

一. Web自动化测试

1. 什么是自动化

由机器代替人为完成指定任务的过程

优点:

1. 代替人工
2. 提高效率
3. 统一标准
4. 批量生产

2. 什么是自动化测试

1. 让程序代替手工去验证系统功能是否符合预期(测试工作)

2. 解决什么问题:

1. 回归测试--版本回归
2. 兼容性测试--web浏览器
3. 压力测试--模拟虚拟用户
4. 提高测试效率,保证产品质量

3. 优点:

1. 更少的时间执行更多的测试
2. 重复使用
3. 减少人为失误
4. 克服手工的局限性

4. 注意:

1. 自动化不能完全代替手工测试
2. 自动化测试跟手工测试没有可比性
3. 自动化测试主要是保证系统中没有已知缺陷(发掘缺陷的主要是手工测试)
4. 完全自动化无法实现的

5. 自动化测试分类

1. 单元测试的自动化
2. 接口测试的自动化
3. 系统测试的自动化
 1. web(本阶段学习)
 2. 移动

3. 什么是web自动化测试

1. 让程序代替手工验证一个web系统的功能是否符合预期的过程, **主要在手工测试完成之后,属于黑盒(功能)**
2. 什么情况下适合做自动化
 1. 需求稳定
 2. 周期长
 3. 需要回归

总结-记忆

1. 自动化测试的概念?
用程序代替人为验证系统功能是否符合预期的过程(测试过程)
2. 自动化测试能解决什么问题?
回归测试--版本回归
兼容性测试--web浏览器兼容
压力测试--模拟虚拟用户
提高测试效率, 保证产品质量
3. 什么样的web项目适合自动化测试?
需求稳定--周期长--需要回归
4. web自动化测试所属分类?
黑盒测试--功能测试

二. Web自动化测试工具选择

1. Web自动化测试工具

1. QTP -- 自动化测试工具--商业版--web/桌面
2. Selenium -- web自动化测试工具--开源--web

2. 什么是Selenium(特点)

web功能自动化测试工具-开源可扩展-支持多语言-多平台-多浏览器--企业需要

3. Selenium发展历史(了解)

我们使用的WebDriver

1. Selenium 1.0

1. IDE--脚本录制工具--浏览器插件
2. Grid--多台计算机同时执行测试用例(分布式用例执行)
3. RC--1.0中主要实现web自动化测试的工具

2. Selenium 2.0

1. Selenium 1.0 + webdriver
2. webdriver--主要实现web自动化测试的工具(直接操作浏览器,效率比RC高)

3. Selenium 3.0

1. 去掉RC的支持
2. 加强2.0

三. WebDriver环境搭建

1. 什么是WebDriver

它是一个web功能自动化测试工具(selenium),它支持主流的浏览器以及编程语言

2. 基于WebDriver环境搭建

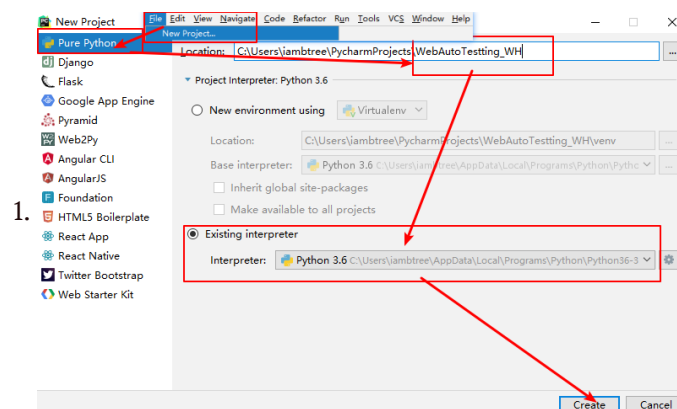
第一项:python解释器

1. 验证解释器安装--python命令是否加入环境变量

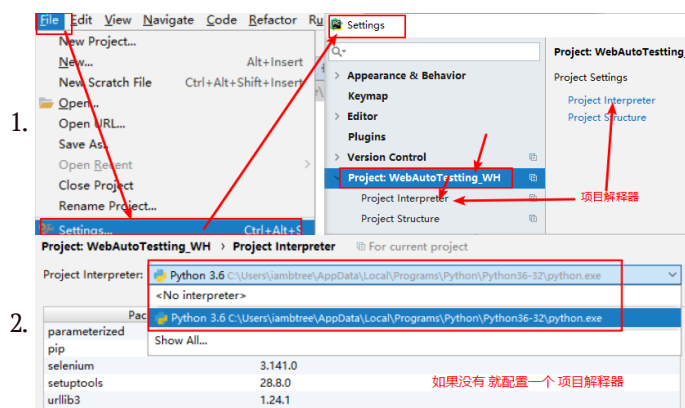
```
C:\Users\jambtree>python
Python 3.6.4 (v3.6.4:4d9e6eb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

第二项:python IDE

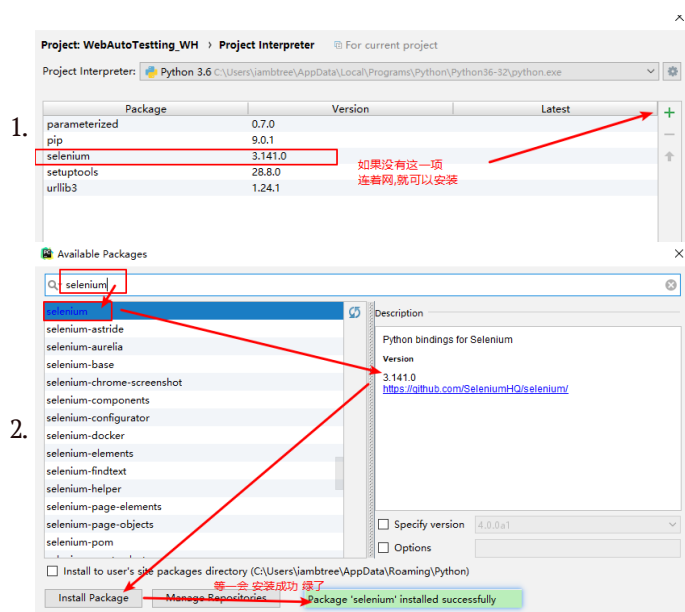
1. 注意1: 创建项目--注意选择系统解释器



2. 注意2: 检查项目中有没有解释器--如果没有设置一个



3. 注意3: 检查解释器中有没有Selenium--安装一个



第三项: webdriver--selenium

使用pip(python包管理工具安装Selenium)(注意--需要网络)

1. 安装--pip install 包名

```
C:\>pip install selenium
Collecting selenium
Using cached https://files.pythonhoste
3/selenium-3.141.0-py2.py3-none-any.whl
Requirement already satisfied: urllib3 i
Installing collected packages: selenium
Successfully installed selenium-3.141.0
```

2. 查看--pip show 包名

```
C:\>pip install selenium
Collecting selenium
Using cached https://files.pythonhoste
3/selenium-3.141.0-py2.py3-none-any.whl
Requirement already satisfied: urllib3 i
Installing collected packages: selenium
Successfully installed selenium-3.141.0
```

3. 卸载--pip uninstall 包名

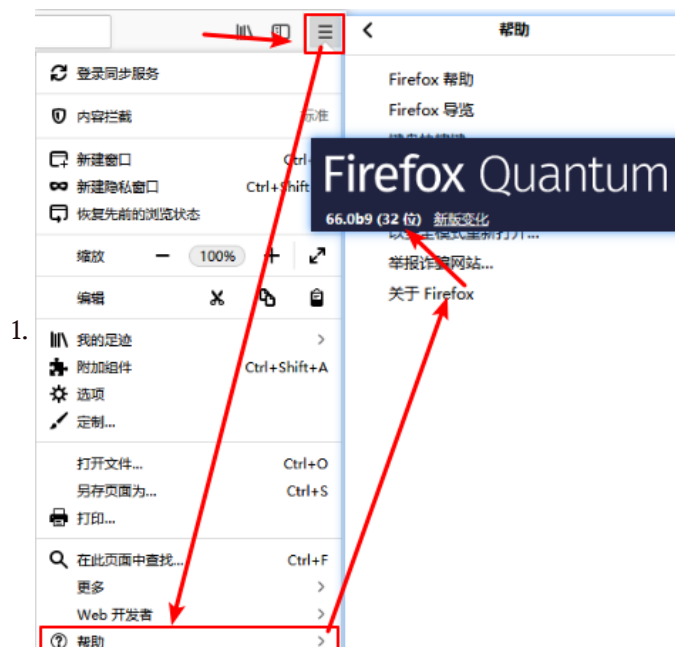
```
C:\>pip uninstall selenium
Uninstalling selenium-3.141.0:
Would remove:
d:\software\python36\lib\site-packages\selenium
d:\software\python36\lib\site-packages\selenium
Proceed (y/n)? y
Successfully uninstalled selenium-3.141.0
```

第四项: 浏览器--驱动(驱动一定要添加到环境变量中)

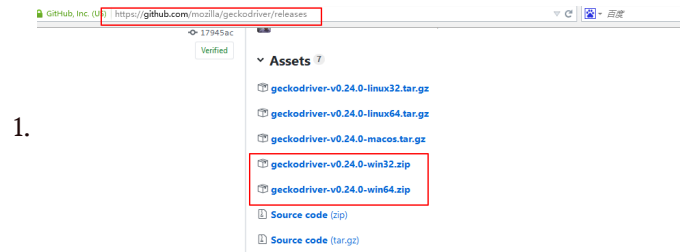
1. 第一步--确定浏览器的版本
2. 第二步--下载对应版本驱动
3. 第三步--配置驱动(环境变量)

火狐浏览器--驱动配置

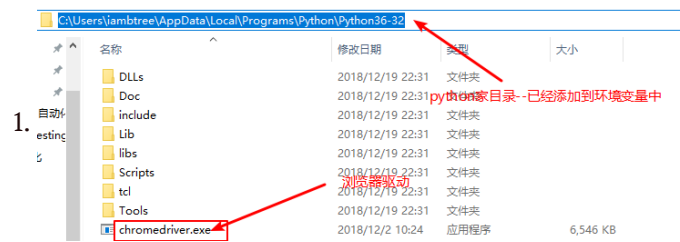
1. 浏览器版本



2. 下载驱动(<https://github.com/mozilla/geckodriver/releases>)

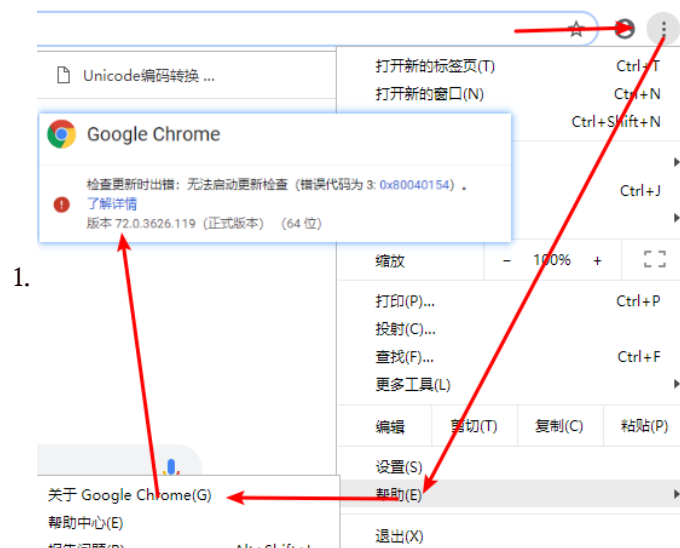


3. 配置驱动--驱动一定要放在环境变量中--验证过python已经加入环境变量--把驱动放入python家目录(安装目录)



谷歌浏览器--驱动配置

1. 浏览器版本

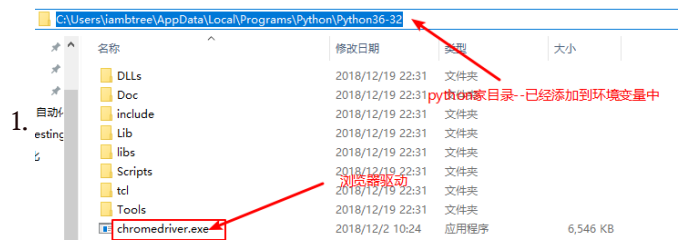


2. 下载驱动

1. 网址:<http://npm.taobao.org/mirrors/chromedriver>
2. 哪个版本



3. 配置驱动--驱动一定要放在环境变量中--验证过python已经加入环境变量--把驱动放入python家目录(安装目录)



3. 入门示例--验证环境--基本框架

准备工作

1. 创建项目
 1. 解释器
 2. selenium
2. 创建一个今天的代码目录day1

Web自动化测试脚本编写基本步骤--模拟手工执行的过程



1. 工具导进来--导包操作
2. 打开浏览器--实例化一个浏览器驱动对象
3. 输入网址--浏览器驱动对象调用get("完整的网址字符串")

4. 业务操作--元素定位和元素操作
5. 关闭浏览器--退出浏览器驱动对象

```
# 1. 工具导进来--导包操作 webdriver
from selenium import webdriver
import time

# 2. 打开浏览器--实例化一个浏览器驱动对象
# 谷歌浏览器
driver = webdriver.Chrome()
# driver = webdriver.Firefox()

# 3. 输入网址--浏览器驱动对象调用get("完整的网址字符串")
driver.get("https://www.baidu.com")

# 4. 业务操作--元素定位和元素操作
time.sleep(3)

# 5. 关闭浏览器--退出浏览器驱动对象    浏览器驱动对象调用quit()
driver.quit()
```

总结--记忆

1. WebDriver环境搭建中涉及到的软件?
 - python解释器
 - pycharm集成开发环境
 - webdriver--selenium
 - 浏览器&驱动
2. pip 安装、卸载、查看selenium命令?
 - pip install 包名
 - pip uninstall 包名
 - pip show 包名
3. web自动化测试脚本编写的基本步骤?
 1. 导入webdriver
 2. 实例化一个浏览器驱动对象
 3. 浏览器驱动对象调用get方法--参数--完整的网址字符串
 4. 元素定位&元素操作
 5. 浏览器驱动对象调用quit()方法

四. 元素定位

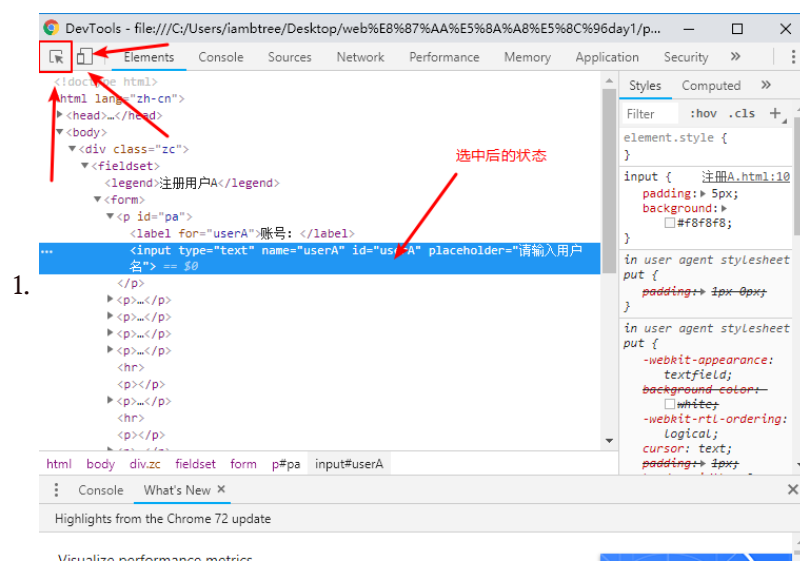
1. 为什么要学习元素定位,如何定位

1. 为了让脚本能够发现页面元素,进而执行操作
2. 依据:元素的属性信息,层级关系

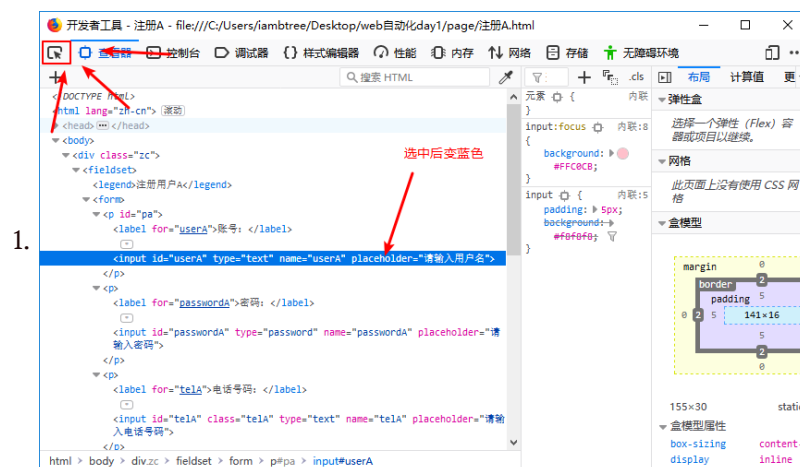
2. 快速查看页面元素信息

浏览器开发工具(F12--右键元素检查/查看元素)--快速定位元素,查看元素信息

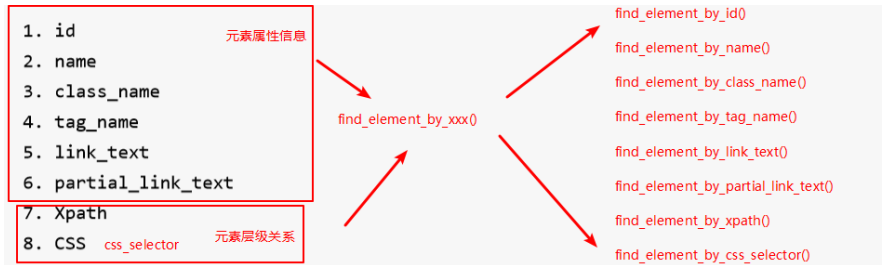
谷歌浏览器



火狐浏览器



3. 元素定位的方式和方法



`element = driver.find_element_by_xxx("定位依据")`

1. 调用--浏览器驱动对象的方法--`driver.find_element_by_xxx()`
2. 参数--定位时所依据的内容字符串--`driver.find_element_by_xxx("定位依据")`
3. 返回--定位到的页面元素--第一个--元素对象

id定位

1. 说明: 通过元素的id属性定位元素
2. 前提: 必须有id属性
3. 方法: `element = driver.find_element_by_id("id属性值")`
4. 注意: id属性页面唯一, 所以优先使用

```
# id 定位

# a. 定位账号A, 输入用户名: admin
driver.find_element_by_id("userA").send_keys("admin")
# 元素输入操作 send_keys()
# 调用-- 元素对象的方法 element.send_keys()
# 参数-- 输入内容的字符串 element.send_keys("输入内容")
# 返回-- 无

# b. 定位密码A, 输入密码: 123456
element = driver.find_element_by_id("passwordA")
element.send_keys("123456")

# 连写定位和操作--当元素操作只有一次的時候
# 如果对一个元素对象的操作是多次, 定义一个变量保存对象
```

name定位

1. 说明: 通过元素的name属性定位元素
2. 前提: 必须有name属性

3. 方法: `element = driver.find_element_by_name("name属性值")`
4. 注意: `name`属性在HTML文档中是可以重复的,如果存在多个,返回的是页面上的第一个定位到的元素对象

```
# name
# a.定位注册A userA admin
driver.find_element_by_name("userA").send_keys("admin")

# b.定位密码A passwordA 123456
driver.find_element_by_name("passwordA").send_keys("123456")
```

class_name定位

1. 说明: 通过元素的`class`属性中的类名定位元素
2. 前提: 必须有`class`属性
3. 方法: `element = driver.find_element_by_class_name("class一个类名")`
4. 注意: 如果`class`属性存在多个类名,只能使用其中的一个类名

```
# class_name
# a.电话号码 telA 18611111111
driver.find_element_by_class_name("telA").send_keys("18611111111")

# b.电子邮箱中 emailA dzyxA 123@qq.com
# 只能填写一个类名
driver.find_element_by_class_name("dzyxA").send_keys("123@qq.com")

# 错误演示 输入多个类名
# driver.find_element_by_class_name("emailA dzyxA").send_keys("123@qq.com")
```

tag_name定位

1. 说明: 通过元素标签名称定位元素
2. 方法: `element = driver.find_element_by_tag_name("标签名称")`
3. 注意: 同一个标签在页面上回存在多个,不建议使用该方法定位元素

```
# tag_name
# a.页面上的第一个输入框--用户名输入框 input admin
driver.find_element_by_tag_name("input").send_keys("admin")
```

link_text定位

1. 说明: 专门用来定位超链接元素
2. 方法: `element = driver.find_element_by_link_text("超链接的全部文本内容")`
3. 注意: 传递的单数不许是超链接的完整文本内容

```
# link_text --- 完整的超链接文本内容
# a.访问 新浪 网站 超链接 定位 点击
# 元素点击操作 click()
# 调用--元素对象的方法 element.click()
# 参数--没有
# 返回--没有
driver.find_element_by_link_text("访问 新浪 网
站").click() # 预期 弹出新页面
```

partial_link_text定位

1. 说明: 专门用来定位超链接元素
2. 方法: `element = driver.find_element_by_partial_link_text("超链接的部分连续文本内容")`
3. 注意: 传递的单数尽量保证页面上唯一

```
# partial_link_text --- 连续的部分超链接文本内容
# a.访问 新浪 网站 超链接 定位 点击
# 元素点击操作 click()
# 调用--元素对象的方法 element.click()
# 参数--没有
# 返回--理解成没有
driver.find_element_by_partial_link_text("访问
新").click()

# 错误--不连续
# driver.find_element_by_partial_link_text("访问
网站").click()
```

总结--记忆

1. `id`、`name`、`class_name` -- 元素的属性--class中可能有多个类名
2. `tag_name` -- 标签名 -- 如果页面有唯一标签
3. `link_text`(完整的文本内容)、`partial_link_text`(连续的部分文本内容) -- 超链接a标签--文本内容

4. xpath定位策略和方法

xml path --xml (嵌套的标签组成)元素路径,html是一种特殊的xml实现,所以可以使用xpath进行定位

方法: find_element_by_xpath("xpath表达式")

层级路径-定位

一般xpath表达式使用的是相对路径定位(不怕页面结构修改) 但是如果实在定位不到目标元素,可以使用绝对路径查找目标元素

1. 格式: 绝对路径

1. 以/html开头--逐层不间隔--写到目标元素

2. 格式: 相对路径

1. //开头 -- //元素 -- //input -- //*

```
# xpath 路径

# a.绝对路径 用户名 admin
/html/body/div/fieldset/form/p[1]/input
driver.find_element_by_xpath("/html/body/div/fieldset/form/p[1]/input").send_keys("admin")

time.sleep(2)
# b.相对路径 用户名 123 //input
driver.find_element_by_xpath('//input').send_keys("123")
```

元素属性-定位

相对路径定位元素,数量多,不精确--限制属性信息进行精确定位

1. 格式: //标签名[@属性名 = "属性值"]//*[@属性名 = "属性值"]

```
# xpath 属性限制

# a.通过属性精确定位到用户名输入框 //*[@name="userA"]
//input[@name="userA"] admin
driver.find_element_by_xpath('//input[@name="userA"]').send_keys("admin")
```

属性逻辑-定位

限制属性信息进行精确定位,发现,一个属性还是定位不准确,多加几个属性

1. 格式: //元素[@属性名1 = "属性值1" and @属性名2 = "属性值2"]

```
# xpath 属性 逻辑and

# 引号嵌套关系
# a.第一个输入框 test1 中输入
//input[@class="login" and @name="user"] admin
driver.find_element_by_xpath('//input[@class="login" and @name="user"]').send_keys("admin")
```

层级属性-定位

目标元素没有特殊的属性能够限制,但是父级元素有 --- 属性限制先找父级元素--进而定位目标元素(层级和属性的结合使用)

1. 格式: //父级元素[@属性名 = "属性值"]/子级元素

```
# xpath 层级 和属性

# 目标元素没有特殊的属性能够限制,但是父级元素有 --- 属性限制先找父级元素--进而定位目标元素
# a.第一个输入框 test01
//p[@id="p1"]/input[@name="user"]
driver.find_element_by_xpath('//p[@id="p1"]/input[@name="user"]').send_keys("admin")
```

扩展--属性信息限制

1. 双标签的文本信息<标签名>文本信息</标签名> -- text()
2. 包含 -- //*[contains(属性信息, "内容")]
 1. //*[contains(@属性名, "属性值")]
 2. //*[contains(text(), "文本")]
3. 开头--//*[starts-with(属性信息, "内容")]
 1. //*[starts-with(@属性名, "属性值")]
 2. //*[starts-with(text(), "文本")]

```

# xpath 属性信息限制扩展
# 标签文本 开头 结尾
# a. 标签文本是xxx的元素  //*[text()="文本内容"]
# driver.find_element_by_xpath('//a[text()="新浪"]').click()

# b. 属性中含有xxx的元素  //*[contains(@属性名, "属性值")]
driver.find_element_by_xpath('//input[contains(@id, "user")]').send_keys("admin")

# c. 属性开头有xxx的元素  //*[starts-with(@属性名, "属性值")]
driver.find_element_by_xpath('//input[starts-with(@id, "user")]').send_keys("admin")

```

5. css_selector定位略和方法

css选择器的作用就是在页面中定位指定元素设置样式的作用,webdriver中可以借助它定位页面元素 方法: find_element_by_css_selector("选择器字符串")

简单选择器

1. id-- #id属性值
2. class -- .一个类名
3. 元素-- 标签名
4. 属性选择 -- [属性名="属性值"] -- 元素[属性名="属性值"]

```

# 打开'注册A.html'页面
# 2.
# 使用CSS定位方式中id选择器定位用户名输入框, 并输入:
admin #userA
driver.find_element_by_css_selector("#userA").send_keys("admin")

# 3.
# 使用CSS定位方式中属性选择器定位密码输入框, 并输入:
123456 [id="passwordA"] input[id="passwordA"]
driver.find_element_by_css_selector('input[id="passwordA"]').send_keys("123456")

# 4.
# 使用CSS定位方式中class选择器定位电话号码输入框, 并输入:
18600000000 .telA
driver.find_element_by_css_selector(".telA").send_keys("18600000000")

```

```
time.sleep(2)
# 5.
# 使用css定位方式中元素选择器定位注册用户按钮，并点击
button
driver.find_element_by_css_selector("button").click()
```

复合选择器

1. 层级选择器--目标元素是子孙元素
 1. 父子关系 父元素>子元素
 2. 祖孙关系 祖元素 孙元素

```
# css_selector 层级选择器
# 层级 在用户名输入框 admin p[id="p1"]>input
# 父子关系的选择器
driver.find_element_by_css_selector('p[id="pa"]>input').send_keys("admin")

time.sleep(2)
# 祖孙关系的选择器
driver.find_element_by_css_selector('p[id="pa"]input').send_keys("admin")

#
driver.find_element_by_css_selector('p#pa>input').send_keys("admin")
# driver.find_element_by_css_selector('p#painput').send_keys("admin")
```

扩展--属性信息限制

1. 元素[属性名^="属性值"] ---- input[type^='p']
2. 元素[属性名\$="属性值"] ---- input[type\$='d']
3. 元素[属性名*="属性值"] ---- input[type*='w']

五. 定位扩展

1. 定位多元素--find_elements_by_xxx

find_elements_by_xxx()--把页面上定位到的元素都返回,结果通过索引操作

1. 调用--浏览器驱动对象的方法

1. driver.find_elements_by_xxx()
2. 参数--都是定位元素的依据字符串
 1. driver.find_elements_by_xxx("定位元素的依据")
3. 返回--多个元素对象--装在列表中返回--使用索引(从0开始)或者下标来操作
 1. driver.find_elements_by_xxx("定位元素的依据")[index]

```
# find_elements_by_xxx
# 定位页面上的第二个输入框input标签 密码A 123456

# a.定位到页面上所有的input标签
input_list = driver.find_elements_by_tag_name("input")
print(type(input_list))
print(input_list)

# b.从中选出第二个,索引为1的元素对象
target_input = input_list[1]
print(type(target_input))

# c.输入操作
target_input.send_keys("123456")
```

2. 定位方法--find_element

find_elements_by_xxx("定位依据")---find_elements(By.XXX,"定位依据")

1. 调用--浏览器驱动对象的方法
 1. driver.find_element()
2. 参数--定位策略By.XXX, "定位依据内容"
 1. driver.find_element(定位策略By.XXX, "定位依据内容")
3. 返回--返回定位到的元素对象
 1. element = driver.find_element(定位策略By.XXX, "定位依据内容")

```
# find_element 定位
# 使用find_element方法定位用户名输入框,输入admin
# 快速导包操作 -- alt + 回车
# 快速导包操作 -- ctrl + alt + 空格
# driver.find_element(By.ID, "userA").send_keys('admin')
driver.find_element(By.NAME, "userA").send_keys('admin')
# 查看源代码调用的方式 -- 按住ctrl,鼠标左键点击方法名
```

六. 元素操作,浏览器操作方法

1. 元素操作

模拟的用户的点击,输入,清空

方法	作用
<code>element.click()</code>	点击
<code>element.send_keys("输入")</code>	输入--参数是输入内容的字符串
<code>element.clear()</code>	清空 修改=清空+输入

```
# 1.
# 通过脚本执行输入
# 用户名: admin ; 密码: 123456; 电话号码: 18611111111; 电子邮件: 123@qq.com
driver.find_element_by_id("userA").send_keys("admin")
driver.find_element_by_id("passwordA").send_keys("123456")
driver.find_element_by_id("telA").send_keys("18611111111")
driver.find_element_by_name("emailA").send_keys("123@qq.com")

# 2.
# 间隔3秒, 修改电话号码为: 18600000000
time.sleep(3)
# 修改=清空+输入
driver.find_element_by_id("telA").clear()
time.sleep(2)
driver.find_element_by_id("telA").send_keys("18600000000")

# 3.
# 间隔3秒, 点击注册用户A
time.sleep(3)
driver.find_element_by_tag_name("button").click()
```

2. 浏览器操作

模拟的是人为对浏览器的动作

方法	作用
driver.maximize_window()	最大化窗口 一般来说在打开浏览器之后,进行窗口最大化的操作,保证页面内容尽可能的多显示
driver.set_window_size(width,height)	设置窗口的大小 width--像素宽 height--像素高
driver.set_window_position(x,y)	设置窗口的位置 x--距离屏幕左边的像素距离 y--距离屏幕顶部的像素距离
driver.back()	回退--当前窗口打开新页面
driver.forward()	前进--有过回退
driver.refresh()	刷新--当页面元素加载不全时,可以进行刷新操作
driver.close()	关闭当前窗口---一个
driver.quit()	退出浏览器驱动对象 --- 所有
driver.title	窗口标题--页面标题--当前页面的-title标签内容
driver.current_url	页面网址--当前页面的

```
# 浏览器窗口最大化
driver.maximize_window()
# 2秒 设置浏览器窗口大小
time.sleep(2)
driver.set_window_size(width=500,height=400)

# 2秒 设置浏览器窗口位置
time.sleep(2)
driver.set_window_position(x=300,y=300)
```

```
# 在当前窗口打开页面 点击"新浪"超链接
driver.find_element_by_link_text("新浪").click()

# 等待2秒 后退
time.sleep(2)
driver.back()

# 等待2秒 前进
time.sleep(2)
driver.forward()

# 等待2秒 刷新
time.sleep(2)
driver.refresh()
```

```
# close quit 区别 要打开多个窗口 --- 点击一个 在新窗口中打开
的连接 访问 新浪 网站
driver.find_element_by_link_text("访问 新浪 网站").click()

# close -- 关闭当前窗口---一个
time.sleep(2)
driver.close()

# quit -- 退出浏览器驱动对象 --- 所有
# time.sleep(2)
# driver.quit()
```

```
# 获取当前页面title
print("页面标题 title:",driver.title)

# 获取当前页面URL
print("页面url current_url:",driver.current_url)
```

3. 获取元素信息

因为需要对页面内容进行判断--所以要获取到

方法	作用
element.size	元素大小-字典-高宽-像素
element.text	元素文本字符串(双标签中的文本)
element.get_attribute("属性名")	指定属性的值(当指定的属性不存在是,返回的是None)
element.isdisplayed()	判断元素是否可见--布尔值
element.isenabled()	判断元素是否可用--布尔值

```
# 1. 获取用户名输入框的大小
input_element = driver.find_element_by_id("userA")
```

```

print("像素大小:",input_element.size)

# 2.获取页面上第一个超链接的文本内容
link_element = driver.find_element_by_tag_name("a")
print("文本内容:",link_element.text)
# 3.获取页面上第一个超链接的地址 href属性
print("超链接的地址:",link_element.get_attribute("href"))

# 4.判断页面中的span标签是否可见
span_element = driver.find_element_by_tag_name("span")
print("是否可见",span_element.is_displayed()) # 假

# 5.判断页面中取消按钮是否可用
cancel_element = driver.find_element_by_id("cancelA")
print("是否可用",cancel_element.is_enabled()) # 假

# 6.判断页面中"旅游"对应的复选框是否选中的状态
check_element = driver.find_element_by_id("lyA")
print("是否选中",check_element.is_selected()) # 真

```

总结-记忆

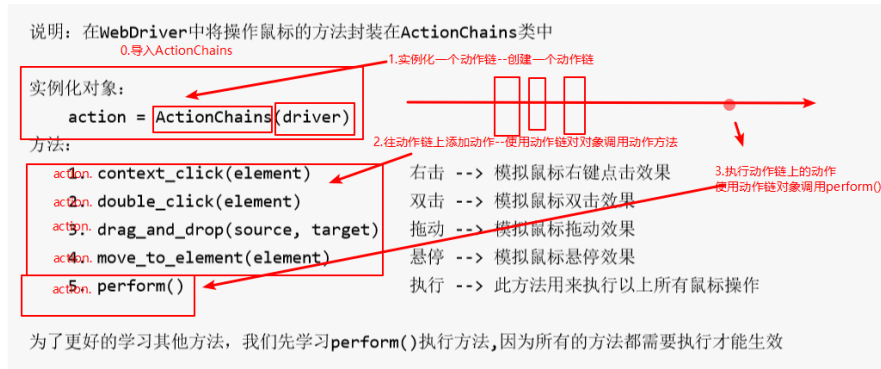
1. 常用的元素操作方法？
 - `element.click()`--元素点击
 - `element.clear()`--清空输入框中的内容
 - `element.send_keys("输入内容")`--模拟输入框输入
一般在输出之前,先清空一下输入框,保证输入内容正确性
2. 常用的操作浏览器方法？
 - `driver.maximize_windows()`--最大化浏览器窗口
 - `driver.refresh()`--模拟刷新操作--页面加载不全时
 - `driver.title`--当前窗口标题--页面中title标签的内容
 - `driver.close()` -- 关闭当前窗口--一个
 - `driver.quit()` -- 退出浏览器驱动对象 -- 关闭所有窗口
3. 常用的获取元素信息的方法？
 - `element.text`---元素文本字符串(双标签内的文本内容)
 - `element.get_attribute("目标属性名")`--对应属性值字符串(目标属性不存在,None)

七. WebDriver鼠标、键盘操作

1. WebDriver鼠标

web应用所支持的鼠标操作越来越丰富--右键单击--左键双击--拖放--悬停

实现流程



流程:

```
# 实例化动作链的对象-- obj = 类名()
# ActionChains类需要导入
# 实例化时需要填入一个参数----浏览器驱动对象

# 使用动作链对象调用动作方法--右键点击--context_click(目标元素对象)
# 调用--动作链对象的方法
# 参数--目标元素对象
# 返回--动作链对象

# 执行动作链动作--动作链对象调用perform()
# 动作链上的动作一定要执行perform才会起效
```

操作	方法
实例化	# 实例化动作链的对象-- ac = ActionChains(driver)
ActionChains类	# ActionChains类需要导入 # 实例化时需要填入一个参数---浏览器驱动对象
右键单击	ac.context_click(目标元素对象)
左键双击	ac.double_click(目标元素对象)
拖放操作	ac.drag_and_drop(source,target) source--拖着哪个元素对象 target--放到哪个元素对象
鼠标悬停	ac.move_to_element(目标元素对象)
执行动作	ac.perform()

```
# 右键点击操作
# 实例化动作链的对象-- obj = 类名()
# ActionChains类需要导入
# 实例化时需要填入一个参数----浏览器驱动对象
action = ActionChains(driver)

# 使用动作链对象调用动作方法--右键点击--context_click(目标元素对象)
# 调用--动作链对象的方法
# 参数--目标元素对象
# 返回--动作链对象
target_element = driver.find_element_by_id("userA")
action.context_click(target_element)

# 执行动作链动作--动作链对象调用perform()
# 动作链上的动作一定要执行perform才会起效
action.perform()

# 连写
#
ActionChains(driver).context_click(driver.find_element_by_id("userA")).perform()
```

```
# 鼠标左键双击
# 先在 用户名输入框中输入 admin
userA_element = driver.find_element_by_id("userA")
userA_element.send_keys("admin")
# 等三秒 在双击 用户名输入框
time.sleep(1)

# 实例化一个动作链对象 ActionChains driver
action = ActionChains(driver)
# 动作链对象调用动作方法--double_click() 目标元素对象
action.double_click(userA_element)
# 执行动作--动作链对象调用perform
action.perform()

# 连写
#
ActionChains(driver).double_click(userA_element).perform()
```

```
# 拖放元素
time.sleep(2)
# 实例化动作链对象 action
act = ActionChains(driver)
```

```

# 使用 action 调用 drag_and_drop(拖着谁,放到哪儿)
source_element = driver.find_element_by_id("div1")
target_element = driver.find_element_by_id("div2")
act.drag_and_drop(source=source_element,target=target_element)

# 执行动作Action调用perform
act.perform()

# 连写
#
ActionChains(driver).drag_and_drop(source=source_element
, target=target_element).perform()

```

```

# 鼠标悬停
# 悬停在注册A按钮上 button
# 实例化一个动作链对象
from selenium.webdriver.common.action_chains import
ActionChains

# from selenium.webdriver import ActionChains
act = ActionChains(driver)
# 动作链对象调用动作方法--move_to_element(目标元素对象)
target_element =
driver.find_element_by_tag_name("button")
act.move_to_element(target_element)
# 动作链对象调用perform()
act.perform()

# 连写
#
ActionChains(driver).move_to_element(driver.find_element
_by_tag_name("button")).perform()

```

总结-记忆

```

# 实例化动作链的对象-- obj = 类名()
# ActionChains类需要导入
# 实例化时需要填入一个参数----浏览器驱动对象

# 使用动作链对象调用动作方法--右键点击--context_click(目标元素对象)

```



```
# 调用--动作链对象的方法
# 参数--目标元素对象
# 返回--动作链对象

# 执行动作链动作--动作链对象调用perform()
# 动作链上的动作一定要执行perform才会起效
```

2. WebDriver键盘

填写表单是经常会用到特殊的键值--所以,需要模拟这些动作

webdriver--Keys--特殊的键值---from selenium.webdriver.common.keys
import Keys

实现

0	send_keys("输入的内容")	输入	元素对象的方法 element.send_keys()
1.	send_keys(Keys.BACK_SPACE)		删除键 (BackSpace)
2.	send_keys(Keys.SPACE)		空格键 (Space)
3.	send_keys(Keys.TAB)	特殊键	制表键 (Tab)
4.	send_keys(Keys.ESCAPE)		回退键 (Esc)
5.	send_keys(Keys.ENTER)		回车键 (Enter)
6.	send_keys(Keys.CONTROL, 'a')		全选 (Ctrl+A)
7.	send_keys(Keys.CONTROL, 'c')		复制 (Ctrl+C)
	send_keys(Keys.CONTROL, 'v')	组合键	

```
# 键盘操作
# 1).输入用户名: admin1, 暂停2秒, 删除1 userA
userA_element = driver.find_element_by_id("userA")
userA_element.send_keys("admin1")
time.sleep(2)
userA_element.send_keys(Keys.BACK_SPACE)
time.sleep(2)

# 2).全选用用户名: admin, 暂停2秒
userA_element.send_keys(Keys.CONTROL, 'a')
time.sleep(2)

# 3).复制用户名: admin, 暂停2秒
userA_element.send_keys(Keys.CONTROL, 'c')
time.sleep(2)
```

```
# 4).粘贴到密码框, 暂停2秒 passwordA
driver.find_element_by_id('passwordA').send_keys(Keys.CONTROL, 'v')
```

总结-记忆

```
1. Keys类的作用
封装的是键盘的特殊键值
from selenium.webdriver.common.keys import Keys
2. 键盘操作调用方法
element.send_keys(输入_特殊键_组合键)
```

八. 元素等待

1. 问题现象(为什么要设置元素的等待):由于某些原因(网络, 客户端, 服务端), 页面加载速度过慢--脚本直接去定位的定位不到---报错--脚本停下来---脚本等到元素出现---元素等待

分类

1. **显式等待--守株待兔**
 1. 指定一个元素定位
 2. 等着这个元素出现
 3. 定位到元素(如果超时定位不到抛出异常)
2. **隐式等待--撒网排查**
 1. 针对页面所有元素定位
 2. 等着目标元素出现
 3. 定位到元素(如果超时定位不到抛出异常)
3. **硬等待--time.sleep(秒数)**

2. 等待流程

1. 第一次定位时如果定位到元素, 则返回, 不触发等待
2. 第一次定位时如果没有定位到元素, 则触发等待:

1. 在等待时间(timeout)内,进行多次定位(每次定位间隔指定时长)
2. 如果在等待时间内,成功定位到,则返回元素
3. 如果在等待时间内,没有定位到元素,则抛出异常

3. 隐式等待

概念

0. 针对页面中所有的元素定位

1. 第一次定位时如果定位到元素,则返回,不触发等待
2. 第一次定位时如果没有定位到元素,则触发等待:
 1. 在等待时间(timeout)内,进行多次定位(每次定位间隔指定时长)
 2. 如果在等待时间内,成功定位到,则返回元素
 3. 如果在等待时间内,没有定位到元素,则抛出异常 `NoSuchElementException`

实现步骤

1. 一般只用设置一次--针对全局
2. `driver.implicitly_wait(等待时间秒)`

```
# 隐式等待 -- 设置一次 针对全局--所有元素定位
# 调用-使用浏览器驱动对象调用
# 参数-等待时长秒
# 能定到 --- 异常
# selenium.common.exceptions.NoSuchElementException
driver.implicitly_wait(10)
# driver.find_element_by_id("userA").send_keys("admin")

# 失败情况
# try:
#     print("start time:", time.time())
#
driver.find_element_by_id("userAd").send_keys("admin")
#
# except Exception as e:
#     print("end time:", time.time())
#     print(type(e))
```

3. 显式等待

概念

0.指定一个元素定位

1. 第一次定位时如果定位到元素,则返回,不触发等待
2. 第一次定位时如果没有定位到元素,则触发等待:
 1. 在等待时间(timeout)内,进行多次定位(每次定位间隔指定时长)
 2. 如果在等待时间内,成功定位到,则返回元素
 3. 如果在等待时间内,没有定位到元素,则抛出异常 `TimeoutException`

实现步骤(两步)

0.导入WebDriverWait

1.实例化一个等待类WebDriverWait对象

`wait`
`driver`
`timeout--等待时长--秒`
`pf--秒--定位间隔时间`

2.WebDriverWait对象调用until()

`调用--wait.until(method)`
`参数--定位方法 lambda d:d.find_element_by_xxx()`
`返回--定位到的元素对象`



1. `wait = WebDriverWait(driver,10,1)`
2. `userA = wait.until(lambda d:d.find_element_by_id("userA"))`
3. `userA.send_keys("操作")`

```
# 显式等待的实现 -- 真的能订到 -- 定位失败是的抛出异常
TimeoutException
# selenium.common.exceptions.TimeoutException
# 用户名
# 实例化一个WebDriverWait类的对象
wait = WebDriverWait(driver, timeout=10,
poll_frequency=1)

# WebDriverWait类的对象调用until
# 定位方法不会自动提示,需要手写,别写错
# 元素操作方法不会自动提示,需要手写,别写错
userA_element = wait.until(lambda d:
d.find_element_by_id("userA"))
userA_element.send_keys("admin")

# 失败情况
# try:
#     print("start time:", time.time())
#     element = WebDriverWait(driver, 10,
1).until(lambda d: d.find_element_by_id("userAa"))
```

```
# element.send_keys("admin")
# except Exception as e:
#     print("end time:", time.time())
#     print(type(e))
```

显式等待和隐式等待的区别(不能共存)

1. 作用域

1. 显示等待--指定一个元素定位
2. 隐式等待--页面中所有元素定位

2. 抛出的异常不同

1. 显示等待--TimeoutException
2. 隐式等待--NoSuchElementException

3. 使用方式不同

1. 显示等待--WebDriverWait(两个--实例化--调用until)
2. 隐式等待--driver.implicitly_wait(等待时间秒)

总结--记忆

1. 为什么要设置元素等待

由于某些原因(网络, 客户端, 服务端), 页面加载速度过慢--脚本直接去定位的定位不到---报错--脚本停下来---脚本等到元素出现---元素等待

2. 元素等待的过程

1. 第一次定位时如果定位到元素, 则返回, 不触发等待
2. 第一次定位时如果没有定位到元素, 则触发等待:
 1. 在等待时间(timeout)内, 进行多次定位(每次定位间隔指定时长)
 2. 如果在等待时间内, 成功定位到, 则返回元素
 3. 如果在等待时间内, 没有定位到元素, 则抛出异常

3. 元素等待的类型

显式等待--守株待兔

隐式等待--撒网排查

4. 显式等待与隐式等待区别

1. 作用域
 1. 显示等待--指定一个元素定位
 2. 隐式等待--页面中所有元素定位
2. 抛出的异常不同
 1. 显示等待--TimeoutException
 2. 隐式等待--NoSuchElementException
3. 使用方式不同

1. 显示等待--webdriverwait(两个--实例化--调用until)
2. 隐式等待--driver.implicitly_wait(等待时间秒)

九. 下拉选择框、弹出框、滚动条操作

1. 下拉框操作

下拉框----select -- option

人级操作(问题现象):使用value属性值定位,比较单一,如果选项操作过多的话需要多次定位点击比较繁琐--如何简化操作

```
# 定位 点击操作
# 广州 option option[value="gz"]
time.sleep(2)
driver.find_element_by_css_selector('option[value="gz"]')
.click()

# 上海 option option[value="sh"]
time.sleep(2)
driver.find_element_by_css_selector('option[value="sh"]')
.click()

# 北京 option option[value="bj"]
time.sleep(2)
driver.find_element_by_css_selector('option[value="bj"]')
.click()
```

神级操作(处理):

实例化对象:

```
select = Select(element)
```

element: <select>标签对应的元素, 通过元素定位方式获取,
例如: driver.find_element_by_id("selectA")

操作方法:

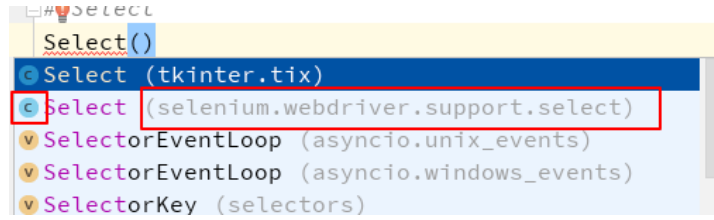
select.select_by_index(index)	--> 根据option索引来定位, 从0开始
select.select_by_value(value)	--> 根据option属性 value值来定位
select.select_by_visible_text(text)	--> 根据option显示文本来定位

1.实例化一个Select类对象--确定目标select标签
导包
参数--目标select标签元素对象

2.Select类对象选择选项--调用选择方法
index
value值
选项的可见文本内容

1. Select

2. 1.实例化一个Select类对象--确定目标select标签 导包



参数--目标select标签元素对象

2.Select类对象选择选项--调用选择方法 index value值 选项的可见文本内容

```
# Select 操作下拉框
# 1.实例化一个Select类对象--确定目标select标签
# 导包
# 参数--目标select标签元素对象
#
# 2.Select类对象选择选项--调用选择方法
# index
# value值
# 选项的可见文本内容

# Select
select_element= driver.find_element_by_id("selectA")
slt = Select(select_element)

# 广州 index 索引值--从0开始 2
slt.select_by_index(2)
time.sleep(2)
# 上海 value option标签的value属性值 sh
slt.select_by_value("sh")
time.sleep(2)

# 北京 文本 option标签的文本内容 A北京
slt.select_by_visible_text('A北京')
```

2. 弹出框处理

1. 弹出框是--浏览器的
2. alert--警告框 -- 文本 --确定按钮
3. confirm--确认框-- 文本 -- 确认按钮--取消按钮
4. prompt--提示框--文本--输入框--确认--取消

问题现象:当页面中意外弹出弹出框--想要关闭--无法定位; 如果不关闭弹出框页面元素也无法定位操作

处理:

三种弹出框封装在了一个alert中



```
# 弹出框处理
# 需求: 打开注册A.html页面, 完成以下操作:
# 1.点击 alert 按钮 alerta
time.sleep(2)
driver.find_element_by_id("alerta").click()
#
# 2.关闭警告框 -- 无法处理 不能定位 也无法查看 - 先不管
# 获取到警告框对象 -- 切换到警告框对象
time.sleep(2)
alt = driver.switch_to.alert
# alt = driver.switch_to_alert()

# 处理,调用方法 text accept() dismiss()
print("打印弹框文本:",alt.text)
# alt.accept()
alt.dismiss()
#
# 3.输入用户名: admin 成功
time.sleep(2)
driver.find_element_by_id("userA").send_keys("admin")
```

3. 滚动条操作

控制页面显示范围的控制件

问题现象:页面元素,受前端技术的影响,默写元素的加载和显示的状态
受滚动条动作的影响--模拟用户操作滚动条的动作--借助js语句实现

```
// window.scrollTo(0, 200)
// 第一个参数--左右滚动--水平滚动--0不滚动 --越大滚动越多
// 第二个参数--上下滚动--垂直滚动--0不滚动 --越大滚动越多
```


处理:

- 写js语句
1. 设置JavaScript脚本控制滚动条 `js="window.scrollTo(0,1000)"`
(0:左边距; 1000: 上边距; 单位像素)
 2. WebDriver调用js脚本方法 `driver.execute_script(js)` ← 脚本语句字符串

```
# 操作滚动条
time.sleep(2)
# a.准备js语句
js = "window.scrollTo(0,100000)"

# b.执行js语句
driver.execute_script(js)

# c.暂停两秒,滚动到最顶部
time.sleep(2)
driver.execute_script("window.scrollTo(0,0)")
```

总结--记忆

通过select类如何选择下拉框?

1. 实例化一个Select类对象
导包
参数--目标select标签元素对象
2. Select类对象调用选择方法
index
option标签的value属性值
option标签的可见文本

如何处理弹出框(无法定位)?

1. 获取到弹出框对象--切换到弹出框
`alt = driver.switch_to.alert`
2. 处理
`alt.text`--文本内容
`alt.accept()`--接受确认
`alt.dismiss()`--忽略取消

JavaScript控制滚动条语句

1. `js = "window.scrollTo(水平偏移量,垂直偏移量)"`
2. `driver.execute_script("脚本语句字符串")`

十. frame切换、多窗口切换

1. frame切换--iframe

frame--前端页面框架--在一个页面(主页)中嵌入另一完整的页面(子页)--都是完整的html页面--相互不影响

1. 表单提交页
2. 后台管理页面
3. 页面广告

问题现象:当你要操作的内容在子页中的时候--不能直接操作--需要先切换到子页中--进行子页业务操作---记得切回主页面

处理:

方法:

```
1). driver.switch_to.frame(frame_reference)  --> 切换frame框架方法
    frame_reference: 可以为frame框架的name、id或者定位到的frame元素
2). driver.switch_to.default_content()  --> 恢复默认页面方法
```

在frame中操作其他页面,必须先回到默认页面,才能进一步操作

主页--子页
子页中的业务操作
子页--主页
切换到子页中处理完后记得切回主页

1. 主页--子页--driver.switch_to.frame(iframe_id_name_标签元素对象)
2. 子页中的业务操作
3. 子页--主页--driver.switch_to.default_content()
4. 在子页中处理完业务之后记得切回主页(保证每次子页切换时起点都在主页中)

```
# frame切换
# 案例: 打开'注册实例.html'页面, 完成以下操作
# 用户名  admin
# 1. 填写主页面的注册信息
driver.find_element_by_id("user").send_keys("admin")
time.sleep(2)

# A子页 -- A子页iframe 标签的信息
# a.主页--子页 --iframe -- id
driver.switch_to.frame("iframe1")

# b.子页操作    A子页 --- 输入操作
# 2. 填写注册页面A中的注册信息
driver.find_element_by_id("userA").send_keys("admin")
time.sleep(2)

# c.子页--主页
driver.switch_to.default_content()
```

```

# B子页 中输入内容
# a.主页--子页 --iframe -- name
driver.switch_to.frame('myframe2')

# b.子页操作    B子页中的业务操作
# # 3. 填写注册页面B中的注册信息
driver.find_element_by_id("userB").send_keys("admin")
time.sleep(2)

# c.子页--主页
driver.switch_to.default_content()

```

2. window切换--handle

由于一些超链接在点击后会在新窗口中打开

问题现象:如果需要操作新窗口中的内容,需要手动的切换driver所指向的窗口(webdriver不会自动切换窗口)

处理:

说明: 在WebDriver中封装了获取当前窗口句柄方法和获取所有窗口句柄的方法以及切换指定句柄窗口的方法:

法: 句柄 英文handle, 窗口的唯一识别码) --字符串格式的内容

方法:

- 1). driver.current_window_handle --> 获取当前窗口句柄
- 2). driver.window_handles --> 获取所有窗口句柄
- 3). driver.switch_to.window(handle) --> 切换指定句柄窗口

1. 确定有哪些窗口(句柄)

2. 确定目标窗口(当前窗口)

3. 切换窗口

4. 查看当前窗口

1. 确定有哪些窗口(句柄)

1. driver.window_handles
2. 所有句柄的列表--索引

2. 确定目标窗口(当前窗口)

1. 从所有句柄的列表--取出目标窗口--索引

3. 切换窗口

1. driver.switch_to.window(一个目标窗口句柄)

4. 查看当前窗口

1. driver.current_window_handle

2. 当前一个窗口句柄

```
# 切换窗口
# a.所有窗口
handles_list = driver.window_handles
# print(type(handles_list))
print("所有窗口:", handles_list)

# b.目标窗口
# 没有切换 -- 知道当前的句柄是什么 ---- 不是当前的---就是目标
print("没有切换_当前窗口", driver.current_window_handle)
target_handle = handles_list[-1]

# c.切换窗口
driver.switch_to.window(target_handle)

# d.当前窗口
print("有切换后_当前窗口", driver.current_window_handle)
```

总结--记忆

1. HTML中常用的frame框架
 iframe
 2. 切换框架的方法 主页--子页
 driver.switch_to.frame(iframe_id_name_元素对象)
 3. 恢复到默认页面的方法 子页--主页
 driver.switch_to.default_content()
-
1. 什么是句柄?
 webdriver中对窗口的唯一标识号--字符串--窗口id
 2. 获取当前窗口句柄方法
 driver.current_window_handle
 3. 获取所有窗口句柄方法
 driver.window_handles
 注意--一般来说driver首次打开的窗口句柄一般都是索引为0
 4. 切换指定句柄窗口方法
 driver.switch_to.window(目标窗口句柄)

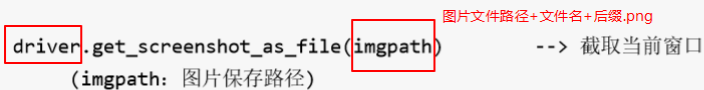
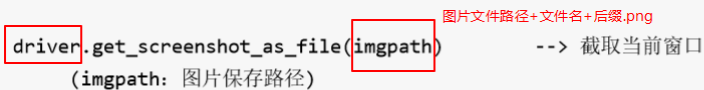
十一. 窗口截图、验证码处理

1. 窗口截图

对浏览器界面的截图

目的:更加生动形象表达用例执行的过程和结果---通过截图

实现:

```
driver.get_screenshot_as_file(imgpath)  --> 截取当前窗口


```

1. 调用--浏览器驱动对象调用
2. 参数--截图存放位置和文件名
3. 返回--无

```
# 打开注册A页面 --用户名中输入 admin
driver.find_element_by_id("userA").send_keys("admin")
driver.find_element_by_id("passwordA").send_keys("123456")
# 截图保存
time.sleep(1)
# driver.get_screenshot_as_file("./a.png")

# pic_name = "./a.png"
# 图片名字不同 -- 在文件名字中加入时间字符串 -- 年月日时分秒
"20190216154823"
# 何如把时间字符串放入文件命中 -- 字符串格式化 "a%s.png" %
"20190216154823"
# pic_name = "./a%s.png" % "20190216154823"

# format()    "./a{}.png".format("放入的内容")
# 调用 -- 目标字符串
# 参数 -- 要放入目标字符串的内容
# 返回--没有

# 自动获取时间字符串的方法
# time.strftime("%Y%m%d%H%M%S")
```

```
pic_name = "./a%s.png" % time.strftime("%Y%m%d%H%M%S")
driver.get_screenshot_as_file(pic_name)
```

2. 验证码处理

web应用中有验证码,影响脚本的执行

处理方式

1. 可以修改项目代码时

1. 去掉验证码--测试
2. 设置万能验证码--测试--生产--(有规律--权限验证)

2. 如果不能修改

1. 验证码识别技术--识别率不高--理论
2. 通过记录用户登录信息--越过登录--越过验证码

3. cookie介绍

服务器保存在浏览器客户端上的一个小文件--用户的身份信息--一般是加密--格式是键值对



唐僧告诉皇帝取经的请求--客户端发送请求

皇帝给唐僧办法通关文牒--服务端保存cookie到客户端

唐僧带着通关文牒请求沿途国家--客户端请求时携带cookie

沿途国家识别唐僧身份提供服务--服务端识别返回响应

webdriver操作cookie

```
1. get_cookie(name) --> 获取指定cookie
   name: 为cookie的名称 参数--cookie名称字符串 返回--对应的cookie值
   driver.xxx()
   1. 获取cookie
   2. 获取指定cookie
   3. 获取所有cookie

2. get_cookies() 没有参数 返回--所有cookie --> 获取本网站所有本地cookies
   2. 设置cookie
   3. 添加一个cookie

3. add_cookie(cookie_dict) --> 添加cookie
   cookie_dict: 一个字典对象, 必选的键包括: "name" and "value"
   {"name": "cookie名字", "value": "cookie值"}
```

1. 获取cookie

1. driver.get_cookie("cookie名字")
 1. 调用--浏览器驱动对象调用
 2. 参数--cookie名字字符串
 3. 返回--返回对应cookie字典
2. driver.get_cookies()
 1. 调用--浏览器驱动对象调用
 2. 参数--无
 3. 返回--所有cookie字典列表

2. 添加cookie

1. driver.add_cookie({ "name": "cookie名字", "value": "cookie值" })
 1. 调用--浏览器驱动对象调用
 2. 参数--{ "name": "cookie名字", "value": "cookie值" }
 3. 返回--无

```
# 获取所有cookie
cookies_list = driver.get_cookies()
print("cookies: ", cookies_list)

# 获取单个cookie
cookie = driver.get_cookie("BAIDUID")
print("cookie: ", cookie)

# 添加cookie
driver.add_cookie({"name": "test", "value": "test"})
# 添加后查看对应cookie信息
cookie = driver.get_cookie("test")
print("cookie_test: ", cookie)
```

```
# 1. 手动登陆baidu, 登陆的时候抓取 (BDUSS)
cookie_dict =
{"name": "BDUSS", "value": "1BvV1lQdx42awtCQm82fkRyZ01ib0xT
VzktbEJmZy1Xdm5CTHg3RVNZdTBMZlZjsUFBQUFBjCQAAAAAAAAAAAAEA
AAAiOKTqdGVzdG10Y2FzdAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAALsgzVy0oM1cY"}

# 2. 使用add_cookie()方法, 添加 (BDUSS)键和值
driver.add_cookie(cookie_dict)

# 3. 调用刷新方法 driver.refresh()
time.sleep(2)
driver.refresh()
```

总结-记忆

1. 截屏方法

```
driver.get_screenshot_as_file("图片路径文件名后缀")
# pic_name = "./a.png"
# 图片名字不同 -- 在文件名字中加入时间字符串 -- 年月日时分
秒 "20190216154823"
# 何如把时间字符串放入文件命中 -- 字符串格式化 "a%s.png" %
"20190216154823"
# pic_name = "./a%s.png" % "20190216154823"
# 自动获取时间字符串的方法
# time.strftime("%Y%m%d%H%M%S")
```

2. 验证码常用的处理方式

可以修改代码

去掉验证码

设置万能验证码--有规律--需要权限验证

不能修改项目代码

验证码识别

越过登录--记录cookie

3. Cookie的作用

服务器保存在浏览器上的一个小文件--用户的身份信息--加密的--
键值对

十二. UnitTest框架

目的	实现	增强
编写用例	定义测试类和测试方法 (TestCase)	封装公共操作(fixture) 自动判断执行结果(断言) 用例和数据分离(参数化)
组织用例	组织测试用例(TestSuite)	加载大量测试用例(TestLoader) 用例执行跳过(skip)
执行用例	运行测试用例(TextTestRunner)	运行时产生报告 (HTMLTestRunner)

1. UnitTest

- 1. 是一个专门用来执行代码测试的功能的集合(框架)
- 2. python语言中的单元测试框架
- 3. 作用
 - 1. 编写组织运行用例
 - 2. 断言用例执行结果(自动判断实际是否符合预期)
 - 3. 生成测试报告
- 4. 核心要素

1. TestCase	测试用例
2. TestSuite	测试套件 -- 组织测试用例
3. TestRunner	运行器 -- 运行测试套件中测试用例
4. TestLoader	测试加载器 -- 大量加载指定目录指定模块用例
5. Fixture	封装非业务公共操作--用例执行的前置条件和后置处理

2. UnitTest核心要素

1). TestCase测试用例方法

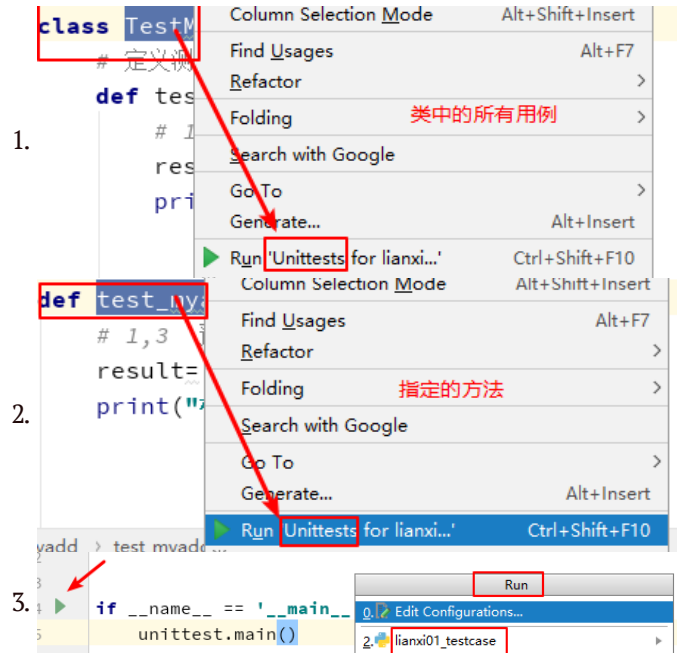
- 1. 测试用例
- 2. 定义:

测试类	class 类名(父类):	class 测试类名(unittest.TestCase):
	pass	测试方法
1. 在测试类定义测试方法	def 方法名(self):	def testxxxx(self):
	被测函数	被测业务

- 2. 定义测试类是一定要继承unittest.TestCase

3. 定义测试方法方法名一定要以test开头(unittest认为test开头的方法就是测试用例)

3. 运行

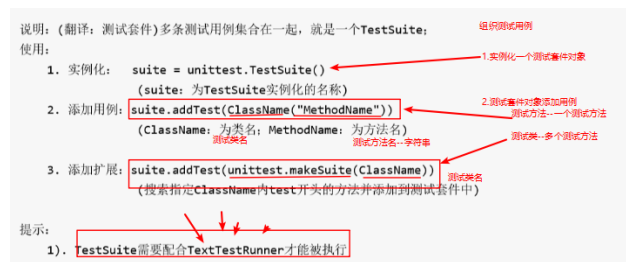


2). TestSuite测试套件

1. 组织测试用例

2. 定义:

1.



2. 实例化测试套件

3. 向测试套件中添加用例

1. 一次添加一个方法

1. 测试类名("测试方法名")

2. 一次添加一个类

1. `unittest.makeSuite`(测试类名)

3). TextTestRunner测试运行器

1. 运行测试套件中的测试用例

2. 定义:

1.

```
1. 实例化: runner = unittest.TextTestRunner() 实例化TextTestRunner对象
           (runner: TextTestRunner实例化名称)
2. 执行: runner.run(suite) runner对象调用run(测试套件)
           (suite: 为测试套件名称)
```

2. 实例化一个TextTestRunner对象

1. runner=unittest.TextTestRunner()

2.

```
unittest.runner
unittest.runner
unittest.runner
# 运行测试 Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor
unittest.TextTestRunner
```

3. TextTestRunner对象调用run(测试套件)

runner.run(suite)

4). TestLoader()测试加载器(扩展)

1. 因为之前学习加载测试用例的方法效率太低--如果需要添加多个文件中的测试用例是,需要多次添加--想要一次性加载指定目录中的指定文件的测试用例

2.

说明: 一次加载一个类 一次加载一个方法 一个目录中又很多测试文件--所有用例都添加到测试套件中 ----一次加载一个类--多次添加

用来加载TestCases到TestSuite中,即加载满足条件的测试用例,并把测试用例封装成测试套件。

使用unittest.TestLoader,通过该类下面的discover()方法自动搜索指定目录下指定开头的.py文件,并将查找到的测试用例组装到测试套件;

实例化加载器对象TestLoader()
loader = unittest.TestLoader()
加载器对象调用discover()
参数--指定的目录,指定文件名
返回--添加好测试用例的套件对象
loader.discover("用例文件目录",pattern="test*.py")

用法:

```
suite = unittest.TestLoader().discover(test_dir, pattern='test*.py')
自动搜索指定目录下指定开头的.py文件,并将查找到的测试用例组装到测试套件
test_dir: 为指定的测试用例的目录
pattern: 为查找的.py文件的格式,默认为'test*.py'
```

也可以使用unittest.defaultTestLoader 代替 unittest.TestLoader() 已经实例化好的加载器对象

运行:

```
runner = unittest.TextTestRunner()
runner.run(suite)
```

3. 实例化加载器对象TestLoader()

1. loader = unittest.TestLoader()

2. unittest.defaultTestLoader

4. 加载器对象调用discover()

1. 参数--指定的目录,指定文件名

返回--添加好测试用例的套件对象

loader.discover("用例文件目录",pattern="test*.py")

总结--记忆

1. unittest框架作用

专门用来执行代码测试的工具的集合
python中的单元测试框架

2. 要使用unittest框架创建测试用例必须继承?

创建测试类--一定要继承unittest.TestCase
在测试类中创建测试方法--就是测试用例--方法名必须以test开头

3. TestSuite作用

1. 实例化一个测试套件对象

```
suite = unittest.TestSuite()
```

2. 测试套件对象添加测试用例

a. 添加单个测试方法

```
suite.addTest(测试类名("测试方法名"))
```

b. 添加整个测试类

```
suite.addTest(unittest.makeSuite(测试类名))
```

4. TestRunner(运行TestSuite)

运行器TextTestRunner

1. 实例化一个运行器对象

```
runner = unittest.TextTestRunner()
```

2. 运行器对象调用run(测试套件)

```
runner.run(测试套件对象)
```

5. TestLoader与TestSuite的区别

1. 实例化加载器对象

```
loader = unittest.TestLoader()
```

2. 调用discover方法

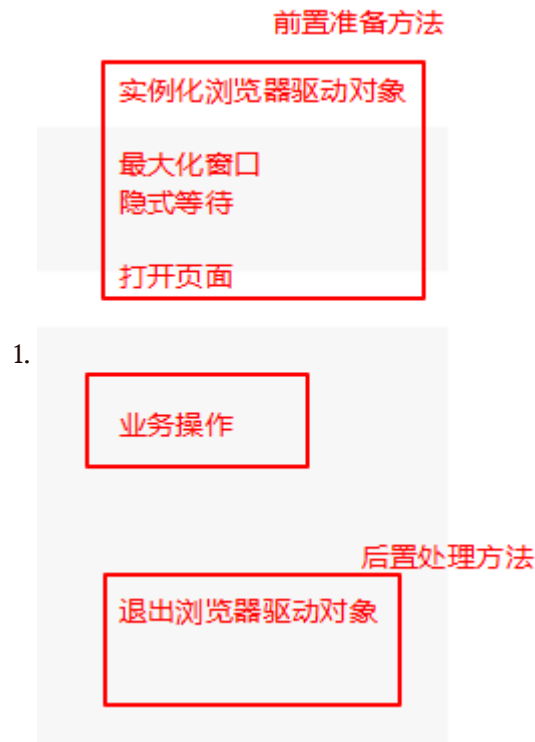
```
suite = loader.discover(start_dir, pattern)
start_dir--目标目录
pattern--test开头, .py结尾的文件--*通配
返回--添加好测试用例的测试套件
```

TestSuite--用例数比较少--能够特殊指定

TestLoader--用例数比较多--无法特殊指定

5). Fixture

1. 为什么:我们的用例中有一些非业务的公共操作--封装起来-减少重复代码



2. 一个概念:在测试用例执行前后所运行的代码固定内容,在unittest中提供多个控制级别的fixture

1. 方法级别

```
1. 初始化(前置处理):
    def setUp(self)          --> 首先自动执行
2. 销毁(后置处理):
    def tearDown(self)       --> 最后自动执行
3. 运行于测试方法的始末, 即: 运行一次测试方法就会运行一次setUp和tearDown
```

每个用例执行前后执行
定义在测试类内
实例方法

2. 类级别

```
1. 初始化(前置处理):
    @classmethod
    def setUpClass(cls):      --> 首先自动执行
2. 销毁(后置处理):
    @classmethod
    def tearDownClass(cls):   --> 最后自动执行
3. 运行于测试类的始末, 即: 每个测试类只会运行一次setUpClass和tearDownClass
```

每个测试类前后执行
定义在测试类内
类方法

3. 模块级别

```
1. 初始化(前置处理):
    def setUpModule()         --> 首先自动执行
2. 销毁(后置处理):
    def tearDownModule()      --> 最后自动执行
3. 运行于整个模块的始末, 即: 整个模块只会运行一次setUpModule和tearDownModule
```

每个测试文件(模块)前后执行
定义在测试文件(模块)内
全局方法

3. 三个级别fixture的执行顺序

1. 模块级别setUpModule()
2. 类级别setUpClass(cls)
3. 方法级别setUp(self)
4. 测试用例
5. 方法级别tearDown(self)
6. 类级别tearDownClass(cls)
7. 模块级别tearDownModule()

```
# 重写父类的setUp方法--方法级别Fixture -- 测试方法-执行前自动执行
def setUp(self):
    print("method start    time:", time.time())

# 重写父类的tearDown方法--方法级别Fixture -- 测试方法-执行后自动执行
def tearDown(self):
    print("method end      time:", time.time())
```

```
# 重写父类的setUpClass方法--类级别Fixture -- 测试类中所有测试方法-执行前自动执行
@classmethod
def setUpClass(cls):
    print("class start    time:", time.time())

# 重写父类的tearDownClass方法--类级别Fixture -- 测试类中所有测试方法-执行后自动执行
@classmethod
def tearDownClass(cls):
    print("class end      time:", time.time())
```

```
# 定义setUpModule方法--模块级别Fixture -- 模块中所有测试方法-执行前自动执行
def setUpModule():
    print("module start    time:", time.time())

# 定义tearDownModule方法--模块级别Fixture -- 模块中所有测试方法-执行后自动执行
def tearDownModule():
    print("module end      time:", time.time())
```

```
class TestLogin(unittest.TestCase):
    def setUp(self):
        # 实例化浏览器驱动对象
        self.driver = webdriver.Chrome()
        self.driver.maximize_window()
        self.driver.implicitly_wait(30)

        # 打开页面
        self.driver.get("http://localhost")

    def tearDown(self):
        # 退出浏览器驱动对象
        time.sleep(2)
        self.driver.quit()
```

```

# 定义测试方法 --- 方法名必须是test开头
def test_login(self):
    # 业务操作
    # a. 首页点击登录--进入登录页
    self.driver.find_element_by_link_text("登
录").click()
    # b. 输入用户名, 密码

    self.driver.find_element_by_id("username").send_keys("1
3012345678")

    self.driver.find_element_by_id("password").send_keys("1
23456")
    # c. 点击登录按钮
    self.driver.find_element_by_css_selector("[name='sbtbutton']").click()
    # d. 获取错误信息
    msg =
self.driver.find_element_by_css_selector(".layui-layer-
content").text
    print("msg:", msg)

```

总结--记忆

1. 什么是Fixture?
在测试用例执行前后所运行的代码固定内容, 在unittest中提供多个控制级别的fixture
2. Fixture控制级别有哪些?
方法级别--类级别--模块级别
3. 如何定义Fixture?
方法级别 -- setUp--tearDown
类级别 -- setUpClass--tearDownClass
模块级别 -- setUpModule--tearDownModule

3. UnitTest断言

让程序代替人为判断用例执行的结果是否符合预期的过程(符合预期,不符合预期)

常用的断言--符合预期断言成功--不符合预期断言失败

1	<code>assertTrue(expr, msg=None)</code> <small>判断真假</small>	验证expr是true, 如果为false, 则fail
2	<code>assertFalse(expr, msg=None)</code>	验证expr是false, 如果为true, 则fail
3	<code>assertEqual(expected, actual, msg=None)</code> <small>判断是否相等</small>	验证expected==actual, 不等则fail 【掌握】
4	<code>assertNotEqual(first, second, msg=None)</code>	验证first != second, 相等则fail
5	<code>assertIsNone(obj, msg=None)</code> <small>判断是否为空</small>	验证obj是None, 不是则fail
6	<code>assertIsNotNone(obj, msg=None)</code>	验证obj不是None, 是则fail
7	<code>assertIn(member, container, msg=None)</code> <small>判断是否包含</small>	验证是否member in container 【掌握】
8	<code>assertNotIn(member, container, msg=None)</code>	验证是否member not in container

unittest判断用例执行结果是否通过的依据:断言异常

1. 如果没有抛出断言异常(通过)
2. 如果抛出断言异常(失败)

```
# 断言是否相等
self.assertEqual(4, result)

# 断言是否为真
# self.assertTrue(True)

# 断言是否包含
# self.assertIn("hello", "hello world")
# self.assertIn(1, [5, 1, 3, 2, 4])
```

```
# 断言是否提示信息是否包含 "验证码不能为空"
# 断言失败后保存截图,继续抛出异常
try:
    # 断言操作
    self.assertIn("验证码不能为空123123213123",
msg)

    # 断言成功
except Exception as e:
    # 断言失败
    img_path =
"./img{}.png".format(time.strftime("%Y%m%d%H%M%S"))
    self.driver.get_screenshot_as_file(img_path)
    # 通过异常判断用例执行的结果,所以要抛出异常
    raise e
```


总结--记忆

1. 什么是断言?
让程序代替人为判断测试用例执行的结果是否符合预期的过程
2. 需要掌握哪个断言?
`self.assertIn("helloasdf", 'hello world')`
`self.assertEqual(4, result)`
3. 断言异常类
`AssertionError`
unittest 判断用例执行结果依据是 断言异常 `AssertionError`

4. 参数化

概念:通过参数的方式来传递数据，从而实现数据和脚本分离

(相同的用例业务 -- 不同数据数量比较大--不断的修改用例数据---用例维护很难)

参数化实现步骤

1. 如何能数据跟业务拆分开
 1. 把用例中的数据都用参数形式来表
 2. 写在用例方法参数列表中括号里
2. 怎么把数据放入参数--工具--parameterized

1. `@parameterized.expand(datas)`
`parameterized C:\Users\iambtree\AppData\Local`
`parameterized (parameterized.parameterized)`

```

# 1.导入parameterized---from parameterized
import parameterized
# 2.@parameterized.expand([[参数值列表],[],[],[],
[]])
# 放在你需要参数化的用例方法前--紧挨着
# 3.数据跟用例参数——位置对应的

# 其他parameterized使用方式
# data_list = [[参数值列表],[],[],[],[]]
# @parameterized.expand(data_list)

# def data_list:
#     return [[参数值列表],[],[],[],[]]
# @parameterized.expand(data_list())导包注意

```

```

# 第一种 -- 直接数据写入参数化方法
# @parameterized.expand([(1, 1, 2), (1, 0, 1), (0,
0, 0)])
# def test_myadd_param01(self, x, y, expect):
#     print("x={}, y={}, expect={}".format(x, y,
expect))
#     result = myadd(x, y)
#     self.assertEqual(expect, result)

# 第二种 -- 通过变量保存测试数据
# test_data = [(1, 1, 2), (1, 0, 1), (0, 0, 0)]
#
# @parameterized.expand(test_data)
# def test_myadd_param02(self, x, y, expect):
#     print("x={}, y={}, expect={}".format(x, y,
expect))
#     result = myadd(x, y)
#     self.assertEqual(expect, result)

# 第三种 -- 通过方法返回测试数据--定义返回数据的方法
@parameterized.expand(return_test_data)
def test_myadd_param02(self, x, y, expect):
    print("x={}, y={}, expect={}".format(x, y,
expect))
    result = myadd(x, y)
    self.assertEqual(expect, result)

```

5. 跳过

概念:对于不满足测试条件的测试方法和测试类, 跳过执行

(根据实际情况确定那些测试方法或者测试类不执行)

跳过实现方法

1. @unittest.skip("跳过原因提示")
2. @unittest.skipIf(判断条件,"跳过原因")

```
# 跳过测试类执行
@unittest.skip("代码未完成")
class TestSkip(unittest.TestCase):

    # 跳过测试方法执行
    @unittest.skip("代码未完成")
    def test01(self):
        print("该测试方法未完成...")

    # 在一定条件下跳过测试方法执行
    @unittest.skipIf(version <= 30, "版本号大于30才需要执行")
    def test02(self):
        print("版本号大于30才需要执行...")

    def test03(self):
        print("test03")
```

6. HTMLTestRunner--HTML报告

```
1. 复制HTMLTestRunner.py文件到项目文件夹
2. 导入HTMLTestRunner、unittest包
3. 生成测试套件
    suite = unittest.TestSuite()
    suite.addTest(TestLogin("test_login"))
4. 设置报告生成路径和文件名
    file_name = file_dir+nowtime+"Report.html"
5. 打开报告 with open(file_name, 'wb') as f:
6. 实例化HTMLTestRunner对象: runner = HTMLTestRunner(stream=f,[title],[description])
    参数说明:
    stream: 文件流, 打开写入报告的名称及写入编码格式)
    title: [可选参数], 为报告标题, 如XXX自动化测试报告
    description: [可选参数], 为报告描述信息; 比如操作系统、浏览器等版本
7. 执行: runner.run(suite)
```

1. 实例化一个测试套件 suite
2. 向测试套件suite添加测试用例
3. 实例化一个运行器对象 runner = HTMLTestRunner(stream, title, description)
4. 运行器对象调用run(测试套件) runner.run(suite)
第三方的工具--文件放入项目目录--导包--类
必须的 stream 测试报告的文件流
非必须 title -- 测试标题
description--测试描述

1. 实例化一个测试套件

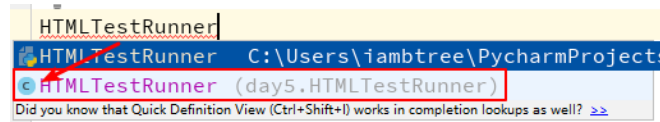
```
1. suite = unittest.TestSuite()
```

2. 向测试套件suite添加测试用例

1. suite.addtest(unittest.makeSuite(测试类))
2. suite.addtest(测试类("测试方法"))

3. 实例化一个运行器对象

1.



2. runner = HTMLTestRunner(stream title,description)

1. stream -- 报告文件流--必填
2. title -- 报告标题--非必填
3. description -- 报告描述--非必填

4. 运行器对象调用run(测试套件)

1. runner.run(suite)

```
# 1.实例化一个测试套件 suite
import unittest

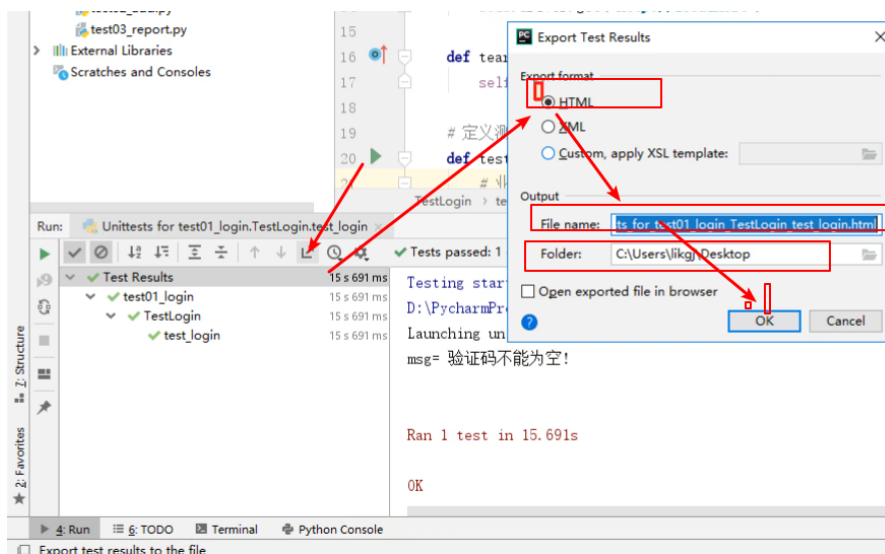
from day5.HTMLTestRunner import HTMLTestRunner
from day5.lianxi02_assert_tpshop_plus import TestLogin
from day5.lianxi03_parameterized import TestMyadd

suite = unittest.TestSuite()

# 2.向测试套件suite添加测试用例
# 类
suite.addTest(unittest.makeSuite(TestLogin))
suite.addTest(unittest.makeSuite(TestMyadd))

# 3a.准备stream title description
# 注意第三方 HTMLTestRunner模块放入项目目录
# s = open(文件名,打开方式)
# s.close()
# with open(文件名,打开方式) as fstream :
#     文件操作---自动关闭文件流
t = "这是一个测试标题"
d = "这是一个测试描述"
file_name = "./testreport.html"
with open(file_name, "wb") as s:
    # 3.实例化一个运行器对象 runner =
    HTMLTestRunner(stream,title,description)
    runner = HTMLTestRunner(stream=s, title=t,
description=d)
    # 4.运行器对象调用run(测试套件) runner.run(suite)
```

```
runner.run(suite)
```



总结--记忆

1. HTMLTestRunner作用

本质是--运行器--能够在运行用例的同时收集运行的结果信息,html格式的报

2. 使用HTMLTestRunner生成报告操作步骤

1. 实例化一个测试套件
2. 向测试套件suite添加测试用例
3. 实例化一个运行器对象
4. 运行器对象调用run(测试套件)