自动化

为什么要做自动化

1、为了提高回归测试的效率，避免回归测试的重复执行。

2、监控：监控已经实现的功能，没有发生问题

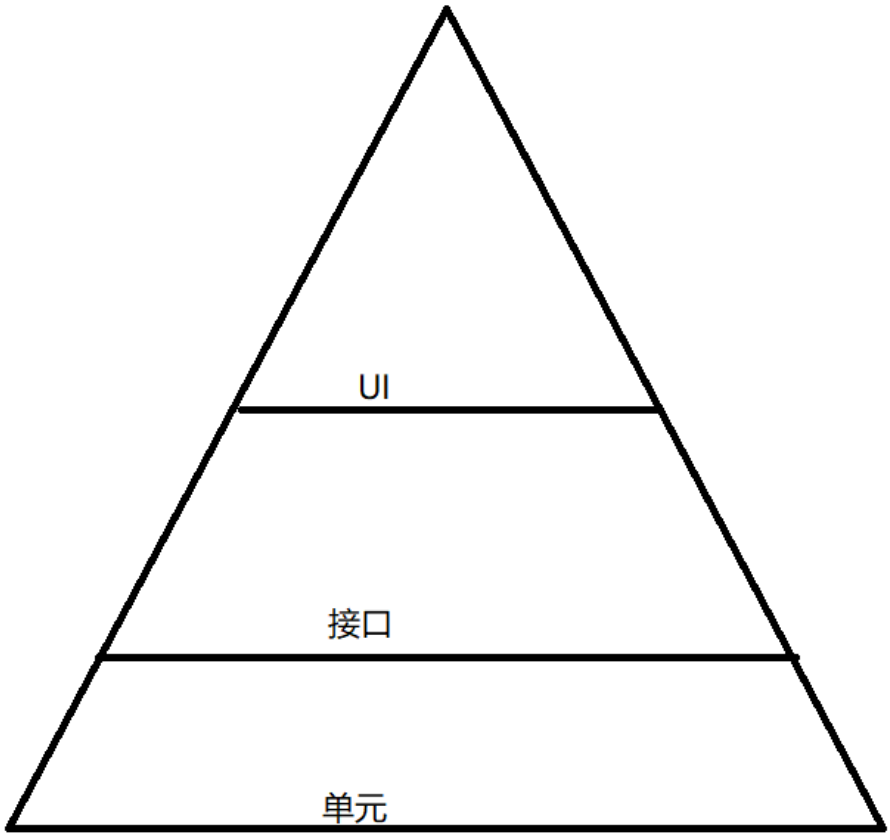
什么情况下可以做自动化

1、需求稳定，不会频繁变更。

2、项目周期长

3、人员具备自动化测试能力

自动化测试分层



UI层自动化：对页面功能做自动化测试验证

接口自动化：对接口层做测试验证

单元测试：对软件的最小单元模块的验证

常见自动化测试工具

–HP QuickTest Professional software简称QTP

Mercury公司产品，后被收购，执行重复的手动测试，主要是用于回归测试和测试同一软件的新版本。

–WinRunner

用于检测应用程序是否能够达到预期的功能及正常运行。通过自动录制、检测和回放用户的应用操作

–SilkTest

用于对企业级应用进行功能测试的产品，可用于测试Web、Java或是传统的C/S结构。

–Selenium

是一款基于web应用程序的开源测试工具，直接运行在浏览器上，可以像真实用户一样操作。

AutoIt3：

Appium：app端的自动化测试框架

selenium框架

Selenium是Thought Works公司开发的一套基于web应用的自动化测试工具，直接运行在浏览器中，模拟用户操作。它 可以被用于单元测试、集成测试、回归测试、系统测试、冒烟测试、验收测试，并且可以运行在各种浏览器和操作系统 上。

Selenium是Thought Works公司开发的一套基于web应用的自动化测试工具，直接运行在浏览器中，模拟用户操作。它可以被用于单元测试、集成测试、回归测试、系统测试、冒烟测试、验收测试，并且可以运行在各种浏览器和操作系统上。

Selenium分为1.0和2.0两个大版本，1.0主要包含ide、Grid、core和rc四大部分。2.0集成了1.0的功能，同时集成了webdriver，WebDriver旨在提供一个更简单，更简洁的编程接口以及解决一些Selenium-RC API的限制。Selenium-Webdriver更好的支持页面本身不重新加载而页面的元素改变的动态网页。WebDriver的目标是提供一个良好设计的面向对象的API，提供了对于现代先进web应用程序测试问题的改进支持。

selenium1.0

IDE：可以通过IDE完成测试过程的录制和回放。主要用来给初学者了解selenium，但不适合直接作为日常自动化的测试。

Grid：是selenium部署、测试及执行。

RC：selenium Remote Control,一个代理与控制器。

Core：selenium的测试机制核心部分，包含测试用例集的执行，断言，由js代码组成，支持夸平台运行。

selenium2.0

2.0 = selenium1.0 + webdriver

selenium环境搭建

selenium环境搭建需要安装以下库：

selenium-3.9

Firefox

geckodriver

1、本地安装

将selenium的安装包下载到本地，解压。解压后，通过cmd进入解压后的目录，执行python setup.py install

2、在线安装

在cmd中执行pip3 install selenium，直接执行，如果你需要安装指定的版本，你可以pip3 install selenium=3.9

3、安装firefox浏览器

4、安装geckodriver，将geckodriver的压缩包解压，将geckodriver.exe文件复制到python安装目录（实际只需要放到 PATH变量能访问到的目录就行）

验证环境：

在cmd中，执行如下操作，执行成功后，会自动打开Firefox浏览器，并跳转到百度页面，则表示环境搭建完成。

```
C:\Users\86159>python
Python 3.5.4 (v3.5.4:3f56838, Aug  8 2017, 02:17:05) [MSC v.1900 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from selenium import webdriver
>>> driver = webdriver.Firefox()
>>> driver.get('https://www.baidu.com')
```

selenium的浏览器基本操作

1. 打开浏览器

   driver = webdriver.Firefox()

2. 跳转地址

   get(url地址)

3. 刷新

   refresh()

4. 前进

   forward()

5. 后退

   back()

6. 关闭

   close()

7. 最大化

   ```
   maximize_window()
   ```

8. 最小化

```
minimize_window()
```

示例代码

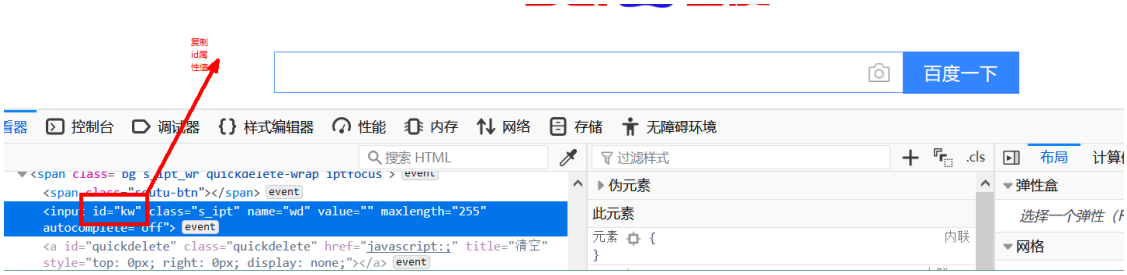
```
#coding=utf-8
__author__ = 'Meteor'
from selenium import webdriver
from time import sleep
#打开一个Firefox浏览器
driver = webdriver.Firefox()
driver.get('http://192.168.70.70:8080/suqi/tms/login')
#等2秒
sleep(2)
#刷新
driver.refresh()
#等2秒
sleep(2)
driver.find_element_by_link_text('公司注册').click()
#等2秒
sleep(2)
#后退
driver.back()
#等2秒
sleep(2)
#前进
driver.forward()
#等2秒
sleep(2)
#最小化
driver.minimize_window()
#等2秒
sleep(2)
#最大化
driver.maximize_window()
#等2秒
sleep(2)
#关闭浏览器
driver.close()
```

selenium元素定位

selenium定位元素的方式有8种。

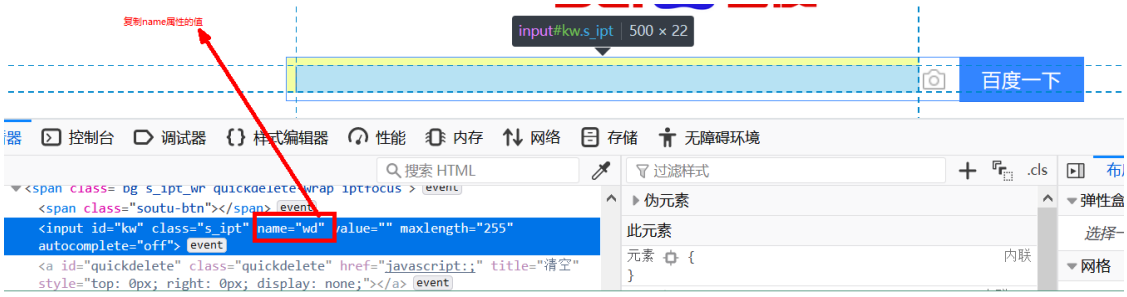
1. find_element_by_id(id属性的值)：根据标签的id属性的值定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_id('kw').send_keys('百度')
```



2. find_element_by_name(name属性的值)：根据标签的name属性值定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_name('wd').send_keys('百度')
```

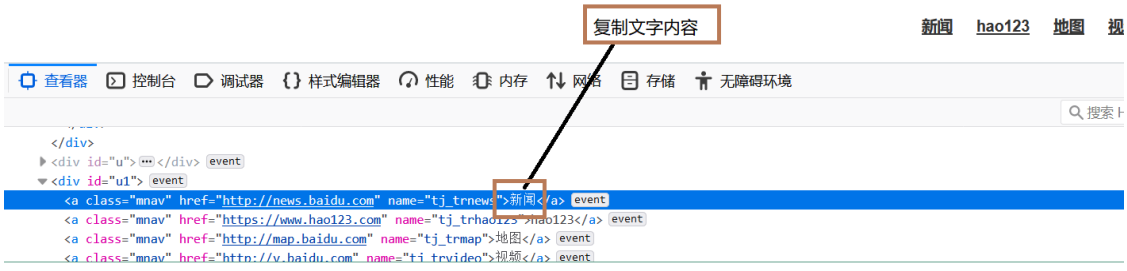


3. find_element_by_class_name(class属性的值)：根据class属性值定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_class_name('s_ipt').send_keys('百度')
```

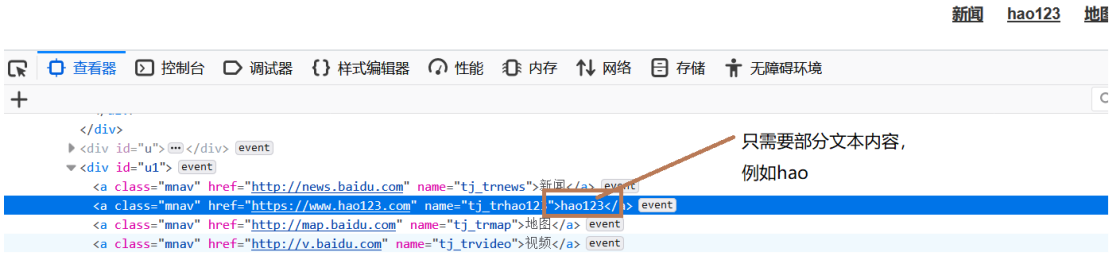
4. find_element_by_link_text('超链接文本内容')：根据超链接文本内容定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_link_text('新闻').click()
```



5. find_element_by_partial_link_text('部分超链接文本内容')：根据部分连接文本内容定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_partial_link_text('hao').click()
```



6. find_element_by_tag_name('标签名')：根据标签名定位元素

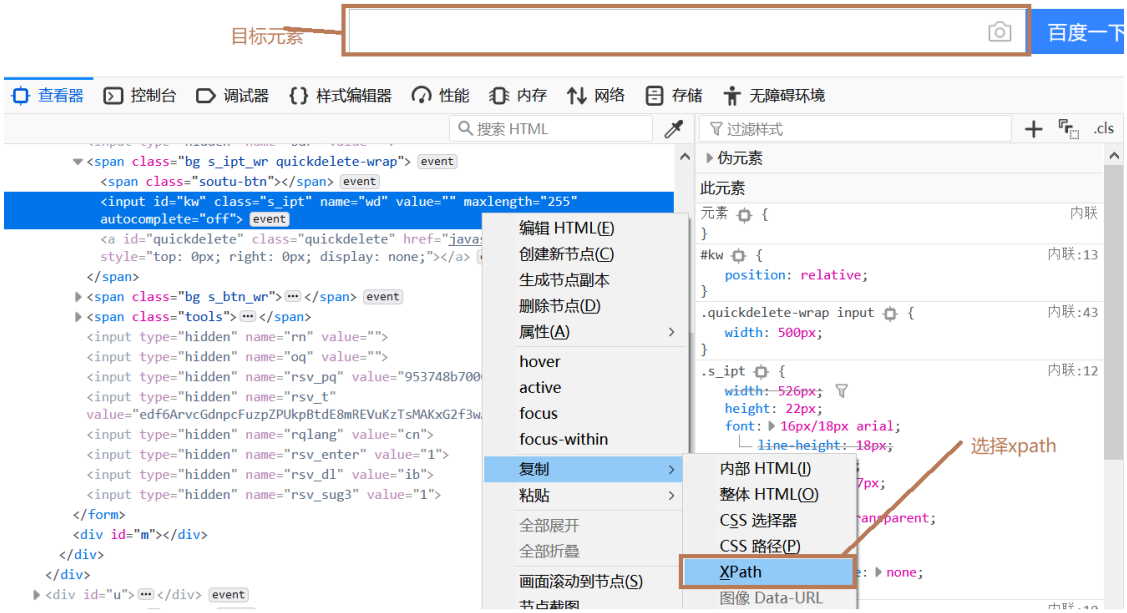
```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_tag_name('area').click()
```

7. find_element_by_css_selector('css选择器')：根据css选择器定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_css_selector('#su').click()
```

8. find_element_by_xpath(xpath路径)：根据xpath路径定位元素

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_xpath('//*[@id="kw"]').click()
```



注：xpath有两种表示方式：

1、绝对路径：

从html标签开始，顺序向下，逐层向下，直到目标元素，形成的最后的路径就是xpath路径,如果元素有多个相同的同级标签（兄弟标签）,则可以在标签后加[数字]，数字表示为第几个标签，数字从1开始。

示例：从下面的html代码中，定位name属性为输入的输入框，xpath路径表示为：

/html/body/div/table/tr/td[1]/input

```
<!DOCTYPE html>
<html>
<head>
    <title>hello</title>
</head>
<body>
    <div>
        <table>
            <tr>
                <td>
                    <input type="text" name="输入">
                </td>
                <td>
                    <button>登录</button>
                </td>
            </tr>
        </table>

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

2、相对路径：

从目标元素定位时，无法确认唯一标识，从该元素开始向上，直到找到一个能唯一定位的父级标签，再从该父级标签向下定位。不从html开始。

示例：在这个案例中，想要定位第一个input标签，通过xpath相对路径定位方式：

首先：在目标input上如果无法唯一定位，可以向上查找他的父级标签是否可以唯一定位，如果不行，可以继续向上查找，直到找了id为in1的这个div标签，开始从该标签向下定位，先定位到这个div标签"//div[@id='in1']"，再向下定位，如果不能确定id为in1的标签是div，可以用*代替。这个路径可以表示为：

1、 //div[@id='in1']/table/tr/td[1]/input

2、 //*[@id='in1']/table/tr/td[1]/input

```
<!DOCTYPE html>
<html>
<head>
    <title>hello</title>
</head>
<body>
    <div></div>
    <div></div>
    <div id="in1">
        <table>
            <tr>
                <td>
                    <input type="text" name="">
                </td>
                <td>
                    <input type="submit" name="">
                </td>
            </tr>
        </table>

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

定位方法：

在firefox浏览器中，将鼠标放到目标元素上，右键选择“查看元素”，会在开发者模式中显示你要定位目标元素。

在目标元素上，可以查看目标元素有哪些属性，如果有id或name属性，可以使用by_id或by_name定位元素。

如果上述方式不能帮助你定位元素，可以采用css选择器方式定位，在目标元素上，右键——>复制——>css选择器

在常用元素定位方式中，最常用的css和xpath两种，因为这两中定位方式基本可以定位所有的页面元素，也可以使用这两种方式实现父子元素定位或兄弟元素定位。

selenium的元素操作

send_keys(内容)：向元素中输入内容

click()：点击元素

text：获取元素中的文本内容，返回文本内容

title：获取当前窗口的title，返回title

is_displayed(): 如果元素可见，返回True，不可见返回False，不存在则产生NoSuchElementException

current_url：返回当前窗口的URL地址

等待：

在我们做自动化测试过程中，往往会发生页面跳转时，由于网络或则是环境问题，导致要定位的目标元素没有及时的加载出来，会产生脚本执行过程有些时候脚本执行是成功的，但是也有可能不成功，也就是说脚本不够稳定。

在做自动化测试时，难免会碰到一些问题，比如你在脚本中操作某个对象时，页面还没有加载出来，你的操作语句已经被执行，从而导致脚本执行失败，针对这样的问题webdriver提供了等待操作，等待一定的时间，或在一个时间段内发现对象，则继续操作。Webdriver提供了隐式等待和显示等待，当然，我们也可以借助包的模块，实现强制等待。

为了解决这些问题，我们可以在脚本中适当加入等待时间，以确保目标元素能够被定位到。

sleep等待

sleep是等待多少秒后，再继续执行后面的代码，要想使用sleep，必须先导入time包。

示例：在百度页面中，输入test，点击百度一下，必须要等待一段时间，才能点击元素

```
from selenium import webdriver
from time import sleep
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_id('kw').send_keys('test')
driver.find_element_by_id('su').click()
#等待2秒，再点击后面的操作
sleep(2)
driver.find_element_by_css_selector('#\31  > h3:nth-child(1) > a:nth-
child(1)').click()
```

注：sleep是强制等待，必须等到达到等待时间，才会继续执行下一步操作。时间如果设置的太长，会导致用例执行时间长，脚本执行效率底，等待时间短，依然可能发生元素没有加载成功。

智能等待：

隐式等待：implicitly_wait

```
from selenium import webdriver
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_id('kw').send_keys('test')
driver.find_element_by_id('su').click()
#智能等待30秒
driver.implicitly_wait(30)
driver.find_element_by_css_selector('#\31  > h3:nth-child(1) > a:nth-
child(1)').click()
```

隐式等待30秒，是总的等待时间，在这30内，如果任意一个时间上页面加载完成，那么等待将不再继续等待，直接执行后面的操作。implicitly_wait每0.5秒判断页面是否加载成功。如果30秒任然没有加载完成，则产生超时。

显示等待：

WebDriverWait

复杂操作：

鼠标操作：

在selenium中如果需要用到鼠标的操作，则需要引入ActionChains类，该类中提供了一系列的鼠标操作方法。

```
#coding=utf-8
__author__ = 'Meteor'
'''
selenium复杂操作
示例：
在百度页面中，进入搜索设置中，修改提示不显示，每页显示条数为50条
'''

from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox()
driver.get('https://www.baidu.com')

seting = driver.find_element_by_link_text('设置')
ac = ActionChains(driver)
ac.move_to_element(seting).perform()
```

ActionChains类在实例化时，需要实现构造方法，构造方法中需要传入Webdirverr实例，即上述示例中的driver。

move_to_element(元素)：该方法是将鼠标移动到指定元素上，但在页面中看不到执行效果，操作是保存在内存中的。

perform()：该方法是执行内存中存储的操作，并显示在页面中。

下拉框处理

在页面中如果要处理下拉框，则可以使用selenium提供的Select类，需要先导入该类。

Select类有构造方法，该方法需要接收一个元素，该方法接收的元素，必须是一个html的Select标签元素，如果不是select标签，则产生UnexpectedTagNameException异常。

如何区分页面下拉框是否可以使用Select类处理，根据下拉框元素标签是否为select标签决定

```
#coding=utf-8
__author__ = 'Meteor'
'''
selenium复杂操作
示例：
在百度页面中，进入搜索设置中，修改提示不显示，每页显示条数为50条
'''

from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.support.select import Select
```

```
driver = webdriver.Firefox()
driver.get('https://www.baidu.com')

seting = driver.find_element_by_link_text('设置')
ac = ActionChains(driver)
ac.move_to_element(seting).perform()

driver.find_element_by_link_text('搜索设置').click()



#定位到下拉框
s = driver.find_element_by_id('nr')
#创建一个select类对象
select = Select(s)

#根据下标选择下拉选项
# select.select_by_index(2)
#根据select标签中的option标签的value值选择下拉选项
# select.select_by_value('50')
#根据下拉选项的文本内容选择
select.select_by_visible_text('每页显示50条')
```

提示框处理

提示框的实现不只有一种方式，可以是开发人员自己通过html实现，也可以是通过alter实现。在selenium中如果是html实现，可以按照正常页面元素处理，如果是alert实现，则需要使用switch_to.alert来处理。

如何判断页面提示框是否为alert处理，能右键或f12定位的，那就用元素处理，不能那就用alert。

```
#coding=utf-8
__author__ = 'Meteor'
'''
selenium复杂操作
示例：
在百度页面中，进入搜索设置中，修改提示不显示，每页显示条数为50条
'''

from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.support.select import Select

driver = webdriver.Firefox()
driver.get('https://www.baidu.com')

seting = driver.find_element_by_link_text('设置')
ac = ActionChains(driver)
ac.move_to_element(seting).perform()

driver.find_element_by_link_text('搜索设置').click()
```

```
#定位到下拉框
s = driver.find_element_by_id('nr')
#创建一个select类对象
select = Select(s)

#根据下标选择下拉选项
# select.select_by_index(2)
#根据select标签中的option标签的value值选择下拉选项
# select.select_by_value('50')
#根据下拉选项的文本内容选择
select.select_by_visible_text('每页显示50条')


driver.find_element_by_xpath('/html/body/div[1]/div[7]/div/div/div/div[1]/form/div/table/tbody/tr[6]/td[2]/div[1]/a[1]').click()

alert = driver.switch_to.alert
#获取提示文本内容
print(alert.text)
#在提示框中的输入框内容输入内容
# alert.send_keys('内容')

#点击确定
alert.accept()
#点击取消
# alert.dismiss()
```

多层框架

如果前端在页面上实现了一个html窗口显示多个页面，则会通过多层框架实现，此时自动化测试过程，如果需要操作不同框架中的元素，则需要在框架中进行切换，使用switch_to.frame实现切换，可以切换到指定框架中，再使用default_content切回默认框架，如果一个页面中有多个框架，先切换到目标框架，操作结束后，需要切回默认框架，再切换到下一个目标框架。

```
#coding=utf-8
__author__ = 'Meteor'

from selenium import webdriver

driver = webdriver.Firefox()

driver.get('file:///G:/test/test.html')

#如果要操作内层框架中的元素，需要先切入到内层框
#框架切换
driver.switch_to.frame('f1')
driver.find_element_by_id('kw').send_keys('test')

#如果内层框架元素操作完成，需要外部框架的元素，你必须切回到外层框架
driver.switch_to.default_content()
driver.find_element_by_name('name').send_keys('test')
```

多窗口处理

```
#coding=utf-8
__author__ = 'Meteor'
'''
多窗口处理
'''
from selenium import webdriver
from time import sleep

driver = webdriver.Firefox()
driver.get('https://www.baidu.com')
driver.find_element_by_id('kw').send_keys('test')
driver.find_element_by_id('su').click()
driver.implicitly_wait(20)
#在百度结果页面中点击test_百度翻译
driver.find_element_by_xpath('/html/body/div[1]/div[5]/div[1]/div[3]/div[2]/h3/a
').click()
print(driver.title)
sleep(2)
#获取当前所有窗口(返回所有的窗口句柄)
handiles = driver.window_handles
#切换窗口
driver.switch_to.window(handiles[1])
print(driver.title)
```

unittest:

python内置的单元测试框架

Unittest是python的单元测试框架，原名为PyUnit，由java的junit演化而来。

在githup中是这样说明的：

python的unittest模块，有时称为“pyunit”，是基于Kentbeck和埃里希gamma的xunit框架设计。相同的模式在许多其他语言中重复，包括c、perl、Java和smalltalk。unittest实现的框架支持固定装置、测试套件和测试流，以便为您的代码启用自动测试。

1. test case：测试用例
2. test suites：测试套件
3. test fixtures：测试固定装置（测试夹具）
4. test runnet：测试运行
5. 断言

测试用例

测试用例：在python中使用unit test，需要先引入该模块，首先应该创建一个测试类，该类必须继承与unittest的TestCase类。继承后，将会拥有父类（TestCasse）中所有的方法和属性。在unittest中什么才算是测试用例?

```
import unittest
#创建一个测试类，该类继承与unittest的TestCase类
class Testa(unittest.TestCase):
    #写一条测试用例
    def test_001(self):
        print('正在执行第一条用例')

    def test_002(self):
        print('正在执行第二条用例')
```

```
#执行上述的测试用例：
if __name__ == '__main__':
    unittest.main()
```

if **name** == '**main**': :如果在当前模块中运行该模块，则每个模块都会有一个默认参数'**name**',如果在当前模块执行，该参数的返回值为**main**，执行时因为在当前模块执行，所以这个判断条件返回结果为真，则执行if的子句，如果是在其他模块中掉用该模块，则**name**的值将不在是**main**，将不会执行if的子句。作用：一般我们写完一个函数，通常会在当前模块中，直接调用该函数，以验证该函数实现是否正确，如果不加这句话，代码是可以正常执行，但是如果其他模块来调用该函数，该函数会被多次执行，加上这句之后，外部调用时，这个条件表达式将不再为真，也就不会被执行多次。

class Testa(unittest.TestCase): 创建一个类，Testa，该类继承了TestCase类。则说明该类为一个测试类。

在这个测试类中，多有以test开头的方法，则作为测试用例，再测试类中以test开头的方法都是测试用例。

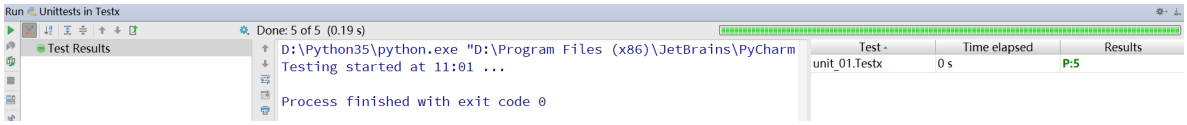
在执行时，用例的先后执行顺序由用例名称的Ascii码的顺序决定。

执行时，也可以选择执行某一条用例。就是将鼠标放到用例上，右键执行。

```
import unittest
'''
有一个函数，用于接收两个数字，并将两个数字除的结果返回。
如果数据不是数字类型，则返回数据类型错误。
如果除数是0，则返回除数不能为0
'''
def a_b(x,y):
    try:
        return x / y

    except TypeError:
        return '数据类型错误'
    except ZeroDivisionError:
        return '除数不能为0'
'''
测试上述函数：
'''
class Testx(unittest.TestCase):
    #测试两个整数除
    def test_001(self):
        a_b(10,2)
    #除数为0
    def test_002(self):
        a_b(10,0)
    #除数为字符
    def test_003(self):
        a_b(10,'a')
    #被除数为列表
    def test_004(self):
        a_b([1,2,3],2)
    #被除数为布尔类型
    def test_005(self):
        a_b(True,1)
```

```
# #执行上述的测试用例：
if __name__ == '__main__':
    unittest.main()
```

用例执行结果



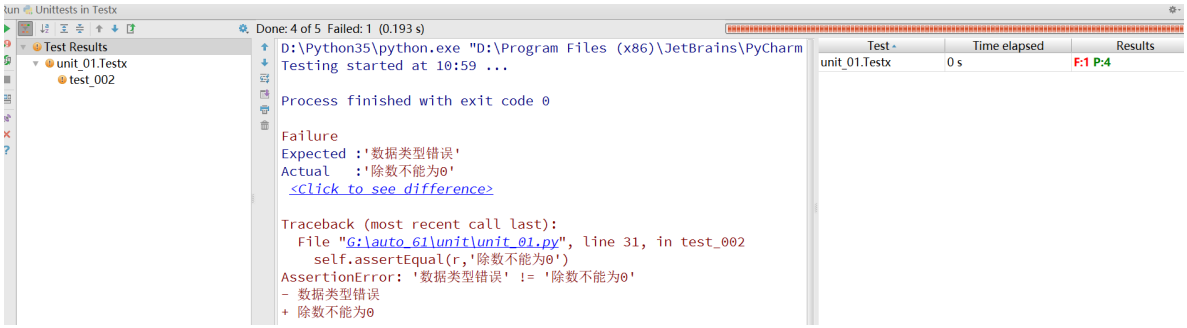
上述案例中，我们能了解unittest的用例和用例执行。但是从执行结果上看，无法判定用例执行是否成功。接下来需要对上述案例进行修改。

修改后如下：

```
import unittest
'''
有一个函数，用于接收两个数字，并将两个数字除的结果返回。
如果数据不是数字类型，则返回数据类型错误。
如果除数是0，则返回除数不能为0
'''
def a_b(x,y):
    try:
        return x / y

    except TypeError:
        return '数据类型错误'
    except ZeroDivisionError:
        return '数据类型错误'
'''
测试上述函数：
'''
class Testx(unittest.TestCase):

    def test_001(self):
        #断言用例执行是否与预期相符
        r = a_b(10,2)
        self.assertEqual(r,5)

    def test_002(self):
        r = a_b(10,0)
        self.assertEqual(r,'除数不能为0')

    def test_003(self):
        r = a_b(10,'a')
        self.assertEqual(r ,'数据类型错误')

    def test_004(self):
        r = a_b([1,2,3],2)
        self.assertEqual(r ,'数据类型错误')

    def test_005(self):
        r = a_b(True,1)
        self.assertEqual(r , 1)
```

```
# #执行上述的测试用例：
if __name__ == '__main__':
    unittest.main()
```

执行结果：



断言：

判定一个执行他的结果是否满足你的预期。unittest提供了多种断言方式。

assertEqual(first, second, msg=None)：如果first与second相等，则断言成功，否则断言失败，msg默认为空，也可以设置msg的值，这个值只有在断言失败的时候，才会有用

assertTrue(expr)：expr是一个表达式，如果表达式返回结果为True，则断言成功，否则断言失败

assertFalse(expr)：expr是一个表达式，如果表达式返回结果为False，则断言成功，否则断言失败

assertNotEqual(first,second)：如果first与second不相等，则断言成功，否则断言失败

assertNotTrue(expr)：expr是一个表达式，如果表达式返回结果不为True，则断言成功，否则断言失败

assertNotFalse(expr)：expr是一个表达式，如果表达式返回结果不为False，则断言成功，否则断言失败

assertIn(member, container)：如果member在container中，则断言成功，否则断言失败

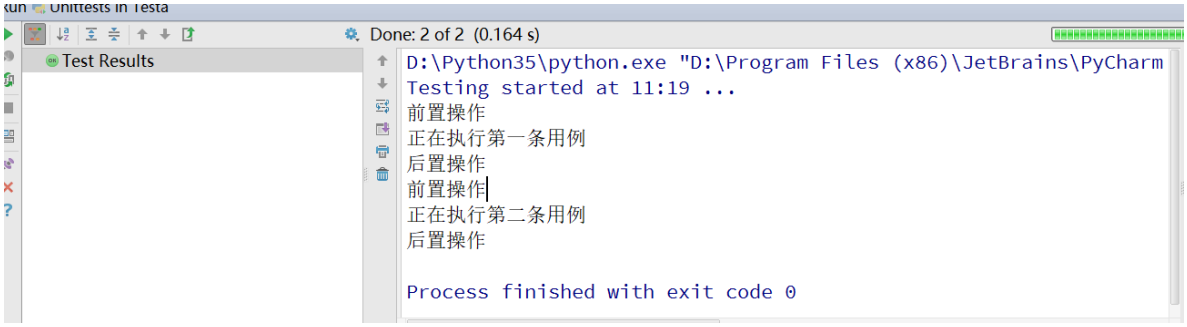
固定装置：

一个固定装置由前置操作+测试用例+后置操作组成。

```
import unittest
#创建一个测试类，该类继承与unittest的TestCase类
class Testa(unittest.TestCase):
    def setUp(self):
        print('前置操作')

    def tearDown(self):
        print('后置操作')
    #写一条测试用例
    def test_add_user(self):
        print('正在执行第一条用例')

    def test_del_user(self):
        print('正在执行第二条用例')
```

```
# #执行上述的测试用例：
if __name__ == '__main__':
    unittest.main()
```

执行结果：



从执行结果中可以看出，每条测试用例在执行之前，会先执行setUp方法，setUp方法执行结束之后，执行测试用例，用例执行结束之后会执行tearDown方法，再次执行下面的用例时与上面逻辑一样

setUp和tearDown方法名是固定的，这个是父类中的方法，在用例中，将父类中的该方法重写。

测试套：

一个测试用例集。一个测试套中可以包含多条测试用例

使用方式：

1、引入unittest模块

2、创建一个空的测试套

3、加载所有要执行的测试用例

4、将加载过来的测试用例放入到测试套中。

5、运行测试套

案例：

```
import unittest
from unit import test_01

#创建一个空的测试套
suite = unittest.TestSuite()

#加载test_01模块中的所有测试用例
tests = unittest.TestLoader().loadTestsFromModule(test_01)

#将测试用例放入到测试套中
suite.addTest(tests)

#执行测试套中所有用例
unittest.TextTestRunner().run(suite)
```

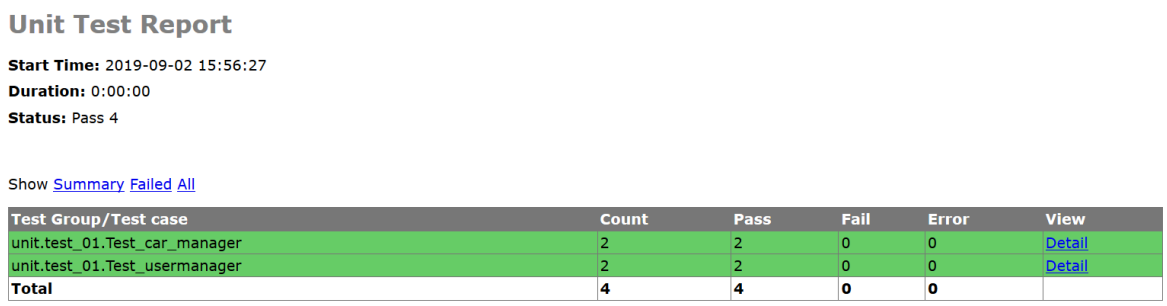
HTMLTestRunner

在做自动化测试过程，我们需要生成一个可视化的测试报告，报告内容包含测试开始时间，测试执行消耗时间，执行用例数、通过、失败、错误。

HTMLTestRunner时一个第三方的html格式的测试报告生成工具。

```
import unittest
from unit import test_01

#创建一个空的测试套
suite = unittest.TestSuite()

#加载test_01模块中的所有测试用例
tests = unittest.TestLoader().loadTestsFromModule(test_01)

#将测试用例放入到测试套中
suite.addTest(tests)
#1、打开一个html文件
file = open('abcd.html','wb')
#执行测试用例，并生成测试报告
HTMLTestRunner.HTMLTestRunner(stream=file).run(suite)

#关闭file
file.close()
```

结果：



Unit Test Report

Start Time: 2019-09-02 15:56:27
Duration: 0:00:00
Status: Pass 4

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
unit.test_01.Test_car_manager	2	2	0	0	Detail
unit.test_01.Test_usermanager	2	2	0	0	Detail
Total	**4**	**4**	**0**	**0**	

项目练习1：

苏汽项目练习，该项目使用PO模式实现。构建结构如下：

```
suqi_auto
        -base       存放selenium封装的库
            selensuqi.py    对selenium的二次封装
        -page_class 所有的页面对象
            页面脚本
        -case       所有测试用例库
            用例
        -data       对象库
            excel文件
        -tools      工具库
            redexcel.py 读取excel库
            访问mysql数据库
            log日志库
```

示例：

selensuqi.py：该模块中实现了SelenBrower类，该类中有一个构造方法，构造方法实现的打开浏览器，需要调用者传入浏览器的类型，决定打开什么浏览器

```
#coding=utf-8
__author__ = 'Meteor'

from selenium import webdriver
from selenium.common.exceptions import
InvalidArgumentException,NoSuchElementException

class SelenBrower:

    def __init__(self,brower=None):
        '''
                该类是对seleuium的封装，构造方法中，需要传入浏览器的类型，根据传入类型，打开对应浏览器
        :param brower:
            s = SelenBrower('Firefox')    打开firefox浏览器
            s = SelenBrower('Ie')    打开Ie浏览器
            s = SelenBrower('Chrome')    打开Chrome浏览器
        :return:None
        '''
        if brower == 'Ie':
            self.driver = webdriver.Ie()
        elif brower == 'Chrome':
            self.driver = webdriver.Chrome()
        else:
            self.driver = webdriver.Firefox()


    def get(self,url):
        '''
        功能：操作浏览器跳转到指定的URL地址，如果传入的URL地址不合法，则提示
        url：http://地址
        '''
        try:
            self.driver.get(url)
        except InvalidArgumentException:
            print('URL地址格式错误%s' %url)
```

```
    def fond_element(self,element):
        '''
        该方法用来定位元素，并不对元素执行任何操作
        :param element:
            self.fond_element('id=compname')
        :return:将定位到的元素返回
        '''
        by = element.split('=')[0]
        self.driver.implicitly_wait(30)
        try:
            if by == 'css':
                return
self.driver.find_element_by_css_selector(element.split('=')[1])
            elif by == 'id':
                return self.driver.find_element_by_id(element.split('=')[1])
            elif by == 'name':
                return  self.driver.find_element_by_name(element.split('=')[1])
            elif by == 'link_text':
                return  self.driver.find_element_by_link_text(element.split('=')
[1])
            elif by == 'xpath':
                return
 self.driver.find_element_by_link_text(element.split('=',1)[1])
        except NoSuchElementException:
            print('元素未能找到%s' %element)

    def send_keys(self,element,keys):
        '''
            该方法用于在元素中输入内容。
            element：要输入内容的元素
            keys：要输入的内容
        '''
        self.fond_element(element).clear()
        #用于清空输入框中的内容
        self.fond_element(element).send_keys(keys)

    def click(self,element):
        '''
        点击元素
        '''
        self.fond_element(element).click()


    def get_text(self,element):
        '''
        获取元素中的文本内容
        '''
        return self.fond_element(element).text
```

login_page.py：登录页面的页面对象，该类继承于上面base包中的SelenBrower类。

```
#coding=utf-8
__author__ = 'Meteor'

from base.selensuqi import SelenBrower
```

```
class Login(SelenBrower):

    def ente_compname(self,compname):
        #输入公司名
        self.send_keys('id=companyName',compname)

    def enter_user(self,username):
        #输入用户名
        self.send_keys('id=account',username)

    def enter_pwd(self,pwd):
        #输入密码
        self.send_keys('id=password',pwd)

    def click_login(self):
        #点击登录
        self.click('link_text=登录')


    def gettext(self):
        #获取登录成功后的文本信息
        return self.get_text('xpath=/html/body/div[2]/div[1]/div/div[2]/font')


    def alert_text(self):
        #获取密码为空提示框信息
        return self.get_text('css=.messager-body > div:nth-child(2)')

    def alert_accept(self):
        #点击提示框中的确定
        pass


    def zhuce(self):
        #点击登录页面的注册
        pass
```

cases目录中存放的是测试用例。用例实现是调用Login类中的方法实现

```
#coding=utf-8
__author__ = 'Meteor'

import unittest
from page_object.login_page import Login
class Test_Login(unittest.TestCase):

    def test_001(self):
        l = Login()
        l.ente_compname('苏汽')
        l.enter_user('sadmin')
        l.enter_pwd('admin')
        l.click_login()
        r = l.gettext()
```

```
        self.assertEqual(r,'')

    def test_002(self):
        l = Login()
        l.ente_compname('苏汽')
        l.enter_user('sadmin')
        l.click_login()
        r = l.gettext()
        self.assertEqual(r,'密码不能为空')
```

html基础：

html：超文本标记语言。

开发会将所有的网页页面用html写成多个html文件，由用户通过http请求，向服务器端请求html文件。

html文件是由一系列的标签组成，每个标签的功能不同，标签由0个或多个属性组成，每个属性的功能也不相同。

属性：属性名='属性值'。如果有多个属性，多个属性之间用空格隔开

id属性：给一个标签设置id属性，该id属性在当前html页面中是唯一的。

name属性：给一个标签设置name，这个名称一般也是唯一。

class属性：类属性，通常是用来标识一类元素

常见的html标签：