Selenium Web 自动化测试实训教程

V1.0

目录

1. 自动化测试概述

1.1 什么是自动化

广义上来讲，自动化包括一切通过工具（程序）的方式来代替或辅助手工测试的行为都可以看做自动化，包括性能测试工具（Loadrunner、Jmeter）,或自己所写的一段程序，用于生成 1 到 100 个测试数据。

狭义上来讲，通工具记录或编写脚本的方式模拟手工测试的过程，通过回放或运行脚本来执行测试用例，从而代替人工对系统的功能进行验证。

1.2 自动化测试的适用范围

① 主体需求明确，不会频繁变动
② 每日构建后的测试验证
③ 比较频繁的回归测试
④ 软件系统界面稳定，变动少
⑤ 需要在多平台上运行的相同测试案例、组合遍历型的测试，大量的重复任务
⑥ 软件维护周期长
⑦ 项目进度压力不太大
⑧ 被测软件系统开发较为规范，能够保证系统的可测性
⑨ 具备大量的自动化测试平台
⑩ 测试人员具备较强的编程能力

以上 10 条，并非需要全部具备才能开展自动化测试。一般来说满足下面三个条件就可以开展自动化测试：

① 软件需求变动不频繁（局部模块也可以）
② 项目周期较长
③ 自动化测试脚本可重复使用

1.3 自动化误区

自动化误区	说明
比人工测试更先进	各有优缺点，互为补充
很快能大幅度减少测试工作量	前期需要投入大量人力
极大提供测试覆盖率	主要由测试用例决定
能发现大量新缺陷	

1.4 常见类型

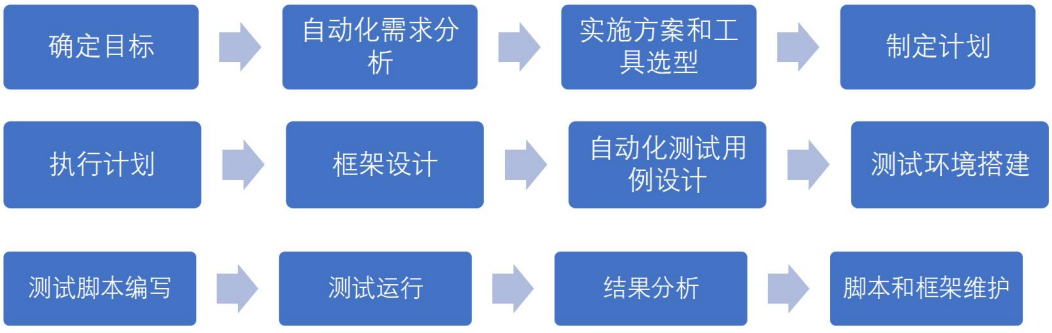
单元测试 、接口测试框架

- java（Junit、testNG）
- C#（Nunit）
- python （unittest、pytest）

UI 测试工具

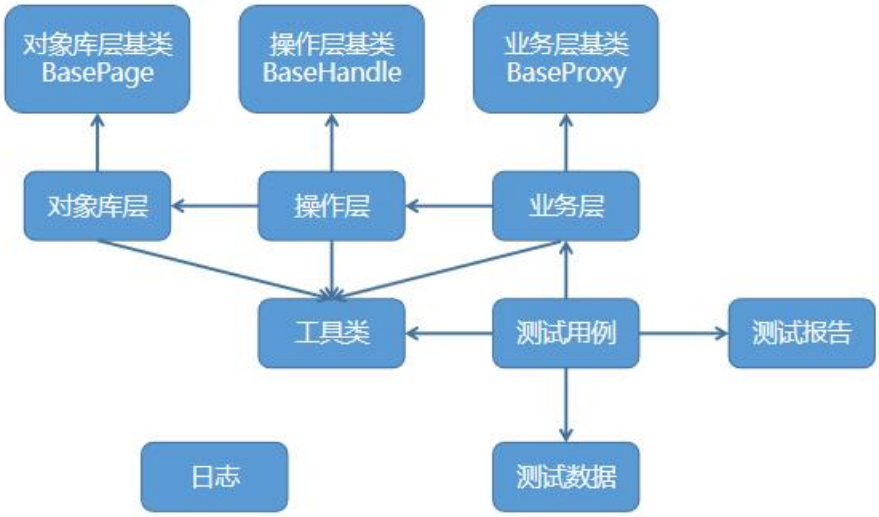
- QTP （ VBScript ）
- Selenium
- Watir （Ruby）
- Autoit（类 Basic 语言）

1.5 Web 自动化实施流程



自动化测试的流程

1. 需求分析
2. 挑选适合做自动化测试的功能
3. 设计测试用例
4. 搭建自动化测试环境 [可选]
5. 设计自动化测试项目的架构 [可选]
6. 编写代码
7. 执行测试用例
8. 生成测试报告并分析结果



1.6 自动化常见工具

自动化测试工具有很多，这里介绍几种常用的：

1. Selenium

- 特点：开源免费，支持多浏览器（Chrome/Firefox/Edge 等）、多编程语言（Python/Java/C# 等），生态成熟，跨平台、跨浏览器兼容性强；可与 Pytest/JUnit 等框架集成，支持复杂场景；Selenium 4 新增相对定位、W3C 标准支持，稳定性提升。
- 适用场景：Web 应用的 UI 回归测试、跨浏览器验证。
- 缺点：需要手动处理元素等待，原生不支持报告生成（需依赖第三方库）。

2. Cypress

- 特点：前端自动化测试工具，基于 JavaScript/TypeScript，自带浏览器环境和录制功能。自动等待元素加载（无需手动设置 WebDriverWait）；实时重载、时间旅行（可回放测试步骤）；内置断言、截图、录屏功能。
- 适用场景：现代前端框架（React/Vue/Angular）的 UI 测试，快速迭代的项目。

- 缺点：仅支持 JavaScript，不支持多浏览器并行（需付费版）。

3. Playwright

- 特点：微软开源，支持多浏览器（Chrome/Edge/Firefox/WebKit）、多语言（Python/Java/JS/C#）。自动等待 + 智能断言，稳定性优于 Selenium；支持单页应用（SPA）、Shadow DOM、iframe 等复杂结构；内置录制工具，可直接生成代码。
- 适用场景：复杂 Web 应用（如金融、电商）的自动化测试，需要跨浏览器一致执行的场景。
- 缺点：生态较新，部分老系统兼容性略差

UFT（QTP）：

企业级自动化测试工具，提供强大、易用的录制回放功能，同时兼容对象和图像两种识别模式，支持 B/S 和 C/S 两种架构的软件测试,属于付费工具 ；

Robot Framework（RF）：

基于 Python 语言编写的自动化测试框架，具备良好的可扩展性，支持关键字驱动，同时可测试多种类型的客户端或接口，可进行移动端测试，开源的；

1.7 自动化测试用例

只要涉及到测试的工作内容，都会涉及到测试用例，测试用例是参考、是思路、是指导，因此自动化测试也需要这样一个参考、思路和指导。否则没有用例的情况下直接编写脚本，想到一点写一点，最终会陷入茫然混乱之中，导致自动化测试的失败。

自动化测试用例与我们手工的测试用例略有不同，主要是自动化测试脚本一旦编写，肯定会是一个较为完整的功能或者流程，而不像手工测试用例，还要每个输入框去考虑输入输出。

以下是一些供参考的自动化测试用例编写原则：

1. 一个用例为一个完整的场景，从用户登录系统到最终退出并关闭浏览器；
2. 一个用例值验证一个功能点，不要试图在用户登录系统后把所有功能都验证一遍；
3. 尽可能少的编写逆向逻辑用例。一方面因为逆向逻辑的用例太多；另一方面自动化脚本本分比较脆弱，复杂的逆向逻辑用例实现起来较为麻烦且容易出错；
4. 用例与用例之间尽量避免产生依赖；
5. 一条用例完成测试之后，需要考虑对测试场景的还原，以免影响其他用例的执行；
6. 用例尽可能细化到每一个操作，如打开什么地址、操作什么控件、点击什么按钮等。

1.8 自动化测试的架构方式

1.8.1 线性测试

通过录制工具录制或编写对 web 程序的操作步骤,产生相应的脚本,每个脚本相对独立,没有依赖与调用，也就是直接通过脚本来模拟用户的操作，这种脚本就是线性的脚本。

这种脚本的优点就是每一个脚本都很独立，都可以直接执行并产生结果。但是缺点也相

当明显，脚本的开发和维护成本非常高，比如登录，每个脚本都必须去重复编写，并且当对应的界面或需求发生变化，就必须逐一的修改。

1.8.2 模块化驱动测试

由于线性驱动的局限性，在自动化的过程中就逐渐引入模块化的概念，把一些常用的重复操作独立成公共模块，脚本需要用到这一模块操作时就可以直接调用，这样就最大限度的消除了重复，提高测试的可维护性。就比如每次需要操作系统时都需要登录，那么把登录模块化为函数，其他的脚本用到的时候直接调用该函数即可。

这种的好处时显而易见的，减少了脚本的重复代码量，如果后期界面和需求改变，只需要到对应的模块中进行修改即可。

1.8.3 数据驱动测试

虽然通过模块化解决了脚本重复的问题，但是依然有诸多不便，比如登录的时候，我第一次测试想用“张三”的用户名登录，下一个测试用例要换成“李四”的用户名登录。这种情况下，还是需要重复的编写脚本，因为虽然登录的步骤相同，但登录所用的测试数据不同。

数据驱动（DDT 或者叫 TDD），就是在这种情况下被引入的。从字面意思来理解，就是数据的改变从驱动自动化测试用例的执行，最终引起测试结果的改变。数据驱动说白了就是参数化。

在参数化过程中，我们可以读取数组、字典中的数据，也可以从外部（txt、csv、excel、xml、数据库等等）获取数据，这些都是数据驱动，目的就是实现数据与脚本的分离。

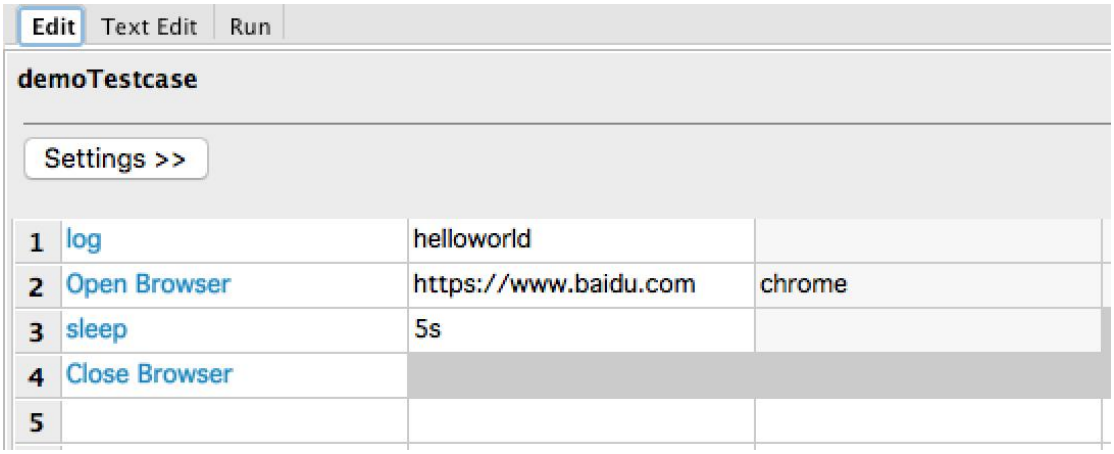
这样就进一步增强了脚本的复用性。

1.8.4 关键字驱动测试

既然都是驱动测试，其实关键字驱动就是把数据换成了关键字，通过关键字的改变引起结果的改变。

目前主流的商业自动化测试工具（如 QTP），都是以关键字为卖点。通过对底层代码的封装，提供给用户图形界面的方式，用户只需要考虑每一步的工作，而不用考虑内部编码的实现；测试人员在写脚本时以近乎类似 excel 写测试用例的方式去编写，从而降低脚本的编写难度。

我们来看一个典型的关键字自动化测试工具 RobotFramework：



上例：蓝色为关键字，黑色字体为数据：

log helloword： 以日志的形式输出 helloworld；

Open Browser … chrome：用 chrome 浏览器打开百度页面；

sleep 5s：等待 5 秒；

Close Browser：关闭浏览器。

2. Selenium 自动化基础

2.1 Selenium 简介

Selenium 是 ThoughtWorks **专为 Web 应用程序编写的一个测试工具。**

主要用于 Web 功能测试和浏览器兼容性测试。

名字起源：在 Selenium 开发的阶段，另外一个测试框架叫 Mercury Interactive。这个框架是最初提出 QTP 的公司研发的。 因为硒可以解水银的毒，所以 Jason 起了这样一个名字。

Selenium 的发展历程

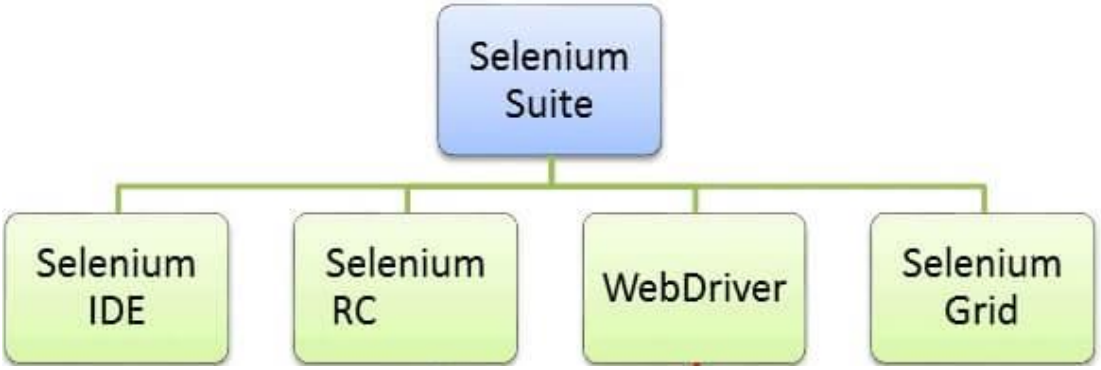
Selenium 的发展经历了多个阶段，逐步从简单工具演进为成熟的自动化测试体系：

- **Selenium IDE**（2004 年）：最初由 Jason Huggins 开发，是一个浏览器插件，支持通过“录制-回放”方式快速生成测试脚本，适合入门级用户。
- **Selenium RC**（Remote Control，远程控制）：解决了浏览器同源策略限制，允许通过编程语言（如 Java、Python）编写测试脚本，但架构较复杂。
- **WebDriver**（2006 年）：由 Simon Stewart 开发，直接与浏览器内核交互（而非通过 JavaScript 注入），稳定性和速度大幅提升，支持更复杂的用户操作。
- **Selenium 2**（2011 年）：将 Selenium RC 与 WebDriver 合并，以 WebDriver 为核心，成为主流版本。
- **Selenium 3**（2016 年）：移除了对 Selenium RC 的支持，完全基于 WebDriver，并增强了对浏览器驱动的管理。
- **Selenium 4**（2021 年）：进一步优化架构，支持 W3C WebDriver 标准（各浏览器

统一接口），新增相对定位、Grid 4 等功能，稳定性和易用性显著提升。

Selenium 的核心组件

Selenium 不是单一工具，而是由多个组件组成的工具集，各自承担不同角色：



1. Selenium IDE

- **定位**：浏览器插件（支持 Chrome、Firefox），可视化录制工具。
- **功能**：通过录制用户操作自动生成测试脚本（支持导出为 Python、Java 等语言），支持简单的断言和回放。
- **适用场景**：快速创建基础测试脚本、演示自动化流程、非技术人员入门。

2. Selenium WebDriver

- **定位**：Selenium 的核心组件，提供编程接口控制浏览器。
- **功能**：通过代码（如 Python、Java、C# 等）直接操作浏览器，支持元素定位、点击、输入、页面跳转等几乎所有用户行为。
- **特点**：
  - 直接与浏览器内核交互，无 RC 的性能瓶颈；
  - 支持所有主流浏览器（Chrome、Firefox、Edge、Safari 等）；
  - 可集成到单元测试框架（如 pytest、JUnit）中，实现复杂场景测试。

3. Selenium Grid

- **定位**：分布式测试工具。
- **功能**：允许将测试用例分发到多台机器（或虚拟机）的不同浏览器/版本上并行执行，大幅缩短测试时间。
- **适用场景**：跨浏览器兼容性测试、大规模测试用例的并行执行。

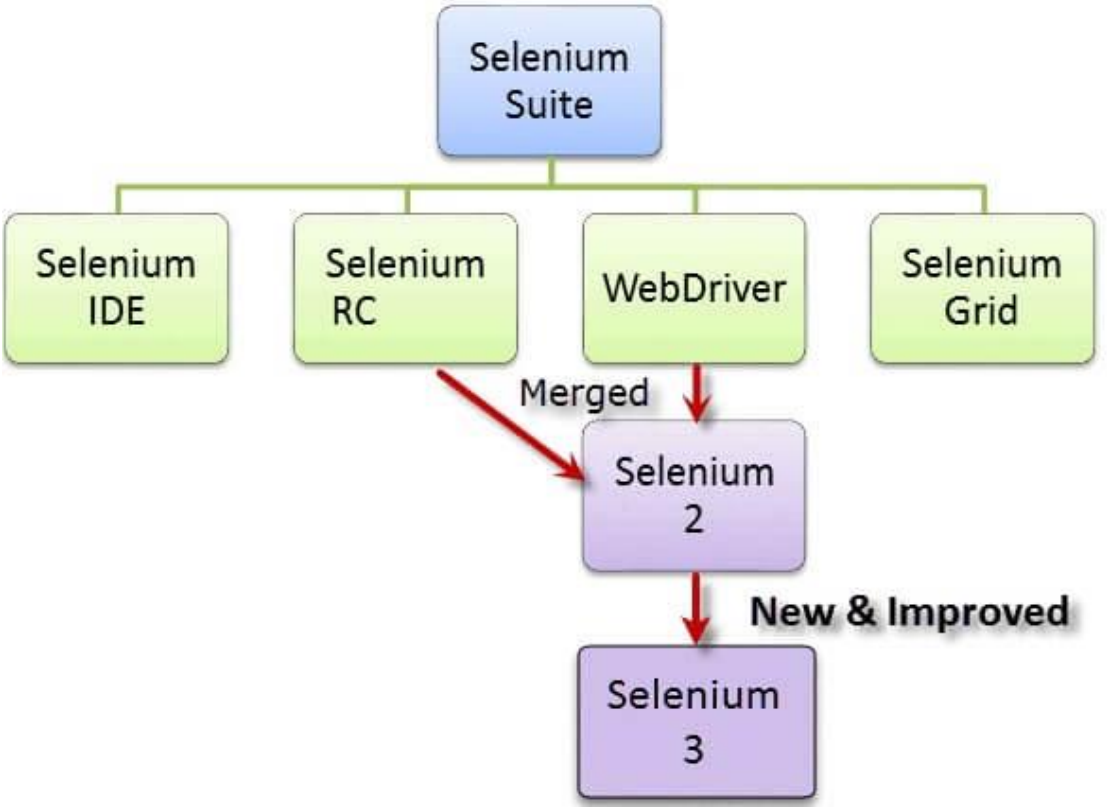
Selenium 的核心特点

1. **开源免费**：无需付费即可使用全部功能，社区活跃，问题解决资源丰富。
2. **跨平台/跨浏览器**：支持 Windows、macOS、Linux 系统，兼容 Chrome、Firefox、Edge 等所有主流浏览器。
3. **多语言支持**：提供 Python、Java、C#、JavaScript 等多种编程语言接口，适配不同技术栈团队。
4. **模拟真实用户操作**：支持点击、输入、下拉选择、鼠标悬停、键盘操作等几乎所有

用户行为，贴近真实使用场景。

5. **可扩展性强**：可与测试框架（pytest、JUnit）、报告工具（Allure、BeautifulReport）、CI/CD 工具（Jenkins）无缝集成，构建完整自动化测试体系。
6. **Selenium 4 新特性**：
   - 相对定位（通过元素间位置关系定位，如“在某个元素下方”）；
   - 内置 Chrome DevTools 协议支持（可控制浏览器性能、网络等）；
   - Grid 4 重构（支持容器化部署，简化分布式测试配置）。

Selenium 演进



Selenium 2 = Selenium RC+WebDriver

Selenium 3 - 停止支持 SeleniumRC。同时增加了支持浏览器类型（Safari 等）

2.2 Selenium 环境搭建

Selenium Python 提供了一个简单的 API 便于我们使用 Selenium WebDriver 编写 功能/验收测试。 通过 Selenium Python 的 API，可以直观地使用所有的 Selenium WebDriver 功能。

Selenium Python 提供了一个很方便的接口来驱动 Selenium WebDriver ，例如 Firefox、Chrome、Ie,以及 Remote。

基于 Python + Selenium 的自动化测试环境安装步骤如下：

1. 安装 python

   在 Python 官网下载对应的电脑配置的 Python 安装包程序，通常使用 Wind64 位的安装包，版本选择目前主流采用 Python3.x,这里我们选择 Python3.7。Python 官网地址：https://www.python.org/

2. 安装 selenium

   selenium 的安装直接在 cmd 命令行安装，安装命令：pip install selenium

3. 安装浏览器及其驱动

   通常自动化测试用的浏览器一般采用主流的浏览器，比如：Chrome、Firefox、IE 等，相应的浏览器都有各种版本，而我们可以采用比较稳定的版本，并下载与之相称的浏览器驱动。

   - Chrome 浏览器驱动

     下载地址： http://chromedriver.storage.googleapis.com/index.html

     不同的 Chrome 的版本对应的 chromedriver.exe 版本也不一样，下载时不要搞错了。

     如果是最新的 Chrome, 下载最新的 chromedriver.exe 就可以了。

   - Firefox 浏览器驱动

     下载地址为：https://github.com/mozilla/geckodriver/releases/

     根据自己的操作系统下载对应的驱动即可，使用的话，需要把驱动的路径和火狐浏览器的路径加入到环境变量里面才可以。

   - IE 浏览器驱动

     下载地址为：http://selenium-release.storage.googleapis.com/index.html

     根据自己 selenium 版本下载对应版本的驱动即可，python 的话，下载里面的 IEDriverServerxxx.zip 即可，这个是区分 32 和 64 位系统的，根据自己的系统下载即可，需要注意的是，如果要打开 IE 浏览器的话，需要在浏览器的 Internet 选项中的安全页里有 4 个安全选项，Internet、本地 Internet、受信任的站点、受限制的站点，这 4 个里面都有一个启用保护模式，都需要勾选上才可以，还得把驱动的路径加入到环境变量中。

   下载浏览器驱动，解压后添加到环境变量，一般放在 python 目录下即可。

   脚本开发工具

4. 脚本开发工具（Selenium 相关依赖包，开发工具）：

   Python IDE 推荐：Pycharm;

2.3 Selenium 浏览器操作

Selenium 体系中用来操作浏览器的一套 API 就是 WebDriver，因此我们讲 Selenium 控制方法，其实就是讲的 WebDriver。WebDriver 针对多种语言都实现了一遍这个 API，因此它可以支持多种编程语言；对于 Python 来说，selenium.webdriver 就是用于 Web 自动化测

试的一个第三方的库。

以下就是 Webdirver 提供的主要几类方法：

WebDriver 控制方法	浏览器操作	打开、关闭、最大化、窗口切换等
	页面元素定位	id、name、linktext、css、xpath 等
	页面元素操作	click、send_keys、clear、text、get_attribute 等
	其他方法	页面等待、frame 切换、调用 JS 等

2.2.1 启动浏览器

对于 WebDriver 的控制方法，先从对浏览器的基本控制说起。要进行 web 页面的测试，首先需要启动浏览器，下面的代码就实现启动浏览器，并打开一个 url（站点）：

```
# 引入 webdriver
from selenium import webdriver
# 启动 chrome 浏览器，也是实例化 Chrome 类
driver = webdriver.Chrome(executable_path=driver_path)
# 打开百度首页
driver.get("http://www.baidu.com")
```

*由于 selenium3.0 的更新，驱动已经独立；按官方解释只要把驱动的目录加到环境变量就可以直接使用不用输入任何参数；如果不成功，需要用 executable_path 参数指明驱动文件的绝对或相对路径。

2.2.2 浏览器控制

1. 控制浏览器大小：

对于 chrome 浏览器，用 WebDriver 启动的时候，是一个缩小的窗口状态，为了方便观察，需要最大化浏览器，那么就要用到 maximize_window()方法；当然，如果你想改成指定大小的话，就要用 set_window_size(480, 800)，那么结果就是设置浏览器为 480 像素宽，800 像素高的窗口。

```
driver.set_window_size(480, 800)        # 480,800 为像素点大小
driver.maximize_window()                # 最大化浏览器
```

2. 后退、前进：

在使用浏览器的过程中，为了方便在浏览过的网页之间切换，WebDriver 也提供了对应的 back()和 forword()方法来模拟后退和前进按钮。

```
driver.back()           # 后退到上一个页面
driver.forward()        # 前进到下一个页面
```

*此方法应用时需谨慎，切换页面最好通过页面元素操作，否则脚本可读性很差，前进后退

很容易搞混。

3. 模拟浏览器刷新：

有时候需要手动刷新（F5）页面，例如页面上有些统计字段，当数据发生改变时，并不会及时变化，需要刷新页面后才会变化。这种时候就需要用到模拟浏览器刷新的方法 refresh()。

```
driver.refresh()          # 浏览器刷新，与 F5 同理
```

4. 退出：

WebDriver 提供了两种退出方式：close()和 quit()，close()方法只关闭当前窗口，不会清理当前 WebDriver 进程，quit()则完全退出 WebDriver 进程。

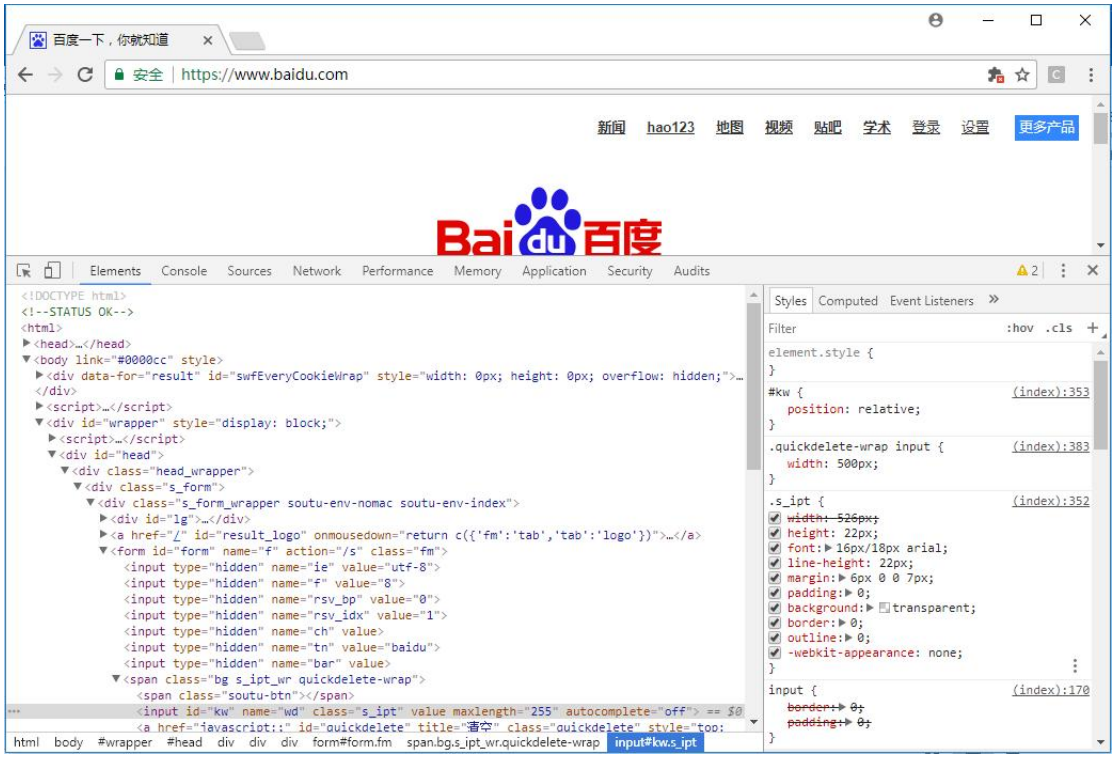
```
driver.close()            # 关闭当前窗口，不会关闭浏览器驱动
driver.quit()             # 退出所有窗口并关闭浏览器驱动
```

*由于 close()仅关闭当前窗口，并不会关闭和清理浏览器驱动以及由此产生的缓存文件等，如果长期如此，将会产生很多垃圾文件，因此结束测试退出时应当使用 quit()。

2. 4 Selenium 元素定位

现在我们就进入 WebDriver 的最重要部分，也是 Selenium 中的最重要部分，元素定位。我们做自动化测试，就是需要通过模拟手工对元素的操作来实现自动化。但是脚本的模拟又不像手工操作，你想操作哪个元素，鼠标放上去点击就可以了；对于脚本来说，必须要先找到对应的元素，才能对元素进行操作。

首先，我们打开百度首页（http://www.baidu.com），在这个页面上有搜索的输入框、有搜索按钮、以及图片和各种链接等。自动化测试就是要通过模拟鼠标和键盘操作来实现输入、点击等。就如上面所说，要完成对元素的输入、点击等操作的话，必须先找到它们。怎么找到？我们需要用到浏览器的开发者工具(很多浏览器都有,建议用 Firefox 或者 chrome，本教材中用到的浏览器都是 chrome），快捷键 F12，打开后如下图：



我们通过上图可以看到，页面上得元素都是由一行行的 HTML 代码组成，它们之间是有层级组织起来的，每个元素有不同的标签名和属性值。WebDriver 就是通过元素的标签名或属性值来找元素的。

观察上图中的代码，可以看到 HTML 代码有如下特征：

1. 由标签对构成

```
<html></html>
<body></body>
<div></div>
```

html、div 就是标签名 tag name。

2. 标签有各种属性

```
<div id="wrapper" style="display: block;">
<input id="kw" name="wd" class="s_ipt" value="" maxlength="255" autocomplete="off">
```

属性有 id，name，class 等。

3. 标签对之间有文本数据

```
<label class="checkbox"><input type="checkbox" name="autologin"> 三日内自动登陆</label>
<a href="http://news.baidu.com" name="tj_trnews" class="mnav">新闻</a>
```

在两个标签对之间的文字，就是这个标签对应的文本。思考一下，第一个例子中“三日内自动登陆”属于哪个标签的文本？

4. 标签有层级关系

```
<div>
    <form>
        <input />
    </form>
</div>
```

对于上面的结构，如果把 input 标签看作子标签，那么其父标签就是 form。

上面一直在说标签，那么上面的标签和我们找的元素又有什么关系呢？我们先来看一下 HTML 中元素和标签的定义：***HTML 元素指的是从开始标签(start tag)到结束标签(end tag)的所有代码。***那么其实我们就是通过标签去找元素，通过 WebDriver 去找这样一个匹配的 HTML 元素对象。

了解了上面的特征，那么怎么去找元素呢？WebDriver 提供了八种定位方法，方法如下：

id	find_element_by_id()
name	find_element_by_name()
class_name	find_element_by_class_name()
tag_name	find_element_by_tag_name()
link_text	find_element_by_link_text()
partial_link_text	find_element_by_partial_link_text()
css_selector	find_element_by_css_selector()
xpath	find_element_by_xpath()

id、name、class name、tag name 相当于是元素的本身的属性，类似于人的手机号、身份证号码等等；而 link_text、partial_link_text 仅针对于链接地址；css selector、xpath 就相当于说在没有 id 或 name 等可以精确标识某个元素的时候(这种情况是我们自动化测试遇到的主流)，通过其他一些方式，比如属性值匹配、定位父级再定位子级等方式去找元素。

下面我们就逐一来了解这些方法(由于每个 web 站点用到的技术不相同，因此介绍过程中可能会用到很多的站点，学习时请注意！)：

1. Id 定位：

HTML 规定 id 属性在 HTML 文档中必须唯一。因此 id 是一个类似身份证的概念，唯一性很强。但是这个唯一性要求也不是强制的，笔者以前就遇到过一个项目 id 并不唯一，当然这是个案；还有对于单选框（radio button），一般是一组 input 的元素，id 值都是一样的。这两点在使用时需要注意。

WebDriver 提供的 id 定位方法，就是通过 id 属性去查找元素。下面的例子通过 id 定位百度输入框和百度搜索按钮（为了能看到效果，我们先加上一些简单的操作方法 send_keys()、click()，对于这些操作方法，后面会详细讲解）如下：

```
driver.find_element_by_id("kw").send_keys("刘德华")     # 找到输入框并输入
driver.find_element_by_id("su").click()                 # 找到按钮并点击
```

后面的例子请执行添加 send_keys()、click()，以便看效果。

2. Name 定位：

HTML 规定 name 来指定元素的名称，类似人的姓名，可以不唯一；不过一般情况下，name 属性重复的概率相对较低，当可以确定 name 唯一的情况下，也可以优先使用 name 属性。通过 name 来定位百度输入框。

```
driver.find_element_by_name("wd")
```

3. class name 定位：

HTML 规定 class 来指定元素的类名，常用于 css 样式等。

```
driver.find_element_by_class_name("s_ipt")
```

用 class 属性来定位需要注意，class name 并不唯一，且一个 class 属性中可能包含多个 class name，如百度搜索按钮的 class 属性就是多个，注意观察，在"bg"和"s_btn"之间有空格，有空格就表示是两个类名：

```
<input type="submit" id="su" value="百度一下" class="bg s_btn">
```

对于此类 class，如果用 class name 的方式直接输入"bg s_btn"就会出错，必须单独用"bg"或者"s_btn"。

```
~~driver.find_element_by_class_name("bg s_btn ")~~          #错误
driver.find_element_by_class_name("s_btn ")              #正确
```

另外就是要注意 class 属性的唯一性。并且 class name 可以用 css_selector 代替，后面会讲到。

4. tag name 定位:

在 HTML 页面中，tag name 会大量存在同名的。仔细观察 HTML 代码就可以看出来，一大堆的 、<input>、<a>等等 tag name，所以很难通过 tag name 去区分不同的元素。但是某些情况下 tag name 还是很有用的，后面我们会讲到。

```
driver.find_element_by_tag_name("input")
```

5. link text 定位:

link 定位与前面几种定位方法不同，是一种特殊的定位方式，专门用来定位文本链接的，也就是<a>标签。这种定位方式也是一种效率较高，定位比较准确的方式。因为一般一个页面不会出现两个文本相同的<a>标签，因为如果出现，会让用户疑惑。我们试试用这种方法来定位百度右上角的链接。

```
driver.find_element_by_link_text("新闻")
driver.find_element_by_link_text("地图")
```

注意：

- 这种方式有一个缺点，就是在这些链接文本中有空格或其他字符，而这种在页面或者开发者工具中不太容易注意到，那么这种情况就会导致定位失败。因此使用时最好从开发者工具中去复制文本。

6. partial link text 定位：

partial link text 是 link text 定位的一种补充，比如上面提到的可能有空格或其他

字符，或者链接地址特别长的时候，那么可以取链接中的一部分来进行定位，不过要保证这部分信息可以唯一的标识这个链接。比如百度首页底部的备案号，链接到公安部网站。

```
<a id="jgwab" target="_blank" href="http://...recordcode=11000002000001">京公网安备 11000002000001 号</a>
```

我们可以用 partial link text 的方式定位

```
driver.find_element_by_partial_link_text("京公网")
```

用 partial link text 的时候，就特别要注意唯一性了。

7. css selector 定位:

CSS(Cascading Style Sheets)是一种语言，用来描述 HTML 和 XML 文档的样式。CSS 使用选择器来为页面元素绑定属性。这些选择器可以被 WebDriver 用作另外的定位策略。

CSS 可以较为灵活的选择控件的任意属性,一般情况下定位速度比后面要讲的 Xpath 快。Selenium 官方推荐使用 CSS 进行定位。

要使用 CSS 进行定位，就需要熟悉 CSS 选择器的语法：

选择器	例子	例子描述
.class	.intro	选择 class="intro" 的所有元素。
#id	#firstname	选择 id="firstname" 的所有元素。
element>element	div>p	选择 元素的所有 <p> 子元素。
element element	div p	选择 元素内部的所有 <p> 元素。
element+element	div+p	选择同一级中紧接在 元素之后的所有<p> 元素。
[attribute^=value]	a[src^="https"]	选择其 src 属性值以 "https" 开头的每个 <a> 元素。
[attribute$=value]	a[src$=".pdf"]	选择其 src 属性以 ".pdf" 结尾的所有 <a> 元素。
[attribute*=value]	a[src*="abc"]	选择其 src 属性中包含 "abc" 子串的每个 <a> 元素。
:nth-child	input:nth-child(1)	选择其父元素中的第一个子元素，如果第一个子元素不为 input，则结果为 0。
	:nth-child(1)	选择父级元素中的第一个子元素
:nth-of-type	input:nth-of-type(1)	选择父元素中的第一个 input 子元素。

*以上只截取了较为常用的 CSS 选择器，更多资料请参考：

http://www.w3school.com.cn/cssref/css_selectors.asp

下面同样以百度输入框和搜索按钮为例介绍 css selector 定位方法，HTML 代码如下：

```
…
<span class="bg s_ipt_wr quickdelete-wrap">
        <span class="soutu-btn"></span>
        <input id="kw" name="wd" class="s_ipt"
                        value="" maxlength="255" autocomplete="off">
```

```
</span>
<span class="bg s_btn_wr">
        <input type="submit" id="su" value="百度一下" class="bg s_btn">
</span>
…
```

1） 通过 class 属性定位：

css selector 中的 class 用"."来标识。

```
driver.find_element_by_css_selector(".s_ipt")
driver.find_element_by_css_selector(".bg.s_btn ")
```

注意，之前 class name 定位提到过的多类名的情况，可以采用加多个"."的方式。

2） 通过 id 属性定位：

css selector 中也提供了用 id 属性定位的方式，和前面讲的 id 定位写法上稍有区别，css selector 中用#标识 id，如：#kw。

```
driver.find_element_by_css_selector("#kw")
driver.find_element_by_css_selector("#su")
```

上面的这种写法和 id 定位中的效果是一样的。

3） 通过标签名定位：

直接通过标签名定位，和我们前面讲 tag name 的时候一样，因为同样的标签名在一个页面中有很多，因此想通过下面的标签名去定位显然是不精确的。

```
driver.find_element_by_css_selector("input")
```

不过 css selector 中提供了很多其他的方式，这一点要比 tag name 定位强大很多。

```
driver.find_element_by_css_selector("span>input")   #通过父级找子级
driver.find_element_by_css_selector("span+input")   #通过哥哥找弟弟
driver.find_element_by_css_selector("span input")   #找 span 内部的所有 input
```

4） 通过精确匹配的属性值定位：

css selector 中允许使用元素的任意属性来定位元素，当然前提是这个属性是唯一的。通过属性来定位元素，属性值（如 type="submit"）引号不是必须的。但是对于有空格或其他符号的时候，需要加上引号(如 class="bg s_btn")。外层用了双引号，因此内层用单引号。注意写法！

```
driver.find_element_by_css_selector("[type=submit]")  #百度搜索按钮 type 属性
driver.find_element_by_css_selector("[class='bg s_btn']")#百度搜索 class 属性
```

注意：

- 我们之前遇到的多个 class 名的情况（上面的第二个例子），在这里就必须输入完全，因为这个时候 class 被看成了一个普通属性。
- 属性值外层用的方括号，不要漏掉了。

5） 通过模糊匹配的属性值定位：

上面讲了通过属性值定位，不过是精确匹配，很多时候属性值比较长，不可能全部输入，且部分属性值已经可以唯一标识了。那么就要用到下面几种模糊匹配方式。

```
driver.find_element_by_css_selector("[type^=sub]")    #符号(^)表示以什么开头
driver.find_element_by_css_selector("[type$=mit]")    #符号($)表示以什么结尾
driver.find_element_by_css_selector("[type*=bmi]")    #符号(*)表示只要包含
```

依然要注意引号的使用。

6） 组合定位：

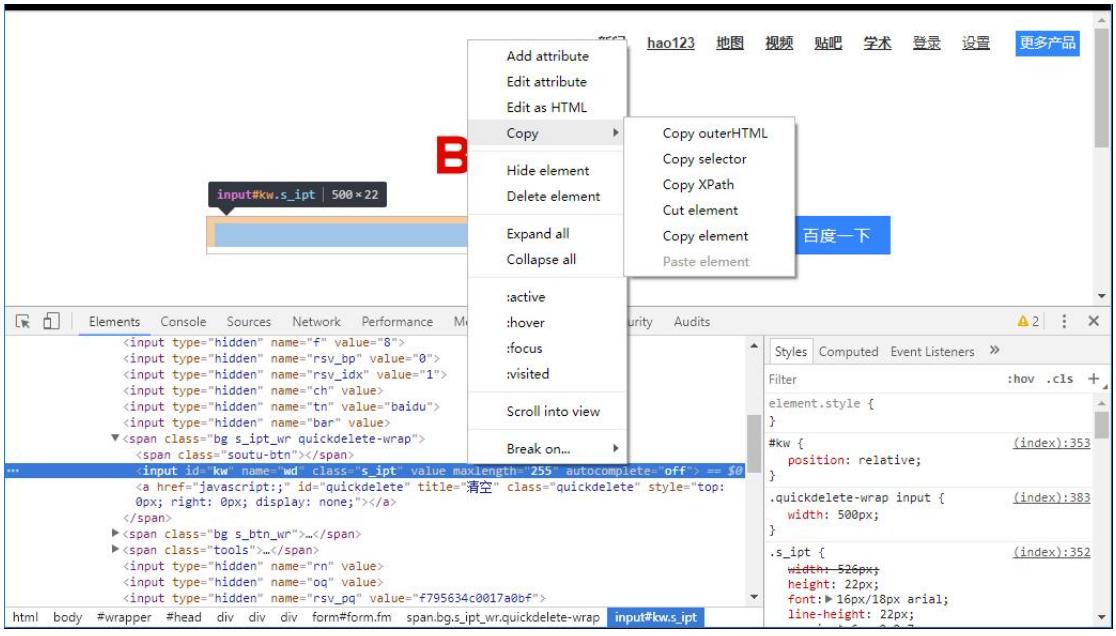
一般来说，我们都会组合进行使用，增加定位的精确性。那我们来试试把上面学的进行组合使用：

```
driver.find_element_by_css_selector("span.bg.s_ipt_wr>input#kw.s_ipt")
driver.find_element_by_css_selector("span[class*='s_btn_wr']>input[type^=sub]#su")
```

标签名可以和属性、class、id 组合查询。能看懂上面的两个例子么？仔细观察其中的组合情况。

7）快速获取 CSS 路径：

我们也可以用开发者工具，选中某个元素，“右键>Copy>Copy selector”拷贝 CSS 路径。



CSS 相关的内容远远不止上面介绍的这些，通过 css selector 方法可以找到页面上的任意一个元素，灵活运用最重要。

8. Xpath 定位：

Xpath 是一种用在 XML 文档中定位元素的语言。HTML 本身可以看作 XML 的一种，因此我们也可以用 Xpath 来定位页面中的元素。同样我们以百度中的 HTML 代码来看。

```
<form id="form" name="f" action="/s" class="fm">
…
```

```
      <span class="bg s_ipt_wr quickdelete-wrap">
            <span class="soutu-btn"></span>
                  <input id="kw" name="wd" class="s_ipt"
                              value="" maxlength="255" autocomplete="off">
      </span>
      <span class="bg s_btn_wr">
                <input type="submit" id="su"
                              value="百度一下" class="bg s_btn">
      </span>
…
</form>
```

1） **绝对定位：**

Xpath 中最直观的定位策略就是绝对路径。我们同样以百度中的输入框和按钮为例。

```
driver.find_element_by_xpath("/html/body/div[2]/div/div/div/div/form/span/input")
driver.find_element_by_xpath("/html/body/div[2]/div/div/div/div/form/span[2]/input")
```

find_element_by_xpath()方法使用 Xpath 来定位元素，Xpath 主要用标签名的层级来定位元素的绝对路径，用/html 标签开始，一级级的往下找，如果一个层级下又多个相同的标签名，那么就按上下顺序确定是第几个（**注意，在 Xpath 中元素的顺序下标是从 1 开始，要注意与之前遇到的其他下标区分**）。例如 div[2]表示当前层级下的第二个 div 标签。

2） **相对定位：**

绝对路径虽然可以查找元素，但是其绝对的书写方式导致如果界面稍微发生点结构变化，哪怕是按钮的位置改变一下，也会导致定位失败，这极大的增加了脚本的维护成本，因此我们做自动化测试时，应当尽可能少的使用 Xpath 的绝对路径。

那么除了绝对路径，Xpath 还提供了一种更为灵活的方式，相对路径定位方法，以“//”开头。

a） **元素属性定位：**

同样以百度输入框和搜索按钮为例：

```
driver.find_element_by_xpath("//input[@id='kw']")
driver.find_element_by_xpath("//input[@id='su']")
```

注意写法，//表示当前页面的某个相对目录，input 表示元素的标签名，[@id='kw']表示这个元素的 id 属性值等于 kw。注意和 css selector 中区别开，Xpath 中属性前面要加@符号，并且这里的***属性值必须完全匹配且必须加引号***。

b） **层级属性结合定位：**

假设现在 input 按钮中没有 id 无法精确定位，那么我们可以找其父级再定位子级。

```
driver.find_element_by_xpath("//form[@id='form']/span/input")
```

```
driver.find_element_by_xpath("//form[@id='form']/span[2]/input")
```

这种方法就是当你发现当前元素无法直接定位时，可以一级级的往上找，直到找到一个可以用来唯一定位的节点，然后写法是从这个唯一节点一层层的往下写。

c） 使用逻辑运算符：

如果元素的属性都无法精确定位到这个元素，我们还可以用逻辑运算符 and 连接多个属性进行定位，以百度输入框为例。

```
driver.find_element_by_xpath("//input[@name='wd' and @class='s_ipt']")
```

d） 使用 contains：

上面讲了，通过属性的方式定位，需要属性值完全匹配，那有没有一种能像 css selector 中的部分匹配的方式呢？有，那就是 contains()方法。以百度搜索按钮为例：

```
driver.find_element_by_xpath("//input[contains(@class,'bg')]")
```

注意写法，contains()用在属性值的位置，contains(attr_name，attr_value)具有两个参数，第一个是属性名，第二个是属性值。属性值可以是部分匹配。这样就表示其 class 属性包含'bg'的 input 元素。

contains()还可以利用文本进行定位，前面讲过文本就是标签对中间的文字符号部分，以百度首页右上角的链接为例：

```
<div id="u1">
  <a href="http://news.baidu.com" name="tj_trnews" class="mnav">新闻</a>
  <a href="http://www.hao123.com" name="tj_trhao123" class="mnav">hao123</a>
  <a href="http://map.baidu.com" name="tj_trmap" class="mnav">地图</a>
  ...
</div>
```

那么上面 3 个<a>标签中的文本分别是“新闻”、“hao123”、“地图”，那如何用文本来定位呢？两种方法：

```
driver.find_element_by_xpath("//a[contains(text(),'新闻')]")
driver.find_element_by_xpath("//a[text()='新闻']")
```

text()方法用于标识 HTML 中的文本。

注意：

- 这种用 contains()定位部分属性或文本的方式效率很低，非常容易出错，因此我们如果有其他选择，尽可能不使用 contains()。

另外，Xpath 也可以用 css selector 小节最后讲到 Copy 来拷贝 Xpath。

xpath 和 css selector 都是非常强大的定位方法，几乎都能唯一定位到页面上的任意元素。但是应用起来也较为复杂，需要多练习各种情况。就这两种查找方式而言，***css selector 更简洁，执行效率更高，因此 selenium 官方推荐使用 css selector。***

9. 定位一组元素：

比如一次性把百度输入框和搜索按钮都找到，写法如下。

```
driver.find_elements_by_css_selector("#form>span>input")
```

注意观察一组元素和单个元素的区别。看出来了么？其实就是单个元素是 find_element 而一组元素是 find_elements。一组元素也支持前面讲的八种方式。

特别注意：

- find_element 返回的是一个 WebElement 对象，而 find_elements 返回的是一个 list,其中的每个元素是一个 WebElement 对象。因此 find_element 可以进行输入、单击等操作，而 find_elements 不能直接操作。

10. 使用 By 类：

Selenium 4.0 以后针对上面讲的 8 种定位方法以及 find_elements，WebDriver 提供了另外一种写法，即统一用 find_element 或 find_elements 调用，这种方法与我们前面讲的八种方式的底层是一摸一样的。笔者后来也更喜欢这种写法,因为看起来更简洁可读性更强；不过 selenium 官方更推荐前面的写法。因此看个人习惯和具体应用场景选择即可。

要使用 By 之前，必须先引入 By 类：

```
from selenium.webdriver.common.by import By
…#打开百度过程省略
driver.find_element(By.ID, "kw")  #找到百度输入框
driver.find_elements(By. CSS_SELECTOR, "#form>span>input")#同时找输入框和按钮
```

find_element()方法只用于定位元素。它需要两个参数，第一个是定位的类型，由 By 提供；第二个参数为定位的具体方式，与我们前面讲八种方式的时候输入的内容一致。

八种方式对照表：

id	By.ID
name	By.NAME
class_name	By.CLASS_NAME
tag_name	By.TAG_NAME
css_selector	By.CSS_SELECTOR
xpath	By.XPATH
link_text	By.LINK_TEXT
partial_link_text	By.PARTIAL_LINK_TEXT

仔细观察我们就会发现，其实就是把小写改成了大写。

相对定位

selenium4 带来了一种新的定位方式-**相对定位器**，添加相对定位器是为了帮助定位可定位元素相邻的元素，可用的相对元素定位器有 5 种

- above() 返回指定元素上方的元素
- below() 返回指定元素下方的元素

- to_left_of() 返回指定元素左侧的元素
- to_right_of() 返回指定元素右侧的元素
- near() 返回指定元素附件的一个元素，要求该元素离指定元素不超过 50px

在 selenium4 中，find_element()接收一个新的参数**with_tag_name("")用于实现相对定位**，而不是使用传统的 8 大定位方式，示例：现有如下格式的表单，使用相对定位的方法如下

```
from selenium.webdriver.support.relative_locator import with_tag_name

# 定位"用户名"输入框下方的"密码"输入框
password_input = driver.find_element(
      with_tag_name("input").below({By.ID: "username"})
)

# 常用相对定位方法：above() / below() / to_left_of() / to_right_of() / near()

# 定位"地图"右边的"hao123"

locator = driver.find_element(By.XPATH, '//div[@id="s-top-left"]/a[3]') # 地图
driver.find_element(with_tag_name('a').to_right_of(locator)).click()    # hao123
```

2. 5 Selenium 元素操作

上面讲了定位方式，定位是我们操作 Web 页面的第一步，定位之后就需要对元素进行单击、输入等操作，接下来介绍 WebDriver 中最常用的几个操作方法。

2.5.1 Webdriver 的常用操作

1. **清除文本：**

   很多输入框，特别是登录的地方，很多 Web 站点都有记住用户名的功能；或者由一个默认值诸如“手机/邮箱/用户名”之类的默认值。对于这种如果直接向输入框输入数据，就会直接接在原来记住的用户名或者默认值后面，如“手机/邮箱/用户名 username”，这样会直接导致登录失败。那么就需要在输入之前先清理掉原来已经存在于输入框中的内容。clear()方法用于清除输入框中的内容：

```
driver.find_element_by_id("kw").clear()
```

2. **模拟按键输入：**

   send_keys()方法用于模拟键盘输入，括号中的参数就是你想输入的内容，比如我们需

要通过百度输入框搜索“刘德华”。

```
driver.find_element_by_id("kw").send_keys("刘德华")
```

3. 单击元素：

click()方法用于模拟鼠标左键单击，比如我们通过上面的方法在百度输入框中输入了“刘德华”，那么我们可以通过 click()点击搜索按钮。

```
driver.find_element_by_id("su").click()
```

当然现在由于百度不需要点击搜索按钮也会执行搜索，因此我们可以试试点击右上角的“新闻”链接，看看新闻去！

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

4. 提交表单：

WebDriver 还提供了一种方法 submit()用于提交表单， submit()操作就很类似我们在输入搜索关键字后，点击了一下回车按钮。这样的好处是，不用再去找一次搜索按钮。

```
driver.find_element_by_id("kw").send_keys("刘德华")
driver.find_element_by_id("kw").submit()
```

submit()也可以用来提交一个按钮，因此其功能很类似 click()。但是 submit()应用范围远不及 click()。我们更常用的是 click()。

5. 获取元素信息：

WebDriver 还提供了一些常用的方法用于获取元素的某些信息，如下：

```
driver.current_url      #获取当前 url
driver.find_element_by_link_text("新闻").text     #获取元素的文本
driver.find_element_by_id("kw").size     #获取输入框的大小
driver.find_element_by_id("kw").get_attribute("class") #获取输入框元素的 class 属性的属性值
driver.title                #获取当前 HTML 的 title
```

- ✧ current_url：用来获取当前浏览器地址栏中的 url，这本来是一个方法，只是转换成了属性来使用，因此要注意这里是没有括号；
- ✧ text：用来获取当前元素的文本，也是方法转属性，因此当作属性来用；
- ✧ size：用于获取当前元素的尺寸，也是方法转属性，因此当作属性来用；
- ✧ get_attribute(attr_name)：用于获取当前元素的某一个属性的属性值，参数就是属性名，如 class、id、name、href、value 等，需要加引号；
- ✧ title：用来获取当前 HTML 的<title>标签的文本。

6. 元素再操作：

前面讲过 find_element 方法获取到的是一个 WebElement 对象，那么这个对象其实就是对应的一个 HTML 元素，HTML 元素中不但包含了当前标签的属性，还完整的包含了子集元素的所有内容，因此我们还可以再次进行 find_element 操作。比如找到一个元素，还可以对这个元素再次定位子集。拿百度右上角的链接举例，先看一下 HTML 代码：

```
<div id="u1">
```

```
<a href="http://news.baidu.com" name="tj_trnews" class="mnav">新闻</a>
<a href="http://www.hao123.com" name="tj_trhao123" class="mnav">hao123</a>
<a href="http://map.baidu.com" name="tj_trmap" class="mnav">地图</a>
...
</div>
```

我们可以先找到 id="u1"的 div 元素。

```
driver.find_element(By.ID, "u1")
```

我们还可以再从找到的 div 元素中查找具体的链接：

```
driver.find_element(By.ID,"u1").find_element(By.PARTIAL_LINK_TEXT, "新 闻
").click()
```

2.5.1Webdriver 多窗口切换

首先我们先理解下什么是窗口的概念：浏览器的 window 概念，拿 Firefox 为例，一个 tab 就是一个 window。那么我们先执行下面的代码：

```
driver.get("http://www.jd.com")
driver.find_element_by_partial_link_text("家用电器").click()
```

现在我们就得到两个 window：

当我们点击某个链接，新建了一个浏览器窗口，此时如果需要在新窗口中进行操作，会报错！因为 WebDriver 中的窗口还停留在之前的窗口中。需要从原来的窗口跳转到新建的窗口才能操作新窗口中的元素。这里依然要用到 switch_to 方法，只是现在需要跳转的时 window 而不是 frame 了。

在我们用 switch_to 之前，我们要先确定窗口，否则我们跳转没有目标啊！但是要怎么去顶窗口呢？这里会涉及到一个句柄的概念，简单理解句柄就是一个内存地址，用来标识浏览器窗口的内存地址的。

首先需要获取到所有窗口的句柄：

```
handles = driver.window_handles  #以 list 形式返回当前浏览器所有 window 的句柄
```

一般来说窗口的句柄都是按顺序的，比如第一个窗口就是 handles[0]，第二个窗口 handles[1]，以此类推。

那么跳转最简单的方法就是，你知道你跳转的窗口是第几个，那么 handles[1]表示第二个窗口：

```
driver.switch_to.window(handles[1])
```

现在就成功的跳转到新打开的浏览器窗口,如果你想回到原来的窗口该怎么做？思考一下。

那对于只有两个窗口频繁切换的情况，有一个较为方便的方法，同时也可以避免跳来跳

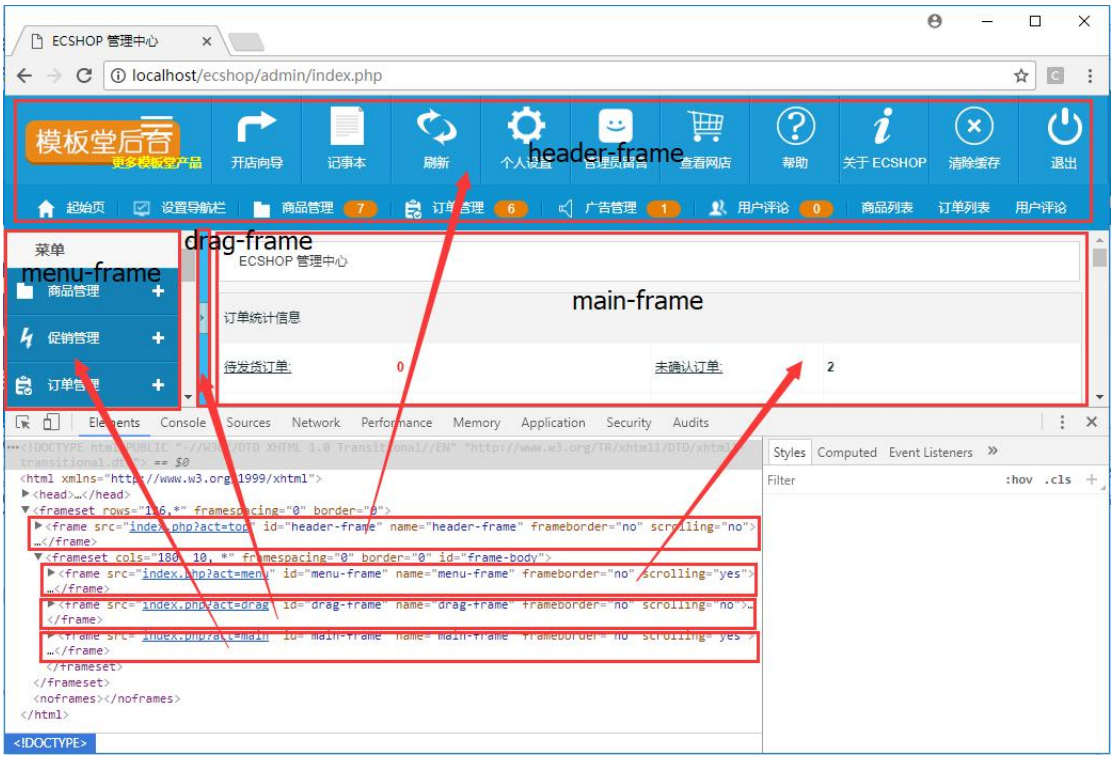
去把自己跳晕了：

```
for handle in driver.window_handles:     #循环遍历当前所有窗口
    if handle != driver.current_window_handle:  #判断是否与当前窗口句柄一致
        driver.switch_to.window(handle)    #不是则跳转
        break
```

- ✧ dirver.current_window_handles 方法用于获取当前窗口句柄，这个循环其实就是判断 handle 如果非当前句柄，就直接跳转过去。
- ✧ switch_to.window()还可以用另一种方式来确定窗口的跳转，那就是窗口的 name 属性 window.name:但是一般浏览器窗口是没有这个属性的，因此需要额外处理一下，后续课程我们将会介绍。

2.5.2 Webdriver 内联框架 frame 切换

在 web 应用中，经常会遇到 frame/iframe 表单嵌套页面的应用，每一个 frame/iframe 就相当于一个独立的页面，这就是我们所说的“内嵌页面”或者叫“内嵌框架”。WebDriver 只能对当前页面上的元素识别和定位，对于 frame/iframe 表单内嵌页面上的元素无法直接定位。例如我们的实战项目 ecshop 的后台管理系统：



仔细观察开发者工具中的 HTML 代码，发现整个页面分为两个<frameset>，其中第一个<frameset>中包含了一个<frame>：

- header-frame，包含页面顶部的各种按钮、链接等；

第二个<frameset>包含了三个<frame>：

- menu-frame，左侧菜单栏，
- drag-frame，中间的收起按钮，
- main-frame，主体页面，包含各模块的操作界面，各类数据的处理、展示等。

注意：

- ***<frameset>标签不统计。<frameset>只是<frame>标签的集合，并不是内嵌页面。***

至于为何要去这样设计，主要由开发的页面框架决定，很多后台管理系统都是这种各种框架拼接的页面。因此我们不管为什么会有这样的设计，一旦遇到这种框架，就需要知道如何去处理。

我们可以通过在页面上“点击右键>查看网页源代码”或“CTRL+U”，我们看到的网页代码结果如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>ECSHOP 管理中心</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="Text/Javascript" language="JavaScript">
<!--

if (window.top != window)
{
  window.top.location.href = document.location.href;
}

//-->
</script>

<frameset rows="126,*" framespacing="0" border="0">
  <frame src="index.php?act=top" id="header-frame" name="header-frame" frameborder="no" scrolling="no">
  <frameset cols="180, 10, *" framespacing="0" border="0" id="frame-body">
    <frame src="index.php?act=menu" id="menu-frame" name="menu-frame" frameborder="no" scrolling="yes">
    <frame src="index.php?act=drag" id="drag-frame" name="drag-frame" frameborder="no" scrolling="no">
    <frame src="index.php?act=main" id="main-frame" name="main-frame" frameborder="no" scrolling="yes">
  </frameset>
</frameset><noframes></noframes>
  <frameset rows="0, 0" framespacing="0" border="0">
  <frame src="http://ecshop.ecmoban.com/record.php?mod=login&url=http%3A%2F%2Flocalhost%2Fecshop%2F" id="hidd-frame" name="hidd-frame" frameborder="no" scrolling="no">
  </frameset>
</head>
<body>
</body>
</html>
```

我们看到当前页面，只有两个 frameset 元素和 4 个 frame 元素。其他肉眼所见的按钮、链接、图片等等，都不在当前页面。而我们说过 WebDriver 只能定位当前页面，因此我们必须要特殊方式来处理这种情况。switch_to.frame()方法就是 WebDriver 用来处理这种特殊情况的方法，该方法的参数可以有 4 种形式：

- Index 起始为 0
- Id
- Name
- WebElement 对象

我们来看实际用法，以 ecshop 后台菜单为例：

```
driver.switch_to.frame(1)  #用 frame 的 index 定位，起始为 0
driver.switch_to.frame("menu-frame") #用 frame 的 id 定位
driver.switch_to.frame("menu-frame") #用 frame 的 name 定位
driver.switch_to.frame(driver.find_element_by_id("menu-frame"))      #    用
WebElement 对象来定位
```

现在我们就进入菜单栏所在的 frame：menu-frame，但是现在又出现一个新的问题，当我们在菜单栏所在的 frame 中，想要操作右侧页面时就会出现无法定位的问题，因为这是另一个 frame:main-frame。要如何处理这种情况呢？有点麻烦：）

首先，我们要从当前的 frame 跳出来，跳出当前 frame 有两种方式：

```
driver.switch_to.parent_frame()   #跳到上一层
driver.switch_to.default_content()  #跳回主文档，也就是最外层
```

其次，我们要进入 main-frame：

```
driver. switch_to.frame("main-frame") #进入右侧的 main-frame
```

现在，我们可以操作右侧的页面了。

听起来是不是有点麻烦，还有更麻烦的。如果存在 frame 有层次结构，看下面的例子。这是笔者假设的结构，用来说明多层次结构的情况下该如何处理。

```
|<frame src="…" id="main-frame" name="main-frame">
|     …
|——<frame src="…" id="header-frame" name="header-frame">
|          …
|————————<frame src="…" id="drag-frame" name="drag-frame">
|                 …
|——<frame src="…" id="menu-frame" name="menu-frame">
|          …
```

我们看到 main-frame 在顶层，而 header-frame 和 menu-frame 是它的子级，drag-frame 又是 header-frame 的子级。

1. **逐层进入：**

假设我们进入当前页面，需要先操作 drag-frame：

```
driver.switch_to.frame("main-frame")    #先进入 main-frame 这一层
driver.switch_to.frame("header-frame")  #再进入 header-frame
driver.switch_to.frame("drag-frame")    #最后才能进入 drag-frame
```

2. **跳转到父级的兄弟：**

假设我们现在需要从 drag-frame 出来，进入 menu-frame：

```
driver.switch_to.parent_frame("")       #跳到 header-frame
driver.switch_to.parent_frame("")       #再跳到 main-frame
driver.switch_to.frame("menu-frame")    #才能进入 menu-frame
```

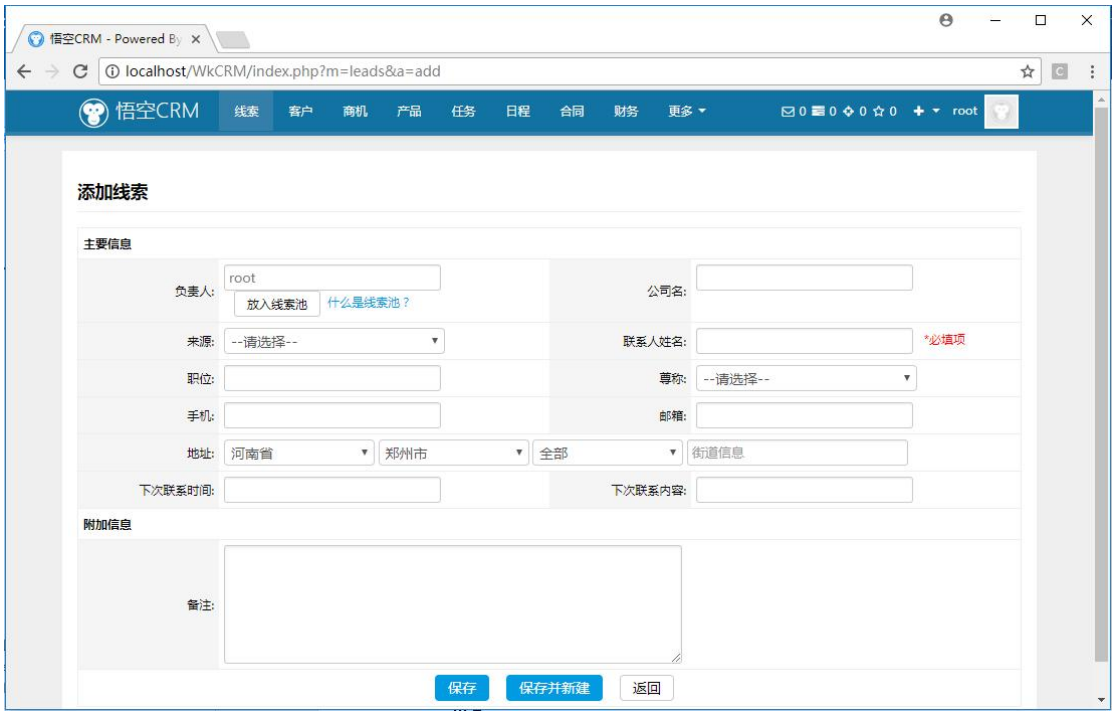
3. **跳出所有的 frame：**

上面的方式很容易把人绕晕，那我们可以采用先直接跳出所有的 frame，再重新进入：

```
driver.switch_to.default_content()  #跳出所有的 frame
driver.switch_to.frame("main-frame")    #先进入 main-frame 这一层
driver.switch_to.frame("menu-frame")    #进入 menu-frame
```

2.5.3 Selenium Select 类

WebDriver 提供了一个专门的类 Select 类来处理 web 页面中的下拉菜单（select 元素）。打开我们的实践项目“悟空 CRM”，登录进入“线索>新建线索”，如下图：



我们来看一下这个页面，下拉框有哪些？来源、尊称、地址都是下拉菜单。通过开发者工具，我们可以看到“来源”下拉框的结构：

```
<select id="source" name="source">
    <option value="">--请选择--</option>
    <option value="网络营销">网络营销</option>
    <option value="公开媒体">公开媒体</option>
    <option value="合作伙伴">合作伙伴</option>
    ...
</select>
```

从上面可以看到，下拉菜单是有两种标签的，<select>用来定义整个下拉菜单元素，<option>用来定义所有的下拉选项，有多少选项就有多少<option>标签。<option>有一个属性 value，其次还有文本，还有一个看不到的 index（从 0 开始）。

对于下拉菜单，我们首先要定位到<select>标签，然后通过对<option>标签的操作对下拉菜单进行赋值（也就是选中某一个选项）。这种操作与我们之前讲的操作普通元素的方式完全不同。因此为了应付下来菜单这种特殊的元素，WebDriver 专门提供了 Select 类。

Select 赋值的三种方法：

```
Select(element).select_by_visible_text("网络营销") #通过可见的文本赋值
Select(element).select_by_value("网络营销")    #通过 value 赋值
Select(element).select_by_index(0)   #通过索引赋值
```

由于 CRM 中的下拉菜单 value 和文本都一样，因此看不出区别，但是实际上是两个不同的方式。

用法：

```
from selenium.webdriver.support.select import Select    #引入 Select 类
```

```
...
locator = driver.find_element_by_id("source")    #定位到“来源”下拉菜单
select = Select(locator)
select.select_by_index(1)    #为“来源”赋值“网络营销”
```

下面我们对用法进行分解说明：

- ✧ select = Select(locator): locator 变量存储定位到的下拉菜单元素对象；实例化 Select 类，以定位后的元素对象为参数传入，这里传入的是 locator 变量；
- ✧ select_by_index(1)：调用 select_by_index()方法，通过 index 的方式赋值，除了 index 方式，还有上面讲的文本和 value 属性赋值。

2.5.4 设置等待

在自动化测试脚本的运行过程中，WebDriver 操作浏览器的时候，对于元素的定位是有一定的超时时间，大致应该在 1-3 秒的样子，如果这个时间内仍然定位不到元素，就会抛出异常，中止脚本执行。很多时候由于网络原因、脚本运行机器本身的原因，都会页面元素加载缓慢，或者造成浏览器卡顿，会导致本来定位准确的脚本出现偶然定位失败的情况。一旦出现这种情况，就会导致测试失败。因此，我们在设计脚本的时候，就必须考虑到如何增加脚本的健壮性。为此，本节，我们就来讨论，如何通过在脚本中设置等待的方式来避免由于网络延迟或浏览器卡顿导致的偶然失败。常用的等待方式有三种：

1. 强制等待(休眠方法)：

强制等待是利用 Python 语言自带的 time 库中的 sleep()方法：

```
from time import sleep
sleep(10)
```

sleep()顾名思义就是睡觉的意思，就是脚本一旦执行到这条语句 sleep(10)就睡 10 秒，再执行后面的语句。因此它是一个强制的方式，使整个脚本暂停。由于这种方式会导致脚本运行时间过长（比如你一个页面上等待 10 秒，一个用例中有 10 个操作用到了这个页面，整个脚本有 10 个用例，这种算下来 sleep 的时间就是成指数增长），因此除非万不得已就尽可能少用 sleep()。

2. 隐式等待：

强制等待显然并不是最好的等待方式，这里介绍一种比强制等待更智能的等待方式：隐式等待。“隐式等待”顾名思义，就是我们在脚本中一般看不到等待语句，但是它会在每个页面加载的时候自动等待，***隐式等待只需要声明一次，一般在打开浏览器后进行声明。声明之后对整个 driver 的生命周期都有效，后面不用重复声明***。：

```
...
driver = webdriver.Chrome()
driver.implicitly_wait(30)
```

implicitly_wait()方法：用来等待页面加载完成（直观的就是浏览器 Tab 页上的小圈

圈转完）。implicitly_wait(30)，超时时间 30s，30 秒内一旦加载完毕，就执行下一条语句；如果 30 秒页面都没有加载完，就超时抛出异常。

但是隐式等待依然存在一个问题，现在网站为了让页面尽快呈现在客户面前，会把所有的 Javascript 放在页面的末尾加载，这样就算 JS 加载较慢也不会影响用户使用。但是，这种时候使用 implicitly_wait()就会有问题，因为 implicitly_wait()方法会等待页面完全加载，才会执行下一句，JS 加载太慢就会一直等待，而这时候其实页面元素已经可以操作了，这样也直接影响了脚本的执行效率，甚至有的加载 JS 过慢的网站，会出现超时。

另外，对于由内嵌框架，内嵌的页面是一个外部页面的时候，有可能还没加载完成，这时候去 switch_to 这个 frame 的时候就可能会抛异常。

3. 显式等待：

隐式等待 implicitly_wait()用起来很方便，整个 WebDriver 过程中只需要声明一次。但是由于 Javascript 加载的原因，因此也会有不适用的时候。遇到这种隐式等待不适用的情况，webDriver 提供了一种更加智能的等待方式：显式等待。显式等待与隐式等待相对，显式等待必须在每个需要等待的元素前面进行声明。例子如下：

```
#引入 WebDriverWait
from selenium.webdriver.support.ui import WebDriverWait
#引入 expected_conditions 类，并重命名为 EC
from selenium.webdriver.support import expected_conditions as EC
…
#设置等待
wait = WebDriverWait(driver, 10,0.5)
wait.until(EC.presence_of_element_located((By.ID, “kw”)))
```

上面的例子的含义为:根据设定的时间间隔检查当前页面 id 为“kw”的元素是否存在，元素存在才进行下一步，如果超过设置时间检测不到则抛异常。

说明：

显式等待需要用到两个类：

WebDriverWait 和 expected_conditions 两个类。

先说 WebDriverWait()：

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

- dirver：浏览器驱动
- Timeout：最长超时时间，默认以秒为单位
- poll_frequency：检测的间隔步长，默认为 0.5s
- ignored_exceptions 超时后的抛出的异常信息，默认抛 NoSuchElementException 异常。
- WebDriverWait()的 until()和 until_not()方法：
- 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
- 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

- until()或 until_not()方法只接收函数或方法作为参数，这里就是需要用到 expected_conditions 类中的各种方法作为参数。

expected_conditions 类：

用于设置一些预期条件，比如元素是否加载出来、元素是否可见等等。

本例中，通过调用 expected_conditions 类中的预期条件判断方法来作为 until()的参数。

Expected_conditions 类提供的预期条件判断方法：

title_is	判断当前页面的 title 是否精确等于预期
title_contains	判断当前页面的 title 是否包含预期字符串
presence_of_element_located	判断某个元素是否被加到了 DOM 树里，并不代表该元素一定可见
visibility_of_element_located	判断某个元素是否可见。可见代表元素非隐藏，并且元素的宽和高都不等于 0
visibility_of	跟上面的方法做一样的事情，只是上面的方法要传入 locator，这个方法直接传定位到的 element 就好了
presence_of_all_elements_located	判断是否至少有 1 个元素存在于 DOM 树中。举个例子，如果页面上有 n 个元素的 class 都是'column-md-3'，那么只要有 1 个元素存在，这个方法就返回 True
text_to_be_present_in_element	判断某个元素中的 text 是否包含了预期的字符串
text_to_be_present_in_element_value	判断某个元素中的 value 属性是否包含了预期的字符串
frame_to_be_available_and_switch_to_it	判断该 frame 是否可以 switch 进去，如果可以的话，返回 True 并且 switch 进去，否则返回 False
invisibility_of_element_located	判断某个元素中是否不存在于 DOM 树或不可见
element_to_be_clickable	判断某个元素中是否可见并且是 enable 的，这样的话才叫 clickable
staleness_of	等某个元素从 DOM 树中移除，注意，这个方法也是返回 True 或 False
element_to_be_selected	判断某个元素是否被选中了,一般用在下拉列表
element_selection_state_to_be	判断某个元素的选中状态是否符合预期
element_located_selection_state_to_be	跟上面的方法作用一样，只是上面的方法传入定位到的 element，而这个方法传入 locator
alert_is_present	判断页面上是否存在 alert

小提示：

显式等待是一种智能程度较高的等待方式，可以有效的增强脚本的健壮性。但是由于使用起

来稍微麻烦一点，因此可以在有页面元素加载这种过程的时候去使用，比如：

- 当打开一个新页面，执行第一个元素操作的时候；
- 当某一步操作会引发页面的加载，并且加载的内容包含了下一步需要操作的元素。

一句话，就是当某个元素有加载过程的时候，就需要加上显式等待。

2.5.5 警告框处理

警告框，经常我们会遇到比如账号密码输错，弹出一个提示框，告知你账号或密码错误；或者在操作页面的过程中，有部分重要或危险的操作（删除）会弹出一个框让你确认。这些都是由 Javascript 生成的警告框。基于警告框的特性，当有警告框的时候，你是无法操作页面元素的。因此警告框必须被处理。

警告框的形式有三种：

1. alert:只是一个提示信息，只有一个确定按钮，你点确定或者点“x”关闭是一样的效果；

2. confirm：确认框，点击确定和取消按钮会得到不同的响应，点“x”关闭与取消效果一样；

3. prompt：带输入框的确认框，输入的数据会返回页面做处理，点确定和取消会得到不同的相应，点“x”关闭与取消效果一样。

警告框一旦弹出，就脱离于当前页面了，也就是说警告框并不是当前页面的元素，因此普通定位方法是无效的。这里就又要用到 switch_to 方法了，这次是 switch_to.alert。跳转到输入框后，还需要其他的操作：

- text：返回 alert/confirm/prompt 中的文本信息；
- accept()：接受现有的警告框；
- dismiss()：取消现有的警告框；
- send_keys(value)：发送文本至警告框，主要针对某些需要输入的警告框。

实际例子不太好找，那么我们依然自己写一个：

```
<html>
        <head>
                <title>认识 alert/confirm/prompt</title>
        </head>
        <body>
                <input id = "alert" value = "alert" type = "button"
                        onclick = "alert('这是一个 alert 警告框');"/>
                <input id = "confirm" value = "confirm" type = "button"
                        onclick = "if(confirm('这是一个 confirm 确认框'))
                        {document.write('你选择了确认！')}
                        else{document.write('你选择了取消！')}"/>
                <input id = "prompt" value = "prompt" type = "button"
                        onclick = "var str = prompt('请输入:',
                        '这是可输入的 prompt');document.write(str) "/>
        </body>
</html>
```

保存为 alert.html，然后在浏览器中打开试试。

然后试试用 Webdriver 操作一下：

```
driver.get("file:///E:/alert.html") #地址直接拷贝浏览器的地址栏
driver.find_element_by_id("alert").click()      #点击出现一个警告框
driver.switch_to.alert.text    #获取 alert 上的文本
driver.switch_to.alert.accept()      #确定警告框
```

对于 prompt 有点例外：

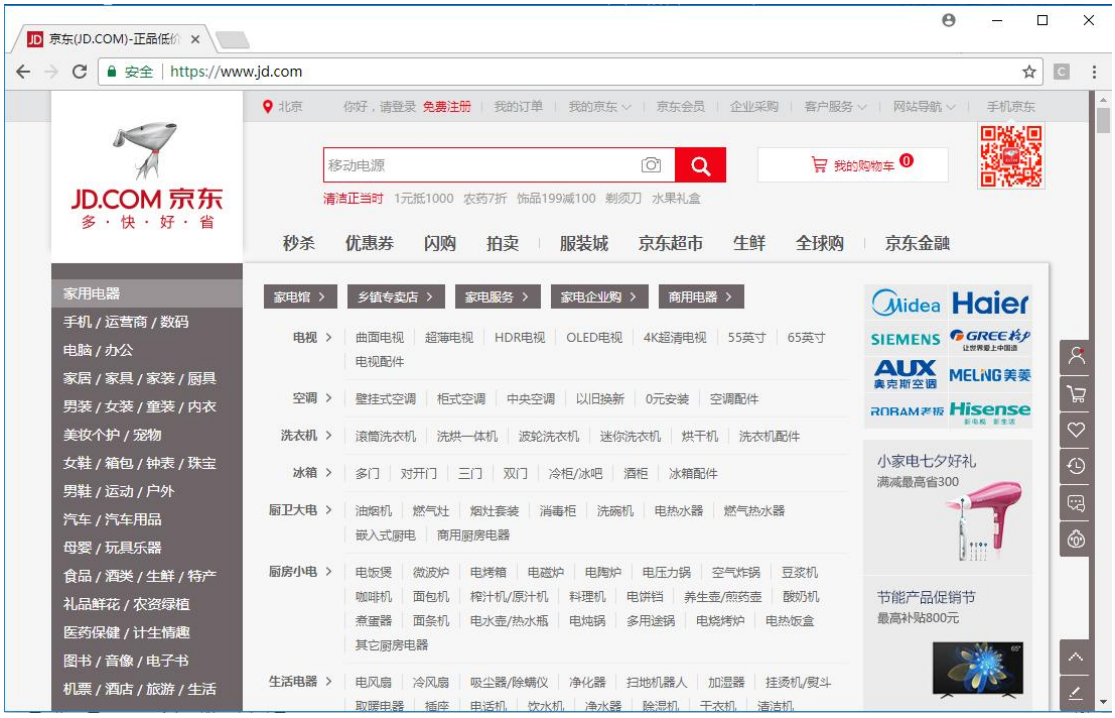
```
driver.get("file:///E:/alert.html") #地址直接拷贝浏览器的地址栏
driver.find_element_by_id("prompt").click()      #点击出现一个 prompt 警告框
driver.switch_to.alert.text    #这里只能获取到“请输入：”
driver.switch_to.alert.send_keys("这是一个例子")  #注意这里 send_keys 在警告框上是看不出变化的，需要 accept 以后才能看到页面变化
driver.switch_to.alert.accept()      #确定警告框
```

prompt 的输入内容，可以通过.send_keys("")方法传入要输入的内容，但是弹出框中的内容不会变化。

dismiss()方法请自行尝试。

2.5.6 鼠标事件

前面我们已经了解到，可以通过 click()模拟鼠标单击，但是现在 Web 页面中经常会有更丰富的鼠标交互方式，比如京东首页左侧的商品分类，必须要鼠标悬停在分类上，才能展示二级菜单：



以京东为例，我们来学习如何使用 WebDriver 中的鼠标事件。

首先 WebDriver 中的各种事件，由 ActionChains 类提供，此例我们需要用到鼠标悬停方法：

```
from selenium.webdriver.common.action_chains import ActionChains    # 引入 ActionChains 类
…打开京东过程省略
locator = driver.find_element_by_link_text("家用电器")    #定位到元素
action = ActionChains(driver)  #实例化 ActionChains 类
action.move_to_element(locator).perform()  #实现鼠标悬停
```

经过上面的代码，我们就可以看到鼠标悬停在家用电器的位置，与我们手动把鼠标放在上面的效果一致。下面我们来对实现鼠标悬停的方法进行分解说明：

- ✧ action = ActionChains(driver)： 实例化 ActionChains()类，将浏览器驱动作为参数传；

- ✧ move_to_element()：此方法是 ActionChains 类中的一个方法，实现鼠标悬停，需要将定位器作为参数传入，也就是要先定位到元素，你才能实现鼠标悬停，例子中我们是先定位一个变量 locator 用于储存定位到的元素对象,然后将该变量作为参数传入 move_to_element()方法；
- ✧ perform()：执行所有 ActionChains 中的存储行为，可以理解为对前面操作的提交动作。

除了例子中的鼠标悬停，ActionChains 类还提供了其他的一些鼠标事件，下表中是一些常用的鼠标事件：

context_click(locator)	鼠标右键单击
double_click(locator)	左键双击
drag_and_drop(source_locator, target _locator)	鼠标拖动，从 source_locator 拖动到 target _locator
move_to_element(locator)	鼠标悬停
click_and_hold(locator)	在元素上按下鼠标左键

**** 不要忘记 perform()***

**** 如果有鼠标按下操作，需要 release()方法释放***

2.5.7 键盘事件

Keys 类中提供了键盘上几乎所有按键的方法。前面我们将简单元素操作的时候，讲到用 send_keys 模拟键盘输入，除此之外，我们还可以模拟键盘上的特殊按键，比如 F1~F12，以及其他组合键，如：Ctrl+A 等。这次我们以京东首页的搜索框为例

```
from selenium.webdriver.common.keys import Keys  # 引入 keys 类
…# 打开京东过程省略
driver.find_element_by_id("key").send_keys("蒙牛") # 定位搜索框并输入条件
driver.find_element_by_id("key").send_keys(Keys.ENTER) # 发送键盘回车事件
```

其他常用键盘事件：

Keys 值	**含义**
send_keys(Keys.BACK_SPACE)	删除键（回退）BackSpace
send_keys(Keys.SPACE)	空格键
send_keys(Keys.TAB)	Tab 键
send_keys(Keys.ESCAPE)	ESC 键
send_keys(Keys.ENTER)	回车键
send_keys(Keys.CONTROL, "a")	Ctrl+A 全选
send_keys(Keys.CONTROL, "c")	Ctrl+C 复制
send_keys(Keys.CONTROL, "x")	Ctrl+X 剪贴
send_keys(Keys.CONTROL, "v")	Ctrl+V 粘贴

send_keys(Keys.F1)	F1~F12
…	
send_keys(Keys.F12)	

2.5.8 调用 Javascript

WebDriver 中提供了很多定位元素、操作元素的方法，但是有些地方还是不太方便。因此 WebDriver 增加了一种调用“神器”的方法，这个“神器”就是 Javascript。我们先来看看 JS 的 定义：***JavaScript 是一种属于网络的脚本语言，已经被广泛用于 Web 应用开发，常用来为网页添加各式各样的动态功能，为用户提供更流畅美观的浏览效果。***既然 JS 能为网页添加各种各样的动态功能，那么我们也能用 JS 来帮助我们处理页面元素。这可以作为 WebDriver 辅助的一个非常有用的扩展。本教材不提供 JS 的教程，有兴趣的同学可以到 W3CSchool 中搜索相关教程。

这里我们介绍几种 JS 的用法：

1. **实现窗口滚动：**

   我们用百度搜索“刘德华”，我们会发现，搜索的结果很长，一屏看不完。当我们想拖到最底部要怎么办呢？我们手工时怎么做的？按住滚动条拖动、或者用鼠标滚轮对吧！但是 WebDriver 中并没有提供窗口滚动的方法。因此我们想将页面拖动到最底部，就需要借助 JS 的力量了。

   首先我们获取以下当前页面的长和宽：

```
xy = driver.find_element_by_tag_name("html").size
```

   得到当前页面的 size 为 xy = {'height': 2403, 'width': 1017}，这是一个长度和宽度的像素长度，2403 就表示有 2403 个像素点这么长。这里'height'长度就是我们需要的值。要实现窗口滚动，我们需要用到 JS 中的 window.scroll(x,y)方法，x 表示宽度，y 表示长度，与我们刚才通过 size 方法获取到的顺序有点不一样。因为 JS 语句稍微有点长，因此我们先进行一个 JS 语句的组装。以下两种方式等同：

```
js = "window.scroll(0,"+ str(xy["height"])+")" #通过拼接的方式
js = "window.scroll(0,%s)"%(xy["height"])   #通过格式化的方式
```

   组装好后，就需要调用 except_script()方法执行 JS 语句：

```
driver.execute_script(js)
```

   好了，看看效果呢？有没有滚动到最底部？

2. **执行元素定位和操作：**

   除了我们上面说的 WebDriver 中没有提供的方法，我们还可以完全模拟前面学到的 WebDriver 中的元素定位和操作，以百度输入框为例：

```
…#省略页面打开过程
driver.find_element_by_id("kw").send_keys("刘德华")
```

   我们现在来完全用 JS 语句实现：

```
js = "document.getElementById('kw').value='刘德华'"
driver.execute_script(js)
```

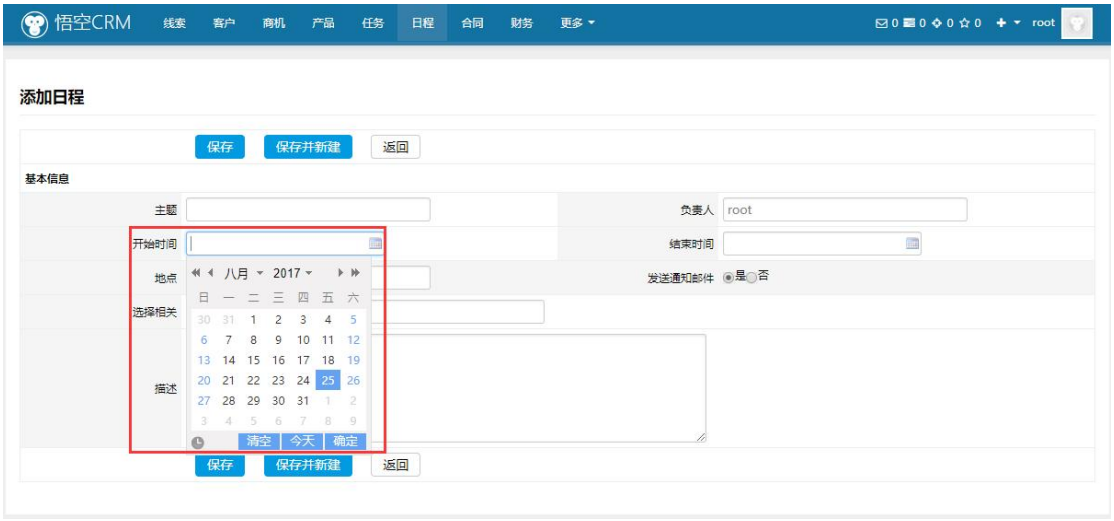
试试看，是不是结果一样？

限于篇幅，上面只介绍了 2 种简单的应用。如果能将 Javascript 学好，将是 web 自动化测试中的一大助力。

2.5.9 其他常见控件

1. 日期控件处理：

关于日期控件，我们先来看一个例子：



一般来说,日期控件都是一个<input>标签，点击后弹出一个日期选择框，在这里甚至是一个 iframe。看起来这么复杂，那我们想实现自动选择一个日期那岂不是很麻烦。其实想多了！既然这是一个<input>，那我们是不是可以直接:send_keys()

```
…#省略页面打开过程
driver.find_element_by_id("start_date").send_keys("2017-08-25")
```

当然，有时候开发为了限制不正确的日期输入，会直接将<input>设置为只读，没办法输入；这种时候怎么办呢？这就要用到上节学到的调用 JS 了。一般输入框只读，是因为有一个 readonly 属性，那么我们可以调用 JS 的 removeAttribute()删除属性的方法干掉元素中的 readonly，这样我们就可以 send_keys()。

```
js = " document.getElementById('id').removeAttribute('readonly')"
driver.execute_script(js)
```

当然，由于这个例子不好找，我们可以先来试一下自己加上一个 readonly，再删除来练习。

```
js = "document.getElementById('start_date').setAttribute('readonly','True')"
driver.execute_script(js)
```

试一试，是不是就不能输入了？那我们现在就可以删除'readonly'后试试：

```
js = " document.getElementById('start_date').removeAttribute('readonly')"
```

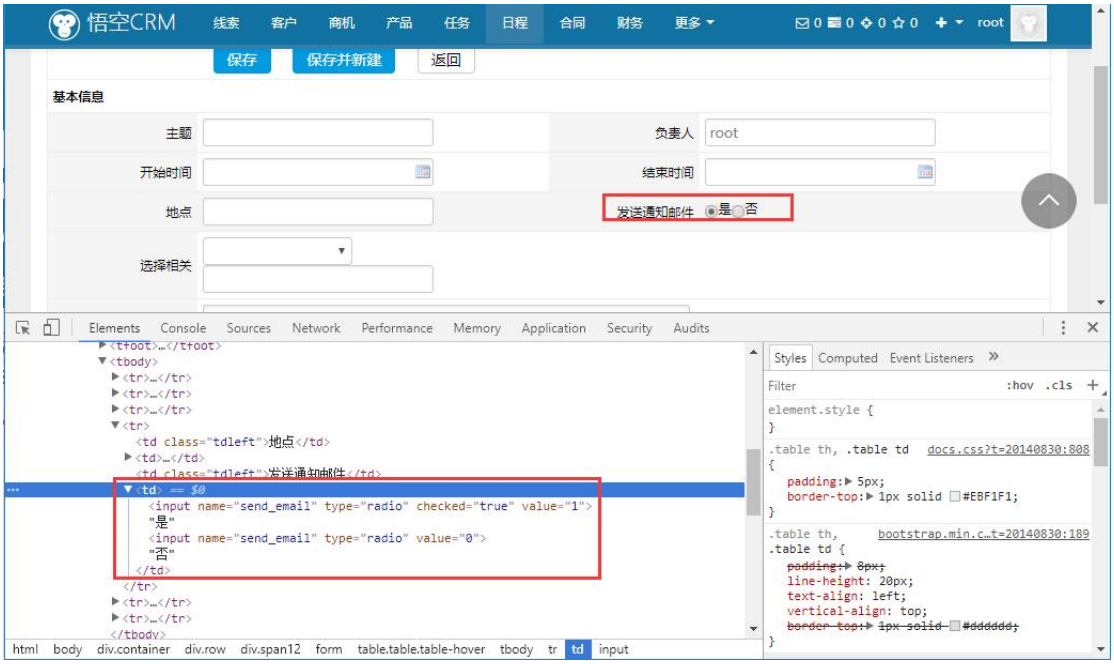
```
driver.execute_script(js)
```

小提示：

- removeAttribute('readonly')，JS 语言中删除属性的方法，参数为属性名，需要引号；
- setAttribute('readonly','True')，JS 语言中添加属性的方法，参数为(属性名，属性值)，都需要引号。

2. 单选框/复选框：

还是我们的 CRM，还是新建日程，看一个单选框的例子：

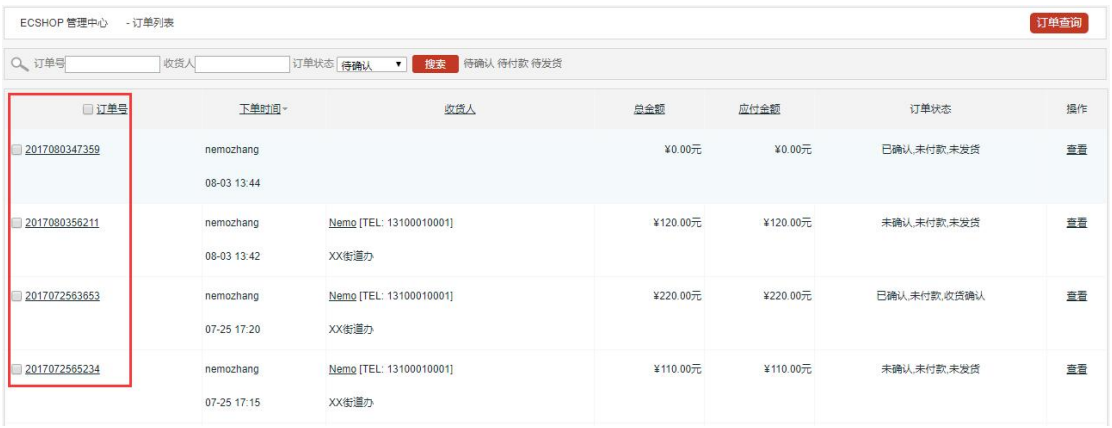


对于单选框来说，其实就是一组 type 属性值为“radio”的<input>标签，这是一组互斥的元素，同时只能有一个被选中。

```
driver.find_elements_by_name("send_email")[1].click()
```

通过 find_elements()定位到这一组 input 元素，那么你想点击哪一个，用下标去点击即可。注意，这个下标不是 xpath 的下标，而是 find_elements()返回的 list 的下标，list 的下标从 0 开始。

对于复选框，主要出现在列表页面，如 ecshop 中的订单列表，



我们通过开发者工具查看 HTML 代码可以看到，复选框是一组拥有 type 属性值为“checkbox”的 input 元素。顶部的订单号复选框，点击可以选中当前列表中的所有“checkbox”。我们要操作这一组复选框，可以单独点击顶部的订单号前的复选框；也可以通过 find_elements 找到所有的“checkbox”进行循环点击。

3. 表格：

selenium 没有提供 table 的处理方法，只有根据需要自己编写脚本。一个简单的例子，新建一个 table.html 文件，输入以下内容：

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>Table</title>
    </head>
    <body>
        <table border="1" id="myTable">
            <tr>
                <th>表头第一格</th>
                <th>表头第二格</th>
                <th>表头第三格</th>
            </tr>
            <tr>
                <td>第一行第一列</td>
                <td>第一行第二列</td>
                <td>第一行第三列</td>
            </tr>
            <tr>
                <td>第二行第一列</td>
                <td>第二行第二列</td>
                <td>第二行第三列</td>
            </tr>
        </table>
    </body>
</html>
```

用浏览器打开，一个简单的 table 就有了：

表头第一格	表头第二格	表头第三格
第一行第一列	第一行第二列	第一行第三列
第二行第一列	第二行第二列	第二行第三列

下面我们来实现以下通过输入行列来取出单元格的文本：

```
driver.get("file:///E:/table.html") #地址直接拷贝浏览器的地址栏
def row_cell_get_table_text(table_loc, row, cell):
    cell = str(cell)
    row = str(row + 1)        #排除表头
    locator = table_loc + "/tbody/tr[" + row + "]/td[" + cell +"]"  #拼接定位语句
    text = driver.find_element_by_xpath(locator).text
    return text
text = row_cell_get_table_text("//table[@id='myTable']",1,2)
```

打印下 text 看看，是不是显示“第一行第二列”。

这是一个利用 xpath 中<tr>和<td>下标参数化实现的通过行列值取对应的单元格文本的例子。其他操作 Table 的方式请大家下来思考。

2.5.10 截图

在测试过程中,有时候对于某些重要的操作完成后展现的页面,通过脚本判断不太放心;或者某些地方出现错误，错误提示不明显，那么可以把当前页面截图保存下来，在分析测试结果的时候，可以看下这些重要地方的截图。下面两种方式得到的结果都是一样的。任选一种即可：

```
driver.save_screenshot(".\screenshot.png")  #返回一个图片文件
driver.get_screenshot_as_file(".\screenshot.png")  #返回一个图片文件
```

上面两个方法的参数都是文件名,例子中是个相对路径,图片保存在和脚本一样的地方。也可以写绝对路径。

```
driver.save_screenshot("E:\\screenshot.png")  #返回一个图片文件
driver.get_screenshot_as_file("E:\\screenshot.png")  #返回一个图片文件
```

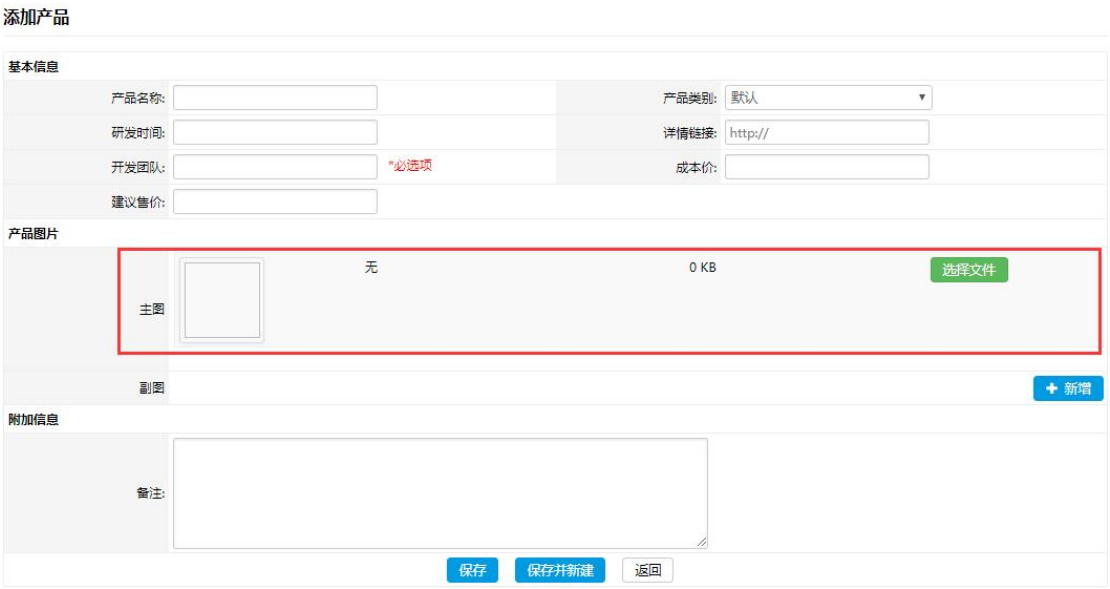
注意：

- Firefox 和 Chrome 浏览器截图，只能截取当前窗口所见，并非整个网页的截图。要想截取非当前窗口的图片，目前有两种方式：

1. 浏览器换成 PhantomJS，这种浏览器在运行时看不到界面。用 PhantomJS 跑脚本，会截取整个网页，而不仅仅是当前可见；
2. 通过 2.2.11 中介绍的窗口滚动，滚动到指定的位置，再截取，就是自己想要的。

2.5.12 上传文件

文件的上传上传为 input 元素，如下图（悟空 CRM>产品>添加产品）：




```
▼<div class="btn btn-success fileinput-button">
  <span>选择文件</span>
  <input type="file" name="main_pic[]" id="main_pic">
</div>
```

仔细观察一下，这里的上传文件实际上就是一个 input 元素，type="file"，这种上传文件的方式，可以直接通过 send_keys()的方式：

```
driver.find_element(By.ID, "id").send_keys("E:\\1.jpg")
```

注意：

- 通过 send_keys()的方式发送文件路径之后，在界面上是看不到缩略图和文件路径信息的，但是在保存后是已经上传了文件。这里需要注意！

在万恶的旧时代，HTML5 还没有出现之前，原生的 file input 表单元素只能一次上传一张图片。无法满足一次上传多图的交互需求，所以，很多场景，就被 swfupload.js 给取代了，有点逐渐淡出人们视野的感觉。

然而，技术发展，日新月异，三十年河东，三十年河西。随着原生 HTML5 表单对多图(multiple 属性)、上传前预览，二进制上传等支持越来越广泛，原生的 file input 表单元素又迎来了新的升级，因此 input file 的方式应用越来越广泛。因此我们以后自动化测试更多的用 send_keys()的方式就行了。

2.5.13 验证码

这是一张网易邮箱的验证码图片，验证码经常出现在我们登录的时候，下面是一段截取百度百科对验证码的介绍：

验证码（CAPTCHA）是“Completely Automated Public Turing test to tell Computers and Humans Apart”（全自动区分计算机和人类的图灵测试）的缩写，是一种区分用户是计算机还是人的公共全自动程序。可以防止：恶意破解密码、刷票、论坛灌水，有效防止某个黑客对某一个特定注册用户用特定程序暴力破解方式进行不断的登陆尝试，实际上用验证码是现在很多网站通行的方式，我们利用比较简易的方式实现了这个功能。这个问题可以由计算机生成并评判，但是必须只有人类才能解答。由于计算机无法解答 CAPTCHA 的问题，所以回答出问题的用户就可以被认为是人类。

目前验证码的技术发展越来越复杂，人辨认起来都非常复杂，何况机器，而我们的自动化测试遇到验证码的时候，是无法自动识别的。在自动化测试的过程中，如果出现验证码要如何解决？大致总结有如下四种方式：

1) **去掉验证码**

去掉验证码不是说系统不用验证码，这肯定是不符合安全要求的，这里说的去掉，是指与开发团队商量，在测试环境中去掉验证码。比如加一个配置，测试环境就关闭验证码，正式环境开启验证码。

2) **设置万能验证码**

依然需要和开发团队商量，设置一个万能的验证码，比如类似“999999”这种，相当于一个后门。每次登录只要用这个验证码就能绕过判断。

3) **验证码识别技术**

如果不能与开发团队取得一致，并且验证码较为简单，可以利用图片识别技术。但是目前识别技术只能识别最简单的数字加字符的验证码，并且部分内容准确率还不能达到 100%，何况现在的验证码不止有数字字符，还各种扭曲、干扰，甚至还有汉字、图片（12306），人辨认都非常困难，何况是图片识别技术。因此这种方式并不适用。

4) **操作 cookie**

最后一种方式就是通过 add_cookie()的方式，我们知道 cookie 指某些网站为

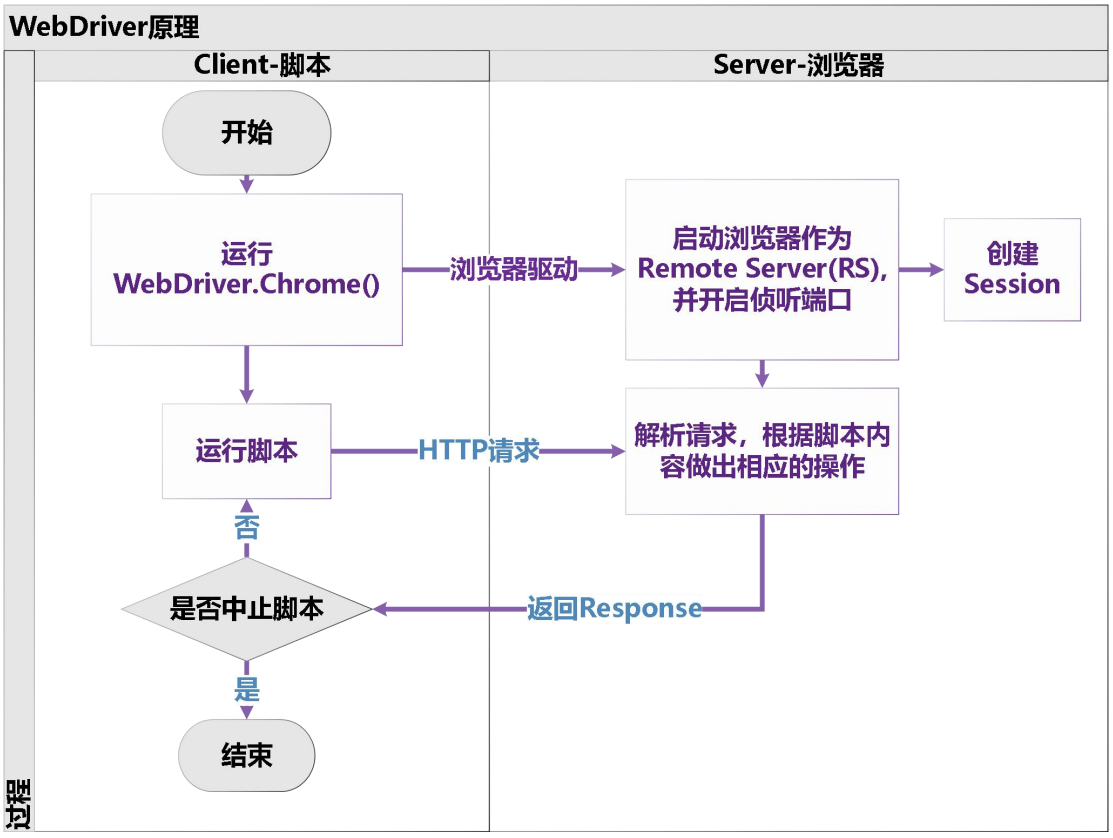
了辨别用户身份、进行 session 跟踪而储存在用户本地终端上的数据（通常经过加密），因此我们如果能知道账户名和密码的 cookie 信息，我们在登录之前直接加入 cookie，那就能绕过登录过程，绕过登录过程也就绕过了验证码。

要通过 add_cookie()方法在浏览器中加入 cookie，就必须要知道账号和密码的键值。下面为 CRM 绕过登录的方式：

```
driver.add_cookie({"name":"PHPSESSID", "value":"***"})
driver.add_cookie({"name":"BIDUPSID", "value":"***"})
```

2.6Webdriver 工作原理

webdiver 原理是经典的 Server-Client 结构（C/S）：



- WebDriver 操作浏览器、页面采用的协议：The WebDriver Wire Protocol，
- Client 和 Server 的通信协议：HTTP。

3. Unittest 测试框架

3.1 测试框架介绍

什么是自动化测试框架？

在了解什么是自动化测试框架之前，先了解一下什么叫框架？框架是整个或部分系统

的可重用设计，表现为一组抽象构件及构件实例间交互的方法；另一种定义认为，框架是可被应用开发者定制的应用骨架。前者是从应用方面，而后者是从目的方面给出的定义。 从框架的定义可以了解，框架可以是被重用的基础平台；框架也可以是组织架构类的东西。其实后者更为贴切，因为框和架本来就是组织和归类所用的。

所以自动化测试框架的定义为：由一个或多个自动化测试基础模块、自动化测试管理模块、自动化测试统计模块等组成的工具集合。

按框架的定义来分，自动化测试框架可以分为：基础功能测试框架、管理执行框架；

基础功能测试框架：

提供元素定位，元素操作，验证结果等，如 Selenium

管理执行框架：

组织、管理、执行测试用例，并反馈测试结果，如 robotframework。

测试框架总体而言可以参考软件开发框架来构建，下面是从软件开发框架原则中对应提取的测试框架的属性：

1、测试框架是测试开发过程中提取特定领域测试方法共性部分形成的体系结构；（软件框架是软件开发过程中提取特定领域软件的共性部分形成的体系结构）

2、测试框架的作用：在其基础上重用测试设计原则和测试经验，调整部分内容便可满足需求，可提高测试用例设计开发质量，降低成本，缩短时间；

3、不同测试技术领域有不同的测试框架类型；

4、测试框架不是一个现成可用的系统，是一个半成品，需要测试工程师基于它结合自己的测试对象知识转化成自己的测试用例；

5、测试框架是提供给测试人员开发相应领域测试用例的测试分析设计工具；

6、测试框架不是测试用例集，而是通用的，具有一般性的系统主体部分。测试人员像做填空一样，根据具体业务完成特定应用系统中与众不同的特殊部分；

7、测试设计模式的思想（等价类/边界值）在测试框架中进行应用。

*现在常说的自己编写自动化测试框架，都是指对 selnium 等基础框架和 unittest、Junit、testNG 等单元测试框架进行二次封装形成的管理执行框架。虽然 unittest 是一个单元测试框架，但是现在也广泛的应用到了 UI 自动化测试、接口自动化测试、APP 自动化测试等功能性的自动化测试中。

我们是用 python+selenium 进行自动化测试，因此我们的框架就是用 unittest 和 selenium 进行二次封装形成我们自己的框架。

那么先看一个 unittest 例子：

```
import unittest
class MyTest_A(unittest.TestCase):
      """测试 A"""
#setUP 和 tearDown 在每个用例执行时都会调用
    def setUp(self):
```

```
        print("test.初始化")
#注意：测试用例必须以 test 开头，否则 unittest 不认为是测试用例
    def test_a(self):
        """测试 a"""
        print("我是测试用例 a！")
    def test_b(self):
        """测试 b"""
        print("我是测试用例 b！")
    def tearDown(self):
        print("test.清理")

if __name__ == '__main__':
        unittest.main()
```

通过对 unittest 中的 TestCase 类的继承，生成的 MyTest_A 类，就是我们的一个测试用例类：

- setUp(), tearDown()这两个方法是用来对用例进行初始化和清理的，也就是在每个用例执行前和执行后，都会自动调用的方法；**注意：是每个用例。**
- test_a(), test_b()这两个方法就是我们的测试用例，方法体就是我们的测试内容；一个以 test 开头的方法就是一个测试用例；
- if __name__ == '__main__'，表示当前文件可以执行，执行的内容就在该方法的方法体中；
- unittest.main()，main 函数是默认的执行参数，按照 ASCII 码的顺序执行用例。ASCII 码的顺序为 0-9，A-Z，a-z，注意大写字母的顺序是在小写字母之前。

3.2 Unittest 框架

unittest 框架中，包含了四个重要的概念，这些概念就概括了整个的框架：

Test Case：

一个测试用例就是一个完整的单元，这个单元包括测试前准备（setUp）、实现测试过程（run）、测试环境清理（tearDown）；

Test Suite：

用来组装单个的测试用例，通过 addTest 加载到 TestCase 的 TestSuite 中，返回一个 TestSuite 实例；

Test Runner：

测试的执行，一般测试框架都会提供丰富的执行策略和执行结果。unittest 中通过 TextTestRunner 类提供的 run()方法来执行 test suite/test case；

Test Fixture：

测试环境的搭建和清理。也就是 setUp 和 tearDown。

3.3 Unittest 测试用例组织

3.3.1 TestSuite

前面讲到 unittest 中使用 main()函数执行用例，是按照 ASCII 码的顺序执行当前文件中所有的测试用例(“test”开头)。那如果我想自己组装用例，只跑自己想要跑的用例或者按自己想要的顺序去跑用例呢？一个框架不会那么菜，当然会有。unittest 中提供了 TestSuite 测试套件来实现这个过程。

将 3.1 中的例子存为文件 mytest_a.py:

```
from mytest_a import MyTest_A
#实例化测试套件 TestSuite
suite = unittest.TestSuite()
#将用例添加到测试套件
suite.addTest(MyTest_A("test_b"))
suite.addTest(MyTest_A("test_a"))
#实例化测试执行 TextTestRunner
runner = unittest.TextTestRunner()
#运行测试套件
runner.run(suite)
```

上面的例子，通过引入前面的 MyTest_A 用例类，将其中的两个测试用例用 addTest()方法加入测试套件，并用 TextTestRunner 类中的 run()方法去执行测试套件。测试套件其实就是一个测试用例的集合。

- addTest(MyTest_A("test_b"))：addTest()方法添加的方式 MyTest_A 是对应的测试用例类，("test_b")表示测试类中的哪个用例（方法名）。可以一直添加。
- run(suite)：通过 TextTestRunner 类中的 run()方法执行已经添加完成的测试套件 suite。

3.3.2 discover 方法

一个大的系统，不可能仅仅就几个文件，特别对于自动化测试来说，文件可能很多，测试用例也是几十上百计。如果按照 TestSuite 去一个个添加，添到手软都不见得能添加出来一个；何况更多的时候要根据不同的情况组织不同的测试用例。因此我们需要一个更为强大方便的方式来组织测试用例的运行。unittest 中提供了一个更为强大的方式来组织测试用例，那就是 discover()方法：

```
discover = unittest.defaultTestLoader.discover(test_dir, pattern="test*.py")
runner = unittest.TextTestRunner()
```

```
runner.run(discover)
```

执行上面的方法，就是执行 test_dir 目录下所有的以 test 开头的 py 文件。相当于 discover()方法就是组织测试用例文件的方式。执行顺序也是按 ASCII 码的顺序执行测试文件，再按 ASCII 码顺序执行文件中的测试用例。如果执行的目录中包含了目录（且目录中有__init__.py 文件），先执行指定目录下的文件，在按 ASCII 的顺序去找子目录中的匹配的文件，再按 ASCII 的顺序执行子目录文件中的测试用例。

discover()方法是 defaultTestLoader 类中提供的方法，共需要三个参数：

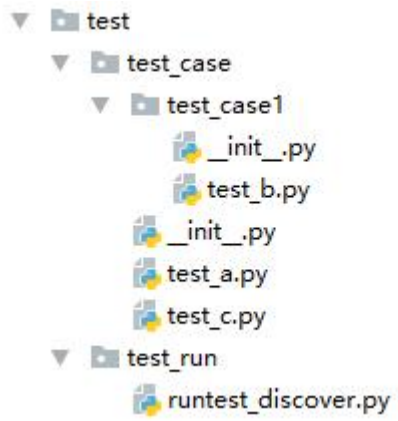
discover(start_dir, pattern="test*.py", top_level_dir=None)

找到指定目录下的所有测试模块，并可递归查到子目录下的测试模块，只有匹配到文件名才会被加载。如果启动的不是顶层目录，那么顶层目录必须单独指定。

- start_dir: 要测试的模块名或测试用例目录；
- pattern="test*.py": 表示用例文件名的匹配规则。此处匹配文件名以"test"开头的".py"类型的文件，星号"*"表示任意多个字符；
- top_level_dir=None: 如果测试用例包含多级目录，而 start_dir 又不是测试用例的顶层目录，那么就需要指定顶层目录，一般默认为 None。

注意：

- 测试目录中包含多个子目录，则需要添加一个空的__init__.py 文件，否则 discover()不会执行没有__init__.py 的子目录。



对于多级目录的用例（上图），需要执行 test\test_case\test_case1\test_b.py 文件，那么写法如下：

```
test_dir = "../../test/test_case/test_case1"
top_level_dir = "../../test/test_case"
discover = unittest.defaultTestLoader.discover(test_dir,
                                               pattern="test*.py"
                                               top_level_dir=top_level_dir
                                               )
runner = unittest.TextTestRunner()
runner.run(discover)
```

不过顶层目录不指定也没什么影响。一般默认 None 就行。

4.自动化脚本设计

4.1 自动化断言

除了四个重要的概念之外，另外两个重要的概念就是断言和参数化。做任何测试的时候，都必须要判断我们的测试是否有效果，当我们用手工测试的时候我们通过肉眼去确认程序的响应结果是否正确，但是在自动化测试中，我们必须预设预期的结果，通过程序响应的实际结果与预期结果的对比，来判断程序的执行是否符合我们的预期，这个过程就叫断言。

本节，我们将来了解一下 unittest 中提供的断言方法：

```
import unittest

class MyTest_C(unittest.TestCase):
    """测试 C"""

    def setUp(self):
        self.s = "登录成功"

    def test_a(self):
        """测试用例 a"""
        self.assertEqual(self.s, "登录成功!", "测试不通过")
        print("我是测试用例 a！")

    def test_b(self):
        """测试 b 用例"""
        self.assertEqual(self.s, "登录成功", "测试不通过")
        print("我是测试用例 b！")

    def tearDown(self):
        print("test.清理")

if __name__ == '__main__':
        unittest.main()
```

其实断言就是通过两个不同的参数去对比，类似：

```
if s == "登录成功":
    pass
```

```
else:
    raise AssertionError("测试不通过")
```

常用的断言方法：

assertEqual(arg1, arg2, msg=None)	验证 arg1=arg2，不等则 fail
assertNotEqual(arg1, arg2, msg=None)	验证 arg1 != arg2，相等则 fail
assertTrue(expr, msg=None)	验证 expr 是 true，如果为 false，则 fail
assertFalse(expr,msg=None)	验证 expr 是 false，如果为 true，则 fail
assertIsNone(expr, msg=None)	验证 expr 是 None，不是则 fail
assertIsNotNone(expr, msg=None)	验证 expr 不是 None，是则 fail
assertIn(arg1, arg2, msg=None)	验证 arg1 是 arg2 的子串，不是则 fail
assertNotIn(arg1, arg2, msg=None)	验证 arg1 不是 arg2 的子串，是则 fail

基本断言方法：基本的断言方法提供了测试结果是 True 还是 False。所有的断言方法都有一个 msg 参数，如果指定 msg 参数的值，则将该信息作为失败的错误信息返回。

4.2 自动化参数化

参数化这个概念相信大家并不陌生,特别是对各种测试工具、性能工具等有一定了解的。

什么是参数化？参数化是一个将测试数据与测试逻辑（步骤）分开， 简化测试用例的过程；方式是将用例中的一些输入、输出等作为参数，数据则单独列出，在执行时选择相应的数据执行。比如一个系统登录的过程，想用不同的人去进行登录操作，要么就需要随时去改脚本，要么就只有重复的写脚本，导致脚本的复用性很差，很难控制和执行；因此如果我们将其中的账号密码作为参数抽取出来，每次执行登录用例的时候，只需要去修改这两个数据即可，大大增加了脚本的复用。这也就想我们写手工测试用例的时候，把数据和操作分开，那么操作步骤只需要写一个,数据可以设计很多,这样就可以完成各种输入输出情况的测试。

在我们用 Python 进行自动化测试的过程中，参数化方式主要有以下几点;

参数化形式：

自定义变量:list、tuple、dict、String ——for 循环取值；

文件读取：txt、csv、Excel；

数据读取：JDBC、pyMysql；

4.2.1 获取数据作为参数

通过 selenium 中的方法提取需要的数据，作为后续脚本执行的参数；还记得我们讲 2.2.5 简单元素操作的时候获取数据的方法么？

```
driver.current_url      #获取当前 url
```

```
driver.find_element_by_link_text("新闻").text      #获取元素的文本
driver.find_element_by_id("kw").size      #获取输入框的大小
driver.find_element_by_id("kw").get_attribute("class") #获取输入框元素的class属性的属性值
driver.title                    #获取当前HTML的title
```

那么我们通过上面的方式获取到这些数据后，可以用变量存储，也可以作为函数的返回值，在后续的脚本中，可以读取这些参数和返回值，或用来做断言，或作为其他语句的参数。

4.2.2 从文件读取参数

1. txt 文件

Python 课程中已讲过如何读取 txt 文件中的数据

```
with open("E:\\1.txt", "r") as f:
    list1 = f.readlines()
```

这里顺带给大家讲一个新的关键字 with…as…，用 with…as…的方式不用关闭文件，因为 with…as…语句会自动关闭文件。与下面的语句相同，体会一下：

```
f = open("E:\\1.txt", "r")
try:
    data = f.read()
finally:
    f.close()
```

小知识：

- with…as…也叫上下文管理器，基本思想是 with 所求值的对象必须有一个__enter__()方法，一个__exit__()方法。
- 紧跟 with 后面的语句被求值后，返回对象的__enter__()方法被调用，这个方法的返回值将被赋值给 as 后面的变量。当 with 后面的代码块全部被执行完之后，将调用前面返回对象的__exit__()方法。
- with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

其他的读取方式：

- read(): 读取整个文件
- readline(): 读取一行数据
- readlines(): 读取所有行的数据

2. 读取 csv 文件

CSV（Comma-Separated Values，逗号分隔值，有时也称为字符分隔值，因为分隔字符也可以不是逗号），其文件以纯文本形式存储表格数据（数字和文本）。是一种通用的、相对简单的文件格式，被用户、商业和科学广泛应用。最广泛的应用是在程序之间转移表格数

据。

CSV 可以通过文本文档或者 Excel 进行创建和编辑，其格式就是行与行之间用回车符分隔，单元格与单元格之间用逗号分隔。

账号	密码
zhangsan	123
lisi	321
wangwu	777

对于下面这样一组简单的数据，在文本文档中的格式为：

```
账号,密码
zhangsan,123
lisi,321
wangwu,777
```

要用 Python 操作 CSV 文件，需要引入 csv 库：

```
import csv                                  #引入 csv 包
f= open(filename, 'r '))
data = csv.reader(f)                        #读取 csv 文件
for user in data:
    print(user)
f.close()
```

打开 csv 文件和关闭 csv 文件的过程与打开普通 txt 文件方式一样。只是读取方式有所不同。

3. excel 文件读取

这里我们只讲如何通过 excel 去读取数据，Python 从 excel 中读取数据会用到第三方库 xlrd。下面我们来看一下 xlrd 操作 excel 的过程，首先需要下载 xlrd:

```
pip install xlrd
```

下载完成后，引入 xlrd；

```
import xlrd
```

通过 open_workbook 方法打开一个 excel 文件：

```
# 打开一个 excel
data = xlrd.open_workbook(r"E:\data.xlsx")
```

excel 中包含多个 sheet 页，要使用哪个 sheet 页中的数据就打开对应的 sheet 页，打开方式有两种：1.通过 sheet 页的名称；2.通过 sheet 页的 index，index 从 0 开始；

```
# 通过 sheet 名称打开，下面的方式任选一种
table = data.sheet_by_name("login")
table = data.sheet_by_index(0)
```

通过 xlrd 可以获取 excel 中的行和列，以及单元格的数据

```
# 获取整行和整列的值（数组）
row_value = table.row_values(0)
col_value = table.col_values(0)
# 通过行列值获取单元格的值
cell_A1 = table.cell(0, 0).value
cell_C4 = table.cell(3, 2).value
# 使用行或者列来获取单元格数据，
cell_A1 = table.row(0)[0].value  # table.row()获取整行
cell_A2 = table.col(0)[1].value  # table.col()获取整列
```

如果需要遍历获取所有数据，必须要知道行数或列数：

```
# 获取行数和列数
nrows = table.nrows
ncols = table.ncols
```

具体的例子：

存在这样一个文件“E:\data.xlsx”，数据内容如下，我们使用 xlrd 提供的方法遍历整个名为“login”的 sheet 页，并返回一个二位数组：

	A	B	C	D
1	user	pwd	islogin	
2	nemo	123	success	
3	nemo		failed	
4				
5				

login

```
import xlrd
data = xlrd.open_workbook(r"E:\data.xlsx")
table = data.sheet_by_name("login")
e_list = []
for n_row in range(1, table.nrows):      #从 1 开始，去掉表头
        e_list.append(table.row_values(n_row))
print(e_list)
```

4.2.3 数据库读取参数

读取数据库相关的表-字段进行参数化，Python 课程中已经详细讲了如何查询数据库数据，这里不再重复。通过从数据库中取出数据，作为脚本执行过程中的参数。

4.3 自动化测试报告

在运行完测试用例后，需要用一个测试报告来作为自动化运行的一个报告成果，一般是统计用例运行的成功失败，这里就给大家引入两个测试报告模板 HTMLTestRunner 以及 BeautifulReport，这里基于 POM 设计模式实现这篇文章中的代码，来进行改造以及介绍具体的使用。

4.3.1HTMLTestRunner

这是一个 unittest 默认的 TextTestRunner 类生成的报告：

```
我是测试用例b!
.
test_run.清理
----------------------------------------------------------------------
我是测试用例a!
Ran 4 tests in 0.000s
test_run.清理

test_a.初始化
OK
a是1a
test_a.清理
test_a.初始化
b是2b
test_a.清理

Process finished with exit code 0
```

看起来不但不清晰，还和用例中的打印语句显示混乱。用例多了看起来很不方便。因此为了我们能更好看到自动化测试的结果，我们需要用到一个第三方的测试报告 HTMLTestRunner。

HTMLTestRunner 是 unittest 的一个扩展，生成比较直观的 HTML 测试报告。如下图：

第一轮冒烟测试

Start Time: 2017-09-04 18:47:24
Duration: 0:00:00.002002
Status: Pass 3 Failure 1 Error 1

测试用例执行情况

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
test_c.MyTest_C: 测试C	2	1	1	0	Detail
test_c_1: 测试b			pass		
test_c_2: 测试a			fail		
unittest.loader._FailedTest	1	0	0	1	Detail
test_b			error		
test_case2.test_a.MyTest_A: 测试A	2	2	0	0	Detail
test_a_1_1___a____1a__: 测试a			pass		
test_a_1_2___b____2b__: 测试a			pass		
Total	5	3	1	1	

是不是看起来更清晰，哪个用例失败了，哪个用例有异常，哪些用例成功了！那我们来看看怎么用：

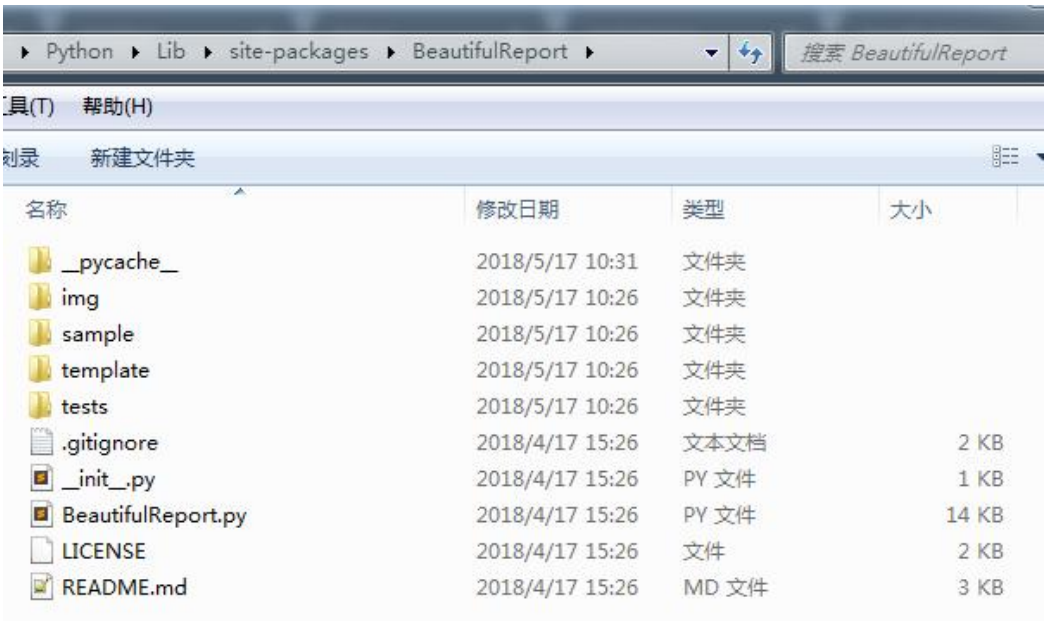
由于 HTMLTestRunner 已经很久没有更新了，目前只支持 python 2，因此需要的话向老师索取。现在假设你已经拿到 HTMLTestRunner 源文件，并放在了“…\Python36\Lib”目录中。

```
# 引入 HTMLTestRunner
```

```
from HTMLTestRunner import HTMLTestRunner
# 定义测试用例目录
test_dir = "./test/test_case"
# 与 discover 结合
discover = unittest.defaultTestLoader.discover(test_dir,
                                               pattern="test*.py")
# 新建报告文件，用 wb 二进制写的方式打开
f = open("./result.html", "wb")
# 实例化 HTMLTestRunner
runner = HTMLTestRunner(stream=f,
                        title="第一轮冒烟测试",
                        description="测试用例执行情况")
runner.run(discover)
f.close()
```

4.3.2BeautifulReport

下载 BeautifulReport 的完整.ZIP 文件，然后解压，把整个文件包放到本地 python 的/Lib/site-packages/目录下，如下图:

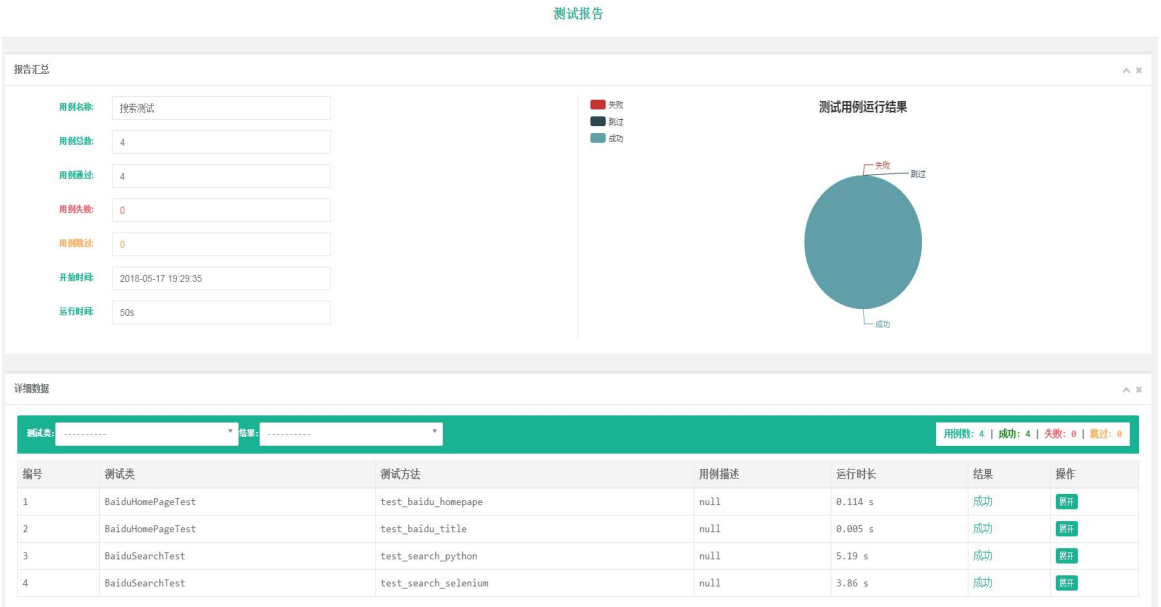


这里用到 unittest.defaultTestLoader.discover()方法批处理整合测试套件，再用 BeautifulReport()方法执行用例。代码如下：

```
from common.BeautifulReport import BeautifulReport
import time
import unittest

#获取用例集合
test_dir = "./test/test_case"
#  与 discover 结合
discover = unittest.defaultTestLoader.discover(test_dir, pattern="test*.py")
#使用 BeautifulReport 报告运行用例，结果写入报告
BeautifulReport(discover).report(
                description=u'自动化测试报告',
                log_path="./report/",
                filename=time.strftime("%Y-%m-%d %H_%M_%S")
)
```

执行脚本后生成的报告截图如下：

```
....
测试已全部完成，可前往C:\Users\Ice\Desktop\se查询测试报告
[Finished in 52.4s]
```

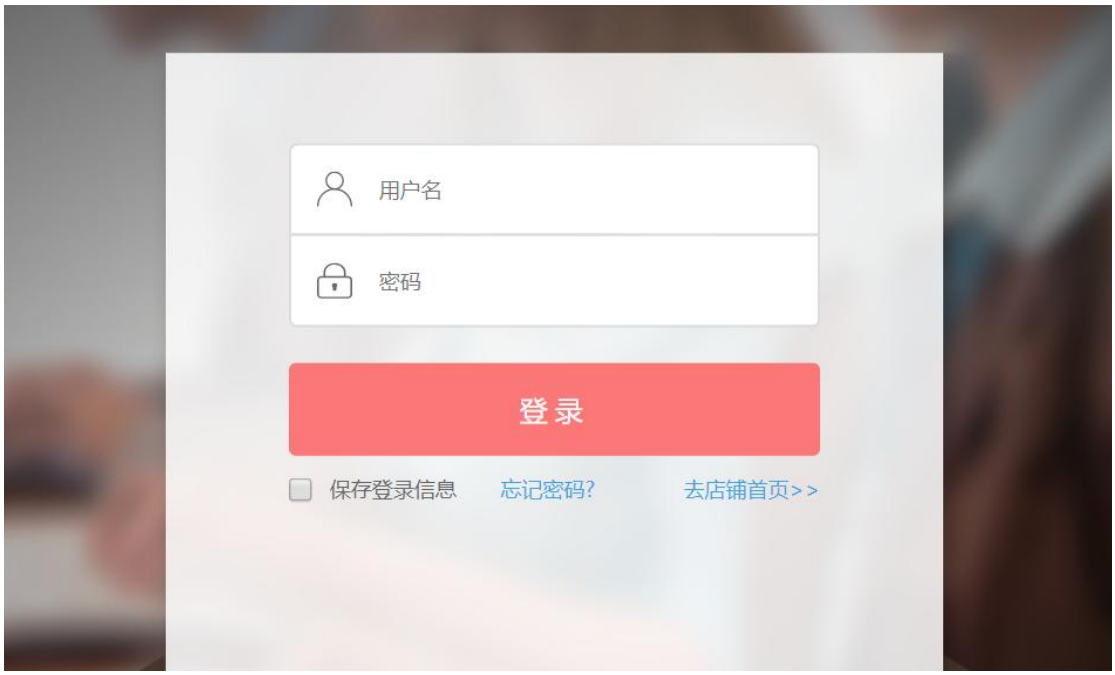


5. 自动化测试框架设计

Page Object 是一种 UI 自动化测试思想，其理念为将页面的交互细节封装起来，使测试用例更关注业务而非界面细节，从而提高测试案例的**可读性**。且对 UI 变动频繁的项目提高**可维护性**。

直观的理解就是把页面上的操作，比如定位元素、获取页面数据、以及元素操作等，都封装在同一个页面类里；而具体的定位元素、获取页面数据、以及元素操作就定义成类中的方法。

还是看例子吧：



就以登录页面为例，我们要实现登录操作，一个简单的登录操作，需要输入用户名、输入密码、点击登录按钮。常规的方法：

```
...
driver.find_element_by_name("username").send_keys("yourusername")
driver.find_element_by_name("password").send_keys("yourpassword")
driver.find_element_by_class_name("btn-a").click()
```

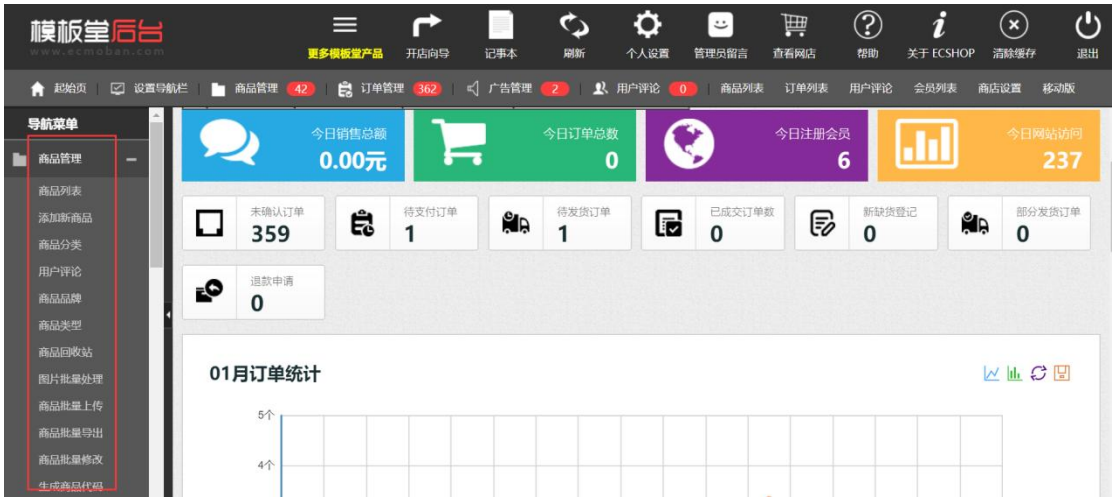
5.1 测试用例脚本设计

以上的脚本，你作为练习，或者说只是一个登录当然没什么问题；但是如果你很多测试用例需要用到登录，甚至说每次登录的账号密码都不一样，那每次都需要去修改脚本或者写很多同样的脚本。为了让我们减少这样的重复代码，我们可以把登录抽象为一个函数：

```
...

def login(uname, passwd):

    driver.find_element_by_name("username").send_keys(uname)

    driver.find_element_by_name("password").send_keys(passwd)

    driver.find_element_by_class_name("btn-a").click()
```

通过将登录操作抽象为函数之后，其他的脚本只要需要用到登录操作，直接调用登录的函数即可，不用再重新编写重复的代码；当需要用不同的用户进行登录操作时，传入不同的参数即可。这样可以减少很多重复的代码。

由于登录页面功能较单一，因此一个函数就可以达到一个较高的代码复用，但是当下图这样一个页面中存在非常多不同的操作流程，这些不同的操作流程中都涉及到一些相同元素的操作。比如需要编辑某一条数据的时候，会用到搜索功能；需要删除时需要用到搜索功能；单独写搜索用例的时候也会用到搜索功能；导出的时候会用到搜索功能等等。



对于这种不同的用例（脚本）需要用到同一个元素，当这个元素发生改变导致定位失效

时，涉及到这个元素的所有脚本都需要修改定位方法。在 web 测试中 web 页面的变化又是最频繁的，因此这种脚本的维护会非常麻烦。因此 Page Object 的思想应运而生，通过将页面元素进行封装后，涉及到元素定位或操作失效的地方，就修改对应页面类中对应的页面方法即可，大大提高了脚本的可维护性。

5.2 业务方法封装及调用

为了举例方便，我们还是拿登录来说明！上面我们已经把登录抽象封装为一个函数，现在我们进一步抽象，将登录页面抽象为 login 类，把页面元素抽象为 login 类中的方法：

```
...
class LoginPage():
    """登录页面类"""

    def __init__(self, driver):
        """声明属性，url 可以用默认，可以修改"""
        self.driver = driver      # 浏览器控制的 driver 不能少
        self.url = http://192.168.1.241/hdshop/admin/index.php # 定义 url

        # 定义定位器
         self.login_username_loc = (By.NAME, "username") #元组数据
         self.login_password_loc = (By.NAME, "password")
         self.login_button_loc = (By.CLASS_NAME, "btn-a")

    def open(self):
        """打开 url"""
        self.driver.get(self.url)
        self.driver.implicitly_wait(self.time)    # 设置隐式等待

    def login_username(self, uname):
        """用户名输入框"""
        self.driver.find_element(*self.login_username_loc).clear()
        self.driver.find_element(*self.login_username_loc).send_keys(uname)

    def login_password(self, passwd):
        """密码输入框"""
        self.driver.find_element(*self.login_password_loc).clear()
        self.driver.find_element(*self.login_password_loc).send_keys(passwd)
```

```
    def login_button(self):
        """密码输入框"""
        self.driver.find_element(*self.login_button_loc).click()

    def login(self, username, password):
        """登录操作"""
        self.open()
        self.login_username(username)
        self.login_password(password)
        self.login_button()
```

通过上面的抽象，那么我们在测试用例中就可以直接调用页面类,先将上面的 LoginPage 类的脚本保存为 login_page.py，新建一个 login_case.py 文件：

```
import unittest
from selenium import webdriver
from login_page import LoginPage

class LoginTestCase(unittest.TestCase):
    """登录测试用例"""
    def setUp(self):
        """初始化测试用例"""
        self.driver = webdriver.Chrome()
        self.driver.maximize_window()

    def test_login_case(self):
        """登录成功"""
        lp = LoginPage(self.driver)
        lp.login("yourusername", "yourpassword")

    def tearDown(self):
        """清理退出"""
        self.driver.quit()
```

通过抽象封装之后，我们在写任意用例的时候当需要用到登录，就直接实例化 LoginPage 类，并调用其中的登录方法。当登录的元素发生变更时，直接修改 LoginPage 中对应的方法即可。

我们还可以进一步抽象，将所有页面都可能用到的方法（比如浏览器的 driver、url、打开浏览器的 open()方法等）抽象出来，作为页面类的基类中的方法。另外我们可以将

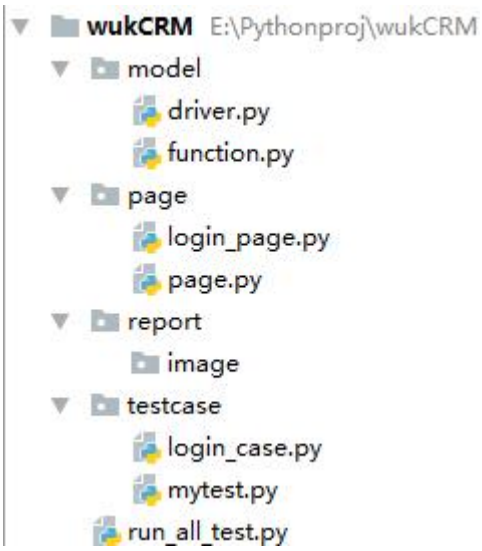
LoginTestCase 类中 setUp()和 tearDown()方法抽象出来，其他的测试用例中只考虑用例的逻辑。

5.3 测试框架搭建

现在我们可以先规范一下项目的文件夹，这样看起来更像一个完整的项目：

```
Project_Name        # 顶层文件夹，项目名称
    |--driver       # 驱动文件夹，用于存放浏览器驱动文件等
    |--model        # 函数文件夹，启动浏览器、发送邮件、数据库操作等函数文件
    |--page         # 页面文件夹，页面的基类以及其他所有的页面类
    |--testcase     # 测试用例，测试用例的基类及所有的测试用例类
    |--report       # 存放测试报告，截图等测试结果文件
```

实例如下：



我们来看看每个文件的内容，试着敲一下，能理解到，这就是你自己的框架了！

：

5.3.1 浏览器驱动封装

封装浏览器驱动方法，以后比如要用到多种浏览器测兼容性等，都可以在这里实现。

wukCRM>model>driver.py 代码如下：

```
from selenium import webdriver


def browser_chrome():
    """启动 Chrome"""
    driver = webdriver.Chrome()
```

```
    driver.maximize_window()
    return driver
```

5.3.2 公共方法封装

页面层的基类，封装所有页面都可能用到的一些方法，比如打开网址、简化元素查找方法、简化显式等待、简化下拉菜单处理等。

wukCRM>page>page.py:代码如下

```
class BasePage():
    """页面类基类"""

    def __init__(self, driver):
        self.driver = driver
        self.url = http://192.168.1.241/hdshop/admin/index.php # 定义 url
        self.timeout = 30

    def open(self):
        """打开 url"""
        self.driver.get(self.url)
        self.driver.implicitly_wait(self.timeout)   # 设置隐式等待

    def find_element(self, loc):
        return self.driver.find_element(*loc)
```

5.3.3 元素定位器封装

登录页面类，这是一个页面类的示例，后面所有的页面类都可以参照该页面来写。在实际操作的过程中，遇到什么页面就参考下面的代码写一个类；页面中的元素，就参考下面代码中的方法,首先将元素定位的方式以类属性的方式写在定位器中,再分析元素的基本操作，写在该元素的方法中。该页面类继承页面基类。**wukCRM>page>login_page.py 代码如下:**

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

class LoginPage(BasePage):
    """登录页面类"""
```

```
    #定位器
    login_username_text_loc = (By.NAME, "username")
    login_password_text_loc = (By.NAME, "password")
    login_button_loc = (By.CLASS_NAME, "btn-a")

    def login_username(self, uname):
        """用户名输入框"""
        self.find_element(self.login_username_text_loc).clear()
        self.find_element(self.login_username_text_loc).send_keys(uname)

    def login_password(self, passwd):
        """密码输入框"""
        self.find_element(self.login_password_text_loc).clear()
        self.find_element(self.login_password_text_loc).send_keys(passwd)

    def login_button(self):
        """登录按钮"""
        self.find_element(self.login_button_loc).click()

    def login(self, username, password):
        """登录操作"""
        self.open(self.url)
        self.login_username(username)
        self.login_password(password)
        self.login_button()
```

用例通用方法封装

测试用例的基类，用于封装所有测试用例都可能用到的一些方法，比如每个用例都会用到初始化和清理，那么最常见的初始化和清理就是打开浏览器和关闭浏览器，其他的用例就不用再去写初始化和清理了。**wukCRM>testcase>mytest.py 代码如下:**

```
import unittest
from model.driver import browser_chrome

class MyTest(unittest.TestCase):
    """测试用例基类"""
```

```
    def setUp(self):
        self.driver = browser_chrome()

    def tearDown(self):
        self.driver.quit()
```

登录的测试用例，这是一个用例的示例，其他测试用例可参照该用例的写法。在写用例的步骤中，用到什么页面上的什么元素，就实例化对应的页面类，然后调用其中对应的元素方法即可。该用例类继承自测试用例基类。**wukCRM>testcase>login_case.py 代码如下：**

```
from page.login_page import LoginPage
from testcase import mytest

class LoginTest(mytest.MyTest):
    """登录测试"""
    def test_login_suss(self):
        """测试登录成功"""
        login = LoginPage(self.driver)
        login.login("yourusername", "yourpassword")
```

5.3.4 批量执行用例生成报告

运行测试的方法，用 discover 组装测试用例，利用 HTMLTestRunner 运行并生成测试结果。wukCRM>run_all_test.py：

```
import unittest,time
from HTMLTestRunner import HTMLTestRunner
from model.function import send_mail

test_dir = "./testcase"
test_report = "./report"
#组织测试用例
discover = unittest.defaultTestLoader.discover(start_dir=test_dir,
                                               pattern="*_case.py")

if __name__ == '__main__':
    #格式化当前日期
```

```
    times = time.strftime("%Y%m%d%H%M%S")
    #组装测试报告路径和文件名
    report_file = test_report + "/WkCRM" + times + "result.html"
    file = open(report_file, 'wb')
    #实例化测试报告
    runner = HTMLTestRunner(stream=file,
                            title="悟空 CRM 自动化测试报告",
                            description="运行环境：window 10,  Chrome")
    #执行测试
    runner.run(discover)
    file.close()
```

现在框架已搭好，当你需要添加页面时，就在 page 文件夹中新建新的 page 类继承自 page 基类;当你需要添加测试用例,就在 testcase 文件夹中新建 testcase 类继承自 mytest 基类。

5.4Page Object 理念

最后，再回过头来理解一下 Page Object 理念：

其实最简单的理解就一句话： ***一个页面一个类，一个元素一个方法。***

由上面我们搭建的框架脚本也可以看出，Page Object 对于测试脚本的设计提供了一种分层思想：

基本层（page 基类）：基本层包括初始化方法以及封装 webdriver 中的最基本的方法(open 方法以及 find_element 方法,还有根据自己的需要封装自己使用到的 webdriver 方法）。

页面层（page 类）：页面层对定位元素的封装。

逻辑层（testcase 类）：为对业务逻辑层面的封装。这样对业务逻辑进行封装之后，测试的时候，只需要传入测试数据就可以了，而不用在去思考业务逻辑层面的东西。还有以后定位元素变了只需要改变页面层的东西,业务逻辑变之后,只需要改变逻辑层的东西。