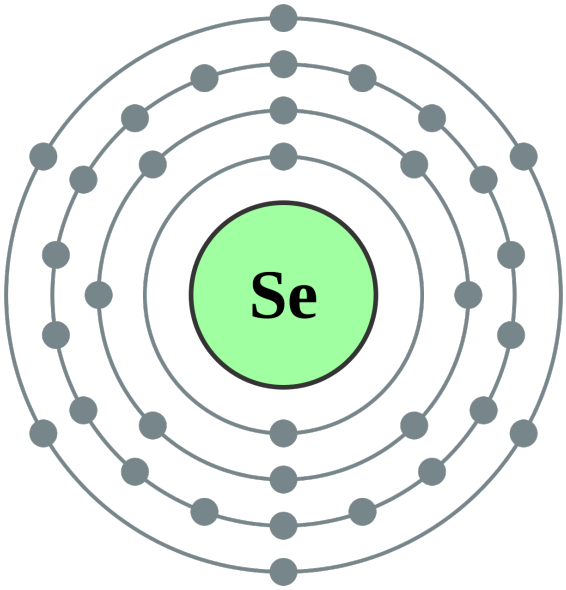
Selenium WebDriver Python 学习笔记 V1.0

司晓龙

SI-Xiaolong

2020 年 8 月 4 日



目录

1 前言 **7**

2 参考资料 **8**

3 简介 **9**

4 了解组件 **10**
4.1 专业术语 . . . 10
4.2 组成部分 . . . 10
4.3 应用框架 . . . 11

5 准备工作 **13**
5.1 安装 selenium . . . 13
5.2 驱动要求 . . . 13
5.3 下载浏览器驱动 . . . 13
5.4 把浏览器驱动放入环境变量（PATH）中，或者直接告知 selenuim 的驱动路径 . . . 13
5.4.1 将.exe 文件放入 PATH 中 . . . 13
5.4.2 将驱动路径告知 selenuim . . . 14
5.4.3 启动浏览器 . . . 14
5.5 引入 WebDriver 库 . . . 15

6 WebElement **16**
6.1 Find Element . . . 16
6.2 Find Elements . . . 16
6.3 Find Element From Element . . . 17
6.4 Find Elements From Element . . . 17
6.5 Get Active Element . . . 17

7 Webelement 定位 **19**
7.1 定位一个元素 . . . 19
7.2 定位多个元素 . . . 19
7.3 元素选择策略 . . . 20
7.4 使用选择器的提示 . . . 20
7.5 相对定位 . . . 21

7.5.1 如何工作 . . . 21
7.5.2 above() 元素上 . . . 21
7.5.3 below() 元素下 . . . 21
7.5.4 toLeftOf() 元素左 . . . 22
7.5.5 toRightOf() 元素右 . . . 22
7.5.6 near() 附近 . . . 22
7.6 查看页面元素 . . . 22
7.7 所有查找元素的方法 . . . 24
7.7.1 id 定位 . . . 24
7.7.2 name 定位 . . . 24
7.7.3 class 定位 . . . 25
7.7.4 tag 定位 . . . 25
7.7.5 link 定位 . . . 26
7.7.6 partial_link 定位 . . . 27
7.7.7 xpath 定位 . . . 28
7.7.8 css 定位 . . . 29

8 Webelement 常用方法 30
8.1 点击和输入 . . . 30
8.2 提交 . . . 30
8.3 其他 . . . 30

9 键盘事件 31
9.1 sendKeys . . . 31
9.2 keyDown . . . 31
9.3 keyUp . . . 32
9.4 clear . . . 32
9.5 常用的键盘操作 . . . 32

10 在 AUT* 中执行 34
10.1 单击元素 . . . 34

11 鼠标操作 35

12 浏览器导航 **36**
12.1 导航到某个网页 . . . 36
12.2 获取当前 URL . . . 36
12.3 浏览器后退，前进 . . . 36
12.4 刷新 . . . 36
12.5 获取当前网页标题 . . . 36

13 窗口和标签页 **37**
13.1 获取窗口句柄 . . . 37
13.2 切换窗口或标签页 . . . 37
13.2.1 切换窗口 . . . 38
13.2.2 创建新窗口或新标签页并且切换 . . . 38
13.3 关闭窗口或标签页 . . . 38
13.4 在会话结束时退出浏览器 . . . 39

14 窗口管理 **41**
14.1 获取窗口大小 . . . 41
14.2 设置窗口大小 . . . 41
14.3 获取窗口位置 . . . 41

15 设置窗口位置 **42**
15.1 最大化窗口 . . . 42
15.2 最小化窗口 . . . 42
15.3 全屏窗口 . . . 42

16 窗口截图 **43**

17 框架 (frames) 和内嵌框架 (iframe) **44**
17.1 使用 WebElement . . . 44
17.2 使用 name 或 id . . . 44
17.3 使用索引 (index) . . . 45
17.4 留下框架 . . . 45

18 页面加载策略 **46**
18.1 normal . . . 46
18.2 eager . . . 46

18.3 none 47

19 等待 **48**
19.1 显示等待 49
19.1.1 选项 50
19.1.2 预期的条件 50
19.1.3 经验 50
19.2 隐式等待 51
19.2.1 经验 52
19.3 流畅等待 52

20 支持的类 (Support classes) **53**

21 JavaScript 警告框, 提示框和确认框 **54**
21.1 Alerts 警告框 54
21.2 Confirm 确认框 54
21.3 Prompt 提示框 55
21.4 经验：警告框处理 55

22 调用 JavaScript 代码 **56**

23 Http 代理 **57**

24 获取断言信息 **58**

25 下拉框选择 **59**

26 文件上传 **60**

27 cookie 操作 **61**

28 附 **62**
28.1 延时效果（time 库） 62
28.1.1 time 库基本介绍 62
28.1.2 时间获取 62
28.1.3 时间格式化 63
28.1.4 程序计时应用 64

29 个人作品 **65**
29.1 平 安 堡 自 动 撕 布 机 (问卷星自动填写) 65

1 前言

本文章为个人学习笔记，内容部分稚嫩部分成熟，部分详细部分有跳步。

本文章内部分代码，部分因 WebDriver 的特性，原网页变动导致无法使用，部分因官方文档语法问题无法使用。本人不敢保证完全更正了所有错误。

内容来自官方文档和互联网上的资料，部分内容可能有雷同，内容过杂，不列完整参考，还请见谅。

联系方式：sixiaolong2018@outlook.com

2 参考资料

Selenium Documentation
Selenium 官方文档
Selenium Python 教程 - 知乎：九四干
Selenium 八种元素定位方法 - eastonliu

3 简介

WebDriver 以本地化方式驱动浏览器，就像用户在本地或使用 Selenium 服务器的远程机器上所做的那样，这标志着浏览器自动化的飞跃。

Selenium WebDriver 指的是语言绑定和各个浏览器控制代码的实现。这通常被称为 WebDriver。

Selenium WebDriver 是 W3C 推荐标准

[*]WebDriver 被设计成一个简单和简洁的编程接口。

[*]WebDriver 是一个简洁的面向对象 API。

[*] 它能有效地驱动浏览器。

4 了解组件

使用 WebDriver 构建测试套件需要理解并有效地使用许多不同的组件。就像软件中的一切一样，不同的人对同一个想法使用不同的术语。下面是在这个描述中如何使用术语的细分。

4.1 专业术语

API: 应用程序编程接口。这是一组用来操作 WebDriver 的“命令”。

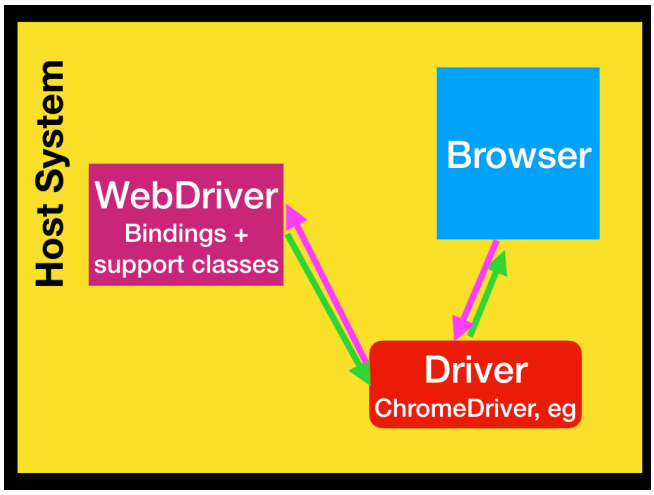
库: 一个代码模块，它包含 api 和实现这些 api 所需的代码。库是对应于具体的语言的，例如 Java 的.jar 文件，.NET 的.dll 文件，等等。

驱动程序: 负责控制实际的浏览器。大多数驱动程序是由浏览器厂商自己创建的。驱动程序通常是与浏览器一起在系统上运行的可执行模块，而不是在执行测试套件的系统上。(尽管它们可能是同一个系统。) 注意: 有些人把驱动称为代理。

框架: 用于支持 WebDriver 套件的附加库。这些框架可能是测试框架，如 JUnit 或 NUnit。它们也可能是支持自然语言特性的框架，如 Cucumber 或 Robotium。还可以编写和使用框架来操作或配置被测试的系统、数据创建、测试预言等等。

4.2 组成部分

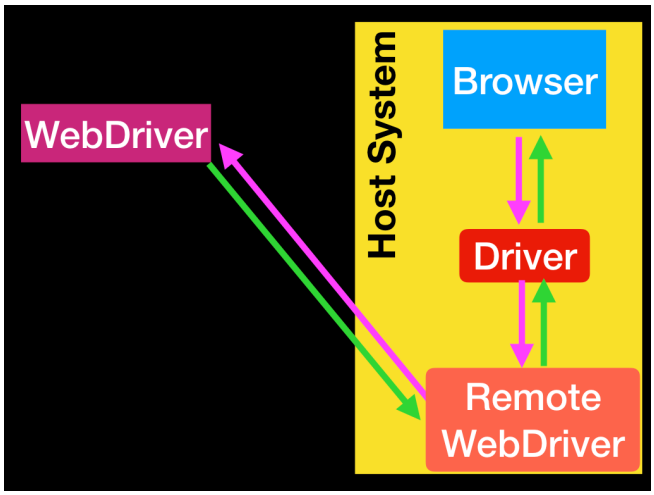
至少，WebDriver 通过一个驱动程序与浏览器对话。通信有两种方式: WebDriver 通过驱动程序向浏览器传递命令，然后通过相同的路径接收信息。



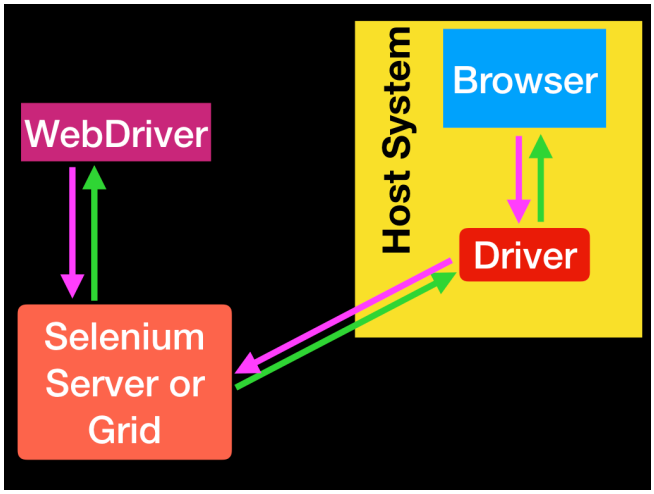
驱动程序是特定于浏览器的，例如 ChromeDriver 对应于谷歌的 Chrome/Chromium，GeckoDriver 对应于 Mozilla 的 Firefox 的，等等。驱动程序在与浏览器相同的系统上运行。这可能与执行测试本身的系统

相同，也可能不同。

上面这个简单的例子就是 **直接**通信。与浏览器的通信也可以是通过 Selenium 服务器或 RemoteWebDriver 进行的 **远程**通信。RemoteWebDriver 与驱动程序和浏览器运行在同一个系统上。



远程通信也可以使用 Selenium Server 或 Selenium Grid 进行，这两者依次与主机系统上的驱动程序进行通信

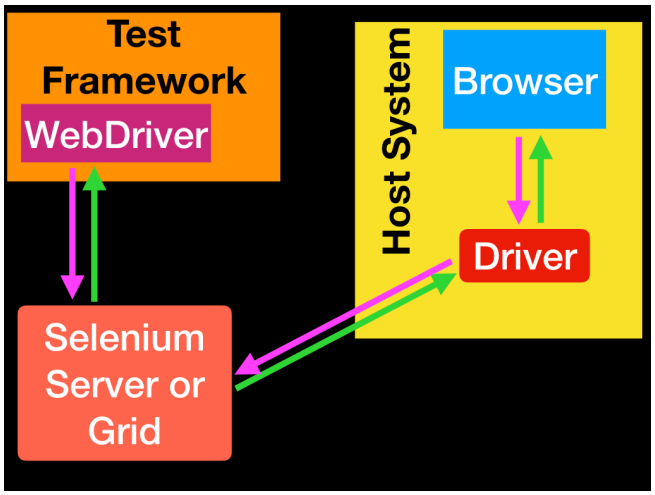


4.3 应用框架

WebDriver 有且只有一个任务: 通过上面的任何方法与浏览器通信。WebDriver 对测试一窍不通：它不知道如何比较事物、断言通过或失败，当然它也不知道报告或 Given/When/Then 语法。

这就是各种框架发挥作用的地方。至少你需要一个与绑定语言相匹配的测试框架，比如. NET 的 NUnit, Java 的 JUnit, Ruby 的 RSpec 等等。

测试框架负责运行和执行 WebDriver 以及测试中相关步骤。因此，您可以认为它看起来类似于下图。



像 Cucumber 这样的自然语言框架/工具可能作为上图中测试框架框的一部分存在，或者它们可能将测试框架完全封装在它们自己的实现中。

5 准备工作

5.1 安装 selenium

```
pip install selenium
```

5.2 驱动要求

通过 WebDriver，Selenium 支持市面上所有主流的浏览器，如 Chrom(ium)、Firefox、Internet Explorer、Opera 和 Safari。尽管并非所有浏览器都对远程控制提供官方支持，但 WebDriver 尽可能使用浏览器的内置自动化支持来驱动浏览器。

WebDriver 的目标是尽可能模拟真实用户与浏览器的交互。

在不同的浏览器中，这可能有不同的级别。有关不同驱动程序特性的详细信息，请参见**驱动程序特性**。

尽管所有的驱动程序共享一个面向用户的界面来控制浏览器，但它们设置浏览器会话的方式略有不同。由于许多驱动程序实现是由第三方提供的，所以它们不包括在标准的 Selenium 发行版中。

驱动程序实例化、配置文件管理和各种特定于浏览器的设置都是具体参数的例子，这些参数根据浏览器有不同的需求。本节介绍了使用不同浏览器的基本要求。

5.3 下载浏览器驱动

Selenium WebDriver 支持市面上绝大多数浏览器。大多数驱动程序需要额外的.exe 文件才能与浏览器交流。

Firefox 浏览器驱动

Chrome 浏览器驱动　Chrome 浏览器驱动 Taobao 镜像

IE 浏览器驱动

Edge 浏览器驱动

Opera 浏览器驱动

PhantomJS 浏览器驱动

5.4 把浏览器驱动放入环境变量（PATH）中，或者直接告知 selenuim 的驱动路径

5.4.1 将.exe 文件放入 PATH 中

可以将.exe 放入现有的 PATH 路径里，也可以创建一个 PATH 路径。我们以创建一个 PATH 路径为例（Windows 环境下）：

[*] 创建一个目录来放置可执行文件，如：C:\WebDriver\bin

[*] 将目录添加至 PATH，以管理员身份打开 Windows PowerShell，运行以下命令，以便将目录永久添加到计算机上所有用户的路径：

```
setx /m path "%path%;C:\WebDriver\bin\"
```

[*] 现在可以测试更改了，关闭所有 Windows PowerShell，打开一个新的 Windows PowerShell。键入在上一步创建的文件夹中创建的一个.exe 文件名称来验证 PATH 路径是否添加，例如：

```
MicrosoftWebDriver
```

[*] 如果 PATH 配置正确，可以看到一些与驱动程序启动相关的输出。

```
Starting MSEdgeDriver 84.0.522.40 (4ede1b0ab15e7ee434735507aa3e79d88ef95c48) on port 9515
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping
    MSEdgeDriver safe.
MSEdgeDriver was started successfully.
```

可以通过按 Ctrl+C 重新控制 Windows PowerShell.

5.4.2 将驱动路径告知 selenuim

在浏览器的参数内填写 xecutable_path，并指向.exe 的位置。

```
Edge(executable_path='/path/to/MicrosoftWebDriver.exe')
```

5.4.3 启动浏览器

一切准备就绪后，可以使用以下代码启动浏览器，同时测试是否正常：

```
from selenium import webdriver

# driver = webdriver.Firefox("驱动路径")

driver = webdriver.Firefox()   # Firefox浏览器

driver = webdriver.Chrome()    # Chrome浏览器

driver = webdriver.Ie()        # Internet Explorer浏览器

driver = webdriver.Edge()      # Edge浏览器

```

```
13 driver = webdriver.Opera()     # Opera浏览器
14
15 driver = webdriver.PhantomJS()   # PhantomJS
```

若成功，以 **Edge** 为例，会打开一个裸浏览器，并提示：“**Microsoft Edge** 正由自动检测软件控制”

5.5 引入 WebDriver 库

```
1 from selenium import webdriver
```

6 WebElement

WebElement 表示 DOM 元素. 可以通过使用 WebDriver 实例从文档根节点进行搜索, 或者在另一个 WebElement 下进行搜索来找到 WebElement.

WebDriver API 提供了内置方法来查找基于不同属性的 WebElement (例如 ID, Name, Class, XPath, CSS 选择器, 链接文本等).

6.1 Find Element

此方法用于查找元素并返回第一个匹配的单个 WebElement 引用, 该元素可用于进一步的元素操作.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Firefox()

driver.get("http://www.google.com")

# Get search box element from webElement 'q' using Find Element
search_box = driver.find_element(By.NAME, "q")

search_box.send_keys("webdriver")
```

6.2 Find Elements

与”Find Element” 相似, 但返回的是匹配 WebElement 列表. 要使用列表中的特定 WebElement, 您需要遍历元素列表以对选定元素执行操作.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Firefox()

# Navigate to Url
driver.get("https://www.example.com")

# Get all the elements available with tag name 'p'
elements = driver.find_elements(By.TAG_NAME, 'p')

```

```
for e in elements:
    print(e.text)
```

6.3 Find Element From Element

此方法用于在父元素的上下文中查找子元素. 为此, 父 WebElement 与"findElement" 链接并访问子元素.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Firefox()
driver.get("http://www.google.com")
search_form = driver.find_element_by_tag_name("form")
search_box = search_form.find_element_by.name("q")
search_box.send_keys("webdriver")
```

6.4 Find Elements From Element

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://www.example.com")

# Get element with tag name 'div'
element = driver.find_element(By.TAG_NAME, 'div')

# Get all the elements available with tag name 'p'
elements = element.find_elements(By.TAG_NAME, 'p')
for e in elements:
    print(e.text)
```

6.5 Get Active Element

此方法用于追溯或查找当前页面上下文中具有焦点的 DOM 元素.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

driver = webdriver.Chrome()
driver.get("https://www.google.com")
driver.find_element(By.CSS_SELECTOR, '[name="q"]').send_keys("webElement")

# Get attribute of current active element
attr = driver.switch_to.active_element.get_attribute("title")
print(attr)
```

7 Webelement 定位

7.1 定位一个元素

使用 WebDriver 时要学习一个最基本的技巧是如何在页面上查找元素。WebDriver 提供了许多内置选择器类型，以 id 属性查找元素为例：

```
driver.find_element_by_id("cheddar")
```

如本例所示，定位 WebDriver 中的元素是在 WebDriver 实例对象上完成的。方法 findElement(By) 返回另一个基本对象类型 WebElement.

[*]WebDriver：表示浏览器

[*]WebElement：表示特定的 DOM 节点（控件，例如链接或输入字段等）

一旦你已经找到一个元素的引用，你可以通过对该对象实例使用相同的调用来缩小搜索范围：

```
cheese = driver.find_element_by_id("cheese")
cheddar = cheese.find_elements_by_id("cheddar")
```

你可以这样做是因为，WebDriver 和 WebElement 类型都实现了搜索上下文接口。在 WebDriver 中，这称为基于角色的接口。基于角色的接口允许你确定特定的驱动程序实现是否支持给定的功能。这些接口定义得很清楚，并且尽量只承担单一的功能。你可以阅读更多关于 WebDriver 的设计，以及在 WebDriver 中有哪些角色被支持，在其他被命名的部分。

因此，上面使用的 By 接口也支持许多附加的定位器策略。嵌套查找可能不是最有效的定位 cheese 的策略，因为它需要向浏览器发出两个单独的命令：首先在 DOM 中搜索 id 为“cheese”的元素，然后在较小范围的上下文中搜索“cheddar”。

为了稍微提高性能，我们应该尝试使用一个更具体的定位器：WebDriver 支持通过 CSS 定位器查找元素，我们可以将之前的两个定位器合并到一个搜索里面:

```
cheddar = driver.find_element_by_css_selector("#cheese #cheddar")
```

7.2 定位多个元素

我们正在处理的文本中可能会有一个我们最喜欢的奶酪的订单列表：

```
<ol id=cheese>
 <li id=cheddar>…
 <li id=brie>…
 <li id=rochefort>…
 <li id=camembert>…
```

```
</ul>
```

因为有更多的奶酪无疑是更好的，但是单独检索每一个项目是很麻烦的，检索奶酪的一个更好的方式是使用复数版本 findElements(By) 。此方法返回 web 元素的集合。如果只找到一个元素，它仍然返回 (一个元素的) 集合。如果没有元素被定位器匹配到，它将返回一个空列表。

```
mucho_cheese = driver.find_elements_by_css_selector("#cheese li")
```

7.3 元素选择策略

在 WebDriver 中有 8 种不同的内置元素定位策略：

定位器 Locator	描述
class name	定位 class 属性与搜索值匹配的元素（不允许使用复合类名）
css selector	定位 CSS 选择器匹配的元素
id	定位 id 属性与搜索值匹配的元素
name	定位 name 属性与搜索值匹配的元素
link text	定位 link text 可视文本与搜索值完全匹配的锚元素
partial link text	定位 link text 可视文本部分与搜索值部分匹配的锚点元素。如果匹配多个元素，则只选择第一个元素。
tag name	定位标签名称与搜索值匹配的元素
xpath	定位与 XPath 表达式匹配的元素

7.4 使用选择器的提示

一般来说，如果 HTML 的 id 是可用的、唯一的且是可预测的，那么它就是在页面上定位元素的首选方法。它们的工作速度非常快，可以避免复杂的 DOM 遍历带来的大量处理。

如果没有唯一的 id，那么最好使用写得好的 CSS 选择器来查找元素。XPath 和 CSS 选择器一样好用，但是它语法很复杂，并且经常很难调试。尽管 XPath 选择器非常灵活，但是他们通常未经过浏览器厂商的性能测试，并且运行速度很慢。

基于链接文本和部分链接文本的选择策略有其缺点，即只能对链接元素起作用。此外，它们在 WebDriver 内部调用 XPath 选择器。

标签名可能是一种危险的定位元素的方法。页面上经常出现同一标签的多个元素。这在调用 findElements(By) 方法返回元素集合的时候非常有用。

建议您尽可能保持定位器的紧凑性和可读性。使用 WebDriver 遍历 DOM 结构是一项性能花销很大的操作，搜索范围越小越好。

7.5 相对定位

在 Selenium 4 中带来了相对定位这个新功能，在以前的版本中被称之为” 好友定位 (Firendly Locators)”。它可以帮助你通过某些元素作为参考来定位其附近的元素。现在可用的相对定位有：

[*]above 元素上

[*]below 元素下

[*]toLeftOf 元素左

[*]toRightOf 元素右

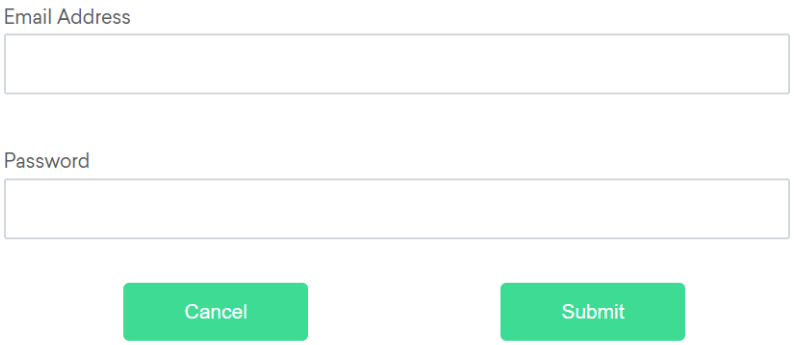
[*]near 附近

findElement 方法现在支持 witTagName() 新方法其可返回 RelativeLocator 相对定位对象。

7.5.1 如何工作

Selenium 是通过使用 JavaScript 函数 getBoundingClientRect() 来查找相对元素的。这个函数能够返回对应元素的各种属性例如：右，左，下，上。

通过下面的例子我们来理解一下关于相对定位的使用:



7.5.2 above() 元素上

返回当前指定元素位置上方的 WebElement 对象

```
#from selenium.webdriver.support.relative_locator import with_tag_name
passwordField = driver.find_element(By.ID, "password")
emailAddressField = driver.find_element(with_tag_name("input").above(passwordField))
```

7.5.3 below() 元素下

返回当前指定元素位置下方的 WebElement 对象

```
#from selenium.webdriver.support.relative_locator import with_tag_name
emailAddressField = driver.find_element(By.ID, "email")
passwordField = driver.find_element(with_tag_name("input").below(emailAddressField))
```

7.5.4 toLeftOf() 元素左

返回当前指定元素位置左方的 WebElement 对象

```
#from selenium.webdriver.support.relative_locator import with_tag_name
submitButton = driver.find_element(By.ID, "submit")
cancelButton = driver.find_element(with_tag_name("button").to_left_of(submitButton))
```

7.5.5 toRightOf() 元素右

返回当前指定元素位置右方的 WebElement 对象

```
#from selenium.webdriver.support.relative_locator import with_tag_name
cancelButton = driver.find_element(By.ID, "cancel")
submitButton = driver.find_element(with_tag_name("button").to_right_of(cancelButton))
```

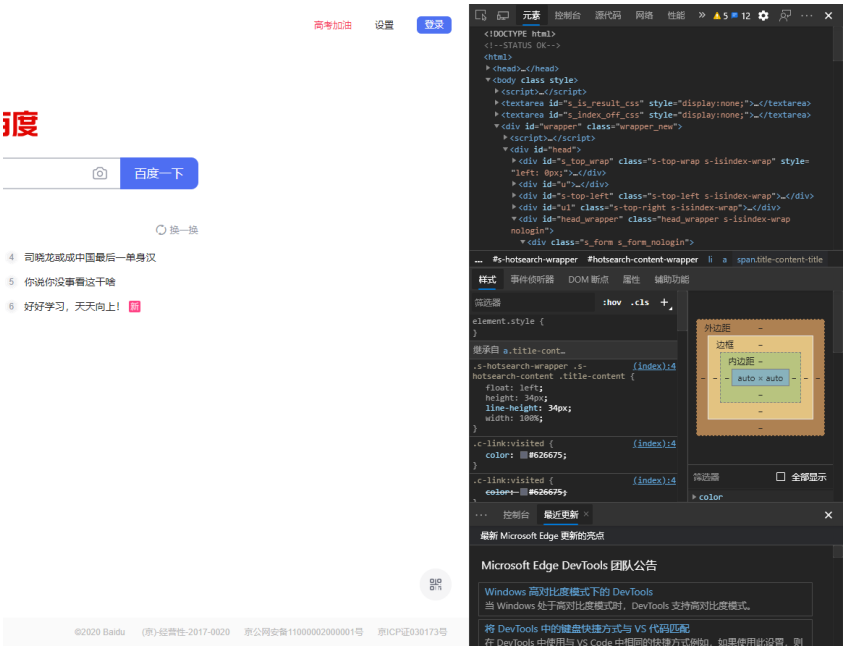
7.5.6 near() 附近

返回当前指定元素位置附近大约 50 像素的 WebElement 对象

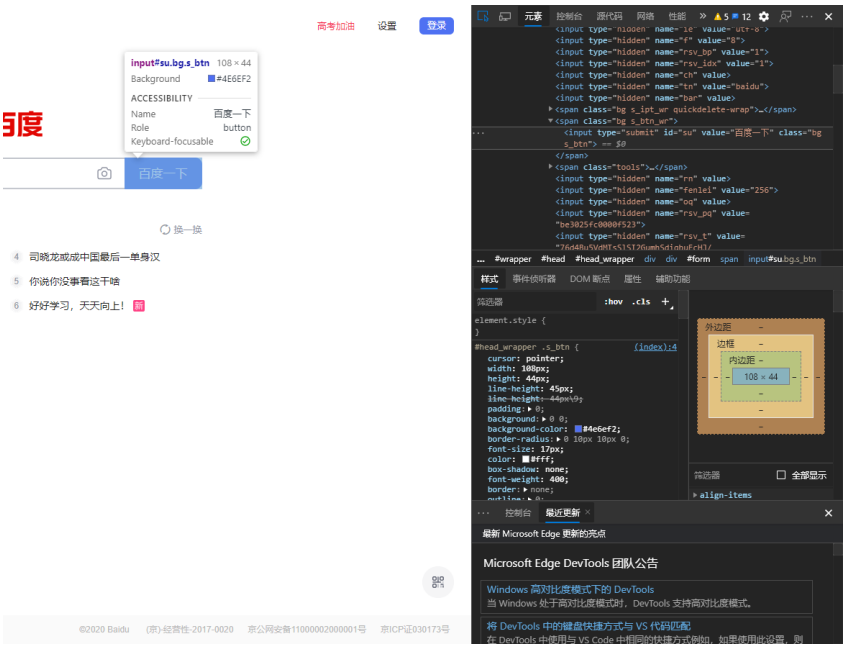
```
#from selenium.webdriver.support.relative_locator import with_tag_name
emailAddressLabel = driver.find_element(By.ID, "lbl-email")
emailAddressField = driver.find_element(with_tag_name("input").near(emailAddressLabel))
```

7.6 查看页面元素

以获取百度的“百度一下”按钮为例（以 Edge 为标准，和 Chrome 操作相同，其他浏览器大体类似）：打开百度首页，按下键盘上的 F12



点击框中左上角的箭头图标，移动鼠标到“百度一下”按钮，就可以自动定位到“百度一下”按钮的 HTML 代码了，查看到按钮的属性，我们可以看到搜索框有 id,name,class 等属性。



7.7 所有查找元素的方法

```
driver.find_element_by_id()
driver.find_element_by_name()
driver.find_element_by_class_name()
driver.find_element_by_tag_name()
driver.find_element_by_link_text()
driver.find_element_by_partial_link_text()
driver.find_element_by_xpath()
driver.find_element_by_css_selector()
```

把 element 变成 elements 就是找所有满足的条件，返回数组。

7.7.1 id 定位

find_element_by_id()

从上面定位到的搜索框属性中，有个 id=”kw” 的属性，我们可以通过这个 id 定位到这个搜索框

代码：

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过id定位搜索框，并输入selenium
driver.find_element_by_id('kw').send_keys('selenium')
# 等待5秒
sleep(5)
# 退出
driver.quit()
```

7.7.2 name 定位

find_element_by_name()

从上面定位到的搜索框属性中，有个 name=”wd” 的属性，我们可以通过这个 name 定位到这个搜索框

代码：

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过name定位搜索框，并输入selenium
driver.find_element_by_name('wd').send_keys('selenium')
# 等待5秒B
sleep(5)
# 退出
driver.quit()
```

7.7.3 class 定位

find_element_by_class_name()

从上面定位到的搜索框属性中，有个 class=”s_ipt” 的属性，我们可以通过这个 class 定位到这个搜索框

代码：

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过class定位搜索框，并输入selenium
driver.find_element_by_class_name('s_ipt').send_keys('selenium')
# 等待5秒
sleep(5)
# 退出
driver.quit()
```

7.7.4 tag 定位

find_element_by_tag_name()

如果懂 HTML 知识，我们就知道 HTML 是通过 tag 来定义功能的, 比如 input 是输入，table 是表格，等等...。每个元素其实就是一个 tag，一个 tag 往往用来定义一类功能，我们查看百度首页的 html 代码，可以看到有很多 div,input,a 等 tag, 所以很难通过 tag 去区分不同的元素。基本上在我们工作中用不到这种定义方法，仅了解就行。下面代码仅做参考，**运行时必定报错**

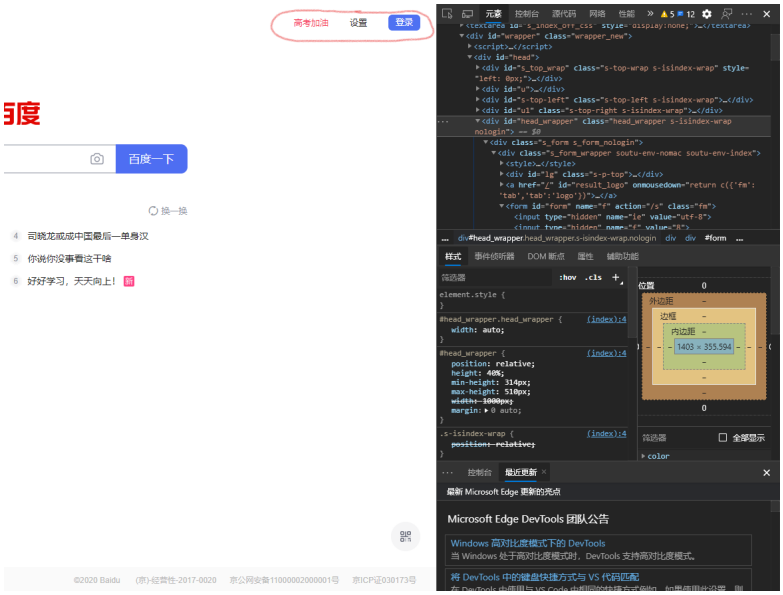
代码：

```
1  from time import sleep
2  from selenium import webdriver
3
4  # 启动浏览器
5  driver = webdriver.Edge()
6  # 打开百度首页
7  driver.get(r'https://www.baidu.com/')
8  # 通过tag定位搜索框，并输入selenium，此处必报错
9  driver.find_element_by_tag_name('input').send_keys('selenium')
10 # 等待5秒
11 sleep(5)
12 # 退出
13 driver.quit()
```

7.7.5 link 定位

find_element_by_link_text()

此种方法是专门用来定位文本链接的，比如百度首页右上角有“高考加油”，“设置”，“登录”等链接



我们来定位“高考加油”这个链接元素

代码：

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过link定位"高考加油"这个链接并点击
driver.find_element_by_link_text('高考加油').click()
# 等待5秒
sleep(5)
# 退出
driver.quit()
```

7.7.6 partial_link 定位

find_element_by_partial_link_text()

有时候一个超链接的文本很长很长，我们如果全部输入，既麻烦，又显得代码很不美观，这时候我们就可以只截取一部分字符串，用这种方法模糊匹配了。

我们用这种方法来定位百度首页的“高考加油”超链接

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过partial_link定位"高考加油"这个链接并点击
driver.find_element_by_partial_link_text('高').click()
# 等待5秒
sleep(5)
# 退出
driver.quit()
```

7.7.7 xpath 定位

find_element_by_xpath()

前面介绍的几种定位方法都是在理想状态下，有一定使用范围的，那就是：在当前页面中，每个元素都有一个唯一的 id 或 name 或 class 或超链接文本的属性，那么我们就可以通过这个唯一的属性值来定位他们。

但是在实际工作中并非有这么美好，有时候我们要定位的元素并没有 id,name,class 属性，或者多个元素的这些属性值都相同，又或者刷新页面，这些属性值都会变化。那么这个时候我们就只能通过 xpath 或者 css 来定位了。

代码：

```
from time import sleep
from selenium import webdriver

# 启动浏览器
driver = webdriver.Edge()
# 打开百度首页
driver.get(r'https://www.baidu.com/')
# 通过xpath定位搜索框，并输入selenium
driver.find_element_by_xpath("//*[@id='kw']").send_keys('selenium')
# 等待5秒
sleep(5)
# 退出
```

```
13 driver.quit()
```

7.7.8 css 定位

find_element_by_css_selector()

这种方法相对 xpath 要简洁些，定位速度也要快些，但是学习起来会比较难理解，这里只做下简单的介绍。

css 定位百度搜索框

```
1  from time import sleep
2  from selenium import webdriver
3
4  # 启动浏览器
5  driver = webdriver.Edge()
6  # 打开百度首页
7  driver.get(r'https://www.baidu.com/')
8  # 通过css定位搜索框，并输入selenium
9  driver.find_element_by_css_selector('#kw').send_keys('selenium')
10 # 等待5秒
11 sleep(5)
12 # 退出
13 driver.quit()
```

8 Webelement 常用方法

8.1 点击和输入

```
driver.find_element_by_id("kw").clear() # 清楚文本
driver.find_element_by_id("kw").send_keys("selenium") # 模拟按键输入
driver.find_element_by_id("su").click() # 单机元素
```

8.2 提交

可以在搜索框模拟回车操作

```
search_text = driver.find_element_by_id('kw') search_text.send_keys('selenium')
search_text.submit()
```

8.3 其他

```
size # 返回元素的尺寸。
text # 获取元素的文本。
get_attribute(name) # 获得属性值。
is_displayed() # 设置该元素是否用户可见。
```

9 键盘事件

Keyboard 代表一个键盘事件. Keyboard 操作通过使用底层接口允许我们向 web 浏览器提供虚拟设备输入.

9.1 sendKeys

即使遇到修饰符键序列, sendKeys 也会在 DOM 元素中键入键序列. 这里是 WebDriver 能够支持的键位列表.

```
1  from selenium import webdriver
2  from selenium.webdriver.common.by import By
3  from selenium.webdriver.common.keys import Keys
4  driver = webdriver.Firefox()
5
6  # Navigate to url
7  driver.get("http://www.google.com")
8
9  # Enter "webdriver" text and perform "ENTER" keyboard action
10 driver.find_element(By.NAME, "q").send_keys("webdriver" + Keys.ENTER)
```

9.2 keyDown

keyDown 用于模拟按下辅助按键 (CONTROL, SHIFT, ALT) 的动作.

```
1  from selenium import webdriver
2  from selenium.webdriver.common.by import By
3  from selenium.webdriver.common.keys import Keys
4  driver = webdriver.Chrome()
5
6  # Navigate to url
7  driver.get("http://www.google.com")
8
9  # Enter "webdriver" text and perform "ENTER" keyboard action
10 driver.find_element(By.NAME, "q").send_keys("webdriver" + Keys.ENTER)
11
12 # Perform action ctrl + A (modifier CONTROL + Alphabet A) to select the page
13 webdriver.ActionChains(driver).key_down(Keys.CONTROL).send_keys("a").perform()
```

9.3 keyUp

keyUp 用于模拟辅助按键 (CONTROL, SHIFT, ALT) 弹起或释放的操作.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
driver = webdriver.Chrome()

# Navigate to url
driver.get("http://www.google.com")

# Store google search box WebElement
search = driver.find_element(By.NAME, "q")

action = webdriver.ActionChains(driver)

# Enters text "qwerty" with keyDown SHIFT key and after keyUp SHIFT key (QWERTYqwerty)
action.key_down(Keys.SHIFT).send_keys_to_element(search, "qwerty").key_up(Keys.SHIFT).send_keys("qwerty").
    perform()
```

9.4 clear

清除可编辑元素的内容. 这仅适用于可编辑且可交互的元素, 否则 Selenium 将返回错误 (无效的元素状态或元素不可交互).

```
from selenium import webdriver
from selenium.webdriver.common.by import By
driver = webdriver.Chrome()

# Navigate to url
driver.get("http://www.google.com")
# Store 'SearchInput' element
SearchInput = driver.find_element(By.NAME, "q")
SearchInput.send_keys("selenium")
# Clears the entered text
SearchInput.clear()
```

9.5 常用的键盘操作

```
send_keys(Keys.BACK_SPACE) # 删除键（BackSpace）
send_keys(Keys.SPACE) # 空格键(Space)
send_keys(Keys.TAB) # 制表键(Tab)
send_keys(Keys.ESCAPE) # 回退键（Esc）
send_keys(Keys.ENTER) # 回车键（Enter）
send_keys(Keys.CONTROL,'a') # 全选（Ctrl+A）
send_keys(Keys.CONTROL,'c') # 复制（Ctrl+C）
send_keys(Keys.CONTROL,'x') # 剪切（Ctrl+X）
send_keys(Keys.CONTROL,'v') # 粘贴（Ctrl+V）
send_keys(Keys.F1) # 键盘 F1
……
send_keys(Keys.F12) # 键盘 F12
```

```
# 输入框输入内容
driver.find_element_by_id("kw").send_keys("seleniumm")

# 删除多输入的一个 m
driver.find_element_by_id("kw").send_keys(Keys.BACK_SPACE)
```

10 在 AUT* 中执行

*AUT: 待测试应用

使用 sendKeys 方法设置元素的文本，如下所示:

```
name = "Charles"
driver.find_element(By.NAME, "name").send_keys(name)
```

一些 web 应用程序使用 JavaScript 库来实现拖放功能。下面是一个简单的例子，拖动一个元素到另一个元素:

```
source = driver.find_element_by_id("source")
target = driver.find_element_by_id("target")
ActionChains(driver).drag_and_drop(source, target).perform()
```

10.1 单击元素

你可以使用 click 方法单击一个元素:

```
driver.find_element_by_css_selector("input[type='submit']").click()
```

11 鼠标操作

在 WebDriver 中，将这些关于鼠标操作的方法封装在 ActionChains 类提供。

ActionChains 类提供了鼠标操作的常用方法：

```
perform() # 执行所有 ActionChains 中存储的行为
context_click() # 右击
double_click() # 双击
drag_and_drop() # 拖动
move_to_element() # 鼠标悬停
```

举个例子：

```
from selenium import webdriver
# 引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Chrome()
driver.get("https://www.baidu.cn")

# 定位到要悬停的元素
above = driver.find_element_by_link_text("设置")
# 对定位到的元素执行鼠标悬停操作
ActionChains(driver).move_to_element(above).perform()
```

12 浏览器导航

12.1 导航到某个网页

```
1    driver.get('https://www.github.com')
```

12.2 获取当前 URL

```
1 driver.current_url
```

12.3 浏览器后退，前进

```
1 driver.back() # 后退
2 driver.forward() # 前进
```

12.4 刷新

```
1 driver.refresh() # 刷新
```

12.5 获取当前网页标题

```
1 driver.title
```

13 窗口和标签页

13.1 获取窗口句柄

WebDriver 没有区分窗口和标签页。如果你的站点打开了一个新标签页或窗口，Selenium 将允许您使用窗口句柄来处理它。每个窗口都有一个唯一的标识符，该标识符在单个会话中保持持久性。你可以使用以下方法获得当前窗口的窗口句柄:

```
driver.current_window_handle
```

13.2 切换窗口或标签页

单击在 https://seleniumhq.github.io 中打开链接，则屏幕会聚焦在新窗口或新标签页上，但 WebDriver 不知道操作系统认为哪个窗口是活动的。要使用新窗口，您需要切换到它。如果只有两个选项卡或窗口被打开，并且你知道从哪个窗口开始，则你可以遍历 WebDriver，通过排除法可以看到两个窗口或选项卡，然后切换到你需要的窗口或选项卡。

不过，Selenium 4 提供了一个新的 api NewWindow ，它创建一个新选项卡 (或) 新窗口并自动切换到它。

（官方的演示代码因网页问题无法使用，我略作修改）

```
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# 启动驱动程序
with webdriver.Edge() as driver:
    # 打开网址 （百度）
    driver.get("https://www.baidu.com/")

    # 设置等待
    wait = WebDriverWait(driver, 10)

    # 存储原始窗口的 ID
    original_window = driver.current_window_handle

    # 检查一下，我们还没有打开其他的窗口
    assert len(driver.window_handles) == 1

    # 单击在新窗口中打开的链接（hao123）
```

```
    driver.find_element_by_link_text("hao123").click()

    # 等待新窗口或标签页
    wait.until(EC.number_of_windows_to_be(2))

    # 循环执行，直到找到一个新的窗口句柄
    for window_handle in driver.window_handles:
        if window_handle != original_window:
            driver.switch_to.window(window_handle)
            break

    # 等待新标签页完成加载内容
    wait.until(EC.title_is("hao123_上网从这里开始"))

# 上述内容完成后，浏览器会自动退出。
```

13.2.1 切换窗口

```
driver.switch_to_window("windowName")
```

13.2.2 创建新窗口或新标签页并且切换

创建一个新窗口或标签页，屏幕焦点将聚焦在新窗口或标签在上。不需要切换到新窗口或标签页。如果除了新窗口之外，打开了两个以上的窗口或标签页，可以通过遍历 WebDriver 看到两个窗口或选项卡，并切换到非原始窗口。

注意: 该特性适用于 Selenium 4 及其后续版本。

```
# 打开新标签页并切换到新标签页
driver.switch_to.new_window('tab')

# 打开一个新窗口并切换到新窗口
driver.switch_to.new_window('window')
```

13.3 关闭窗口或标签页

当使用完窗口或标签页，并且他**不是浏览器打开的最后一个窗口或标签页**时，应该关闭它并切换回以前使用的窗口。假设您遵循了前一节中的代码示例，您将把前一个窗口句柄存储在一个变量中。把它们放在一起，我们得到：

```
#关闭标签页或窗口
driver.close()

#切回到之前的标签页或窗口
driver.switch_to.window(original_window)
```

如果在关闭一个窗口后忘记切换回另一个窗口句柄，WebDriver 将在当前关闭的页面上执行，并触发一个 No Such Window Exception 无此窗口异常。必须切换回有效的窗口句柄才能继续执行。

13.4 在会话结束时退出浏览器

在完成浏览器会话后，应该调用退出 (quit)，而不是关闭 (close)：

```
driver.close() # 关闭单个窗口
driver.quit() # 关闭所有窗口
```

退出 (quit) 将：

[*] 关闭所有与 WebDriver 会话相关的窗口和标签页

[*] 结束浏览器进程

[*] 结束后台驱动进程

[*] 通知 Selenium Grid 浏览器不再使用，以便可以由另一个会话使用它 (如果您正在使用 Selenium Grid)

调用 quit() 失败将留下额外的后台进程和端口运行在机器上，这可能在以后导致一些问题。

有的测试框架提供了一些方法和注释，您可以在测试结束时放入 teardown() 方法中。

```
# unittest teardown
# https://docs.python.org/3/library/unittest.html?highlight=teardown#unittest.TestCase.tearDown
def tearDown(self):
    self.driver.quit()
```

如果不在测试上下文中运行 WebDriver，您可以考虑使用 try / finally，这是大多数语言都提供的，这样一个异常处理仍然可以清理 WebDriver 会话。

```
try:
    #WebDriver code here...
finally:
    driver.quit()
```

Python 的 WebDriver 现在支持 Python 上下文管理器，当使用 with 关键字时，可以在执行结束时自动退出驱动程序。6.2 中使用的案例就使用了这个方法。

```
with webdriver.Firefox() as driver:
  # WebDriver 代码…

# 在此缩进位置后 WebDriver 会自动退出
```

14 窗口管理

窗口分辨率会影响 Web 应用的呈现方式，因此 WebDriver 提供了移动和调整浏览器窗口的机制。

14.1 获取窗口大小

以像素为单位获取浏览器窗口的大小。

```
# 分别获取每个尺寸
width = driver.get_window_size().get("width")
height = driver.get_window_size().get("height")

# 或者存储尺寸并在以后查询它们
size = driver.get_window_size()
width1 = size.get("width")
height1 = size.get("height")
```

14.2 设置窗口大小

恢复窗口并设置窗口大小。

```
driver.set_window_size(1024, 768)
```

14.3 获取窗口位置

获取浏览器窗口左上角的坐标。

```
# 分别获取每个尺寸
x = driver.get_window_position().get('x')
y = driver.get_window_position().get('y')

# 或者存储尺寸并在以后查询它们
position = driver.get_window_position()
x1 = position.get('x')
y1 = position.get('y')
```

15 设置窗口位置

将窗口移动到设定的位置。

```
# 将窗口移动到主显示器的左上角
driver.set_window_position(0, 0)
```

15.1 最大化窗口

放大窗口。对于大多数操作系统，窗口将填满屏幕，而不会挡住操作系统自己的菜单和工具栏。

```
driver.maximize_window()
```

15.2 最小化窗口

最小化当前浏览上下文的窗口。此命令的确切行为特定于各个窗口管理器。

最小化窗口通常将窗口隐藏在系统托盘中。

注意：此功能适用于 Selenium 4 和更晚的版本。

```
river.minimize_window()
```

15.3 全屏窗口

填充整个屏幕，类似于在大多数浏览器中按 F11.

```
driver.fullscreen_window()
```

16 窗口截图

```
1 driver.get_screenshot_as_file("D:\\baidu_img.jpg")
2 # 截取当前窗口，并指定截图图片的保存位置
```

17 框架 (frames) 和内嵌框架 (iframe)

框架 (frame) 是一种现在已被弃用的方法，用于从同一域中的多个文档构建站点布局。除非你使用的是 HTML5 之前的 webapp，否则你不太可能与他们合作。内嵌框架 (iframe) 允许插入来自完全不同领域的文档，并且仍然经常使用。

如果您需要使用框架或 iframe, WebDriver 允许您以相同的方式使用它们。考虑 iframe 中的一个按钮。如果我们使用浏览器开发工具检查元素，我们可能会看到以下内容:

```
<div id="modal">
  <iframe id="buttonframe" name="myframe"  src="https://seleniumhq.github.io">
   <button>Click here</button>
 </iframe>
</div>
```

如果不是 iframe，我们系统使用类似下面的操作单击按钮：

```
# 事实上这不会工作，下面有解释
driver.find_element_by_tag_name('button').click()
```

但是，如果 iframe 之外没有按钮，那么您可能会得到一个 no such element 无此元素 的错误。这是因为 Selenium 只知道顶层文档中的元素。为了与按钮进行交互，我们需要首先切换到框架，这与切换窗口的方式类似。WebDriver 提供了三种切换到框架的方法。

17.1 使用 WebElement

使用 WebElement 进行切换是最灵活的选择。您可以使用首选的选择器找到框架并切换到它。

```
# 存储网页元素
iframe = driver.find_element_by_css_selector("#modal > iframe")

# 切换到选择的 iframe
driver.switch_to.frame(iframe)

# 单击按钮
driver.find_element_by_tag_name('button').click()
```

17.2 使用 name 或 id

如果 frames 或 iframe 具有 name 或 id 属性，则可以使用此项。如果 name 或 id 在页面上不唯一，则找到的第一个 name 或 id 将被切换到。

```
# 通过 id 切换框架
driver.switch_to.frame('buttonframe')

# 单击按钮
driver.find_element_by_tag_name('button').click()
```

17.3 使用索引 (index)

使用 frame 的 index，例如：使用 JaveScript 中的 Window.frame 进行查询。

```
# 切换到第 2 个框架
driver.switch_to.frame(1)
```

17.4 留下框架

离开 iframe 或 frameset，切换回默认内容，如下所示:

```
# 切回到默认内容
driver.switch_to.default_content()
```

18 页面加载策略

定义当前会话的页面加载策略. 默认情况下, 当 Selenium WebDriver 加载页面时, 遵循 normal 的页面加载策略. 始终建议在页面加载缓慢时, 停止下载其他资源 (例如图片, css, js) .

document.readyState 属性描述当前页面的加载状态. 默认情况下, 在页面就绪状态是 complete 之前, WebDriver 都将延迟 driver.get() 的响应或 driver.navigate().to() 的调用.

在单页应用程序中 (例如 Angular, react, Ember) , 一旦动态内容加载完毕 (即 pageLoadStrategy 状态为 COMPLETE) , 则点击链接或在页面内执行某些操作的行为将不会向服务器发出新请求, 因为内容在客户端动态加载, 无需刷新页面.

单页应用程序可以动态加载许多视图, 而无需任何服务器请求, 因此页面加载策略将始终显示为 COMPLETE 的状态, 直到我们执行新的 driver.get() 或 driver.navigate().to() 为止.

WebDriver 的页面加载策略支持以下内容:

18.1 normal

此配置使 Selenium WebDriver 等待整个页面的加载. 设置为 normal 时, Selenium WebDriver 将保持等待, 直到返回 load 事件

默认情况下, 如果未设置页面加载策略, 则设置 normal 为初始策略.

```
1 from selenium import webdriver
2 from selenium.webdriver.chrome.options import Options
3 options = Options()
4 options.page_load_strategy = 'normal'
5 driver = webdriver.Chrome(options=options)
6 # Navigate to url
7 driver.get("http://www.bing.com")
8 driver.quit()
```

18.2 eager

这将使 Selenium WebDriver 保持等待, 直到完全加载并解析了 HTML 文档, 该策略无关样式表, 图片和 subframes 的加载.

设置为 eager 时, Selenium WebDriver 保持等待, 直至返回 DOMContentLoaded 事件.

```
1 from selenium import webdriver
2 from selenium.webdriver.chrome.options import Options
3 options = Options()
4 options.page_load_strategy = 'eager'
```

```
driver = webdriver.Chrome(options=options)
# Navigate to url
driver.get("http://www.bing.com")
driver.quit()
```

18.3 none

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
options = Options()
options.page_load_strategy = 'none'
driver = webdriver.Chrome(options=options)
# Navigate to url
driver.get("http://www.bing.com")
driver.quit()
```

19 等待

WebDriver 通常可以说有一个阻塞 API(Blocking API)。因为它是一个指示浏览器做什么的进程外库，而且 web 平台本质上是异步的，所以 WebDriver 不跟踪 DOM 的实时活动状态。这伴随着一些我们将在这里讨论的挑战。

根据经验，大多数由于使用 Selenium 和 WebDriver 而产生的间歇性 (intermittent) 问题都与浏览器和用户指令之间的竞争条件有关。例如，用户指示浏览器导航到一个页面，然后在试图查找元素时得到一个 no such element 的错误。

考虑下面的文档：

```
<!doctype html>
<meta charset=utf-8>
<title>Race Condition Example</title>

<script>
  var initialised = false;
  window.addEventListener("load", function() {
    var newElement = document.createElement("p");
    newElement.textContent = "Hello from JavaScript!";
    document.body.appendChild(newElement);
    initialised = true;
  });
</script>
```

例如下面的这个 WebDrive 指令可能看起来很简单而且很正确：

```
driver.navigate("file:///race_condition.html")
el = driver.find_element(By.TAG_NAME, "p")
assert el.text == "Hello from JavaScript!"
```

但实际上并不如此，这里的问题是 WebDriver 中使用的默认页面加载策略听从 document.readyState 在返回调用 navigate 之前将状态改为”complete” 。因为 p 元素是在文档完成加载之后添加的，所以这个 WebDriver 脚本可能是间歇性 (intermittent) 的。它“可能”间歇性 (intermittent) 是因为无法做出保证说异步触发这些元素或事件不需要显式等待或阻塞这些事件。

幸运的是，WebElement 接口上可用的正常指令集——例如 WebElement.click 和 WebElement.sendKeys —是保证同步的，因为直到命令在浏览器中被完成之前函数调用是不会返回的 (或者回调是不会在回调形式的语言中触发的)。高级用户交互 APIs, 键盘和鼠标是例外的，因为它们被明确地设计为“按我说的做”的异步命令。

等待是在继续下一步之前会执行一个自动化任务来消耗一定的时间。

为了克服浏览器和 WebDriver 脚本之间的竞争问题，大多数 Selenium 客户都附带了一个 wait 包。在使用等待时，您使用的是通常所说的显式等待。

19.1 显示等待

显式等待使 WebdDriver 等待某个条件成立时继续执行，它允许代码停止执行或冻结线程，直到条件成立。换句话说，只要条件的返回值是 false，他将继续尝试和等待。直到在达到最大时长时抛出超时异常（TimeoutException）。

由于显式等待允许您等待条件的发生，所以它们非常适合在浏览器及其 DOM 和 WebDriver 脚本之间同步状态。

对于之前的错误代码，我们可以使用显式等待，让 findElement 调用等待，直到脚本中动态添加的元素被添加到 DOM 中:

```
1 from selenium.webdriver.support.ui import WebDriverWait
2 def document_initialised(driver):
3     return driver.execute_script("return initialised")
4
5 driver.navigate("file:///race_condition.html")
6 WebDriverWait(driver).until(document_initialised)
7 el = driver.find_element(By.TAG_NAME, "p")
8 assert el.text == "Hello from JavaScript!"
```

我们将条件作为函数引用传递，等待将会重复运行直到其返回值为 true。“truthful”返回值是在当前语言中计算为 boolean true 的任何值，例如字符串、数字、boolean、对象 (包括 WebElement) 或填充 (非空) 的序列或列表。这意味着空列表的计算结果为 false。当条件为 true 且阻塞等待终止时，条件的返回值将成为等待的返回值。

有了这些知识，并且因为等待实用程序默认情况下会忽略 no such element 的错误，所以我们可以重构我们的指令使其更简洁:

```
1 from selenium.webdriver.support.ui import WebDriverWait
2
3 driver.navigate("file:///race_condition.html")
4 el = WebDriverWait(driver).until(lambda d: d.find_element_by_tag_name("p"))
5 assert el.text == "Hello from JavaScript!"
```

在这个示例中，我们传递了一个匿名函数 (但是我们也可以像前面那样显式地定义它，以便重用它)。传递给我们条件的第一个，也是唯一的一个参数始终是对驱动程序对象 WebDriver 的引用 (在本例中称为 d)。在多线程环境中，您应该小心操作传入条件的驱动程序引用，而不是外部范围中对驱动程序的引用。

因为等待将会吞没在没有找到元素时引发的 no such element 的错误，这个条件会一直重试直到找到元素为止。然后它将获取一个 WebElement 的返回值，并将其传递回我们的脚本。

如果条件失败，例如从未得到条件为真实的返回值，等待将会抛出/引发一个叫 timeout error 的错误/异常。

19.1.1 选项

等待条件可以根据您的需要进行定制。有时候是没有必要等待缺省超时的全部范围，因为没有达到成功条件的代价可能很高。

等待允许你传入一个参数来覆盖超时（超时重试次数）:

```
1 WebDriverWait(driver, timeout=3).until(some_condition)
```

19.1.2 预期的条件

由于必须同步 DOM 和指令是相当常见的情况，所以大多数客户端还附带一组预定义的预期条件。顾名思义，它们是为频繁等待操作预定义的条件。

不同的语言绑定提供的条件各不相同，但这只是其中一些:

[*]alert is present

[*]element exists

[*]element is visible

[*]title contains

[*]title is

[*]element staleness

[*]visible text

可以参考每个客户端绑定的 API 文档，以找到期望条件的详尽列表（官方文档提供）:

Java’s org.openqa.selenium.support.ui.ExpectedConditions class

Python’s selenium.webdriver.support.expected_conditions class

.NET’s OpenQA.Selenium.Support.UI.ExpectedConditions type

19.1.3 经验

```
1 from selenium import webdriver
2 from selenium.webdriver.common.by import By
3 from selenium.webdriver.support.ui import WebDriverWait
4 from selenium.webdriver.support import expected_conditions as EC
```

```
5
6  driver = webdriver.Firefox()
7  driver.get("http://www.baidu.com")
8
9  element = WebDriverWait(driver, 5, 0.5).until(
10                       EC.presence_of_element_located((By.ID, "kw"))
11                       )
12 element.send_keys('selenium')
13 driver.quit()
```

WebDriverWait 类是由 WebDirver 提供的等待方法。在设置时间内，默认每隔一段时间检测一次当前页面元素是否存在，如果超过设置时间检测不到则抛出异常。具体格式如下：

```
1  WebDriverWait(driver, timeout, poll_frequency=0.5, ignored_exceptions=None)
```

```
1  driver # 浏览器驱动。
2  timeout # 最长超时时间，默认以秒为单位。
3  poll_frequency # 检测的间隔（步长）时间，默认为0.5S。
4  ignored_exceptions # 超时后的异常信息，默认情况下抛NoSuchElementException异常。
```

WebDriverWait() 一般由 until() 或 until_not() 方法配合使用，下面是 until() 和 until_not() 方法的说明。

```
1  until(method, message= '' ) 调用该方法提供的驱动程序作为一个参数，直到返回值为True。
2  until_not(method, message= '' ) 调用该方法提供的驱动程序作为一个参数，直到返回值为False。
```

在本例中,通过 as 关键字将 expected_conditions 重命名为 EC,并调用 presence_of_element_located() 方法判断元素是否存在。

19.2 隐式等待

还有第二种区别于显示等待类型的隐式等待。通过隐式等待，WebDriver 在试图查找**任何**元素时在一定时间内轮询 DOM。当网页上的某些元素不是立即可用并且需要一些时间来加载时是很有用的。

默认情况下隐式等待元素出现是禁用的，它需要在单个会话的基础上手动启用。将显式等待和隐式等待混合在一起会导致意想不到的结果，就是说即使元素可用或条件为真也要等待睡眠的最长时间。

警告：不要混合使用隐式和显式等待。这样做会导致不可预测的等待时间。例如，将隐式等待设置为 10 秒，将显式等待设置为 15 秒，可能会导致在 20 秒后发生超时。

隐式等待是告诉 WebDriver 如果在查找一个或多个不是立即可用的元素时轮询 DOM 一段时间。默认设置为 0，表示禁用。一旦设置好，隐式等待就被设置为会话的生命周期。

```
driver = Firefox()
driver.implicitly_wait(10)
driver.get("http://somedomain/url_that_delays_loading")
my_dynamic_element = driver.find_element(By.ID, "myDynamicElement")
```

19.2.1 经验

如果某些元素不是立即可用的，隐式等待是告诉 WebDriver 去等待一定的时间后去查找元素。默认等待时间是 0 秒，一旦设置该值，隐式等待是设置该 WebDriver 的实例的生命周期。

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.implicitly_wait(10) # seconds
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element_by_id("myDynamicElement")
```

19.3 流畅等待

流畅等待实例定义了等待条件的最大时间量，以及检查条件的频率。

用户可以配置等待来忽略等待时出现的特定类型的异常，例如在页面上搜索元素时出现的 NoSuchElementException。

```
driver = Firefox()
driver.get("http://somedomain/url_that_delays_loading")
wait = WebDriverWait(driver, 10, poll_frequency=1, ignored_exceptions=[ElementNotVisibleException,
    ElementNotSelectableException])
element = wait.until(EC.element_to_be_clickable((By.XPATH, "//div")))
```

20 支持的类 (Support classes)

WebDriver 提供了一些用于简化代码维护的类. 其提供了一种不错的抽象, 使得将 HTML 元素建模为域对象的操作, 变得更加容易, 还提供了一些更有帮助的方法, 使用此类对象时更容易操作. 我们将学到以下方法:

[*]Locator Strategies

[*]Events

[*]LoadableComponent

[*]ThreadGuard

[*]etc.

这部分官方文档缺失，仅给出了 ThreadGuard，Java Only。

21 JavaScript 警告框, 提示框和确认框

WebDriver 提供了一个 API, 用于处理 JavaScript 提供的三种类型的原生弹窗消息. 这些弹窗由浏览器提供限定的样式.

21.1 Alerts 警告框

其中最基本的称为警告框, 它显示一条自定义消息, 以及一个用于关闭该警告的按钮, 在大多数浏览器中标记为" 确定"(OK). 在大多数浏览器中, 也可以通过按" 关闭"(close) 按钮将其关闭, 但这始终与"确定"按钮具有相同的作用.

WebDriver 可以从弹窗获取文本并接受或关闭这些警告.

```
# Click the link to activate the alert
driver.find_element_by_link_text("See an example alert").click()

# Wait for the alert to be displayed and store it in a variable
alert = wait.until(expected_conditions.alert_is_present())

# Store the alert text in a variable
text = alert.text

# Press the OK button
alert.accept()
```

21.2 Confirm 确认框

确认框类似于警告框, 不同之处在于用户还可以选择取消消息. 查看样例确认框.

此示例还呈现了警告的另一种实现:

```
# Click the link to activate the alert
driver.find_element_by_link_text("See a sample confirm").click()

# Wait for the alert to be displayed
wait.until(expected_conditions.alert_is_present())

# Store the alert in a variable for reuse
alert = driver.switch_to.alert

# Store the alert text in a variable
```

```
text = alert.text

# Press the Cancel button
alert.dismiss()
```

21.3 Prompt 提示框

提示框与确认框相似, 不同之处在于它们还包括文本输入. 与处理表单元素类似, 您可以使用 WebDriver 的 sendKeys 来填写响应. 这将完全替换占位符文本. 按下取消按钮将不会提交任何文本. 查看样例提示框.

```
# Click the link to activate the alert
driver.find_element_by_link_text("See a sample prompt").click()

# Wait for the alert to be displayed
wait.until(expected_conditions.alert_is_present())

# Store the alert in a variable for reuse
alert = Alert(driver)

# Type your message
alert.send_keys("Selenium")

# Press the OK button
alert.accept()
```

21.4 经验：警告框处理

```
alert = driver.switch_to_alert()
```

```
text # 返回 alert/confirm/prompt 中的文字信息。
accept() # 接受现有警告框。
dismiss() # 解散现有警告框。
send_keys(keysToSend) # 发送文本至警告框。keysToSend: 将文本发送至警告框。
```

22 调用 JavaScript 代码

```
1 js="window.scrollTo(100,450);"
2 driver.execute_script(js) # 通过javascript设置浏览器窗口的滚动条位置
```

通过 execute_script() 方法执行 JavaScripts 代码来移动滚动条的位置。

23 Http 代理

代理服务器充当客户端和服务器之间的请求中介. 简述而言, 流量将通过代理服务器流向您请求的地址, 然后返回.

使用代理服务器用于 Selenium 的自动化脚本, 可能对以下方面有益:

[*] 捕获网络流量 [*] 模拟网站后端响应 [*] 在复杂的网络拓扑结构或严格的公司限制/政策下访问目标站点.

如果您在公司环境中, 并且浏览器无法连接到 URL, 则最有可能是因为环境, 需要借助代理进行访问.

Selenium WebDriver 提供了如下设置代理的方法:

```
from selenium import webdriver

PROXY = "<HOST:PORT>"
webdriver.DesiredCapabilities.FIREFOX['proxy'] = {
    "httpProxy": PROXY,
    "ftpProxy": PROXY,
    "sslProxy": PROXY,
    "proxyType": "MANUAL",

}

with webdriver.Firefox() as driver:
    # Open URL
    driver.get("https://selenium.dev")
```

24 获取断言信息

```
1 title = driver.title # 打印当前页面title
2 now_url = driver.current_url # 打印当前页面URL
3 user = driver.find_element_by_class_name('nums').text # 获取结果数目
```

25 下拉框选择

```
1 from selenium import webdriver
2 from selenium.webdriver.support.select import Select
3 from time import sleep
4
5 driver = webdriver.Chrome()
6 driver.implicitly_wait(10)
7 driver.get('http://www.baidu.com')
8 sel = driver.find_element_by_xpath("//select[@id='nr']")
9 Select(sel).select_by_value('50')  # 显示50条
```

26 文件上传

```
1 driver.find_element_by_name("file").send_keys('D:\\upload_file.txt')
2 # 定位上传按钮，添加本地文件
```

27 cookie 操作

WebDriver 操作 cookie 的方法：

```
1 get_cookies() # 获得所有cookie信息。
2 get_cookie(name) # 返回字典的key为“name”的cookie信息。
3 add_cookie(cookie_dict) # 添加cookie。“cookie_dict”指字典对象，必须有name 和value 值。
4 delete_cookie(name,optionsString)
5 # 删除cookie信息。“name”是要删除的cookie的名称，“optionsString”是该cookie的选项，
6 # 目前支持的选项包括“路径”，“域”。
7 delete_all_cookies() # 删除所有cookie信息
```

28 附

这里的内容，对初学者 or 零基础菜鸟很有用

28.1 延时效果（time 库）

28.1.1 time 库基本介绍

概述：time 库是 Python 中处理时间的标准库

```
import time
time.<b>()
```

time 库包含三类函数：

```
time()   ctime()   gmtime() # 时间获取
strftime()   strptime() # 时间格式化
sleep()   perf_counter() # 程序计时
```

28.1.2 时间获取

```
time()
获取当前时间戳，即计算机内部时间值，浮点数
>>>time.time()
1516939876.6022282

ctime()
获取当前时间并以易读方式表示，返回字符串
>>>time.ctime()
'Fri Jan 26 12:11:16 2018'

gmtime()
获取当前时间，表示为计算机可处理的时间格式
>>>time.gmtime()
time.struct_time(tm_year=2018, tm_mon=1,
tm_mday=26, tm_hour=4, tm_min=11, tm_sec=16,
tm_wday=4, tm_yday=26, tm_isdst=0)
```

28.1.3 时间格式化

```
strftime(tpl, ts)
tpl是格式化模板字符串，用来定义输出效果
ts是计算机内部时间类型变量
>>>t = time.gmtime()
>>>time.strftime("%Y-%m-%d %H:%M:%S",t)
'2018-01-26 12:55:20'
```

格式化控制符

格式化字符串	日期/时间说明	值范围和实例
%Y	年份	0000~9999，例如 1900
%m	月份	01~12，例如：10
%B	月份名称	January~December，例如：April
%b	月份名称缩写	Jan~Dec，例如：Apr
%d	日期/时间说明	01~31，例如：25
%A	星期	Monday~Sunday，例如：Wednesday
%a	星期缩写	Mon~Sun，例如：Wed
%H	小时（24h 制）	00~23，例如：12
%h	小时（12h 制）	01~12，例如：7
%p	上/下午	AM，PM，例如：PM
%M	分钟	00~59，例如：26
%s	秒	00~59，例如：26

```
strptime(str, tpl)
str是字符串形式的时间值
tpl是格式化模板字符串，用来定义输入效果
>>>timeStr = '2018-01-26 12:55:20'
>>>time.strptime(timeStr, "%Y-%m-%d %H:%M:%S")
time.struct_time(tm_year=2018, tm_mon=1,
tm_mday=26, tm_hour=4, tm_min=11, tm_sec=16,
tm_wday=4, tm_yday=26, tm_isdst=0)
```

```
>>>t = time.gmtime()
>>>time.strftime("%Y-%m-%d %H:%M:%S",t)

```

```
>>>timeStr = '2018-01-26 12:55:20'
>>>time.strptime(timeStr, “%Y-%m-%d %H:%M:%S” )
```

28.1.4 程序计时应用

```
perf_counter()
返回一个CPU级别的精确时间计数值，单位为秒
由于这个计数值起点不确定，连续调用差值才有意义
>>>start = time.perf_counter()
318.66599499718114
>>>end = time.perf_counter()
341.3905185375658
>>>end - start
22.724523540384666
```

```
sleep(s)
s拟休眠的时间，单位是秒，可以是浮点数
>>>def wait():
time.sleep(3.3)
>>>wait() #程序将等待3.3秒后再退出
```

29 个人作品

29.1 平安堡自动撕布机(问卷星自动填写)

```python
from selenium import webdriver

import time

# 获取问卷星地址
a = input("请输入今日问卷星网址: ")

# 输出网址供确认
print("今日问卷星网址: ",a)

# 打开浏览器
driver = webdriver.Edge()

# 进入网址
driver.get(a)

# 根据路径找到按钮,并模拟进行点击
driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[1]/div[2]').click()

# 延迟防止加载不全
time.sleep(10)

# 地图是内嵌网页，确定它的iframe
iframe=driver.find_element_by_id("yz_popwinIframe")

# 进入iframe
driver.switch_to.frame(iframe)

# 根据class找到按钮,并模拟进行点击“确定”按钮
driver.find_element_by_xpath('/html/body/div/div[4]/div[1]/a').click()

# 根据路径找到按钮,并模拟进行点击
driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[2]/div[2]/div[1]').click()

# 根据路径找到按钮,并模拟进行点击
```

```
36 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[3]/div[2]/div[2]').click()
37
38 # 根据路径找到文本框,并模拟进行输入
39 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[4]/div[2]/input').send_keys("36")
40
41 # 根据路径找到文本框,并模拟进行输入
42 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[5]/div[2]/input').send_keys("司晓
       龙")
43
44 # 根据路径找到按钮,并模拟进行点击
45 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[6]/div[2]/div[2]/span/a').click()
46
47 # 根据路径找到按钮,并模拟进行点击
48 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[7]/div[2]/div[2]/span/a').click()
49
50 # 根据路径找到按钮,并模拟进行点击
51 driver.find_element_by_xpath('/html/body/form/div[5]/div[2]/fieldset/div[8]/div[2]/div[2]/span/a').click()
52
53 # 根据路径找到按钮,并模拟进行点击
54 driver.find_element_by_xpath('/html/body/form/div[5]/div[6]/div[3]/div[1]/div/div').click()
55
56 # 关闭浏览器
57 driver.quit()
```