
            手机号:<input type="text" id="phone_num" disabled value="13611111111">
        </p>
        <input type="button" onclick="myFunction()" value="删除">
        <p id="result"></p>
    </div>
    <script>
    function myFunction(){
        var x;
        var r=confirm("是否删除?");
        if (r==true){
            document.getElementById("my_number").innerHTML = ""
            x="数据已删除!";
        }
        else{
            x="您已取消删除!";
        }
        document.getElementById("result").innerHTML=x;
    }
    </script>
</body>
</html>
```

示例中,笔者还是使用 input 标签的 onclick 属性。当用户单击“删除”按钮时,代码会调用 myFunction()方法,该方法会弹出一个二次确认框,让用户确认是否删除手机号,浏览器的效果如图 5-27 所示。

图 5-27 JavaScript 二次确认框

5. 二次确认框操作

有了 JavaScript 警告框的处理经验后，笔者认为 JavaScript 二次确认框的处理方式应该相同，所以笔者直接编写代码以查看执行结果是否正确，代码如下：

```
//第 5 章/new_selenium/my_sel_html_19.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test14.html')
driver.maximize_window()
#单击"删除"按钮
time.sleep(1)
driver.find_element_by_xpath("//input[@value = '删除']").click()
#处理删除弹框
time.sleep(1)
#driver.switch_to.alert.accept()          #确定
driver.switch_to.alert.dismiss()          #取消
```

示例中，笔者先调用 WebDriver 的 switch_to.alert 切换到二次确认框，然后调用 Alert 类 dismiss()方法单击二次确认框的取消按钮，表示不删除手机号。读者可以参考警告框的操作，尝试单击二次确认框的确定按钮。

5.4.10 JavaScript 操作

Selenium WebDriver 除了可以处理 JavaScript 弹框外，还可以执行 JavaScript 命令。笔者先带领读者了解一下 JavaScript 如何定位标签元素，然后带领读者使用 JavaScript 处理下只读属性的标签元素。

1. JavaScript 基础

读者可以通过 JavaScript 中的 document 对象来定位、操作标签元素，所以接下来笔者将重点介绍 document 对象。

1) JavaScript 对象

JavaScript 的主要对象包括两个，一个是 window 对象；另一个是 document 对象。

(1) window 对象：在 JavaScript 中，一个浏览器窗口就是一个 window 对象。

(2) document 对象：window 对象存放了这个页面的所有信息，为了更好地分类处理这些信息，window 对象下面又分为很多对象，其中 document 对象包含了整个 HTML 文档，可以访问文档中的内容及其所有页面元素。JavaScript 通过 document 对象获取标签元素。

2) document 定位元素

笔者编写了一个简单的页面，页面中只包含一个输入框，代码如下：

```
//第 5 章/new_selenium/test15.html
< html lang = "en">
< head >
    < meta charset = "UTF - 8">
</head >
< body >
    < div >
        < p id = "my_number">
            手机号:< input type = "text" id = "phone_num" disabled value = "13611111111">
        </p >
    </div >
</body >
</html >
```

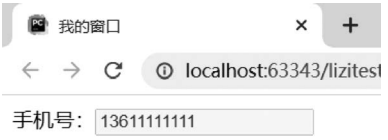

图 5-28　只读输入框

示例中,input 标签元素为只读模式,输入框中的值为电话号“13611111111”,浏览器的效果如图 5-28 所示。

仔细观察 input 标签元素的属性,笔者发现 input 标签元素包含 id 属性,所以笔者决定使用 id 属性进行定位。首先需要按 F12 快捷键打开开发者工具,然后进入控制台页面并调用了 document 对象的 getElementById()方法,参数传入的是 input 标签的 id 属性值,代码如下:

```
document.getElementById('phone_num')
```

在控制台页面输入示例代码后,按 Enter 键后就可以看到控制台中打印出 input 标签元素,表示定位 input 标签元素成功,如图 5-29 所示。

图 5-29　JavaScript 的 id 定位

3) document 修改属性值

定位到标签元素后,读者可以通过 value 属性获取标签的值,代码如下:

```
document.getElementById('phone_num').value
```

如果读者想要修改标签元素的值,则可以先获取标签元素的值,然后对其进行重新赋值,代码如下:

```
document.getElementById('phone_num').value = 13622222222
```

示例中，笔者对电话号码输入框进行了重新赋值，如图 5-30 所示。

图 5-30　JavaScript 重新赋值

2. JavaScript 中 xpath 定位

前面示例中笔者讲解了如何使用 JavaScript 进行 id 定位、赋值标签元素，但笔者在后面的代码中需要统一使用 xpath 定位，所以读者需要了解如何使用 JavaScript 进行 xpath 定位。

1) JavaScript 的 xpath 定位方法

JavaScript 中的 xpath 定位需要用到 document 的 evaluate()方法，格式如下：

```
document.evaluate(
        xpathExpression,
        contextNode,
        namespaceResolver
        resultType,
        result)
```

document 的 evaluate()方法可以根据传入的 xpath 表达式及其他参数返回一个 xpathResult 对象，读者可以通过这个 xpathResult 对象来定位和操作元素。evaluate()方法的参数的具体含义如下。

(1) xpathExpression：需要定位的元素的 xpath 路径。

(2) contextNode：本次查询的上下文节点，通常传入 document。

(3) namespaceResolver：通常用来解析 xpath 内的前缀，以便对文档进行匹配。通常传入 null。

(4) resultType：指定所返回的 xpathResult 的类型。具体类型如下。

- XPathResult.ANY_TYPE：适合于 xpath 表达式的数据类型。
- XPathResult.ANY_UNORDERED_NODE_TYPE：返回匹配节点的集合，顺序可能和文档中的不一样。
- XPathResult.BOOLEAN_TYPE：返回 boolean 类型。
- XPathResult.FIRST_ORDERED_NODE_TYPE：返回文档中匹配节点的第 1 个

节点。

- XPathResult.NUMBER_TYPE：返回 num 类型。
- XPathResult.ORDERED_NODE_ITERATOR_TYPE：返回匹配节点的集合，顺序和文档中的一样。
- XPathResult.ORDERED_NODE_SNAPSHOT_TYPE：返回一个节点集合片段，在文档外捕获节点，这样将来对文档的任何修改不会影响节点集合。节点集合中的顺序要和文档中的一样。
- XPathResult.STRING_TYPE：返回一个 string 类型。
- XPathResult.UNORDERED_NODE_ITERATOR_TYPE：返回匹配节点的集合，顺序可能和文档中的不一样。
- XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE：返回一个节点集合片段，在文档外捕获节点，这样将来对文档的任何修改不会影响节点集合。节点集合中的顺序没有必要和文档中的一样。

(5) result：用于存储查询结果，通常传入 null。此时将创建新的 xpathResult 对象。

2) JavaScript 的 xpath 定位方法实战

以上介绍了 document 的 evaluate()方法和参数，为了让读者更好地认识该方法如何使用，笔者还是使用手机号输入框来演示 xpathResult 对象获取、xpath 对应的标签获取和 xpath 对应的标签的值的获取。

(1) 获取 xpathResult 对象。

笔者想在文档中根据 xpath 获取输入框标签元素，并且如果文档中有多个相同的输入框，则按顺序获取第 1 个，所以将 resultType 参数传入 XPathResult.FIRST_ORDERED_NODE_TYPE，代码如下：

```
#获取 xpathResult 对象
document.evaluate("//input[@id = 'phone_num']", document, null, XPathResult.FIRST_ORDERED_
NODE_TYPE, null)
```

示例中，读者只需关注 evaluate()方法的 xpathExpression 参数和 resultType 参数，其他参数按照笔者的写法传入。XPathResult.FIRST_ORDERED_NODE_TYPE 表示按顺序匹配文档中节点的第 1 个节点。控制台的效果如图 5-31 所示。

图 5-31　获取 xpathResult 对象

(2) 获取 xpath 对应的标签。

获取 xpathResult 对象后，读者可以使用该对象获得 input 标签元素，代码如下：

```
#获取 xpath 对应的标签
document.evaluate("//input[@id = 'phone_num']", document, null, XPathResult.FIRST_ORDERED_
NODE_TYPE, null).singleNodeValue
```

示例中，singleNodeValue 用于匹配文档中的第 1 个节点，如果没有匹配到节点，则返回 null。控制台的效果如图 5-32 所示。

图 5-32　获取 input 标签元素

(3) 获取 xpath 对应标签的 value 值。

获取 input 标签元素后，可以使用 value 属性获取 input 标签元素的值，代码如下：

```
#获取 xpath 对应标签的 value 值
document.evaluate("//input[@id = 'phone_num']", document, null, XPathResult.FIRST_ORDERED_
NODE_TYPE, null).singleNodeValue.value
```

笔者将 input 标签的 value 属性的默认值设置为“13611111111”，所以使用 JavaScript 命令获取的值也应该是该值，控制台的效果如图 5-33 所示。

图 5-33　获取 value 值

3. JavaScript 处理只读标签元素

笔者已经将 JavaScript 如何操作标签元素讲解了一遍，相信读者也有了一定的了解。接下来笔者将举例说明在 WebDriver 中如何使用 JavaScript 的 xpath 定位来处理只读标签的问题。

1) 只读标签

为了让读者记忆深刻，笔者再次对 input 标签进行介绍，代码如下：

```
手机号:< input type = "text" id = "phone_num" disabled value = "13611111111">
```

示例中，input 标签元素的 disabled 属性表示标签只读，即用户不能修改其值。

2) WebDriver 操作只读标签

上述 input 标签元素如果使用 WebDriver 操作，则直接进行操作时代码会报错，代码如下：

```
//第 5 章/new_selenium/my_sel_html_21.py
import time
```

```
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test15.html')
driver.maximize_window()
#修改只读输入框
time.sleep(1)
driver.find_element_by_xpath("//input[@id = 'phone_num']").send_keys("136222222")

#执行结果
selenium.common.exceptions.ElementNotInteractableException: Message: element not interactable
```

示例中,执行结果提示元素不可交互,即元素为只读,表示不能被修改,所以如果读者想修改只读属性的元素,则需要借助 JavaScript 脚本进行操作。

3) WebDriver 执行 JavaScript 命令操作只读标签方法(1)

前面学习过如何使用 JavaScript 对标签元素进行定位和操作,但当时的操作都是在开发者工具的控制台中完成的。如果想使用 Selenium WebDriver 调用 JavaScript 命令,则需要用到 WebDriver 的 execute_script()方法,该方法的参数传入 JavaScript 命令即可,代码如下:

```
//第 5 章/new_selenium/my_sel_html_22.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test15.html')
driver.maximize_window()
#执行 JavaScript 代码
time.sleep(1)
js_command = "document.evaluate({}, document, null, XPathResult.FIRST_ORDERED_NODE_TYPE,
null)" \
             ".singleNodeValue" \
             ".removeAttribute('disabled')"\
            .format("\"//input[@id = 'phone_num']\"")
print(js_command)
driver.execute_script(js_command)
#修改只读输入框
driver.find_element_by_xpath("//input[@id = 'phone_num']").clear()
driver.find_element_by_xpath("//input[@id = 'phone_num']").send_keys("13622222222")

#执行结果
document.evaluate("//input[@id = 'phone_num']", document, null, XPathResult.FIRST_ORDERED_
NODE_TYPE, null).singleNodeValue.removeAttribute('disabled')
```

示例中,笔者将 JavaScript 命令作为字符串赋值给 js_command 变量。值得注意的是,笔者在 JavaScript 命令中使用了 removeAttribute()方法,表示删除标签元素的属性。笔者

的目的是删除 disabled 属性，所以在 removeAttribute() 方法中传入 disabled 参数。

有了删除标签元素 disabled 属性的 JavaScript 命令后，笔者调用 WebDriver 的 execute_script()方法并传入该 JavaScript 命令，执行后即可删除 input 标签元素的 disabled 属性了。删除了 input 标签元素的只读属性之后就可以正常对其进行操作了，浏览器的效果如图 5-34 所示。

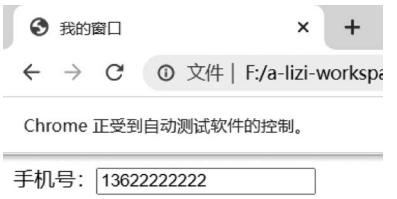

图 5-34　删除 disabled 属性后操作

4) WebDriver 执行 JavaScript 命令操作只读标签方法(2)

在开发过程中，前端开发人员可能不会设置 disabled 属性，而是设置了 readonly 属性，该属性也可以让输入框变为只读，代码如下：

```
手机号:< input type = "text" id = "phone_num" readonly value = "13611111111">
```

如果此时读者在 JavaScript 脚本的 removeAttribute()方法中仍然传入 disabled 参数，则执行代码后会报错，报错信息如下：

```
selenium.common.exceptions.InvalidElementStateException: Message: invalid element state
```

如果想要执行代码后不报错，则读者只需向 JavaScript 脚本的 removeAttribute()方法中传入 readonly 参数，其他代码无须修改，代码如下：

```
js_command = "document.evaluate({}, document, null, XPathResult.FIRST_ORDERED_NODE_TYPE,
null)" \
             ".singleNodeValue" \
             ".removeAttribute('readonly')"\
            .format("\"//input[@id = 'phone_num']\"")
```

5.4.11　获取属性值与断言

在 UI 自动化测试过程中，如果读者进行了登录操作，则该如何判断登录是否成功呢？此时就需要找到系统登录后的一些特有内容，在自动化脚本执行登录后判断这些内容是否存在，在这个过程中涉及如何获取内容、如何进行判断，本节中笔者将对此一一进行介绍。

1. 获取属性值

读者可以根据需要，通过 WebElement 的 get_attribute()方法获取不同的属性值，如属性值、文本值、标签元素内部的标签等。笔者编写了一个简单的 HTML 代码，用于演示如何获取这些属性值，代码如下：

```
< div id = "hobby">< span >打篮球</span ></div >
```

1) 获取文本属性值

获取标签元素的文本内容，需要先获取标签元素，再使用 WebElement 的 get_attribute()

方法,向方法中传入参数名 innerText 即可,代码如下:

```
//第 5 章/new_selenium/my_sel_html_23.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test16.html')
driver.maximize_window()
#获取标签元素
time.sleep(1)
hobby = driver.find_element_by_xpath("//div[@id = 'hobby']")
#获取文本信息
div_innerText = hobby.get_attribute("innerText")
print(div_innerText)

#执行结果
打篮球
```

示例中,笔者想获取 div 标签元素中包含的文本信息,首先定位 div 标签元素,然后使用该标签元素调用 get_attribute()方法,并传入 innerText,获取 div 标签中的文本内容“打篮球”。

2) 获取属性值

获取标签元素属性值也需要先获取标签元素,再使用 WebElement 的 get_attribute()方法进行获取,但此时 get_attribute()方法的参数需要传入属性名,例如笔者想获取 id 属性值,则传入 id 即可,代码如下:

```
//第 5 章/new_selenium/my_sel_html_23.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test16.html')
driver.maximize_window()
#获取标签元素
time.sleep(1)
hobby = driver.find_element_by_xpath("//div[@id = 'hobby']")
#获取 id 属性值
div_id = hobby.get_attribute("id")
print(div_id)
#执行结果
hobby
```

示例中,笔者想获取 div 标签元素的 id 属性值,还是先定位 div 标签元素,然后使用该标签元素调用 get_attribute()方法,并传入 id。获取 id 的属性值 hobby。

3) 获取标签内部的 HTML

获取标签元素内部的 HTML,使用 WebElement 的 get_attribute()方法且参数需要传入 innerHTML,获取的内容包含内部的标签、内部标签的属性、内部的文本,代码如下:

```
//第 5 章/new_selenium/my_sel_html_23.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test16.html')
driver.maximize_window()
#获取标签元素
time.sleep(1)
hobby = driver.find_element_by_xpath("//div[@id = 'hobby']")
#获取 div 标签中的 HTML
div_innerHTML = hobby.get_attribute("innerHTML")
print(div_innerHTML)

#执行结果
<span>打篮球</span>
```

示例中，笔者想获取 div 标签元素内部的 HTML，首先定位 div 标签元素，然后使用该标签元素调用 get_attribute()方法，并传入 innerHTML。获取的内容包括 span 标签和内部的文本信息。

2. 断言

断言的意思就是判断是否符合预期结果，Python 中使用 assert 进行断言，如果断言的结果为 True，则代码不打印任何信息；如果断言结果为 False，则程序会触发异常并打印错误信息，代码如下：

```
#断言成功，即断言结果为 True
assert 1 == 1
#断言失败，即断言结果为 False
assert 1 == 2, "1 不等于 2"

#执行结果
AssertionError: 1 不等于 2
```

示例中，笔者先使用 assert 断言 1 等于 1，执行结果没有打印错误信息，说明断言 1 等于 1 的结果为 True；接着笔者又使用 assert 断言 1 等于 2，并设置在断言结果为 False 时打印“1 不等于 2”，从执行结果可以看出打印了自定义的错误信息，即断言结果为 False。

3. 获取属性值并断言

有了对 Python 断言的基本了解之后，笔者将 UI 自动化测试代码和断言相结合，用例执行完成后对结果进行断言操作。

1）页面跳转断言

在 UI 自动化测试中经常需要进行页面跳转，每次页面跳转之后读者可以断言跳转是否成功。

(1) 百度首页代码。

笔者首先展示百度首页的 HTML 代码,由于笔者将使用 title 进行断言,所有主要关注的也是 head 标签中的 title 标签,代码如下:

```
<head>
    <meta charset="UTF-8">
    <title>百度一下,你就知道</title>
</head>
```

(2) 页面跳转断言。

要使用 title 进行断言,首先需要调用 WebDriver 的 title()方法获取页面的 title 值,然后使用 assert 将获取的 title 值与预期 title 值进行比较,代码如下:

```
//第5章/new_selenium/my_sel_html_25.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
driver.maximize_window()
#页面跳转断言
time.sleep(1)
assert driver.title == "百度一下,你就知道", "跳转百度首页失败!"
print(driver.title)

#执行结果
百度一下,你就知道
```

示例中,笔者断言时添加了错误返回内容,当断言结果为 False 时打印“跳转百度首页失败!”,当断言结果为 True 时不打印任何内容。

读者可以思考下,如果断言页面跳转失败,则断言后边的打印语句是否还会继续执行?为了演示,笔者修改断言预期以让断言结果失败,代码如下:

```
//第5章/new_selenium/my_sel_html_25.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
driver.maximize_window()
#页面跳转断言
time.sleep(1)
#assert driver.title == "百度一下,你就知道", "跳转百度首页失败!"
assert driver.title == "百度一下", "跳转百度首页失败!"
print(driver.title)

#执行结果
```

```
Traceback (most recent call last):
  File "F:/a - lizi - workspace/lizitest - 1/new_selenium_html/my_sel_html_25.py", line 10,
in < module >
    assert driver.title == "百度一下", "跳转百度首页失败!"
AssertionError: 跳转百度首页失败!
```

示例中,断言失败,并且只打印了笔者自定义错误信息,没有再执行断言后的打印语句,所以读者需要记住断言失败时其后面的语句不会继续执行。

2) 文本断言

除了页面跳转可以断言外,新增用户的场景同样可以断言。在实际工作中新增用户单击“确定”按钮后,系统会跳转到用户列表页并重新请求列表数据,新增数据一般会放在列表的第 1 条。接下来以此场景为例,笔者演示新增用户后的断言操作。

(1) 表格示例代码。

笔者写了一段简单的 HTML 表格,表格中包含一条数据,即代表新增的一条数据,代码如下:

```
//第 5 章/new_selenium/test17.html
< div >
    < table border = "1" id = "user_table" width = "500">
       < tr >
         < th >名称</th >
         < th >电话</th >
       </tr >
       < tr >
         < td >栗子测试</td >
         < td > 15611111111 </td >
       </tr >
    </table >
</div >
```

(2) 新增后断言。

假设需要规定用户名是唯一的,此时读者就可以获取表格中的用户名,然后将其与 UI 自动化测试时输入的用户名进行比较,代码如下:

```
//第 5 章/new_selenium/my_sel_html_26.py
import time
from selenium import webdriver

driver = webdriver.Chrome()
driver.get('F:\\a - lizi - workspace\\lizitest - 1\\new_html\\test17.html')
driver.maximize_window()
#操作
time.sleep(1)
first_user = driver.find_element_by_xpath("//table[@id = 'user_table']//td[1]")
#断言
```

```
assert first_user.text == '栗子测试', '第 1 条数据不是栗子测试'
print(first_user.text)

#执行结果
栗子测试
```

示例中,笔者调用 WebDriver 的 title()方法获取标签元素的文本,然后将该文本与 UI 自动化输入内容进行比较,执行结果没有报错且打印了第 1 条数据的用户名,表示断言成功。

5.4.12 下载文件操作

笔者在实际工作中很少使用 UI 自动化测试进行下载操作,但各个公司要求不同,所以笔者在这里简单介绍下自动下载文件操作,其实使用 Selenium WebDriver 下载文件只需配置浏览器的一些参数,然后到下载文件页面单击需要下载的文件。接下来笔者将使用不同浏览器演示如何在 http://chromedriver.storage.googleapis.com/网站下载指定的 chromedriver。

1. Chrome 浏览器下载文件

当使用 Chrome 浏览器下载文件时,需要调用 WebDriver 的 ChromeOptions()方法获取 Options 对象,然后调用 Options 的 add_experimental_option()方法对浏览器进行设置,设置内容包括禁止下载弹窗,以及设置文件下载路径,设置完成后就可以通过单击下载链接直接下载了,代码如下:

```
//第 5 章/new_selenium/my_sel_html_27.py
import time
from selenium import webdriver

#浏览器设置
options = webdriver.ChromeOptions()
prefs = {'profile.default_content_settings.popups': 0, #禁止弹出下载窗口
         'download.default_directory': os.getcwd()} #设置文件下载路径
options.add_experimental_option('prefs', prefs)
#实例化驱动时传入设置参数
driver = webdriver.Chrome(chrome_options = options)
driver.get('http://chromedriver.storage.googleapis.com/index.html?path = 111.0.5563.19/')
driver.maximize_window()
#下载 Windows 的驱动
time.sleep(1)
driver.find_element_by_xpath("//a[text() = 'chromedriver_win32.zip']").click()
```

2. Firefox 浏览器下载文件

当使用 Firefox 浏览器下载文件时,需要调用 WebDriver 的 FirefoxProfile()方法获取 FirefoxProfile 对象,然后调用 FirefoxProfile 的 set_preference()方法对浏览器进行设置,设置内容包括设置下载路径和下载文件类型,设置完成后就可以通过单击下载链接直接下载

了,代码如下:

```
//第 5 章/new_selenium/my_sel_html_28.py
import os
import time
from selenium import webdriver

#浏览器设置
fp = webdriver.FirefoxProfile()
fp.set_preference("browser.download.folderList", 2) #可以自定义下载目录
fp.set_preference("browser.download.dir", os.getcwd()) #指定下载目录
fp.set_preference("browser.helperApps.neverAsk.saveToDisk", "application/zip") #指定下载
                                                                                #文件类型
#实例化驱动时传入设置参数
driver = webdriver.Firefox(Firefox_profile=fp)
driver.get('http://chromedriver.storage.googleapis.com/index.html?path=111.0.5563.19/')
driver.maximize_window()
#下载 Windows 的驱动
time.sleep(1)
driver.find_element_by_xpath("//a[text()='chromedriver_win32.zip']").click()
```

5.5 WebDriver 元素等待

在实际工作中有时在页面上明明可以看到控件,但代码依然报错并提示找不到元素。此时读者可以考虑是否代码运行得太快,即在控件还没有加载完成就开始定位元素,从而导致定位不到元素并报错。由此引出一个概念叫作元素等待,意思是等待元素加载完成后再进行操作。元素等待有 3 种方式:强制等待、隐式等待、显式等待。

5.5.1 强制等待

强制等待方式笔者已经在前面的代码中使用过了很多次,即 time.sleep()方法。之所以叫作强制等待是因为使用 sleep()方法设置 3s 等待时间后,代码就会等待 3s,不会智能地去判断到底应该等待几秒,所以叫作强制等待。以操作百度首页输入框为例,假设笔者设置打开首页 3s 后操作输入框,可能会发生如下问题。

(1) 如果百度首页输入框在 1s 之内加载出来,则强制等待就会多等待 2s,导致脚本执行速度变慢。

(2) 如果百度首页输入框在 10s 后才加载出来,则强制等待 3s 显然不够,脚本一定会报元素不存在。

虽然强制等待有缺点,但也不是说就一定不能使用。如果读者的测试场景是在新增数据提交后必须等待 2s 才能数据同步成功,就可以用 sleep()方法等待 2s,然后去做断言,读者需要记住所有的事物存在即合理。

5.5.2 隐式等待

隐式等待是全局性的等待,只需设置一次就可以在 WebDriver 的生命周期内一直生效。隐式等待设置了一个等待时间,当被操作元素在等待时间内加载完成时,可以正常操作该元素;当被操作元素在等待时间内未加载完成时,操作该元素时代码会报错。

1. 等待时间内可以找到元素

以操作百度首页输入框为例,假设需求是最多等待 10s,笔者调用 WebDriver 的 implicitly_wait()方法进行隐式等待,代码如下:

```
//第 5 章/new_selenium/my_sel_html_29.py
from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.maximize_window()
print(time.strftime("%Y-%m-%d %X", time.localtime()))
driver.implicitly_wait(10)
driver.get("https://www.baidu.com/")
driver.find_element_by_xpath("//input[@id='kw']").send_keys("栗子测试")
print(time.strftime("%Y-%m-%d %X", time.localtime()))

#执行结果
2023-02-17 23:47:00
2023-02-17 23:47:03
```

示例中,笔者先使用 driver.implicitly_wait()方法设置隐式等待,将超时时间设置为 10s。从执行结果中可以看出,从打开百度到操作百度输入框用时 3s,说明隐式等待很智能,不需要等待 10s,定位到元素后就可以直接对其进行操作。

2. 等待时间内找不到元素

还是以操作百度首页输入框为例,为了让隐式等待超时,笔者故意将输入框的 xpath 改成错误的值,代码如下:

```
//第 5 章/new_selenium/my_sel_html_30.py
from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.maximize_window()
print(time.strftime("%Y-%m-%d %X", time.localtime()))
driver.implicitly_wait(10)
driver.get("https://www.baidu.com/")
driver.find_element_by_xpath("//input[@id='kw2']").send_keys("栗子测试")
print(time.strftime("%Y-%m-%d %X", time.localtime()))

#执行结果
```

```
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"xpath","selector":"//input[@id='kw2']"}
```

示例中，笔者故意将百度输入框的 xpath 写错，目的是让代码定位不到元素。此种情况下，隐式等待会在 10s 内不停地轮询寻找元素，到超时时间 10s 后，报错并提示定位不到元素。

5.5.3 显式等待

跟隐式等待不同，显式等待需要在每个需要操作的元素的前面进行声明。显式等待需要调用 WebDriverWait 类和 expected_conditions 模块。

1. WebDriverWait 类

WebDriverWait 类的主要作用是设置等待超时时间，在 WebDriverWait 类的使用过程中，笔者一般只传入驱动 driver 和超时时间 timeout 两个参数，其他使用默认值即可。格式如下：

```
WebDriverWait(driver,timeout,poll_frequency=0.5,ignored_exceptions=None)
- driver:浏览器驱动.
- timeout:超时时间,单位:秒.
- poll_frequency:检测的间隔步长,默认为 0.5s.
- ignored_exceptions:超时后抛出的异常信息,默认抛出 NoSuchElementExeception.
```

2. expected_conditions 模块

expected_conditions 模块的主要作用是设置预期判断条件，在 expected_conditions 模块的使用过程中，判断条件可以有很多个，笔者在这里只列出了两个，这两个条件都是判断元素是否可见，只不过参数不同而已，一个是传定位器 locator；另一个是传元素 element，代码如下：

```
expected_conditions 模块
- visibility_of_element_located 类:判断元素是否可见,参数为定位器 locator.
- visibility_of 类:判断元素是否可见,参数为元素 element.
```

3. 显式等待应用

接下来笔者以百度输入框为例，分别使用定位器和元素进行显式等待操作。

1）定位器显式等待

定位器 locator 的获取需要调用 WebDriver 的 By 类，然后通过 locator 定位元素，代码如下：

```
#定位器 locator
locator = (By.XPATH, "//input[@id='kw']")
bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of_element_
located(locator))
```

2）元素显式等待

元素的获取笔者已经讲过很多次，只需调用 WebDriver 的 find_element_by_xpath()方

法,代码如下:

```
#元素 element
element = driver.find_element_by_xpath("//input[@id='kw']")
bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of(element))
```

4. 等待时间内可以找到元素

还是以百度首页输入框为例,笔者使用元素为参数进行显式等待,代码如下:

```
//第 5 章/new_selenium/my_sel_html_31.py
import time
from selenium import webdriver
from selenium.webdriver.support import expected_conditions
from selenium.webdriver.support.wait import WebDriverWait

driver = webdriver.Chrome()
driver.maximize_window()
print(time.strftime("%Y-%m-%d %X", time.localtime()))
driver.get("https://www.baidu.com/")
#显式等待
element = driver.find_element_by_xpath("//input[@id='kw']") #定位到元素
bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of(bd_input))
#显式等待,条件是直到元素可见
#操作
bd_input.send_keys("栗子测试")
print(time.strftime("%Y-%m-%d %X", time.localtime()))

#执行结果
2023-02-19 08:53:10
2023-02-19 08:53:13
```

示例中,笔者使用 WebDriverWait 将超时时间设置为 10s,使用 expected_conditions 模块的 visibility_of 类将条件设置为元素可见。从执行结果可以看出,代码用了 3s 的时间完成了操作,说明显式等待也是智能等待。

5. 等待时间内找不到元素

笔者还是故意将百度首页输入框 xpath 写错,想以此来验证显式等待超时的情况,代码如下:

```
//第 5 章/new_selenium/my_sel_html_32.py
import time
from selenium import webdriver
from selenium.webdriver.support import expected_conditions
from selenium.webdriver.support.wait import WebDriverWait

driver = webdriver.Chrome()
driver.maximize_window()
print(time.strftime("%Y-%m-%d %X", time.localtime()))
```

```
driver.get("https://www.baidu.com/")
#显式等待
element = driver.find_element_by_xpath("//input[@id='kw2']") #定位到元素
bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of(bd_input))
#显式等待,条件是直到元素可见
#操作
bd_input.send_keys("栗子测试")
print(time.strftime("%Y-%m-%d %X", time.localtime()))

#执行结果
Traceback (most recent call last):
  File "F:/a-lizi-workspace/lizitest-1/new_selenium_html/my_sel_html_32.py", line 11,
in <module>
    element = driver.find_element_by_xpath("//input[@id='kw2']")
selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate
element: {"method":"xpath","selector":"//input[@id='kw2']"}
```

示例中,当代码执行时很快就提示不能定位到元素,这个提示是 find_element_by_xpath()方法提示的,所以代码并没有运行到 WebDriverWait,也就不能验证超时是否生效。

接下来笔者改用定位器显式等待的方式进行测试,代码如下:

```
//第5章/new_selenium/my_sel_html_33.py
import time
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions
from selenium.webdriver.support.wait import WebDriverWait

driver = webdriver.Chrome()
driver.maximize_window()
print(time.strftime("%Y-%m-%d %X", time.localtime()))
driver.get("https://www.baidu.com/")
#显式等待
locator = (By.XPATH, "//input[@id='kw2']")
bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of_element_
located(locator)) #显式等待,条件是直到元素可见
bd_input.send_keys("栗子测试")
print(time.strftime("%Y-%m-%d %X", time.localtime()))

#执行结果
Traceback (most recent call last):
  File "F:/a-lizi-workspace/lizitest-1/new_selenium_html/my_sel_html_33.py", line 13,
in <module>
    bd_input = WebDriverWait(driver, 10).until(expected_conditions.visibility_of_element_
located(locator))
selenium.common.exceptions.TimeoutException: Message:
```

示例中,使用定位器进行显式等待执行时,代码运行到 WebDriverWait 后一直在不停地定位元素,直到 10s 后报错并提示 TimeoutException 表示定位元素超时,证明显式等待

生效。

在实际工作中页面上的每个标签元素的加载时间是不同的,而显式等待是针对每个标签元素进行等待的,那么当读者知道某个标签元素加载时间较长时,可以将等待时间设置得长一些,这样会比隐式等待更加灵活。

5.6 WebDriver 鼠标操作

对于鼠标操作,笔者在自动化测试代码编写过程中不经常使用,这里做一些简单介绍。鼠标操作一般包括单击、双击、右击、拖动、移动到元素上、按下左键等,这些操作方法都包含在 ActionChains 类中。

笔者在前面的小节中已经对 ActionChains 类介绍过多次,以鼠标悬停为例回顾如何使用鼠标操作单击百度首页设置中的“高级搜索”项,代码如下:

```
//第 5 章/new_selenium/my_sel_html_34.py
from selenium import webdriver
from selenium.webdriver import ActionChains
import time

driver = webdriver.Chrome()
driver.get('https://www.baidu.com')
driver.maximize_window()
#实例化动作链
action = ActionChains(driver)
time.sleep(3)
#将鼠标移动到设置按钮上
setting = driver.find_element_by_xpath("//span[text() = '设置']")
action.move_to_element(setting).perform()
#鼠标单击高级搜索
search = driver.find_element_by_xpath("//span[text() = '高级搜索']")
action.click(search).perform()
```

示例中,将鼠标移动到指定标签元素需要调用 ActionChains 类的 move_to_element()方法;鼠标单击操作需要调用 ActionChains 类的 click()方法。需要注意的是,如果想让与鼠标操作相关的方法生效,则一定要调用 perform()方法。

为了方便记忆,笔者对鼠标操作的方法进行了总结。

1. 基本操作

单击、双击、右击等都是鼠标的基本操作,代码如下:

```
action.move_to_element(元素).perform()          #移动到元素上
action.click(元素).perform()                    #单击元素
action.double_click(元素).perform()             #双击元素
action.context_click(元素).perform()            #在元素上右击
action.click_and_hold(元素).perform()           #在元素上单击并按下
```

2. 将元素 1 拖曳到元素 2 的位置

如果想将元素 1 拖曳到元素 2 的位置，则读者可以使用不同的方式实现。方式一，直接调用 action.drag_and_drop()方法；方式二，先调用 action.click_and_hold()方法实现鼠标左键单击元素 1 不放，然后调用 release()方法在元素 2 的位置释放鼠标；方式三，先调用 action.click_and_hold()方法实现鼠标左键单击元素 1 不放，然后调用 move_to_element()方法将鼠标移动到元素 2 处，最后调用 release()方法释放鼠标，代码如下：

```
action.drag_and_drop(元素 1, 元素 2).perform()
action.click_and_hold(元素 1).release(元素 2).perform()
action.click_and_hold(元素 1).move_to_element(元素 2).release().perform()
```

3. 将元素移动到指定位置

如果想将元素移动到指定位置，则有两种不同的方式。方式一，可以调用 ActionChains 类的 drag_and_drop_by_offset()方法，传入元素和指定坐标；方式二，可以先调用 ActionChains 类的 click_and_hold()方法按住元素，再调用 drag_and_drop_by_offset()方法移动到指定坐标，最后调用 release()方法释放鼠标，代码如下：

```
action.drag_and_drop_by_offset(元素, 400, 150).perform()
action.click_and_hold(元素).move_by_offset(400, 150).release().perform()
```

5.7 WebDriver 键盘操作

在 Selenium 中提供了 Keys 类，在该类中定义了不同的按键属性，读者可以通过调用属性来完成键盘操作或键盘组合操作。读者可以使用 WebElement 类的 send_keys()方法来模拟键盘上的所有按键操作。例如笔者想用键盘组合键实现全选、复制、粘贴等功能，代码如下：

```
//第 5 章/new_selenium/my_sel_html_35.py
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
import time

driver = webdriver.Chrome()
driver.maximize_window()
#操作百度
driver.get('http://www.baidu.com')
time.sleep(2)
bd_input = driver.find_element_by_xpath("//input[@id='kw']")
bd_input.send_keys("栗子测试")
bd_input.send_keys(Keys.CONTROL, 'a')        #全选
bd_input.send_keys(Keys.CONTROL, 'c')        #复制
bd_input.send_keys(Keys.ENTER)               #按 Enter 键
#为了显示效果等待 5s
```

```
time.sleep(5)
#操作必应
driver.get('https://cn.bing.com/')
time.sleep(2)
by_input = driver.find_element_by_xpath("//input[@id='sb_form_q']")
by_input.send_keys(Keys.CONTROL, 'v')          #粘贴
by_input.send_keys(Keys.BACK_SPACE)            #删除最后一个字
by_input.send_keys(Keys.ENTER)                 #按 Enter 键
```

示例中,当笔者想实现全选操作时,调用 send_keys()方法并传入两个参数 Keys.CONTROL 和'a',表示按组合键 Ctrl+A,其他键盘操作读者可以查看 Keys 类自行尝试。

5.8 本章总结

本章将 WebDriver API 中的重点内容一一进行了举例讲解,虽然还有一些内容没有讲到,但所讲解内容已经足够应对日常工作,对于没有讲解的内容如果读者在工作过程中遇到,则可以随时在网上进行查找分析。

学习了本章后,读者应该能够熟练地使用 Firefox 插件 Ruto-XPath Finder 找到每个元素的 xpath,并根据实际情况使用基本操作小节中的内容对元素进行操作。在元素操作过程中,读者需要注意使用显示等待来增加代码的稳定性和执行效率。对于定位不到的元素读者可以从三方面进行思考,第一考虑元素的 xpath 是否正确;第二考虑该元素是否在 iframe 中;第三考虑元素是否在另一个窗口。相信读者仔细学习本章内容后,不会再对 WebDriver API 感到陌生,遇到问题也能自己尝试解决。