selenium库

参考链接

Java

通过maven安装依赖，pom.xml中插入下面的代码

```xml
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.X</version>
</dependency>
```

如果只在指定浏览器设置，例如火狐，则如下

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-firefox-driver</artifactId>
  <version>3.X</version>
</dependency>
```

Browser manipulation

Browser navigation

Navigate to

The first thing you will want to do after launching a browser is to open your website. This can be achieved in a single line:
启动浏览器后你要做的第一件事就是打开你的网站。这可以通过一行代码实现:

```
//Convenient
driver.get("https://selenium.dev");

//Longer way
driver.navigate().to("https://selenium.dev");
```

Get current URL

You can read the current URL from the browser's address bar using:
您可以从浏览器的地址栏读取当前的 URL，使用:

```
driver.getCurrentUrl();
```

Back Forward Refresh

Pressing the browser's back button:
按下浏览器的后退按钮:
Pressing the browser's forward button:
按下浏览器的前进键:
Refresh the current page:
刷新当前页面:

```
driver.navigate().back();
driver.navigate().forward();
driver.navigate().refresh();
```

Get title

You can read the current page title from the browser:
从浏览器中读取当前页面的标题:

```
driver.getTitle();
```

Windows and tabs

Get window handle

WebDriver does not make the distinction between windows and tabs. If your site opens a new tab or window, Selenium will let you work with it using a window handle. Each window has a unique identifier which remains persistent in a single session. You can get the window handle of the current window by using:
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.getWindowHandle();
```

Switching windows or tabs

Clicking a link which opens in a new window will focus the new window or tab on screen, but WebDriver will not know which window the Operating System considers active. To work with the new window you will need to switch to it. If you have only two tabs or windows open, and you know which window you start with, by the process of elimination you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.

However, Selenium 4 provides a new api NewWindow which creates a new tab (or) new window and automatically switches to it.

单击在 <a href="https://seleniumhq.github.io"target="_blank">新窗口 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。

```
//Store the ID of the original window
String originalWindow = driver.getWindowHandle();

//Check we don't have other windows open already
assert driver.getWindowHandles().size() == 1;

//Click the link which opens in a new window
```

```
driver.findElement(By.linkText("new window")).click();

//Wait for the new window or tab
wait.until(numberOfWindowsToBe(2));

//Loop through until we find a new window handle
for (String windowHandle : driver.getWindowHandles()) {
    if(!originalWindow.contentEquals(windowHandle)) {
        driver.switchTo().window(windowHandle);
        break;
    }
}

//Wait for the new tab to finish loading content
wait.until(titleIs("Selenium documentation"));
```

Create new window (or) new tab and switch

Creates a new window (or) tab and will focus the new window or tab on screen. You don't need to switch to work with the new window (or) tab. If you have more than two windows (or) tabs opened other than the new window, you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.

Note: This feature works with Selenium 4 and later versions.

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。您不需要切换到新窗口 (或) 标签页。如果除了新窗口之外， 您打开了两个以上的窗口 (或) 标签页，您可以通过遍历 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。

```


// Opens a new tab and switches to new tab
driver.switchTo().newWindow(WindowType.TAB);

// Opens a new window and switches to new window
driver.switchTo().newWindow(WindowType.WINDOW);
```

Closing a window or tab

When you are finished with a window or tab and it is not the last window or tab open in your browser, you should close it and switch back to the window you were using previously. Assuming you followed the code sample in the previous section you will have the previous window handle stored in a variable. Put this together and you will get:

当你完成了一个窗口或标签页的工作时，_并且_它不是浏览器中最后一个打开的窗口或标签页时，你应该关闭它并切换回你之前使用的窗口。 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。把这些放在一起，你会得到:

```
//Close the tab or window
driver.close();

//Switch back to the old tab or window
driver.switchTo().window(originalWindow);
```

Forgetting to switch back to another window handle after closing a window will leave WebDriver executing on the now closed page, and will trigger a **No Such Window Exception**. You must switch back to a valid window handle in order to continue execution.

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。

Quitting the browser at the end of a session

When you are finished with the browser session you should call quit, instead of close:

当你完成了浏览器会话，你应该调用 quit 退出，而不是 close 关闭:

```
driver.quit();
```

- Quit will:
  - Close all the windows and tabs associated with that WebDriver session
  - Close the browser process
  - Close the background driver process
  - Notify Selenium Grid that the browser is no longer in use so it can be used by another session (if you are using Selenium Grid)

Failure to call quit will leave extra background processes and ports running on your machine which could cause you problems later.

Some test frameworks offer methods and annotations which you can hook into to tear down at the end of a test.

- 退出将会
  - 关闭所有与 WebDriver 会话相关的窗口和选项卡
  - 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。

```
/**
 * Example using JUnit
 * https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/AfterAll.html
 */
@AfterAll
public static void tearDown() {
    driver.quit();
}
```

If not running WebDriver in a test context, you may consider using try / finally which is offered by most languages so that an exception will still clean up the WebDriver session.

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();
}
```

Frames and Iframes

Frames are a now deprecated means of building a site layout from multiple documents on the same domain. You are unlikely to work with them unless you are working with an pre HTML5 webapp. Iframes allow the insertion of a document from an entirely different domain, and are still commonly used.

If you need to work with frames or iframes, WebDriver allows you to work with them in the same way. Consider a button within an iframe. If we inspect the element using the browser development tools, we might see the following:

框架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>
```

If it was not for the iframe we would expect to click on the button using something like:

如果不是 iframe，我们可能会使用如下方式点击按钮:

```
//This won't work
driver.findElement(By.tagName("button")).click();
```

However, if there are no buttons outside of the iframe, you might instead get a no such element error. This happens because Selenium is only aware of the elements in the top level document. To interact with the button, we will need to first switch to the frame, in a similar way to how we switch windows. WebDriver offers three ways of switching to a frame.

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 提供了三种切换到帧的方法。

Using a WebElement

Switching using a WebElement is the most flexible option. You can find the frame using your preferred selector and switch to it.

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。

```
//Store the web element
WebElement iframe = driver.findElement(By.cssSelector("#modal>iframe"));

//Switch to the frame
driver.switchTo().frame(iframe);

//Now we can click the button
driver.findElement(By.tagName("button")).click();
```

Using a name or ID

If your frame or iframe has an id or name attribute, this can be used instead. If the name or ID is not unique on the page, then the first one found will be switched to.

如果您的 frame 或 iframe 具有 id 或 name 属性，则可以使用该属性。如果名称或 id 在页面上不是唯一的，那么将切换到找到的第一个。

```
//Using the ID
driver.switchTo().frame("buttonframe");

//Or using the name instead
driver.switchTo().frame("myframe");

//Now we can click the button
```

```
driver.findElement(By.tagName("button")).click();
```

Using an index

It is also possible to use the index of the frame, such as can be queried using window.frames in JavaScript.

还可以使用frame的索引， 例如可以使用JavaScript中的 window.frames 进行查询.

```
// Switches to the second frame
driver.switchTo().frame(1);
```

Leaving a frame

To leave an iframe or frameset, switch back to the default content like so:

离开 iframe 或 frameset，切换回默认内容，如下所示:

```
// Return to the top level
driver.switchTo().defaultContent();
```

Window management

Screen resolution can impact how your web application renders, so WebDriver provides mechanisms for moving and resizing the browser window.

屏幕分辨率会影响 web 应用程序的呈现方式，因此 WebDriver 提供了移动和调整浏览器窗口大小的机制。

Get window size

Fetches the size of the browser window in pixels.

获取浏览器窗口的大小(以像素为单位)。

```
//Access each dimension individually
int width = driver.manage().window().getSize().getWidth();
int height = driver.manage().window().getSize().getHeight();

//Or store the dimensions and query them later
Dimension size = driver.manage().window().getSize();
int width1 = size.getWidth();
int height1 = size.getHeight();
```

Set window size

Restores the window and sets the window size.

恢复窗口并设置窗口大小。

```
driver.manage().window().setSize(new Dimension(1024, 768));
```

Get window position

Fetches the coordinates of the top left coordinate of the browser window.

获取浏览器窗口左上角的坐标。

```
// Access each dimension individually
int x = driver.manage().window().getPosition().getX();
int y = driver.manage().window().getPosition().getY();

// Or store the dimensions and query them later
Point position = driver.manage().window().getPosition();
int x1 = position.getX();
int y1 = position.getY();
```

Set window position

Moves the window to the chosen position.

将窗口移动到设定的位置。

```
// Move the window to the top left of the primary monitor
driver.manage().window().setPosition(new Point(0, 0));
```

Maximize window

Enlarges the window. For most operating systems, the window will fill the screen, without blocking the operating system's own menus and toolbars.

扩大窗口。对于大多数操作系统，窗口将填满屏幕，而不会阻挡操作系统自己的菜单和工具栏。

```
driver.manage().window().maximize();
```

Minimize window

Minimizes the window of current browsing context. The exact behavior of this command is specific to individual window managers.

Minimize Window typically hides the window in the system tray.

Note: This feature works with Selenium 4 and later versions.

最小化当前浏览上下文的窗口. 这种命令的精准行为将作用于各个特定的窗口管理器.

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以及更高版本.

```
driver.manage().window().minimize();
```

Fullscreen window

Fills the entire screen, similar to pressing F11 in most browsers.

填充整个屏幕，类似于在大多数浏览器中按下 F11。

```
driver.manage().window().fullscreen();
```

TakeScreenshot

Used to capture screenshot for current browsing context. The WebDriver endpoint screenshot returns screenshot which is encoded in Base64 format.

用于捕获当前浏览上下文的屏幕截图. WebDriver端点 屏幕截图 返回以Base64格式编码的屏幕截图.

```
import org.apache.commons.io.FileUtils;
import org.openqa.selenium.chrome.ChromeDriver;
import java.io.*;
import org.openqa.selenium.*;

public class SeleniumTakeScreenshot {
    public static void main(String args[]) throws IOException {
        WebDriver driver = new ChromeDriver();
        driver.get("http://www.example.com");
        File scrFile =
((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(scrFile, new File("./image.png"));
        driver.quit();
    }
}
```

TakeElementScreenshot

Used to capture screenshot of an element for current browsing context. The WebDriver endpoint screenshot returns screenshot which is encoded in Base64 format.

用于捕获当前浏览上下文的元素的屏幕截图. WebDriver端点 屏幕截图 返回以Base64格式编码的屏幕截图.

```
import org.apache.commons.io.FileUtils;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import java.io.File;
import java.io.IOException;

public class SeleniumelementTakeScreenshot {
  public static void main(String args[]) throws IOException {
    WebDriver driver = new ChromeDriver();
    driver.get("https://www.example.com");
    WebElement element = driver.findElement(By.cssSelector("h1"));
    File scrFile = element.getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("./image.png"));
    driver.quit();
  }
}
```

Execute Script

Executes JavaScript code snippet in the current context of a selected frame or window.

在当前frame或者窗口的上下文中，执行JavaScript代码片段.

```
    //Creating the JavascriptExecutor interface object by Type casting
      JavascriptExecutor js = (JavascriptExecutor)driver;
    //Button Element
      WebElement button =driver.findElement(By.name("btnLogin"));
    //Executing JavaScript to click on element
      js.executeScript("arguments[0].click();", element);
    //Get return value from script
      String text = (String) javascriptExecutor.executeScript("return
arguments[0].innerText", element);
    //Executing JavaScript directly
      js.executeScript("console.log('hello world')");
```

Print Page

Prints the current page within the browser.

Note: This requires Chromium Browsers to be in headless mode

打印当前浏览器内的页面

注意: 此功能需要无头模式下的Chromium浏览器

```
    import org.openqa.selenium.print.PrintOptions;

    driver.get("https://www.selenium.dev");
    printer = (PrintsPage) driver;
```

```java
        PrintOptions printOptions = new PrintOptions();
        printOptions.setPageRanges("1-2");

        Pdf pdf = printer.print(printOptions);
        String content = pdf.getContent();
```

Loacating elements

Locating one element

One of the most fundamental techniques to learn when using WebDriver is how to find elements on the page. WebDriver offers a number of built-in selector types, amongst them finding an element by its ID attribute:

使用 WebDriver 时要学习的最基本的技术之一是如何查找页面上的元素。 WebDriver 提供了许多内置的选择器类型，其中包括根据 id 属性查找元素:

```java
WebElement cheese = driver.findElement(By.id("cheese"));
```

As seen in the example, locating elements in WebDriver is done on the WebDriver instance object. The findElement(By) method returns another fundamental object type, the WebElement.

- WebDriver represents the browser
- WebElement represents a particular DOM node (a control, e.g. a link or input field, etc.)

Once you have a reference to a web element that's been "found", you can narrow the scope of your search by using the same call on that object instance:

如示例所示，在 WebDriver 中定位元素是在 WebDriver 实例对象上完成的。 findElement(By) 方法返回另一个基本对象类型 WebElement。

- WebDriver 代表浏览器
- WebElement 表示特定的 DOM 节点（控件，例如链接或输入栏等)

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：

```java
WebElement cheese = driver.findElement(By.id("cheese"));
WebElement cheddar = cheese.findElement(By.id("cheddar"));
```

You can do this because both the WebDriver and WebElement types implement the SearchContext interface. In WebDriver, this is known as a role-based interface. Role-based interfaces allow you to determine whether a particular driver implementation supports a given feature. These interfaces are clearly defined and try to adhere to having only a single role of responsibility. You can read more about WebDriver's design and what roles are supported in which drivers in the Some Other Section Which Must Be Named.

Consequently, the By interface used above also supports a number of additional locator strategies. A nested lookup might not be the most effective cheese location strategy since it requires two separate commands to be issued to the browser; first searching the DOM for an element with ID “cheese”, then a search for “cheddar” in a narrowed context.

To improve the performance slightly, we should try to use a more specific locator: WebDriver supports looking up elements by CSS locators, allowing us to combine the two previous locators into one search:

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:

```
driver.findElement(By.cssSelector("#cheese #cheddar"));
```

Locating multiple elements

It is possible that the document we are working with may turn out to have an ordered list of the cheese we like the best:

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>…
</ol>
```

Since more cheese is undisputably better, and it would be cumbersome to have to retrieve each of the items individually, a superior technique for retrieving cheese is to make use of the pluralized version findElements(By). This method returns a collection of web elements. If only one element is found, it will still return a collection (of one element). If no element matches the locator, an empty list will be returned.

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。

```
List<WebElement> muchoCheese = driver.findElements(By.cssSelector("#cheese li"));
```

Element selection strategies

There are eight different built-in element location strategies in WebDriver:

在 WebDriver 中有 8 种不同的内置元素定位策略：

Locator	Description
class name	Locates elements whose class name contains the search value (compound class names are not permitted)
css	Locates elements matching a CSS selector
id	Locates elements whose ID attribute matches the search value
name	Locates elements whose NAME attribute matches the search value
link test	Locates anchor elements whose visible text matches the search value
partial link test	Locates anchor elements whose visible text contains the search value. If multiple elements are matching, only the first one will be selected.
tag name	Locates elements whose tag name matches the search value
xpath	Locates elements matching an XPath expression

Tips on using selectors

In general, if HTML IDs are available, unique, and consistently predictable, they are the preferred method for locating an element on a page. They tend to work very quickly, and forego much processing that comes with complicated DOM traversals.

If unique IDs are unavailable, a well-written CSS selector is the preferred method of locating an element. XPath works as well as CSS selectors, but the syntax is complicated and frequently difficult to debug. Though XPath selectors are very flexible, they are typically not performance tested by browser vendors and tend to be quite slow.

Selection strategies based on linkText and partialLinkText have drawbacks in that they only work on link elements. Additionally, they call down to XPath selectors internally in WebDriver.

Tag name can be a dangerous way to locate elements. There are frequently multiple elements of the same tag present on the page. This is mostly useful when calling the findElements(By) method which returns a collection of elements.

The recommendation is to keep your locators as compact and readable as possible. Asking WebDriver to traverse the DOM structure is an expensive operation, and the more you can narrow the scope of your search, the better.

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。

Relative Locators

Selenium 4 brings Relative Locators which are previously called as Friendly Locators. This functionality was added to help you locate elements that are nearby other elements. The Available Relative Locators are:

- above
- below
- toLeftOf
- toRightOf
- near

findElement method now accepts a new method withTagName() which returns a RelativeLocator.

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 (Friendly 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。

How does it work

Selenium uses the JavaScript function getBoundingClientRect() to find the relative elements. This function returns properties of an element such as right, left, bottom, and top.

Let us consider the below example for understanding the relative locators.

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

above()

Returns the WebElement, which appears above to the specified element

返回当前指定元素位置上方的WebElement对象

```
import static org.openqa.selenium.support.locators.RelativeLocator.withTagName;

WebElement passwordField= driver.findElement(By.id("password"));
WebElement emailAddressField =
driver.findElement(withTagName("input").above(passwordField));
```

below()

Returns the WebElement, which appears below to the specified element

返回当前指定元素位置下方的WebElement对象

```
import static org.openqa.selenium.support.locators.RelativeLocator.withTagName;

WebElement emailAddressField= driver.findElement(By.id("email"));
WebElement passwordField =
driver.findElement(withTagName("input").below(emailAddressField));
```

toLeftOf()

Returns the WebElement, which appears to left of specified element

返回当前指定元素位置左方的WebElement对象

```
import static org.openqa.selenium.support.locators.RelativeLocator.withTagName;

WebElement submitButton= driver.findElement(By.id("submit"));
WebElement cancelButton=
driver.findElement(withTagName("button").toLeftOf(submitButton));
```

toRightOf()

Returns the WebElement, which appears to right of the specified element

返回当前指定元素位置右方的WebElement对象

```
import static org.openqa.selenium.support.locators.RelativeLocator.withTagName;

WebElement cancelButton= driver.findElement(By.id("cancel"));
WebElement submitButton=
driver.findElement(withTagName("button").toRightOf(cancelButton));
```

near()

Returns the WebElement, which is at most 50px away from the specified element.

返回当前指定元素位置附近大约px50像素的WebElement对象

```
import static org.openqa.selenium.support.locators.RelativeLocator.withTagName;

WebElement emailAddressLabel= driver.findElement(By.id("lbl-email"));
WebElement emailAddressField =
driver.findElement(withTagName("input").near(emailAddressLabel));
```

JavaScript alerts, prompts and confirmations

WebDriver provides an API for working with the three types of native popup messages offered by JavaScript. These popups are styled by the browser and offer limited customisation.

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.

Alerts

he simplest of these is referred to as an alert, which shows a custom message, and a single button which dismisses the alert, labelled in most browsers as OK. It can also be dismissed in most browsers by pressing the close button, but this will always do the same thing as the OK button. See an example alert.

WebDriver can get the text from the popup and accept or dismiss these alerts.

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. 查看样例警告框.

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.findElement(By.linkText("See an example alert")).click();

//Wait for the alert to be displayed and store it in a variable
Alert alert = wait.until(ExpectedConditions.alertIsPresent());

//Store the alert text in a variable
String text = alert.getText();

//Press the OK button
alert.accept();
```

Confirm

A confirm box is similar to an alert, except the user can also choose to cancel the message. See a sample confirm.

This example also shows a different approach to storing an alert:

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.findElement(By.linkText("See a sample confirm")).click();

//Wait for the alert to be displayed
wait.until(ExpectedConditions.alertIsPresent());

//Store the alert in a variable
Alert alert = driver.switchTo().alert();

//Store the alert in a variable for reuse
String text = alert.getText();

//Press the Cancel button
alert.dismiss();
```

Prompt

Prompts are similar to confirm boxes, except they also include a text input. Similar to working with form elements, you can use WebDriver's send keys to fill in a response. This will completely replace the placeholder text. Pressing the cancel button will not submit any text. See a sample prompt.

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.findElement(By.linkText("See a sample prompt")).click();

//Wait for the alert to be displayed and store it in a variable
Alert alert = wait.until(ExpectedConditions.alertIsPresent());

//Type your message
alert.sendKeys("Selenium");

//Press the OK button
alert.accept();
```

Waits

WebDriver can generally be said to have a blocking API. Because it is an out-of-process library that instructs the browser what to do, and because the web platform has an intrinsically asynchronous nature, WebDriver does not track the active, real-time state of the DOM. This comes with some challenges that we will discuss here.

From experience, most intermittent issues that arise from use of Selenium and WebDriver are connected to race conditions that occur between the browser and the user's instructions. An example could be that the user instructs the browser to navigate to a page, then gets a no such element error when trying to find an element.

Consider the following document:

WebDriver通常可以说有一个阻塞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>
```

The WebDriver instructions might look innocent enough:

这个 WebDriver的说明可能看起来很简单:

```
driver.get("file:///race_condition.html");
WebElement element = driver.findElement(By.tagName("p"));
assertEquals(element.getText(), "Hello from JavaScript!");
```

The issue here is that the default page load strategy used in WebDriver listens for the document.readyState to change to "complete" before returning from the call to navigate. Because the p element is added after the document has completed loading, this WebDriver script might be intermittent. It “might” be intermittent because no guarantees can be made about elements or events that trigger asynchronously without explicitly waiting—or blocking—on those events.

Fortunately, the normal instruction set available on the WebElement interface—such as WebElement.click and WebElement.sendKeys—are guaranteed to be synchronous, in that the function calls will not return (or the callback will not trigger in callback-style languages) until the command has been completed in the browser. The advanced user interaction APIs, Keyboard and Mouse, are exceptions as they are explicitly intended as “do what I say” asynchronous commands.

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step.

To overcome the problem of race conditions between the browser and your WebDriver script, most Selenium clients ship with a wait package. When employing a wait, you are using what is commonly referred to as an explicit wait.

这里的问题是WebDriver中使用的默认页面加载策略页面加载策略听从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脚本可能是间歇性的。它"可能"间歇性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。

Explicit wait

Explicit waits are available to Selenium clients for imperative, procedural languages. They allow your code to halt program execution, or freeze the thread, until the condition you pass it resolves. The condition is called with a certain frequency until the timeout of the wait is elapsed. This means that for as long as the condition returns a falsy value, it will keep trying and waiting.

Since explicit waits allow you to wait for a condition to occur, they make a good fit for synchronising the state between the browser and its DOM, and your WebDriver script.

To remedy our buggy instruction set from earlier, we could employ a wait to have the findElement call wait until the dynamically added element from the script has been added to the DOM:

显示等待 是Selenium客户可以使用的命令式过程语言。它们允许您的代码暂停程序执行，或冻结线程，直到满足通过的 条件 。这个条件会以一定的频率一直被调用，直到等待超时。这意味着只要条件返回一个假值，它就会一直尝试和等待

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。

为了弥补我们之前的错误指令集，我们可以使用等待来让 findElement 调用等待直到脚本中动态添加的元素被添加到DOM中:

```
WebDriver driver = new ChromeDriver();
driver.get("https://google.com/ncr");
driver.findElement(By.name("q")).sendKeys("cheese" + Keys.ENTER);
// Initialize and wait till element(link) became clickable - timeout in 10
seconds
WebElement firstResult = new WebDriverWait(driver, Duration.ofSeconds(10))
        .until(ExpectedConditions.elementToBeClickable(By.xpath("//a/h3")));
// Print the first result
System.out.println(firstResult.getText());
```

We pass in the condition as a function reference that the wait will run repeatedly until its return value is truthy. A “truthful” return value is anything that evaluates to boolean true in the language at hand, such as a string, number, a boolean, an object (including a WebElement), or a populated (non-empty) sequence or list. That means an empty list evaluates to false. When the condition is truthful and the blocking wait is aborted, the return value from the condition becomes the return value of the wait.

With this knowledge, and because the wait utility ignores no such element errors by default, we can refactor our instructions to be more concise:

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:

```
WebElement foo = new WebDriverWait(driver, Duration.ofSeconds(3))
          .until(driver -> driver.findElement(By.name("q")));
assertEquals(foo.getText(), "Hello from JavaScript!");
```

In that example, we pass in an anonymous function (but we could also define it explicitly as we did earlier so it may be reused). The first and only argument that is passed to our condition is always a reference to our driver object, WebDriver. In a multi-threaded environment, you should be careful to operate on the driver reference passed in to the condition rather than the reference to the driver in the outer scope.

Because the wait will swallow no such element errors that are raised when the element is not found, the condition will retry until the element is found. Then it will take the return value, a WebElement, and pass it back through to our script.

If the condition fails, e.g. a truthful return value from the condition is never reached, the wait will throw/raise an error/exception called a timeout error.

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。

Options

The wait condition can be customised to match your needs. Sometimes it is unnecessary to wait the full extent of the default timeout, as the penalty for not hitting a successful condition can be expensive.

The wait lets you pass in an argument to override the timeout:

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:

```
new WebDriverWait(driver,
Duration.ofSeconds(3)).until(ExpectedConditions.elementToBeClickable(By.xpath("//
a/h3")));
```

Expected conditions

Because it is quite a common occurrence to have to synchronise the DOM and your instructions, most clients also come with a set of predefined expected conditions. As might be obvious by the name, they are conditions that are predefined for frequent wait operations.

The conditions available in the different language bindings vary, but this is a non-exhaustive list of a few:

- alert is present
- element exists
- element is visible
- title contains
- title is
- element staleness
- visible text

You can refer to the API documentation for each client binding to find an exhaustive list of expected conditions:

- 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

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

您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:

- 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

Implicit wait

There is a second type of wait that is distinct from explicit wait called implicit wait. By implicitly waiting, WebDriver polls the DOM for a certain duration when trying to find any element. This can be useful when certain elements on the webpage are not available immediately and need some time to load.

Implicit waiting for elements to appear is disabled by default and will need to be manually enabled on a per-session basis. Mixing explicit waits and implicit waits will cause unintended consequences, namely waits sleeping for the maximum time even if the element is available or condition is true.

Warning: Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example, setting an implicit wait of 10 seconds and an explicit wait of 15 seconds could cause a timeout to occur after 20 seconds.

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0, meaning disabled. Once set, the implicit wait is set for the life of the session.

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。

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

FluentWait

FluentWait instance defines the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition.

Users may configure the wait to ignore specific types of exceptions whilst waiting, such as NoSuchElementException when searching for an element on the page.

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。

```
// Waiting 30 seconds for an element to be present on the page, checking
// for its presence once every 5 seconds.
```

```
Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
  .withTimeout(Duration.ofSeconds(30))
  .pollingEvery(Duration.ofSeconds(5))
  .ignoring(NoSuchElementException.class);

WebElement foo = wait.until(new Function<WebDriver, WebElement>() {
  public WebElement apply(WebDriver driver) {
    return driver.findElement(By.id("foo"));
  }
});
```

HTTP proxies

A proxy server acts as an intermediary for requests between a client and a server. In simple, the traffic flows through the proxy server on its way to the address you requested and back.

A proxy server for automation scripts with Selenium could be helpful for:

- Capture network traffic
- Mock backend calls made by the website
- Access the required website under complex network topologies or strict corporate restrictions/policies.

If you are in a corporate environment, and a browser fails to connect to a URL, this is most likely because the environment needs a proxy to be accessed.

Selenium WebDriver provides a way to proxy settings:

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

```
import org.openqa.selenium.Proxy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;

public class proxyTest {
  public static void main(String[] args) {
    Proxy proxy = new Proxy();
    proxy.setHttpProxy("<HOST:PORT>");
    ChromeOptions options = new ChromeOptions();
    options.setCapability("proxy", proxy);
    WebDriver driver = new ChromeDriver(options);
    driver.get("https://www.google.com/");
    driver.manage().window().maximize();
    driver.quit();
  }
}
```

Page loading strategy

Defines the current session's page loading strategy. By default, when Selenium WebDriver loads a page, it follows the normal pageLoadStrategy. It is always recommended to stop downloading additional resources (like images, css, js) when the page loading takes lot of time.

The document.readyState property of a document describes the loading state of the current document. By default, WebDriver will hold off on responding to a driver.get() (or) driver.navigate().to() call until the document ready state is complete

In SPA applications (like Angular, React, Ember) once the dynamic content is already loaded (I.e once the pageLoadStrategy status is COMPLETE), clicking on a link or performing some action within the page will not make a new request to the server as the content is dynamically loaded at the client side without a full page refresh.

SPA applications can load many views dynamically without any server requests, So pageLoadStrategy will always show COMPLETE status until we do a new driver.get() and driver.navigate().to()

WebDriver pageLoadStrategy supports the following values:

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. 始终建议您在页面加载缓慢时, 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:

normal

This will make Selenium WebDriver to wait for the entire page is loaded. When set to normal, Selenium WebDriver waits until the load event fire is returned.

By default normal is set to browser if none is provided.

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.

```java
import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.NORMAL);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}
```

eager

This will make Selenium WebDriver to wait until the initial HTML document has been completely loaded and parsed, and discards loading of stylesheets, images and subframes.

When set to eager, Selenium WebDriver waits until DOMContentLoaded event fire is returned.

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.

```java
import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}
```

none

When set to none Selenium WebDriver only waits until the initial page is downloaded.

设置为 none 时, Selenium WebDriver仅等待至初始页面下载完成.

```
import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.NONE);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}
```

Web element

WebElement represents a DOM element. WebElements can be found by searching from the document root using a WebDriver instance, or by searching under another WebElement.

WebDriver API provides built-in methods to find the WebElements which are based on different properties like ID, Name, Class, XPath, CSS Selectors, link Text, etc.

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).

Find Element

It is used to find an element and returns a first matching single WebElement reference, that can be used for future element actions

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.

```
WebDriver driver = new FirefoxDriver();

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

// Get search box element from webElement 'q' using Find Element
```

```
WebElement searchBox = driver.findElement(By.name("q"));

searchBox.sendKeys("webdriver");
```

Find Elements

Similar to 'Find Element', but returns a list of matching WebElements. To use a particular WebElement from the list, you need to loop over the list of elements to perform action on selected element.

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.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import java.util.List;

public class findElementsExample {
    public static void main(String[] args) {
        WebDriver driver = new FirefoxDriver();
        try {
            driver.get("https://example.com");
            // Get all the elements available with tag name 'p'
            List<WebElement> elements = driver.findElements(By.tagName("p"));
            for (WebElement element : elements) {
                System.out.println("Paragraph text:" + element.getText());
            }
        } finally {
            driver.quit();
        }
    }
}
```

Find Element From Element

It is used to find a child element within the context of parent element. To achieve this, the parent WebElement is chained with 'findElement' to access child elements

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.

```
WebDriver driver = new FirefoxDriver();
driver.get("http://www.google.com");
WebElement searchForm = driver.findElement(By.tagName("form"));
WebElement searchBox = searchForm.findElement(By.name("q"));
searchBox.sendKeys("webdriver");
```

Find Elements From Element

It is used to find the list of matching child WebElements within the context of parent element. To achieve this, the parent WebElement is chained with 'findElements' to access child elements

此方法用于在父元素的上下文中查找匹配子WebElement的列表. 为此, 父WebElement与"findElements"链接并访问子元素.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.List;

public class findElementsFromElement {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("https://example.com");

            // Get element with tag name 'div'
            WebElement element = driver.findElement(By.tagName("div"));

            // Get all the elements available with tag name 'p'
            List<WebElement> elements = element.findElements(By.tagName("p"));
            for (WebElement e : elements) {
                System.out.println(e.getText());
            }
        } finally {
            driver.quit();
        }
    }
}
```

Get Active Element

It is used to track (or) find DOM element which has the focus in the current browsing context.

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.

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class activeElementTest {
  public static void main(String[] args) {
    WebDriver driver = new ChromeDriver();
    try {
      driver.get("http://www.google.com");
      driver.findElement(By.cssSelector("[name='q']")).sendKeys("webElement");

      // Get attribute of current active element
```

```
          String attr = driver.switchTo().activeElement().getAttribute("title");
          System.out.println(attr);
        } finally {
          driver.quit();
        }
      }
    }
```

Is Element Enabled

This method is used to check if the connected Element is enabled or disabled on a webpage. Returns a boolean value, True if the connected element is enabled in the current browsing context else returns false.

此方法用于检查网页上连接的元素是否被启用或禁用. 返回一个布尔值, 如果在当前浏览上下文中启用了连接的元素, 则返回True； 否则返回false .

```
    //navigates to url
    driver.get("https://www.google.com/");

    //returns true if element is enabled else returns false
    boolean value = driver.findElement(By.name("btnK")).isEnabled();
```

Is Element Selected

This method determines if the referenced Element is Selected or not. This method is widely used on Check boxes, radio buttons, input elements, and option elements.

Returns a boolean value, True if referenced element is selected in the current browsing context else returns false.

此方法确定是否 已选择 引用的元素. 此方法广泛用于复选框, 单选按钮, 输入元素和选项元素.

返回一个布尔值, 如果在当前浏览上下文中 已选择 引用的元素, 则返回 True, 否则返回 False.

```
   //navigates to url
   driver.get("https://the-internet.herokuapp.com/checkboxes");

   //returns true if element is checked else returns false
   boolean value = driver.findElement(By.cssSelector("input[type='checkbox']:first-
  of-type")).isSelected();
```

Get Element TagName

It is used to fetch the TagName of the referenced Element which has the focus in the current browsing context.

此方法用于获取在当前浏览上下文中 具有焦点的被引用元素的 TagName .

```
//navigates to url
driver.get("https://www.example.com");

//returns TagName of the element
String value = driver.findElement(By.cssSelector("h1")).getTagName();
```

Get Element Rect

It is used to fetch the dimensions and coordinates of the referenced element.

The fetched data body contain the following details:

- X-axis position from the top-left corner of the element
- y-axis position from the top-left corner of the element
- Height of the element
- Width of the element

用于获取参考元素的尺寸和坐标.

提取的数据主体包含以下详细信息:

- 元素左上角的X轴位置
- 元素左上角的y轴位置
- 元素的高度
- 元素宽度

```
// Navigate to url
driver.get("https://www.example.com");

// Returns height, width, x and y coordinates referenced element
Rectangle res =  driver.findElement(By.cssSelector("h1")).getRect();

// Rectangle class provides getX,getY, getWidth, getHeight methods
System.out.println(res.getX());
```

Get Element CSS Value

Retrieves the value of specified computed style property of an element in the current browsing context.

获取当前浏览上下文中元素的特定计算样式属性的值.

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

// Retrieves the computed style property 'color' of linktext
String cssValue = driver.findElement(By.linkText("More
information...")).getCssValue("color");
```

Get Element Text

Retrieves the rendered text of the specified element.

获取特定元素渲染后的文本.

```
// Navigate to url
driver.get("https://example.com");

// Retrieves the text of the element
String text = driver.findElement(By.cssSelector("h1")).getText();
```

Keyboard

Keyboard represents a KeyBoard event. KeyBoard actions are performed by using low-level interface which allows us to provide virtualized device input to the web browser.

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.

sendKeys

The sendKeys types a key sequence in DOM element even if modifier key sequence is encountered. Here are the list of possible keystrokes that WebDriver Supports.

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.

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class HelloSelenium {
  public static void main(String[] args) {
    WebDriver driver = new FirefoxDriver();
    try {
      // Navigate to Url
      driver.get("https://google.com");

      // Enter text "q" and perform keyboard action "Enter"
      driver.findElement(By.name("q")).sendKeys("q" + Keys.ENTER);
    } finally {
      driver.quit();
    }
  }
}
```

keyDown

The keyDown is used to simulate action of pressing a modifier key(CONTROL, SHIFT, ALT) keyDown用于模拟按下辅助按键(CONTROL, SHIFT, ALT)的动作.

```
WebDriver driver = new ChromeDriver();
try {
  // Navigate to Url
  driver.get("https://google.com");

  // Enter "webdriver" text and perform "ENTER" keyboard action
  driver.findElement(By.name("q")).sendKeys("webdriver" + Keys.ENTER);

  Actions actionProvider = new Actions(driver);
  Action keydown = actionProvider.keyDown(Keys.CONTROL).sendKeys("a").build();
  keydown.perform();
} finally {
  driver.quit();
}
```

keyUp

The keyUp is used to simulate key-up (or) key-release action of a modifier key(CONTROL, SHIFT, ALT)

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

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.interactions.Actions;

public class HelloSelenium {
  public static void main(String[] args) {
    WebDriver driver = new FirefoxDriver();
    try {
      // Navigate to Url
      driver.get("https://google.com");
      Actions action = new Actions(driver);

      // Store google search box WebElement
      WebElement search = driver.findElement(By.name("q"));

      // Enters text "qwerty" with keyDown SHIFT key and after keyUp SHIFT key
(QWERTYqwerty)

action.keyDown(Keys.SHIFT).sendKeys(search,"qwerty").keyUp(Keys.SHIFT).sendKeys("
qwerty").perform();
    } finally {
      driver.quit();
    }
  }
}
```

clear

Clears the content of an editable element. This is only applied for the elements which is editable and interactable, otherwise Selenium returns the error (invalid element state (or) Element not interactable)

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).

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class clear {
  public static void main(String[] args) {
    WebDriver driver = new ChromeDriver();
    try {
      // Navigate to Url
      driver.get("https://www.google.com");
      // Store 'SearchInput' element
      WebElement searchInput = driver.findElement(By.name("q"));
      searchInput.sendKeys("selenium");
      // Clears the entered text
      searchInput.clear();
    } finally {
      driver.quit();
    }
  }
}
```