|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 本文所述内容在Windows系统下实践完成，涉及的软件版本信息如下：   |  |  |  | | --- | --- | --- | | **软件** | **版本** | **位数** | | Java | 1.8.0\_152 | 64位 | | Eclipse | eclipse-jee-oxygen-1a-win32-x86\_64 (4.7.1a) | 64位 | | IntelliJ IDEA | 2017.3.1 | 64位 | | Maven | 3.5.2 |  | | Selenium | 3.8.1 |  | | Junit | 4.12 |  | | TestNG | 6.13.1 |  | | AutoIt | 3.3.14.2（AutoIt CN中文版） |  | | Sikuli |  |  | | Firefox | 55 | 32位 |   注：部分软件版本间兼容性十分重要，下文中会提及，使用其他版本时请注意。 |

# 简介

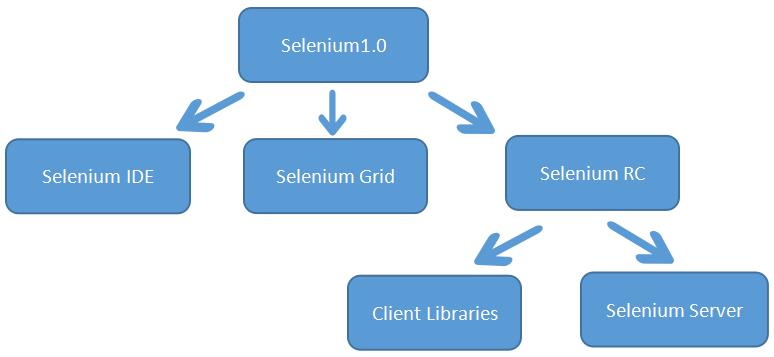
Selenium 是用于测试 Web 应用程序用户界面 (UI) 的常用框架。它是一款用于运行端到端功能测试的超强工具。您可以使用多个编程语言编写测试，并且 Selenium 能够在一个或多个浏览器中执行这些测试。

## 历史

* **Selenium1.0**

2004年，ThoughtWorks公司的Jason Huggins和他所在的团队采用Javascript编写一种测试工具来验证浏览器页面的行为。这个JavaScript类库就是Selenium core，同时也是seleniumRC、Selenium IDE的核心组件。

|  |
| --- |
| **关于命名**  当时QTP mercury是主流的商业自动化工具，是化学元素汞（俗称水银），而Selenium是开源自动化工具，是化学元素硒，硒可以对抗汞。 |



**Selenium 1.0 = Selenium IDE + Selenium Grid + Selenium RC**

|  |
| --- |
| * + **Selenium IDE**   Selenium IDE是嵌入到Firefox浏览器中的一个插件，实现简单的浏览器操作的录制与回放功能。   * + **Selenium Grid**   Selenium Grid是一种自动化的测试辅助工具，Grid通过利用现有的计算机基础设施，能加快Web-App的功能测试。利用Grid可以很方便地实现在多台机器上和异构环境中运行测试用例。   * + **Selenium RC**   Selenium RC（Remote Control）是Selenium家族的核心部分。Selenium RC 支持多种不同语言编写的自动化测试脚本，通过Selenium RC的服务器作为代理服务器去访问应用，从而达到测试的目的。  Selenium RC分为Client Libraries和Selenium Server。Client Libraries库主要用于编写测试脚本，用来控制Selenium Server的库。Selenium Server负责控制浏览器行为。 |

* **Selenium 2.0**

2006年，Google的工程师Simon Stewart发起了WebDriver的项目；因为长期以来Google一直是Selenium的重度用户，但却被限制在有限的操作范围内。

Selenium RC 是在浏览器中运行JavaScript应用，使用浏览器内置的JavaScript翻译器来翻译和执行selenese命令（selenese是Selenium命令集合）。

WebDriver是通过原生浏览器支持或者浏览器扩展来直接控制浏览器。WebDriver针对各个浏览器而开发，取代了嵌入到被测Web应用中的JavaScript，与浏览器紧密集成，因此支持创建更高级的测试，避免了JavaScript安全模型导致的限制。除了来自浏览器厂商的支持之外，WebDriver还利用操作系统级的调用，模拟用户输入。

2011年，Selenium和Webdriver合并， Selenium 2.0诞生。

|  |
| --- |
| Selenium与WebDriver原是属于两个不同的项目，WebDriver的创建者Simon Stewart早在2009年8月的一份邮件中解释了项目合并的原因。  Selenium与WebDriver合并原因：为何把两个项目合并？部分原因是WebDriver解决了Selenium存在的缺点（例如能够绕过JavaScript沙箱，我们有出色的API），部分原因是Selenium解决了WebDriver存在的问题（例如支持广泛的浏览器），部分原因是因为Selenium的主要贡献者和我都觉得合并项目是为用户提供最优秀框架的最佳途径。 |

**Selenium 2.0 = Selenium 1.0 + WebDriver**

需要强调的是，在Selenium 2.0中主推的是WebDriver，可以将其看作Selenium RC的替代品。因为Selenium为了保持向下的兼容性，所以在Selenium 2.0中并没有彻底地抛弃Selenium RC。

Selenium 2 具有来自 WebDriver 的清晰面向对象 API，并能以最佳的方式与浏览器进行交互。Selenium 2 不使用 JavaScript 沙盒，它支持多种浏览器和多语言绑定。Selenium 2 为下列程序提供驱动程序：

* Mozilla Firefox
* Google Chrome
* Microsoft Internet Explorer
* Opera
* Apple iPhone
* Android browsers
* ……

借助 Selenium 2，您可使用 Java、C#、Ruby、和 Python 编写测试。Selenium 2 还提供基于 HtmlUnit 的无外设驱动，是用于测试 Web 应用程序的 Java 框架。HtmlUnit 运行速度特别快，但它不是一个真正与真实浏览器相关联的驱动。

* **Selenium3.0**

2016年7月，Selenium3.0悄悄发布第一个beta版。

**Selenium 3.0 = Selenium 2.0 + Selenium RC（Remote Control）**

## Selenium3现状

### 更新

* + 去掉了对selenium rc的支持。
  + 全面拥抱java 8。
  + 支持macOS (Sierra or later)，支持官方的[safaridriver](https://developer.apple.com/library/prerelease/content/releasenotes/General/WhatsNewInSafari/Articles/Safari_10_0.html)。
  + 通过ms官方的[webdriver server](https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/)支持Edge浏览器。
  + 只支持ie 9.0版本以上。
  + 通过Mozilla官方的[geckodriver](https://github.com/mozilla/geckodriver/releases)来支持firefox。

尽管firefox driver非常的稳定高效，但firefox driver毕竟是google实现的。geckodriver的出现是必然的结果，因为最新版本的firefox换引擎了，老的firefox driver应该是不支持新引擎的。

|  |
| --- |
| firefox driver究竟是什么？为什么你一次都没下载运行过？  这是因为firefox driver包含在selenium各语言版本的分发包里，使用webdriver启动firefox的时候，webdriver会为firefox安装firefox driver扩展。 |

### 重要意义

• webdriver 协议现在已经成为业内公认的浏览器ui测试的标准实现。webdriver协议是google对开源测试领域的重要贡献。

• 各种官方支持意味着以后的浏览器ui测试的速度和稳定性会有较大的提升。

• 浏览器ui自动化测试已经成为了行业标配。

• selenium专注web测试。

# 环境搭建

## 资源列表

|  |  |
| --- | --- |
| 名称 | 网址 |
| Maven | <http://maven.apache.org/> |
| Selenium | <http://selenium-release.storage.googleapis.com/index.html> |
| Firefox | <http://ftp.mozilla.org/pub/firefox/releases/> |
| Gecko driver | <https://github.com/mozilla/geckodriver/releases> |
| Chrome driver | <http://chromedriver.storage.googleapis.com/index.html>  <https://npm.taobao.org/mirrors/chromedriver> |
| IE driver | <http://selenium-release.storage.googleapis.com/index.html> |
| Edge driver | <https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/> |
| Opera driver | <https://github.com/operasoftware/operachromiumdriver/releases> |
| Safari driver | <https://developer.apple.com/library/content/releasenotes/General/WhatsNewInSafari/Introduction/Introduction.html> |
| PhantomJS driver | <http://phantomjs.org/> |

## Java

由于Selenium的限制，需要安装Java8。

32位或64位依系统而定，如果是64位系统则两个版本均可安装。

## Maven

我们需要创建一个测试工程，但该工程需要依赖很多jar包，为了方便管理这些依赖，我们使用Maven进行管理 。

下载Maven解压即可使用。

我们主要需要配置Maven本地库存放目录，以及主仓库地址，如：

|  |
| --- |
| <localRepository>E: /maven</localRepository>  <mirrors>  <mirror>  <id>alimaven</id>  <name>aliyun maven</name>  <url>http://maven.aliyun.com/nexus/content/groups/public/</url>  <mirrorOf>central</mirrorOf>  </mirror>  </mirrors> |

注：上例使用的是阿里云的Maven仓库作为镜像站点，因为比较快一点。

## Eclipse

**安装Eclipse**

（解压）安装Java或JavaEE版均可，但位数应与JDK匹配。

注：如果系统中JDK是绿色版，或其注册表信息缺失，或安装有多个版本的JDK而不想使用默认的JDK启动Eclipse，可以在Eclipse根目录下找到配置文件eclipse.ini，并用文本编辑器打开，再在其中加入如下内容，指定用于启动的JDK：

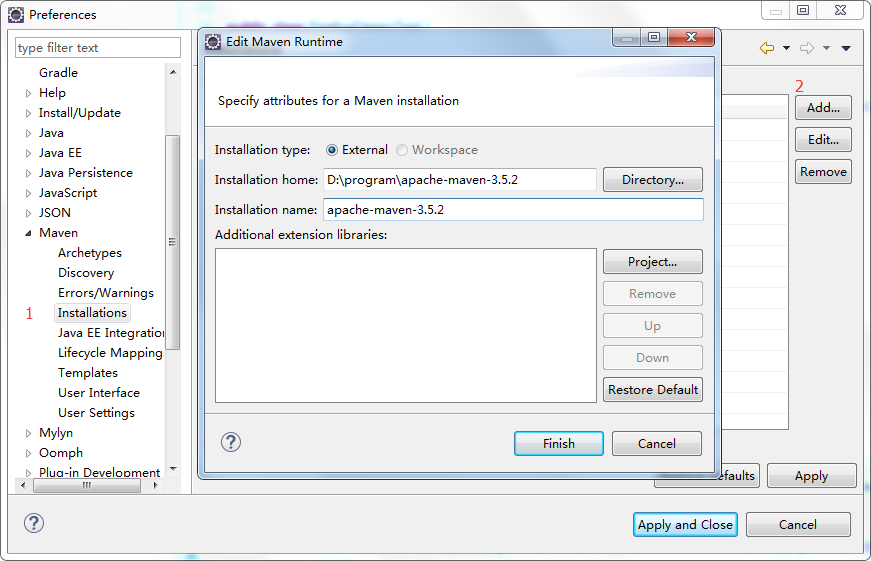
|  |
| --- |
| -vm  <Java>/bin/java.exe |

这样，应该就可以正常启动Eclipse了。

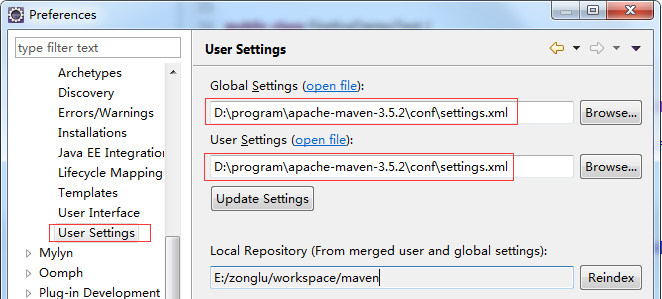
**配置Maven**

然后，我们需要在Eclipse中使用刚刚配置好的Maven。

如下图添加Maven配置，然后勾选使用该新配置。



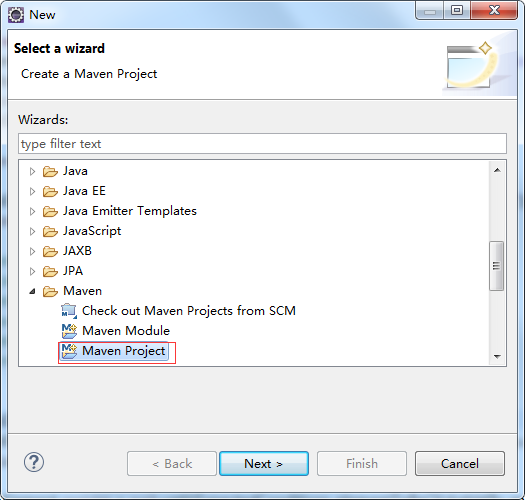
然后，选择使用Maven的配置文件，如下图：



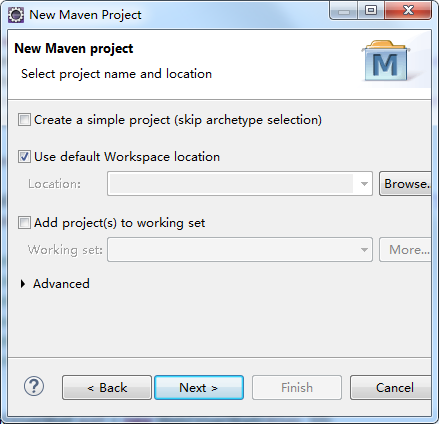
注意：选择了Maven配置文件后，下面的本地仓库路径也会自动改变。最好，点击下“Update Settings”更新下配置。

**创建项目**

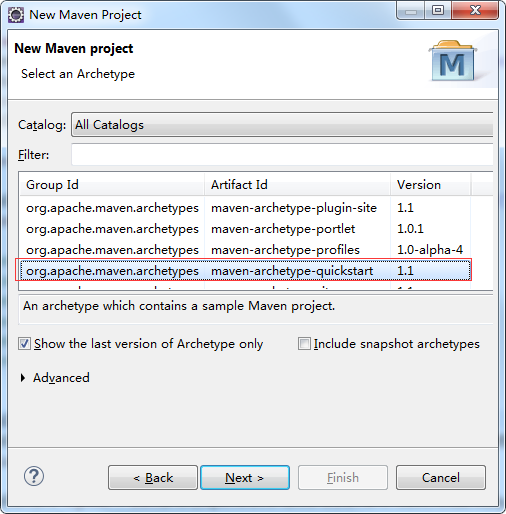
要在项目中使用Maven需要创建Maven工程，因此，菜单中【New】🡪【Other…】，在如下弹出对话框选择“Maven Project”：



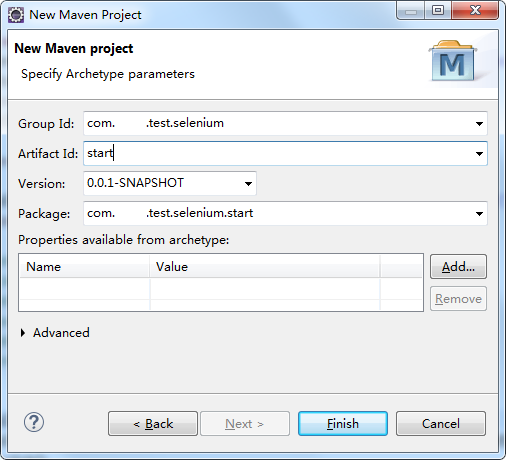
下一步，可自定义工程位置：



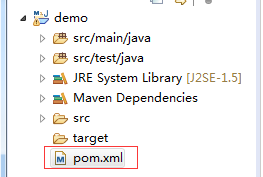
下一步，根据需要选择模板，通常测试选择“quickstart”就可以了。



下一步，填写参数：



完成创建，工程目录结构如下。



其中，pom.xml文件即是工程Maven配置文件，相关依赖也将在其中配置。（具体配置见下文）

## IntelliJ IDEA

待添加。

## Selenium

我们可以在Selenium的官网上下载其软件包，但由于我们使用Maven管理依赖，所以，我们只需要在工程pom.xml配置文件中加依赖即可。

|  |
| --- |
| <dependencies>  …  <dependency>  <groupId>org.seleniumhq.selenium</groupId>  <artifactId>selenium-java</artifactId>  <version>3.8.1</version>  </dependency>  …  </dependencies> |

## JUnit

如果我们使用JUnit编写测试用例，则需要在工程中增加其相关依赖：

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency> |

JUnit的使用这里就不赘述了。

## TestNG

如果使用TestNG编写测试，则需要添加依赖：

|  |
| --- |
| <dependency>  <groupId>org.testng</groupId>  <artifactId>testng</artifactId>  <version>${testng.version}</version>  </dependency> |

为了方便执行测试，我们还需要为Eclipse安装TestNG的插件，在Eclipse Marketplace中搜索TestNG并安装即可。

安装成功后，在TestNG的测试类或xml配置文件右键菜单中则会增加执行菜单项。

TestNG详细使用这里就不赘述了。

## AutoIt

这里我安装的是AutoIt CN版本，可以从这里下载：<http://www.autoit3.cn>。除了是中文版外，还集成了一些插件，使用及转换成exe文件也很方便。

## Sikuli

待添加。

# Selenium WebDriver的使用

简单来说，我们总是从创建WebDriver开始的，并通过WebDriver导航页面和窗口，以及查找元素等等。

## WebDriver的实例化

测试开始前，我们总是要实例化一个WebDriver对象，不同的浏览器构造方式不太一样。

但总体来说，我们需要根据具体测试的浏览器来创建XXXDriver对象。通常的问题在于，浏览器可执行文件和驱动文件的位置。通常，如果浏览器被安装在默认位置，则不需要在代码中指定；而驱动文件如果加入Path路径下，也不需要在代码中指定。此种情况下，就可以直接new XXXDriver()创建WebDriver对象。

但一些情况下，我们可能想要手动配置测试环境，这可以通过代码实现。

### FirefoxDriver

指定Firefox可执行文件和驱动文件路径最简单的方法是，设置系统属性：

|  |
| --- |
| System.setProperty("webdriver.gecko.driver","…/geckodriver.exe");  System.setProperty("webdriver.firefox.bin", "…/firefox.exe"); |

当然，也可以使用GeckoDriverService来创建WebDriver：

|  |
| --- |
| FirefoxBinary bin = new FirefoxBinary(new File("…/firefox.exe"));  GeckoDriverService service = new GeckoDriverService.Builder()  .usingFirefoxBinary(bin)  .usingDriverExecutable(new File("…/geckodriver.exe"))  .build();  WebDriver driver = new FirefoxDriver(service); |

### ChromeDriver

Chrome的配置跟Firefox很类似。

设置系统属性：

|  |
| --- |
| System.setProperty("webdriver.chrome.driver", "…/chromedriver.exe"); |

通过ChromeDriverService创建：

|  |
| --- |
| ChromeOptions options = new ChromeOptions();  options.setBinary(new File(“…/chrome.exe”));  ChromeDriverService service = new ChromeDriverService.Builder()  . usingDriverExecutable(new File(“…/chromedriver.exe”))  .build();  WebDriver driver = new ChromeDriver(service, options); |

## 基本操作

通常，我们常用的基本操作都从WebDriver开始。

### 页面操作

**设置浏览器的窗口大小**

|  |
| --- |
| driver.manage().window().maximize(); // 最大化  driver.manage().window().setSize(new Dimension(480, 800)); |

**导航到测试页面**

|  |
| --- |
| driver.get( "http://www.baidu.com" ); // 导航到页面（方法一）  driver.navigate().to( "http://www.baidu.com" ); // 导航到页面（方法二） |

**“前进”和“后退”**

|  |
| --- |
| driver.navigate().forward(); // 前进  driver.navigate().back(); // 后退 |

**刷新**

|  |
| --- |
| driver.navigate().refresh(); |

### 查找元素

常用的元素查找方法有下面两种：

|  |
| --- |
| WebElement findElement(By by)  java.util.List<WebElement> findElements(By by) |

findElement()返回第一个查找到的元素，如果不存在则抛出异常；findElements()返回查找到的所有元素，如果不存在返回空列表。

需要注意两点：

1. Selenium推崇面向接口编程，因此，返回的元素都是接口WebElement；
2. WebElement也实现了查找接口，因此，也可以进行查找。

### 元素定位

查找元素时，需要指定“定位器”，通常我们使用抽象类org.openqa.selenium.By的静态方法来创建所需的定位器。

* **By ID**

通过ID查找。这是最高效也最常用的定位方式。但是，可能存在的陷阱是：网页中ID并非唯一或者ID是自动生成的。使用中应该避免这些情况。

|  |
| --- |
| WebElement element = driver.findElement(By.id("xxx")); |

* **By Name**

通过name属性查找。

|  |
| --- |
| WebElement cheese = driver.findElement(By.name("cheese")); |

* **By XPath**

通过XPath查找。WebDriver试图在任何地方使用浏览器原生XPath能力，但是对于原生不支持XPath的浏览器，将提供实现。这可能导致一些非预期的行为，除非你清楚不同XPath引擎间的差异。

| **Driver** | **Tag and Attribute Name** | **Attribute Values** | **Native XPath Support** |
| --- | --- | --- | --- |
| [HtmlUnit Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#htmlunit-driver) | Lower-cased | As they appear in the HTML | Yes |
| [Internet Explorer Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#internet-explorer-driver) | Lower-cased | As they appear in the HTML | No |
| [Firefox Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#firefox-driver) | Case insensitive | As they appear in the HTML | Yes |

比如如下页面：

|  |
| --- |
| <input type="text" name="example" />  <INPUT type="text" name="other" /> |

查找语句为：

|  |
| --- |
| List<WebElement> inputs = driver.findElements(By.xpath("//input")); |

匹配结果为：

| **XPath expression** | [HtmlUnit Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#htmlunit-driver) | [Firefox Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#firefox-driver) | [Internet Explorer Driver](http://docs.seleniumhq.org/docs/03_webdriver.jsp#internet-explorer-driver) |
| --- | --- | --- | --- |
| //input | 1 (“example”) | 2 | 2 |
| //INPUT | 0 | 2 | 0 |

注意：有时元素不必声明某些有默认值的属性，比如input元素的type属性可以缺省，缺省值为text。但是在WebDriver中使用XPath时，你不能期望能够匹配未声明的默认属性。

* **By Class Name**

通过class属性查找。实际使用中相同的class属性一般对应多个元素，所以一般会查找到多个元素，然后取用第一个元素。

|  |
| --- |
| List<WebElement> cheeses = driver.findElements(By.className("cheese")); |

* **By Link Text**

通过超链接文本查找超链接。

|  |
| --- |
| WebElement cheese = driver.findElement(By.linkText("cheese")); |

* **By Partial Link Text**

通过部分超链接文本查找超链接。

|  |
| --- |
| WebElement cheese = driver.findElement(By.partialLinkText("cheese")); |

* **By Css Selector**

通过Css Selector查找。如果浏览器默认原生支持css查询，可参阅[w3c css selectors](http://www.w3.org/TR/CSS/#selectors)；否则[Sizzle](http://sizzlejs.com/)被使用。IE6，7和FF3.0都使用Sizzle作为css查询引擎。

注意并非所有的浏览器都是等效的，有的css在一个中起作用也许在另一个中不会。

|  |
| --- |
| WebElement cheese = driver.findElement(By.cssSelector("#food span.dairy.aged")); |

* **By TagName**

通过元素标签名查找。

|  |
| --- |
| WebElement frame = driver.findElement(By.tagName("iframe")); |

### 通用操作

WebElement接口定义了很多元素的通用操作，包括点击、输入、信息获取……，详见API文档。

### 特殊元素操作

有一些元素比较特殊，部分操作不在WebElement接口中定义，或借用某个不太直观的方法。下面列举一些比较特殊的用法。

* **文件域**

输入文件路径：

|  |
| --- |
| file.sendKeys(filePath); |

* **下拉列表**

对于下拉列表这种较复杂的元素，使用通用的方法来操作将不是那么高效的。我们可以将其封装为Select对象，这将提供更多有用的方法便于操作。

|  |
| --- |
| Select select = new Select(driver.findElement(By.tagName("select")));  select.selectByVisibleText(“\*\*\*”);  select.deselectByVisibleText(“\*\*\*”);  select.selectByValue(“\*\*\*”);  select.deselectByValue(“\*\*\*”);  select.deselectAll();  select.getAllSelectedOptions();  select.getFirstSelectedOption(); |

* **表单**

一旦填写完表单，我们就期望提交它。一个提交的方法是找到“提交”按钮并点击它。

|  |
| --- |
| driver.findElement(By.id("submit")).click(); |

另外，元素都提供了一个方便的submit()方法。如果你调用一个处于表单中的元素的submit()方法，将沿DOM向上查找直到找到表单然后提交。如果元素不在表单中，则会抛出NoSuchElementException异常。

|  |
| --- |
| element.submit(); |

* **对话框**

我们可以获取到各种弹出对话框的对象，并可以执行accept、dismiss、读取内容或者向prompt中输入内容等操作。

|  |
| --- |
| Alert alert = driver.switchTo().alert();  alert.accept();  alert.dismiss();  alert.getText();  alert. sendKeys(); |

### 窗口及框架切换

**扩展知识**

|  |
| --- |
| 需要理解的一点是，当浏览器打开一个页面时，可能存在弹出窗口，或者页面是框架集结构，或者包含有iframe。  这些情况下，Selenium的“焦点”是在主窗口的顶级文档之上的。  要操作主窗口顶级文档之外的元素，则需要转换“焦点”。 |

Selenium支持在命名窗口间切换：

|  |
| --- |
| driver.switchTo().window(nameOrHandle); |

怎样知道窗口的名字呢？查看一下打开它的JS或超链接：

|  |
| --- |
| <a href="somewhere.html" target="windowName">Click here to open a new window</a> |

另外，也可以使用窗口句柄，这可能需要迭代所有窗口句柄来获取某个窗口的句柄了：

|  |
| --- |
| for (String handle : driver.getWindowHandles()) {  driver.switchTo().window(handle);  } |

在框架间切换也是类似的：

|  |
| --- |
| driver.switchTo().frame("frameName");  WebElement xf = driver.findElement(by);  driver.switchTo().frame(xf);  driver.switchTo().defaultContent(); |

## 高级操作

### 鼠标操作

|  |
| --- |
| // 拖拽元素到目标元素中  new Actions( driver ).dragAndDrop( element, target ).perform();  // hover  new Actions(driver).moveToElement( text ).perform(); |

### 键盘操作

|  |
| --- |
| sendKeys(Keys.CONTROL, "a"); // 组合键 |

### 等待

**无条件等待**

|  |
| --- |
| Sleeper.SYSTEM\_SLEEPER.sleep(new Duration(5, TimeUnit.SECONDS)); |

**显式等待**

显式等待使用WebDriverWait类和ExpectedCondition接口完成，编码实现ExpectedCondition接口用以判断某种条件，然后等待这种条件达成或超时。

|  |
| --- |
| WebElement myDynamicElement = (new WebDriverWait(driver, 10))    .until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicElement"))); |

ExpectedConditions类提供了许多静态工厂方法，返回常用的ExpectedCondition对象。

**隐式等待**

隐式等待影响设置后全局的相关操作的等待时间。

|  |
| --- |
| // 页面加载超时时间设置为5s。超时异常。  driver.manage().timeouts().pageLoadTimeout(5, TimeUnit.SECONDS);  // 定位对象超时时间设置为10s。超时NoSuchElement异常。  driver.manage().timeouts().implicitlyWait( 10, TimeUnit.SECONDS );  // 异步脚本的超时时间设置为3s。  driver.manage().timeouts().setScriptTimeout(3, TimeUnit.SECONDS); |

### 截图

测试过程中，可以把页面截图，保存成为图片。

需要注意，截取的图片存在临时目录，所以需要将其拷贝到指定的目录，否则测试完成可就找不到截图了。

|  |
| --- |
| File screenshotFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.FILE);  FileUtils.copyFile( screenshotFile, new File("E:\\1.png") ); |

### 执行JS

可以执行任意JS脚本来查找一个元素，并且只要返回一个DOM元素，它将被自动转换为WebElement对象。

|  |
| --- |
| WebElement element = (WebElement) ((JavascriptExecutor)driver).executeScript("return $('.cheese')[0]"); |

### Cookie

|  |
| --- |
| Cookie c1 = new Cookie("name", "key-aaaaaaa");  Cookie c2 = new Cookie("value", "value-bbbbbb");  driver.manage().addCookie(c1);  driver.manage().addCookie(c2);  //获得 cookie  Set<Cookie> coo = driver.manage().getCookies();  //删除所有 cookie  //driver.manage().deleteAllCookies(); |

### 下载

下载文件时有个特殊的问题是，可能会弹出下载窗口，而该窗口属于系统窗口，超出了页面范畴，并且selenium也没有提供接口进行相关操作。

一种处理方案是通过类似autoIt这样的脚本对下载窗口进行处理，另一种方案是配置浏览器使其自动下载到指定目录。

当然，前一种方案已经超出了selenium的范畴。

下面我们分浏览器来分别看下这些方案的具体实现。

* **Firefox**

无论是配置为自动下载还是手动保存，都是通过Firefox的首选项进行设置的。首选项可以通过在地址栏输入“about:config”进行访问修改。

要配置Firefox的首选项，需要构建一个FirefoxProfile对象，并设入创建的FirefoxOptions对象中。option对象会传入driver构造器，用以修改首选项。

模板代码大致如下：

|  |
| --- |
| FirefoxOptions options = new FirefoxOptions();  FirefoxProfile profile = new FirefoxProfile();  // Don't open unknown type of MIME  profile.setPreference("browser.helperApps.alwaysAsk.force", false);  // Don't popup alert dialog  profile.setPreference("browser.download.manager.alertOnEXEopen", false);  profile.setPreference("browser.download.manager.focusWhenStarting", false);  profile.setPreference("browser.download.manager.useWindow", false);  profile.setPreference("browser.download.manager.showAlertOnComplete", false);  profile.setPreference("browser.download.manager.closewhenDone", false);  ...  options.setProfile(profile);  driver = new FirefoxDriver(options); |

需要关注的关键首选项有以下这些：

|  |  |  |
| --- | --- | --- |
| **首选项名称** | **说明** | **类型/候选值** |
| browser.download.useDownloadDir | 是否自动下载 | 布尔 |
| browser.download.folderList | 下载目录 | 0：桌面  1：我的下载  2：自定义 |
| browser.download.dir | 自定义下载目录 |  |

**自动下载**

关键代码如下：

|  |
| --- |
| profile.setPreference("browser.download.useDownloadDir", true); // ①  profile.setPreference("browser.download.folderList", 2); // ②  profile.setPreference("browser.download.dir", "d:\\java"); // ③  profile.setPreference("browser.helperApps.neverAsk.saveToDisk", "binary/octet-stream"); //④ |

第1行代码指定使用自动下载，第2行代码指明使用自定义下载目录，第3行代码指定自定义下载录路径。

需要特别说明的是第4行代码，Firefox浏览器在下载/打开文件时，默认行为与其他浏览器有所差异。无论是否自动下载，都会先弹出一个确认对话框，而如果不想看到这个对话框则需要设置第4行的首选项。该首选项的值是一个字符串，内容为下载文件类型对应的MIME值，多个值间用“,”（英文逗号）分隔。另外，从首选项名称可看出，这只是禁用了保存操作的确认，通常也会禁用打开操作确认，即首选项“browser.helperApps.neverAsk.openFile”。

**手动保存**

手动保存配置就简单一点，关键就是设置“browser.download.useDownloadDir”为false。

当然，下载/打开确认还是禁用为好，否则AutoIt脚本将需要额外操作相关确认对话框。

* **Chrome**

与Firefox配置类似，只是不需要创建profile对象，而是使用一个Map装载，然后设置入option对象即可。

|  |
| --- |
| HashMap<String, Object> chromePrefs = new HashMap<String, Object>();  chromePrefs.put("download.prompt\_for\_download", false);  ...  options.setExperimentalOption("prefs", chromePrefs);  driver = new ChromeDriver(options); |

**自动下载**

关键代码如下：

|  |
| --- |
| chromePrefs.put("profile.default\_content\_settings.popups", 0);  chromePrefs.put("download.prompt\_for\_download", false);  chromePrefs.put("download.default\_directory", downloadFilepath); |

类似地，也是指定为自动下载，并指定了下载目录。

**手动下载**

关键将“download.prompt\_for\_download”设置为true即可。

### 浏览器配置

通常，我们可以通过一些配置选项来设置浏览器的一些特性，上文中已经介绍了下载行为相关的一些设置。

* **Chrome**

初始最大化

|  |
| --- |
| options.addArguments("start-maximized"); |

## 高级特性

### Headless

* htmlunit

selenium内置无头浏览器内核，纯Java编写，页面渲染能力可能与预期不一致。

* PhantomJS

使用 QtWebKit 浏览器内核渲染页面，基本可以和真正浏览器保持一致。

* Chrome-headless

Google 针对 Chrome 浏览器 59版新增加的一种模式，可以让你不打开UI界面的情况下使用 Chrome 浏览器，所以运行效果与 Chrome 保持完美一致。

Mac和Linux下，需要Chrome版本59+；Windows下，需要Chrome版本60+；chromedriver版本2.30+。

|  |
| --- |
| options.setHeadless(true); |

注：默认情况下，Chrome headless模式的窗口大小应该800\*600，这可以从截图结果得知，如果想修改窗口大小，设置相关选项即可：

|  |
| --- |
| options.addArguments("--window-size=1920,1080"); |

* **Firefox-headless**

与Chrome类似。

Windows和Mac下，需要Firefox版本56+；Linux下，需要Firefox版本55+。

|  |
| --- |
| options.setHeadless(true);  options.addArguments("--disable-gpu"); // Temporarily needed if running on Windows |

注：Firefox headless模式下默认窗口大小为1366\*686（类似Chrome设置窗口大小的参数不会生效，尚不清楚设置方法）。

# 附录

## 注意事项

### 关于get()

使用get()方法导航页面时，WebDriver 判断页面加载完成的依据是onload事件被触发。

需要注意的是，如果页面在加载时包含AJAX请求，那么可能就无法得到其是否加载完成了。

### 关于click()

当调用WebElement.click()时，如果被操作元素是不可用的（disabled），click()方法调用仍会是正常的，不会有异常。

所以，如果会出现被点击的元素是不可用的情况，而点击不可用元素又被认为是非法的，那么应该在click()调用前使用isEnable()方法检查元素可用性。

### driver.close() vs.driver.quit()

在测试结束后通常需要关闭浏览器，这个操作通常使用close()或quit()方法完成。但是两者是有区别的，其方法说明也体现了这一点。

|  |  |
| --- | --- |
| close() | Close the current window, quitting the browser if it's the last window currently open. |
| quit() | Quits this driver, closing every associated window. |

对于close()而言，如果当前窗口是最后一个打开窗口，那么关闭当前窗口退出浏览器；quit()则关闭每个关联的窗口退出浏览器。

实际使用中发现，即使是只有最后一个窗口的情况下，调用两个方法还是有差异的。简单来说，close()通常不会关闭驱动进程，会使驱动进程驻留内存；而quit()方法会关闭驱动进程，不过通常会有个进程终止错误，但可忽略。总之，应尽量使用quit()来结束测试。

## FAQ

### 关于Firefox驱动兼容性

* Gecko driver对Firefox的兼容性

Gecko驱动的发布页面会说明推荐的Fireofx版本

Note that with geckodrwer vO.19.O the following versions are recommended: 
• Firefox 55.0 (and greater) 
Selenium 3.5 (and greater) 

* 内置Webdriver对Firefox的兼容性

如果使用webdriver驱动, 则需要在 selenium-server-standalone-x.x.x.jar!\org\openqa\selenium\firefox\webdriver.xpi!\install.rdf

Firefox 
<em targetApplication> 
<Description> 
<em minVersion>3 minVersion> 
<em maxVersion>52 0</em:maxVersion> 
</Description> 
</em targetApplication> 

注：selenium3.8.1已不再使用Webdrver驱动，而转用Gecko驱动，但这些文件还是保留的。

### 暂不支持H5拖放

对于拖放操作而言，我们一般使用如下代码完成操作：

|  |
| --- |
| actions.dragAndDrop(source, target).perform(); |

但是，由于Selenium目前对H5的支持还不好，所以，上面的代码并不能用于执行H5原生拖放操作。

目前，通常使用js来模拟，需要被注入的脚本如下：

|  |
| --- |
| function simulateDragDrop(sourceNode, destinationNode) {  var EVENT\_TYPES = {  DRAG\_END: 'dragend',  DRAG\_START: 'dragstart',  DROP: 'drop'  };  function createCustomEvent(type) {  var event = new CustomEvent('CustomEvent');  event.initCustomEvent(type, true, true, null);  event.dataTransfer = {  data: {  },  setData: function(type, val) {  this.data[type] = val;  },  getData: function(type) {  return this.data[type];  }  };  return event;  }  function dispatchEvent(node, type, event) {  if (node.dispatchEvent) {  return node.dispatchEvent(event);  }  if (node.fireEvent) {  return node.fireEvent('on' + type, event);  }  }  var event = createCustomEvent(EVENT\_TYPES.DRAG\_START);  dispatchEvent(sourceNode, EVENT\_TYPES.DRAG\_START, event);  var dropEvent = createCustomEvent(EVENT\_TYPES.DROP);  dropEvent.dataTransfer = event.dataTransfer;  dispatchEvent(destinationNode, EVENT\_TYPES.DROP, dropEvent);  var dragEndEvent = createCustomEvent(EVENT\_TYPES.DRAG\_END);  dragEndEvent.dataTransfer = event.dataTransfer;  dispatchEvent(sourceNode, EVENT\_TYPES.DRAG\_END, dragEndEvent);  } |

注入js脚本大致的思想就是利用js创建<script>标签，因此，可以预见的方法无非是通过src属性指定js脚本文件地址，要么读取整个脚本文本加入标签中。

对于指定地址而言，简单自不必说，但有一个缺点是我们需要在某个服务器上发布该js脚本文件。而注入整个脚本块的方式不需要这样一个服务器，但是，另一方面，更复杂是显而易见的，而且拼接对语法的要求极其严格，因为换行的因素，可能导致出现一些语法错误。

**另一种方案**

除了使用js脚本操作外，我们也可以模拟鼠标操作。

很容易想到，使用Robot来处理。但是，转而会面临的问题是，我们只能获取到元素在页面中的位置，却不知道其在屏幕上的位置。

一个折衷获取元素屏幕位置的方案是，将浏览器全屏化，这时，元素在页面中的位置和在屏幕上的位置就重合了。

### 元素高亮

在debug过程中，很多时候我们想查看查找到的元素是否就是我们想要的，这时，最直接的方式就是让该元素高亮。我们可以使用js给元素添加临时外边框来实现：

|  |
| --- |
| private static final String JS\_HIGHLIGHT = "element = arguments[0];" +  "original\_style = element.getAttribute('style');" +  "element.setAttribute('style', original\_style + \";" +  "background: yellow; border: 2px solid red;\");" +  "setTimeout(function(){element.setAttribute('style', original\_style);}, 1000);"; |

### 注入jQuery

|  |
| --- |
| public Boolean jqueryIsLoaded(){  Boolean loaded;  try{  loaded = (Boolean)((JavascriptExecutor)driver).executeScript("return jQuery()!=null");  }catch(Exception e){  loaded = false;  }  return loaded;  }  public void injectJQuery(){  String jquery = "var headID=document.getElementsByTagName(\"head\")[0];" +  "var newScript = document.createElement('script');" +  "newScript.type='text/javascript';" +  "newScript.src='.../1.9.1/jquery.min.js';" +  "headID.appendChild(newScript);";  ((JavascriptExecutor)driver).executeScript(jquery);  } |

## 参考

### Chrome driver 与 chrome 版本对应

|  |  |
| --- | --- |
| **chromedriver版本** | **支持的Chrome版本** |
| v2.35 | v62-64 |
| v2.34 | v61-63 |
| v2.33 | v60-62 |
| v2.32 | v59-61 |
| v2.31 | v58-60 |
| v2.30 | v58-60 |
| v2.29 | v56-58 |
| v2.28 | v55-57 |
| v2.27 | v54-56 |
| v2.26 | v53-55 |
| v2.25 | v53-55 |
| v2.24 | v52-54 |
| v2.23 | v51-53 |
| v2.22 | v49-52 |
| v2.21 | v46-50 |
| v2.20 | v43-48 |
| v2.19 | v43-47 |
| v2.18 | v43-46 |
| v2.17 | v42-43 |
| v2.13 | v42-45 |
| v2.15 | v40-43 |
| v2.14 | v39-42 |
| v2.13 | v38-41 |
| v2.12 | v36-40 |
| v2.11 | v36-40 |
| v2.10 | v33-36 |
| v2.9 | v31-34 |
| v2.8 | v30-33 |
| v2.7 | v30-33 |
| v2.6 | v29-32 |
| v2.5 | v29-32 |
| v2.4 | v29-32 |

## 知识扩展

### W3C WebDriver

**参考文档**：

<https://www.w3.org/TR/webdriver/>

<https://www.w3.org/testing/Activity>

<https://github.com/w3c/webdriver>

### MIME

<http://www.w3school.com.cn/media/media_mimeref.asp>

### Headless

Firefox headless: <https://developer.mozilla.org/en-US/Firefox/Headless_mode>

Chrome headless: <https://developers.google.cn/web/updates/2017/04/headless-chrome>