

Web自动化测试

1.元素的交互

1)点击

```
1 | element.click()
```

2)清空

```
1 | element.clear()
```

3)输入

```
1 | element.send_keys("输入的内容")
```

2.元素的定位

1)css选择器

```
1 | element = browser.find_element_by_css_selector("对应的css选择器")
```

2) id选择器

```
1 | element = browser.find_element_by_id("对应的id名")
```

3) class选择器

```
1 | browser.find_element_by_class_name("搜索的class名称")
```

4) 标签选择器

```
1 | element = browser.find_element_by_tag_name("标签名")
```

5) 文本选择器

```
1 | element = browser.find_element_by_link_text("提交")#精准
2 | element = browser.find_element_by_partial_link_text("提交")#模糊
```

6) xPhan选择器

```
1 | element = browser.find_element_by_xpath("元素的xphan位置")
```

7) 拓展xpath

- //: 跳级
- /: 父子级关系
- //div[@class='class名称'], 指定选择div里class属性为"class名称"
- //div[@class='class名称']/div. 指定选择div里class属性为"class名称"的子级div

- //div[@class='class名称']/div[contains(@class,'aaa')] 指定选择div里class属性为"class名称"的子级div的class属性里包含aaa

3.元素的隐性等待

在进入页面时由于程序更新过快，页面请求未加载完成，导致报错

在交互前等待再操作

```
1 element = browser.find_element_by_xpath("提交")
2 time.sleep(2)
3 element.send_keys("Hello world!")
```

上程序会导致主程序休眠，如果是30s呢？所以会将修改为

```
1 element = browser.find_element_by_xpath("提交")
2 #多少秒内找到元素立刻执行，找不到则报错
3 element.implicitly_wait(10)
4 element.send_keys("Hello world!")
```

此运行结果会立即生效

4.单选/多选/下拉交互

定位对应的标签click

注意如果是下拉交互则需要定位到所选择的子元素

```
1 element = browser.find_element_by_xpath("/html/body/div[8]/select/option[3]")
2 element.click()
```

5.日期/评分交互

```
1 #日期交互为输入,在年前加00 如0020250228
2 element.send_keys("0020250225")
3 #评分交互为点击对应元素交互
4 element.click()
```

6.上传交互

```
1 element.send_keys(r'图片本机路径')
```

7.常用操作

1)driver操作

方法	解释
back	浏览器后退
close	关闭tab页但是，driver.exe不主动停止
quit	退出浏览器，关闭driver.exe进程

方法	解释
curent_url	当前的url地址
execute_script	执行js脚本
forward	前进
fullscreen_window	全屏
get	打开指定的url
title	浏览器标题
get_window_position	窗口位置信息
refresh	刷新界面
set_window_size	设置窗口大小
maximize_window	最大化窗口
minimize_window	最小化窗口
name	浏览器名称
page_soure	页面源代码
save_screenshot	保存截图

```
1  #关闭浏览器的当前页
2  driver.close()
3  #关闭浏览器
4  driver.quit
```

2) 元素的操作

方法	说明
clear	去除元素内容
click	点击
find_element	定位元素
get_attribute	获取属性
is_displayed	是否显示
is_enabled	是否使能
is_selected	是否可选
location	位置
rect	矩形
screenshot_as_png	元素存为图片

方法	说明
send_keys	输入信息文本
size	大小
submit	提交
tag_name	标签名称
text	标签文本
location_once_scrolled_into_view	滚动可见

8.三大等待

```

1  #强制等待
2  time.sleep(2)
3  #隐式等待
4  driver.implicitly_wait(10)
5  #显示等待
6  ele = WebDriverWait(driver,10).until(
7      lambda _:driver.find_element(
8          By.XPATH,"/...路径")
9  )

```

lambda为匿名函数

9.Pom

- 1.写出元素定位表达式
- 2.写书适合的等待方式
- 3.维护各个元素的定位表达式

Pager Objects Model 页面对象模型

- 类 表示 页面
- 类属性 表示 页面中的元素
- 类的方法 表示 对页面的操作

10.键盘操作

在sendKeys的基础上操作：

例如要输入ctrl+A: `sendKeys(Keys.CONTROL,"A");`

`sendKeys(Keys.按键,"按键");**`

```

1  private static void test08() throws InterruptedException {
2      WebDriver webDriver = new ChromeDriver();
3      webDriver.get("https://www.baidu.com");
4      webDriver.findElement(By.cssSelector("#kw")).sendKeys("521");
5      // 键盘操作    Keys.
6      // ctrl + A

```

```

7     webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL, "A");
8         sleep(1000);
9         // ctrl + x
10
11    webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL, "X");
12        sleep(1000);
13        // ctrl + v
14
15    webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL, "V");
16        sleep(1000);
17    }

```

11.鼠标操作

通过 Actions 对象来操作: `Actions actions = new Actions(webDriver);`

鼠标右击: `contextClick();`

鼠标双击: `doubleClick();`

鼠标移动: `moveToElement(webElement);`

执行行为: `perform();`

测试框架unittest

一、TestCase (测试用例)

在普通类中如何定义测试类, 发现让程序发现测试类

继承`unittest.TestCase`, 将类继承于此类, 从而定义为测试类

1.默认测试类:

```

1  import unittest
2
3
4  class MyTestCase(unittest.TestCase):
5      def test_something(self):
6          self.assertEqual(True, False) # add assertion here
7
8
9  if __name__ == '__main__':
10     unittest.main()

```

2.第一个测试用例

```

1  # 1、导包
2  # 2、自定义测试类
3  # 3、在测试类中书写测试方法 采用print 简单书写测试方法
4  # 4、执行用例
5
6  import unittest

```

```

7
8 # 2、自定义测试类,需要继承unittest模块中的TestCase类即可
9 class TestDemo(unittest.TestCase):
10     # 书写测试方法,测试用例代码,书写要求,测试方法必须test_ 开头
11     def test_method1(self):
12         print('测试方法1-1')
13
14     def test_method2(self):
15         print('测试方法1-2')
16
17 # 4、执行测试用例
18 # 4.1 光标放在类后面执行所有的测试用例
19 # 4.2 光标放在方法后面执行当前的方法测试用例

```

二、TestSuite(测试套件)

TestSuite（测试套件）：用来组装，打包，管理多个TestCase（测试用例）文件的

```

1 # 1、导包
2 # 2、实例化（创建对象）套件对象
3 # 3、使用套件对象添加用例方法
4 # 4、实例化对象运行
5 # 5、使用运行对象去执行套件对象
6
7 import unittest
8
9 from unittest_Demo2 import TestDemo
10 from unittest_Demo1 import Demo
11
12 suite = unittest.TestSuite()
13
14 # 将一个测试类中的所有方法进行添加
15 # 套件对象.addTest(unittest.makeSuite(测试类名))
16 suite.addTest(unittest.makeSuite(TestDemo))
17 suite.addTest(unittest.makeSuite(Demo))

```

将多个测试用例导入后，利用addTest函数添加到测试套件中

套件对象.addTest(unittest.makeSuite(测试类名))

这是老版本的写法，现在也可以使用

新版本写法如下：

```

1 suite.addTest(TestDemo("test_case1"))

```

```

1 import unittest
2
3
4 class TestDemo(unittest.TestCase):
5     def test_case1(self):
6         print("test_case1")
7
8

```

```

9  class TestDemo2(unittest.TestCase):
10     def test_case2(self):
11         print("test_case2")
12
13
14  if __name__ == '__main__':
15     suite = unittest.TestSuite()
16     suite.addTest(TestDemo("test_case1"))
17     suite.addTest(TestDemo2("test_case2"))

```

三、TestRunner(测试执行)

执行特定的测试函数或者测试套件

```

1  # 1、实例化运行对象
2  runner = unittest.TextTestRunner();
3  # 2、使用运行对象去执行套件对象
4  # 运行对象.run(套件对象)
5  runner.run(suite)

```

注意：

```

1  # 1、导包
2  # 2、实例化（创建对象）套件对象
3  # 3、使用套件对象添加用例方法
4  # 4、实例化对象运行
5  # 5、使用运行对象去执行套件对象
6
7  import unittest
8
9
10 class TestDemo(unittest.TestCase):
11     def test_case1(self):
12         print("test_case1")
13
14     def test_case2(self):
15         print("test_case2")
16
17
18 class Demo(unittest.TestCase):
19     def test_case3(self):
20         print("test_case3")
21
22     def test_case4(self):
23         print("test_case4")
24
25
26 suite = unittest.TestSuite()
27
28 # 将一个测试类中的所有方法进行添加
29 # 套件对象.addTest(unittest.makeSuite(测试类名))
30 suite.addTest(TestDemo())
31 suite.addTest(Demo())
32 if __name__ == '__main__':
33     # 4、实例化运行对象

```

```

34 runner = unittest.TextTestRunner()
35 # 5、使用运行对象去执行套件对象
36 # 运行对象.run(套件对象)
37 runner.run(suite)

```

四、TestLoader（测试加载）

写法：

1. suite = unittest.TestLoader().discover("指定搜索的目录文件","指定字母开头模块文件")
2. suite = unittest.defaultTestLoader.discover("指定搜索的目录文件","指定字母开头模块文件")

【推荐】注意： 如果使用写法1，TestLoader()必须有括号。

```

1 # 1. 导包
2 # 2. 实例化测试加载对象并添加用例 ---> 得到的是 suite 对象
3 # 3. 实例化 运行对象
4 # 4. 运行对象执行套件对象
5
6 import unittest
7
8 # 实例化测试加载对象并添加用例 ---> 得到的是 suite 对象
9 # unittest.defaultTestLoader.discover('用例所在的路径', '用例的代码文件名')
10 # 测试路径：相对路径
11 # 测试文件名：可以使用 * 通配符，可以重复使用
12 suite = unittest.defaultTestLoader.discover('./Case', 'cs*.py')
13 runner = unittest.TextTestRunner()
14 runner.run(suite)

```

就是说运行指定目录下的指定类，规则由该方法指定discover(指定相对路径，指定文件)

注意：指定文件也需要继承unittest.TestCase类是一个测试类

TestSuite与TestLoader区别：

```

1 TestSuite与TestLoader区别：
2     共同点：都是测试套件
3     不同点：实现方式不同
4     TestSuite： 要么添加指定的测试类中所有test开头的方法，要么添加指定测试类中指定某个
5     test开头的方法
6     TestLoader： 搜索指定目录下指定字母开头的模块文件中以test字母开头的方法并将这些方法
7     添加到测试套件中，最后返回测试套件

```

五、Fixture（测试夹具）

是一种代码结构，在某些特定情况下，会自动执行。

1.方法级别

在每个测试方法（用例代码）执行前后都会自动调用的结构

- def setup(),每个测试方法执行之前都会执行（初始化）
- def tearDown(),每个测试方法执行之后都会执行（释放）


```

1  # 初始化
2  def setUp(self):
3      # 每个测试方法执行之前执行的函数
4
5  # 释放
6  def tearDown(self):
7      # 每个测试方法执行之后执行的函数

```

代码补充完整:

```

1  import unittest
2
3
4  class Demo(unittest.TestCase):
5      def test_case3(self):
6          print("Test1:test_case3")
7
8      def test_case4(self):
9          print("Test1:test_case4")
10
11     # 初始化
12     def setUp(self):
13         # 每个测试方法执行之前执行的函数
14         print("Demo:setUp")
15
16     # 释放
17     def tearDown(self):
18         # 每个测试方法执行之后执行的函数
19         print("Demo:tearDown")

```

运行结果为:

```

1  Demo:setUp
2  Test1:test_case3
3  Demo:tearDown
4  Demo:setUp
5  Test1:test_case4
6  Demo:tearDown

```

2.类级别

在每个测试类中所有方法执行前后 都会自动调用的结构(在整个类中 执行之前执行之后各一次)

- `def setUpClass()` ,类中所有方法之前
- `def tearDownClass()` , 类中所有方法之后

特性: 测试类运行之前运行一次setUpClass , 类运行之后运行一次tearDownClass

注意: 类方法必须使用 @classmethod修饰

```

1  import unittest
2
3
4  class Demo(unittest.TestCase):

```

```

5     def test_case3(self):
6         print("Test1:test_case3")
7
8     def test_case4(self):
9         print("Test1:test_case4")
10
11    @classmethod
12    def setUpClass(cls):
13        print('-----类加载前')
14
15    @classmethod
16    def tearDownClass(cls):
17        print('-----类加载后')

```

运行结果：

```

1  -----类加载前
2  Test1:test_case3
3  Test1:test_case4
4  -----类加载后

```

六、断言

让程序代替人工自动的判断预期结果和实际结果是否相符

断言的结果：

- 1)、True，用例通过
- 2)、False，代码抛出异常，用例不通过
- 3)、在unittest中使用断言，需要通过 `self.断言方法`

常用的断言：

```

1  self.assertEqual(ex1, ex2) # 判断ex1 是否和ex2 相等
2  self.assertIn(ex1, ex2) # ex2是否包含 ex1  注意：所谓的包含不能跳字符
3  self.assertTrue(ex) # 判断ex是否为True
4
5  重点讲前两个assertEqual 和 assertIn
6  方法：
7  assertEquals: self.assertEqual(预期结果, 实际结果) 判断的是预期是否相等实际
8  assertIn: self.assertIn(预期结果, 实际结果) 判断的是预期是否包含实际中
9  assertIn('admin', 'admin') # 包含
10 assertIn('admin', 'adminnnnnnnnn') # 包含
11 assertIn('admin', 'aaaaadmin') # 包含
12 assertIn('admin', 'aaaaadminnnnnnn') # 包含
13 assertIn('admin', 'addddadmin') # 不是包含

```

断言的使用：

```

1 import unittest
2 # assertEquals: self.assertEqual(预期结果, 实际结果) 判断的是预期是否相等实际
3 # assertIn: self.assertIn(预期结果, 实际结果) 判断的是预期是否包含实际中
4 def test_001():
5     return 1+1
6 class TestLogin(unittest.TestCase):
7     def test_001(self):
8         self.assertEqual(2, test_001())

```

assertEquals: self.assertEqual(预期结果, 实际结果) 相等择为None, 不相等会报错并输出差异
其余断言的方法:

方法	说明
assertTrue(expr)	验证expr是否为True
assertFalse(expr)	验证expr是否为False
assertEqual(first,second)	验证first是否等于second
assertNotEqual(first,second)	验证first是否不等于second
assertIsNone(obj)	验证obj是否为None
assertIsNotNone(obj)	验证obj是否不为None
assertIn(member,container)	验证container中是否包含member
assertNotIn(member,container)	验证container中是否不包含member

七、测试报告

1.Pyharm软件自动导出

测试成功后点击控制台的导出按钮输出为Html即可

2.第三方HTMLTestRunner 导出

HTMLTestRunner 类的部分实现, 主要看init()初始化方法的参数:

- stream: 指定生成 HTML 测试报告的文件, 必填。
- verbosity: 指定日志的级别, 默认为 1。如果想得到更详细的日志, 则可以将参数 修改为 2。
- title: 指定测试用例的标题, 默认为 None。
- description: 指定测试用例的描述, 默认为 None

代码如下:

定义好生成路径, 测试的代码运行即可

```

1 import unittest
2 from HTMLTestRunner import HTMLTestRunner
3
4 #当前路径
5 test_dir = './'
6

```

```
7 #!iot_*.py表示!iot_开头的所有测试用例
8 discover = unittest.defaultTestLoader.discover(test_dir, pattern='*.py')
9
10 #报告存放的路径
11 fp = open("./iot.html", "wb")
12 runner = HTMLTestRunner(stream=fp, title='测试报告', description='测试用例情况:')
13 runner.run(discover)
14 fp.close
```

测试框架Pytest

[API 参考 - pytest 文档 - pytest 测试框架](#)

这个有项目实践，非常推荐：[pytest 官方文档的中文翻译](#)，但不仅仅是单纯的翻译，也包含自己的理解和实践。

1. 发现测试用例
2. 执行测试用例
3. 判断测试结果
4. 生成测试报告

一、pytest 默认测试用例

pytest 默认测试用例的格式：

- **模块名**：模块名（文件名）通常被统一放在一个 `testcases` 文件夹中，然后需要保证模块名以 `test_` 开头或 `_test` 结尾，例如 `test_demo1` 或 `demo2_test`
- **类名**：测试类类名必须以 `Test` 开头，并且不能带有 `init` 方法
- **方法名**：测试方法名（Case 名）必须以 `test_` 开头，例如 `test_demo1(self)`、`test_demo2(self)`

第一个测试用例：

```
1 class TestDemo:
2     def test_demo1(self):
3         print("测试用例1")
4
5     def test_demo2(self):
6         print("测试用例2")
```

可以发现类旁边与函数旁边都有运行标识符，可以运行使用

二、全局配置文件 pytest.ini

更改配置文件可以将识别规则更改

```
1 [pytest]
2 #参数
3 addopts = -vs
4 # 默认的执行路径，它会默认执行该文件夹下所有的满足条件的测试case
5 testpaths = ./testcases
6 # 文件命名规则
```

```

7 python_files = test_*.py
8 # 类名命名规则
9 python_classes = Test*
10 # Case命名规则
11 python_functions = test_*
12
13 # 标记
14 markers =
15 # 冒烟规则
16 smoke:冒烟用例
17 product_manage:商品管理

```

三、执行pytest

命令行执行:

```

1 # -vs: -v输出详细信息 -s输出调试信息
2 pytest -vs
3
4 # -n: 多线程运行 (前提安装插件: pytest-xdist)
5 pytest -vs -n=2
6
7 # --reruns num: 失败重跑 (前提安装插件: pytest-rerunfailres)
8 pytest -vs --reruns=2
9
10 # -x: 出现一个用例失败则停止测试
11 pytest -vs -x
12
13 # --maxfail: 出现几个失败才终止
14 pytest -vs --maxfail=2
15
16 # --html: 生成html的测试报告,后面 需要跟上所创建的文件位置及文件名称 (前提安装插件:
    pytest-html)
17 pytest -vs --html ./reports/result.html
18
19 # -k: 运行测试用例名称中包含某个字符串的测试用例,我们可以采用or表示或者,采用and表示都
20 pytest -vs -k "qiuluo"
21 pytest -vs -k "qiuluo or weiliang"
22 pytest -vs -k "qiuluo and weiliang"
23
24 # -m: 冒烟用例执行,后面需要跟一个冒烟名称,执行user_manage这个分组
25 pytest -vs -m user_manage

```

添加注解: @pytest.mark.user_manage

命令行执行: pytest -vs -m user_manage

```

1 class TestDemo:
2
3     # 我们在Case上采用@pytest.mark. + 分组名称,就相当于该方法被划分为该分组中
4     # 注意: 一个分组可以有多个方法,一个方法也可以被划分到多个分组中
5     @pytest.mark.user_manage
6     def test_demo1(self):
7         print("user_manage_test1")
8

```

```
9      @pytest.mark.product_manage
10      def test_demo2(self):
11          print("product_manage_test1")
12
13      @pytest.mark.user_manage
14      @pytest.mark.product_manage
15      def test_demo3(self):
16          print("manage_test1")
17
```

Main函数执行:

添加mian: 使用 `main` 方法执行

```
1  if __name__ == '__main__':
2      pytest.main()
3
4  if __name__ == '__main__':
5      pytest.main(["-vs"])
```

完整代码:

```
1  import pytest
2
3  def test_01():
4      print("啥也没有")
5
6  if __name__=='__main__':
7      pytest.main()
```

四、执行参数

main函数执行参数

参数	描述	例子
-v	输出调试信息，如打印信息	pytest.main(['-v','testcase/test_one.py','testcase/test_two.py'])
-s	输出更详细的详细，如文件名，用例名称	pytest.main(['-vs','testcase/test_one.py','testcase/test_two.py'])
-n	多线程或分布式运行测试用例	
-x	只有一个执行用例失败择退出执行测试	pytest.main(['-vsx','testcase/test_one.py'])
- maxfail	出现N个测试用例失败就停止测试	pytest.main(['-vs','-x=2','testcase/test_one.py'])

参数	描述	例子
- html=report.html	输出测试报告	pytest.main(['-vs', '- html=./report.html', 'testcase/test_one.py'])
-m	通过标记表达式执行	
-k	根据测试用例的部分字符串指定测试用例，可以用 and/or	

命令行的参数：

参数	描述	案例
-v	输出调试信息。如：打印信息	pytest -x ./testcase/test_one.py
-q	输出简单信息。	pyets -q ./testcase/test_one.py
-s	输出更详细的信息，如：文件名、用例名	pytest -s ./testcase/test_one.py
-n	多线程或分布式运行测试用例	
-x	只要有一个用例执行失败，就停止执行测试	pytest -x ./testcase/test_one.py
- maxfail	出现N个测试用例失败，就停止测试	pytest --maxfail=2 ./testcase/test_one.py
- html=report.html	生成测试报告	pytest ./testcase/test_one.py -- html=./report/report.html
- html=report.html	生成测试报告	pytest ./testcase/test_one.py -- html=./report/report.html
-k	根据测试用例的部分字符串指定测试用例，可以使用 and, or	pytest -k "MyClass and not method"，这条命令会匹配文件名、类名、方法名匹配表达式的用例，这里这条命令会运行 TestMyClass.test_something，不会执行 TestMyClass.test_method_simple

五、常用运行方式：配置文件

pytest.ini配置文件执行

不管是mian执行方式还是命令执行，最终都会去读取pytest.ini文件
在项目的根目录下创建pytest.ini文件

```
1 [pytest]
2 addopts=-vs -m slow --html=./report/report.html
3 testpaths=testcase
4 test_files=test_*.py
5 test_classes=Test*
6 test_functions=test_*
7 makerers=
8     smock:冒烟测试用例
```

pytset.ini文件尽可能不要出现中文。

pytset.ini文件中的参数

参数	作用
[pytest]	用于标志这个文件是pytest的配置文件
addopts	命令行参数，多个参数之间用空格分隔
testpaths	配置搜索参数用例的范围
python_files	改变默认的文件搜索规则
python_classes	改变默认类搜索规则
python_functions	改变默认的测试用例搜索规则
markers	用例标记，自定义mark，需要先注册标记，运行时才不会出现warnings

六、pytest配置文件pytest.ini文件

pytest的配置文件通常放在测试目录下，名称为pytest.ini，命令行运行时会使用该配置文件中的配置

```
1 #配置pytest命令行运行参数
2 [pytest]
3 addopts = -s ... # 空格分隔，可添加多个命令行参数 -所有参数均为插件包的参数配置测试
  搜索的路径
4 testpaths = ./scripts # 当前目录下的scripts文件夹 -可自定义
5 #配置测试搜索的文件名称
6 python_files = test*.py
7 #当前目录下的scripts文件夹下，以test开头，以.py结尾的所有文件 -可自定义配置测试搜索的测试
  类名
8 python_classes = Test_*
9
10 #当前目录下的scripts文件夹下，以test开头，以.py结尾的所有文件中，以Test开头的类 -可
  自定义配置测试搜索的测试函数名
11
12 python_functions = test_*
13
```


14 | #当前目录下的scripts文件夹下，以test开头，以.py结尾的所有文件中，以Test开头的类内，以test_开头的方法 -可自定义

七、pytest的常用插件

插件列表网址: https://blog.csdn.net/weixin_45467931/article/details/140868984

八、pytest中conftest.py文件

可以有多个: conftest文件, 最顶层的 conftest, 一般写全局的 fixture

九、pytest中fixtrue装饰器

1、前言

虽然setup和teardown可以执行一些前置和后置操作, 但是这种是针对整个脚本全局生效的
如果有以下场景: 1.用例一需要执行登录操作; 2.用例二不需要执行登录操作; 3.用例三需要执行登录操作, 则setup和teardown则不满足要求。fixture可以让我自定义测试用例的前置条件

2、fixtrue的优势

- 命名方式灵活, 不限于setup和teardown两种命名
- conftest.py可以实现数据共享, 不需要执行import 就能自动找到fixture
- scope=module, 可以实现多个.py文件共享前置
- scope="session" 以实现多个.py 跨文件使用一个 session 来完成多个用例

3、Fixture的调用方式:

```
1 | @pytest.fixture(scope =  
  | "function", params=None, autouse=False, ids=None, name=None)
```

Fixture的作用范围:

取值	范围 说明
function	函数级 每一个函数或方法都会调用
class	函数级 模块级 每一个.py文件调用一次
module	模块级 每一个.py文件调用一次
session	会话级 每次会话只需要运行一次, 会话内所有方法及类, 模块都共享这个方法

默认取值为function (函数级别), 控制范围的排序为: session > module > class > function

4、scope = "function"

参数传入

```
1 | import pytest  
2 | # fixture函数(类中) 作为多个参数传入  
3 | @pytest.fixture()  
4 | def login():  
5 |     print("打开浏览器")  
6 |     a = "account"
```

```

7         return a
8
9     @pytest.fixture()
10    def logout():
11        print("关闭浏览器")
12
13    class TestLogin:
14        #传入login fixture
15        def test_001(self, login):
16            print("001传入了login fixture")
17            assert login == "account"
18
19        #传入logout fixture
20        def test_002(self, logout):
21            print("002传入了logout fixture")
22
23        def test_003(self, login, logout):
24            print("003传入了两个fixture")
25
26        def test_004(self):
27            print("004未传入任何fixture哦")

```

执行结果：

```

1  ===== test session starts
   =====
2  collecting ... collected 4 items
3
4  scripts/test_demo1.py::TestLogin::test_001 打开浏览器
5  001传入了login fixture
6  PASSED
7  scripts/test_demo1.py::TestLogin::test_002 关闭浏览器
8  002传入了logout fixture
9  PASSED
10 scripts/test_demo1.py::TestLogin::test_003 打开浏览器
11 关闭浏览器
12 003传入了两个fixture
13 PASSED
14 scripts/test_demo1.py::TestLogin::test_004 004未传入任何fixture哦
15 PASSED
16
17 ===== 4 passed in 0.01s
   =====

```

发现当函数作为参数传入测试类函数时候，会优先执行，并且优先执行完成函数再运行本函数内的语句

Fixture的相互调用

@pytest.fixture()的函数也可以被相互调用

```

1  import pytest
2
3
4  # fixtrue作为参数，互相调用传入

```

```

5 @pytest.fixture()
6 def account():
7     a = "account"
8     print("第一层fixture")
9     return a
10
11
12 # Fixture的相互调用一定是要在测试类里调用这层fixture才会生效，普通函数单独调用是不生效的
13 @pytest.fixture()
14 def login(account):
15     print("第二层fixture")
16
17
18 class TestLogin:
19     def test_1(self, login):
20         print("直接使用第二层fixture,返回值为{}".format(login))
21
22     def test_2(self, account):
23         print("只调用account fixture,返回值为{}".format(account))

```

执行结果：

```

1  ===== test session starts
2  =====
3  collecting ... collected 2 items
4
5  scripts/test_demo1.py::TestLogin::test_1 第一层fixture
6  第二层fixture
7  直接使用第二层fixture,返回值为None
8  PASSED
9  scripts/test_demo1.py::TestLogin::test_2 第一层fixture
10  只调用account fixture,返回值为account
11  PASSED
12
13  ===== 2 passed in 0.01s
14  =====

```

执行顺序发现他会先执行 (test_1 : account -> login -> test_1)

5、scope = "class"

代码如下：

```

1  import pytest
2  # fixture作用域 scope = 'class'
3  @pytest.fixture(scope='class')
4  def login():
5      print("scope为class")
6
7
8  class TestLogin:
9      def test_1(self, login):
10         print("用例1")
11
12     def test_2(self, login):

```

```

13         print("用例2")
14
15     def test_3(self, login):
16         print("用例3")
17
18
19 if __name__ == '__main__':
20     pytest.main()

```

注意：当测试类内的每一个测试方法都调用了fixture，fixture只在该class下所有测试用例执行前**执行一次**

可以优先调用定义账号密码：

```

1  import pytest
2  @pytest.fixture(scope='class')
3  def login():
4      a = '123'
5      print("输入账号密码登陆")
6
7  class TestLogin:
8      def test_1(self):
9          print("用例1")
10
11      def test_2(self, login):
12          print("用例2")
13
14      def test_3(self, login):
15          print("用例3")
16
17      def test_4(self):
18          print("用例4")
19
20 if __name__ == '__main__':
21     pytest.main()

```

6.scope = "module": 与class相同,

只从.py文件开始引用fixture的位置生效

```

1  import pytest
2  # fixture scope = 'module'
3  @pytest.fixture(scope='module')
4  def login():
5      print("fixture范围为module")
6
7
8  def test_01():
9      print("用例01")
10
11
12 def test_02(login):
13     print("用例02")
14
15

```

```

16 class TestLogin():
17     def test_1(self):
18         print("用例1")
19
20     def test_2(self):
21         print("用例2")
22
23     def test_3(self):
24         print("用例3")
25
26 if __name__ == '__main__':
27     pytest.main()

```

7.fixtrue参数详解-autouse

默认False

若为True，则每个测试函数都会自动调用该fixture,无需传入fixture函数名
由此我们可以总结出调用fixture的三种方式：

- 1.函数或类里面方法直接传fixture的函数参数名称
- 2.使用装饰器@pytest.mark.usefixtures()修饰
- 3.autouse=True自动调用，无需传任何参数，作用范围跟着scope走（谨慎使用）

让我们来看一下，当autouse=True的效果：

当参数为True时候，无论函数是否被调用，他都会随着类的函数执行

8.fixtrue参数详解params

Fixture的可选形参列表，支持列表传入

默认None，每个param的值

fixture都会去调用执行一次，类似for循环

可与参数ids一起使用，作为每个参数的标识，详见ids

被Fixture装饰的函数要调用是采用：Request.param(固定写法)

十、Allure测试报告

安装好插件和软件后使用：

更改配置文件：

```
1 | addopts = -s ... -v --color=yes --alluredir=./result --clean-alluredir
```

运行命令：

```
1 | pytest.main()
```

生成好json代码

运行生成html命令

```
1 | allure generate ./"项目文件夹"/"生成的json文件夹" -o ./report
```

参数拓展：

Allure用例描述		
使用方法	参数值	参数说明
@allure.epic()	epic描述	定义项目、当有多个项目是使用。往下是feature
@allure.feature()	模块名称	用例按照模块区分，有多个模块时给每个起名字
@allure.story()	用例名称	一个用例的描述
@allure.title(用例的标题)	用例标题	一个用例标题
@allure.testcase()	测试用例的连接地址	自动化用例对应的功能用例存放系统的地址
@allure.issue()	缺陷地址	对应缺陷管理系统里边的缺陷地址
@allure.description()	用例描述	对测试用例的详细描述
@allure.step()	操作步骤	测试用例的操作步骤
@allure.severity()	用例等级	blocker、critical、normal、minor、trivial
@allure.link()	定义连接	用于定义一个需要在测试报告中展示的连接
@allure.attachment()	附件	添加测试报告附件

十一、pytest跳过测试用例skip、skipif

1.@pytest.mark.skip

跳过执行测试用例，有可选参数 reason：跳过的原因，会在执行结果中打印

- @pytest.mark.skip可以加在函数上，类上，类方法上
- 如果加在类上面，类里面的所有测试用例都不会执行

```

1  import pytest
2
3
4  @pytest.fixture(autouse=True)
5  def login():
6      print("====登录====")
7
8
9  def test_case01():
10     print("我是测试用例11111")
11
12
13  @pytest.mark.skip(reason="不执行该用例！！因为没写好！！")
14  def test_case02():
15     print("我是测试用例22222")
16
17

```

```

18 class Test1:
19
20     def test_1(self):
21         print("%% 我是类测试用例1111 %%")
22
23     @pytest.mark.skip(reason="不想执行")
24     def test_2(self):
25         print("%% 我是类测试用例2222 %%")
26
27
28 @pytest.mark.skip(reason="类x也可以跳过不执行")
29 class TestSkip:
30     def test_1(self):
31         print("%% 不会执行 %%")

```

@pytest.mark.skip加了这个注解的地方都会按要求跳过，不进行执行

运行结果为：

```

1  ====登录====
2  我是测试用例11111
3  PASSEDSKIPPED (不执行该用例！！因为没写好！！)
4  Skipped: 不执行该用例！！因为没写好！！
5  ====登录====
6  %% 我是类测试用例1111 %%
7  PASSEDSKIPPED (不想执行)
8  Skipped: 不想执行
9  SKIPPED (类也可以跳过不执行)
10 Skipped: 类也可以跳过不执行

```

2.pytest.skip()函数基础使用

作用：在测试用例执行期间强制跳过不再执行剩余内容

类似：在Python的循环里面，满足某些条件则break 跳出循环

```

1 def test_function():
2     n = 1
3     while True:
4         print(f"这是我第{n}条用例")
5         n += 1
6         if n == 5:
7             pytest.skip("我跑五次了不跑了")

```

执行结果为：

```

1 这是我第1条用例
2 这是我第2条用例
3 这是我第3条用例
4 这是我第4条用例
5 SKIPPED (我跑五次了不跑了)
6 Skipped: 我跑五次了不跑了

```

3.pytest.skip(msg="",allow_module_level=False)

当 allow_module_level=True 时，可以设置在模块级别跳过整个模块

```
1 import sys
2 import pytest
3
4 if sys.platform.startswith("win"):
5     pytest.skip("跳过仅限 windows 的测试", allow_module_level=True)
6
7
8 @pytest.fixture(autouse=True)
9 def login():
10     print("====登录====")
11
12
13 def test_case01():
14     print("我是测试用例11111")
```

所以当前模块都不能执行

4. @pytest.mark.skipif(condition, reason="xxx")

方法:

skipif(condition, reason=None)

参数:

condition: 跳过的条件，必传参数

reason: 标注原因，必传参数

使用方法:

@pytest.mark.skipif(condition, reason="xxx")

代码演示:

```
1
2 import pytest
3 class Test_ABC:
4     def setup_class(self):
5         print("----->setup_class")
6     def teardown_class(self):
7         print("----->teardown_class")
8     def test_a(self):
9         print("----->test_a")
10        assert 1
11    @pytest.mark.skipif(condition=2>1,reason = "跳过该函数") # 跳过测试函数
test_b
12    def test_b(self):
13        print("----->test_b")
14        assert 0
```

执行结果:


```

1 ----->setup_class
2 ----->test_a
3 PASSEDSKIPPED (跳过该函数)
4 Skipped: 跳过该函数

```

5.跳过标记

- 可以将 `pytest.mark.skip` 和 `pytest.mark.skipif` 赋值给一个标记变量
- 在不同模块之间共享这个标记变量
- 若有多个模块的测试用例需要用到相同
- 的 `skip` 或 `skipif`，可以用一个单独的文件去管理这些通用标记，然后适用于整个测试用例集

代码演示：

```

1 # 标记
2 skipmark = pytest.mark.skip(reason="不能在window上运行=====")
3 skipifmark = pytest.mark.skipif(sys.platform == 'win32', reason="不能在window
  上运行啦啦啦=====")
4
5
6 @skipmark
7 class TestSkip_Mark(object):
8
9     @skipifmark
10     def test_function(self):
11         print("测试标记")
12
13     def test_def(self):
14         print("测试标记")
15
16
17 @skipmark
18 def test_skip():
19     print("测试标记")
20

```

执行效果：

```

1 SKIPPED
2 Skipped: 不能在window上运行啦啦啦=====
3 SKIPPED (不能在window上运行=====)
4 Skipped: 不能在window上运行=====
5 SKIPPED (不能在window上运行=====)
6 Skipped: 不能在window上运行=====

```

6.pytest.importorskip

`pytest.importorskip(modname: str, minversion: Optional[str] = None, reason: Optional[str] = Nonse)`

作用：如果缺少某些导入，则跳过模块中的所有测试

参数列表

- `modname`：模块名

- minversion: 版本号
- reason: 跳过原因, 默认不给也行

代码演示:

```
1 pexpect = pytest.importorskip("pexpect", minversion="0.3")
2 @pexpect
3 def test_import():
4     print("test")
```

7.使用自定义标记 mark

前言

- pytest可以支持自定义标记, 自定义标记可以把一个web项目划分为多个模块, 然后指定模块名称执行
- 譬如我们可以标明哪些用例在window上执行, 哪些用例在mac上执行, 在运行的时候指定mark就行

代码演示:

```
1
2 import pytest
3
4 @pytest.mark.model
5 def test_model_a():
6     print("执行test_model_a")
7
8 @pytest.mark.regular
9 def test_regular_a():
10    print("test_regular_a")
11
12 @pytest.mark.model
13 def test_model_b():
14    print("test_model_b")
15
16 @pytest.mark.regular
17 class TestClass:
18     def test_method(self):
19         print("test_method")
20
21 def testnoMark():
22     print("testnoMark")
```

如何避免warnings

创建一个 pytest.ini 文件

加上自定义mark

pytest.ini 需要和运行的测试用例同一个目录, 或在根目录下作用于全局

```
[pytest]
markers =
model: this is model mark
```

如果不想标记 model 的用例

```
pytest -s -m " not model" test_one.py
```

如果想执行多个自定义标记的用例

```
pytest -s -m "model or regular" 08_mark.py
```

十二、pytest参数化 @pytest.mark.parametrize

pytest允许在多个级别启用测试化参数：

- 1) pytest.fixture()允许fixture有参数化功能
- 2) pytest.mark.parametrize 允许在测试函数和类中定义多组参数和fixtures
- 3) pytest_generate_tests允许定义自定义参数化方案或扩展

def parametrize(self,argnames, argvalues, indirect=False, ids=None, scope=None):

参数的含义

参数	含义	演示
argnames	参数值列表 格式：字符串"arg1,arg2,arg3"	@pytest.mark.parametrize("name,pwd", [("yy1", "123"), ("yy2", "123")])
argvalues:	参数值列表 格式：必须是列表，如：[val1,val2,val3]	-如果只有一个参数，里面则是值的列表如： @pytest.mark.parametrize("username", ["yy", "yy2", "yy3"]) -如果有多个参数例，则需要用元组来存放值，一个元组对应一组参数的值，如： @pytest.mark.parametrize("name,pwd", [("yy1", "123"), ("yy2", "123"), ("yy3", "123")])
ids:	含义：用例的id 格式：传一个字符串列表	作用：可以标识每一个测试用例，自定义测试数据结果的显示，为了增加可读性
indirect:		作用：如果设置成 True，则把传进来的参数当函数执行，而不是一个参数（下一篇文章即讲解

例如：

```
1 | @pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6),  
2 |   ("6*9", 42)])  
3 | def test_eval(test_input, expected):  
4 |     print(f"测试数据{test_input},期望结果{expected}")  
   assert eval(test_input) == expected
```

运行结果为：

```

1
2 test_demo3.py::test_eval[3+5-8] 测试数据3+5,期望结果8
3 PASSED
4 test_demo3.py::test_eval[2+4-6] 测试数据2+4,期望结果6
5 PASSED
6 test_demo3.py::test_eval[6*9-54] 测试数据6*9,期望结果54
7 PASSED

```

1.函数数据参数化

parametrize(argnames, argvalues, indirect=False, ids=None, scope=None)

参数的含义:

参数	含义	
argnames	参数名	
argvalues	参数对应值, 类型必须为list	当参数为一个时格式: [value] 当参数个数大于一个时, 格式为: [(param_value1,param_value2...), (param_value1,param_value2...)] 使用方法:

使用方法:

@pytest.mark.parametrize(argnames,argvalues)

参数值为N个, 测试方法就会运行N次

代码测试:

```

1 import pytest
2
3 class Test_ABC:
4     def setup_class(self):
5         print("----->setup_class")
6     def teardown_class(self):
7         print("----->teardown_class")
8     @pytest.mark.parametrize("a", [3, 6]) # a参数被赋予两个值, 函数会运行两遍
9     def test_a(self, a): # 参数必须和parametrize里面的参数一致
10         print("test data:a=%d" % a)
11         assert a % 3 == 0

```

执行结果:

```

1 test_demo3.py::Test_ABC::test_a[3] ----->setup_class
2 test data:a=3
3 PASSED
4 test_demo3.py::Test_ABC::test_a[6] test data:a=6
5 PASSED----->teardown_class

```

多个参数

代码演示:

```

1 import pytest
2 class Test_demo1:
3     def setup_class(self):
4         print("----->setup_class")
5     def teardown_class(self):
6         print("----->teardown_class")
7     @pytest.mark.parametrize("a,b",[(1,2),(0,3)]) # 参数a,b均被赋予两个值，函数会
运行两遍
8     def test_a(self,a,b): # 参数必须和parametrize里面的参数一致
9         print("test data:a=%d,b=%d"%(a,b))
10        assert a+b == 3

```

执行结果：

```

1
2 test_demo3.py::Test_ABC::test_a[1-2] ----->setup_class
3 test data:a=1,b=2
4 PASSED
5 test_demo3.py::Test_ABC::test_a[0-3] test data:a=0,b=3
6 PASSED----->teardown_class
7

```

函数返回值作为参数

代码演示：

```

1 import pytest
2 def return_test_data():
3     return [(1,2),(0,3)]
4 class Test_ABC:
5     def setup_class(self):
6         print("----->setup_class")
7     def teardown_class(self):
8         print("----->teardown_class")
9     @pytest.mark.parametrize("a,b",return_test_data()) # 使用函数返回值的形式传
入参数值
10    def test_a(self,a,b):
11        print("test data:a=%d,b=%d"%(a,b))
12        assert a+b == 3

```

执行结果：

```

1 test_demo3.py::Test_ABC::test_a[1-2] ----->setup_class
2 test data:a=1,b=2
3 PASSED
4 test_demo3.py::Test_ABC::test_a[0-3] test data:a=0,b=3
5 PASSED----->teardown_class

```

“笛卡尔积”，多个参数化装饰器

代码演示：

```

1
2 import pytest
3
4 # 笛卡尔积，组合数据
5 data_1 = [1, 2, 3]
6 data_2 = ['a', 'b']
7
8
9 @pytest.mark.parametrize('a', data_1)
10 @pytest.mark.parametrize('b', data_2)
11 def test_parametrize_1(a, b):
12     print(f'笛卡尔积 测试数据为 : {a}, {b}')
```

执行结果：

```

1 test_demo3.py::test_parametrize_1[a-1] 笛卡尔积 测试数据为 : 1, a
2 PASSED
3 test_demo3.py::test_parametrize_1[a-2] 笛卡尔积 测试数据为 : 2, a
4 PASSED
5 test_demo3.py::test_parametrize_1[a-3] 笛卡尔积 测试数据为 : 3, a
6 PASSED
7 test_demo3.py::test_parametrize_1[b-1] 笛卡尔积 测试数据为 : 1, b
8 PASSED
9 test_demo3.py::test_parametrize_1[b-2] 笛卡尔积 测试数据为 : 2, b
10 PASSED
11 test_demo3.py::test_parametrize_1[b-3] 笛卡尔积 测试数据为 : 3, b
12 PASSED
```

参数化，标记数据

代码演示：

```

1 # 标记参数化
2 @pytest.mark.parametrize("test_input,expected", [
3     ("3+5", 8),
4     ("2+4", 6),
5     pytest.param("6 * 9", 42, marks=pytest.mark.xfail),
6     pytest.param("6*6", 42, marks=pytest.mark.skip)
7 ])
8 def test_mark(test_input, expected):
9     assert eval(test_input) == expected
```

十三、pytest标记为失败函数和失败重试

1.标记为预期失败的函数

`xfail(condition=None, reason=None, raises=None, run=True, strict=False)`

参数	含义	使用方法
condition	预期失败的条件，必传参数	
reason	失败的原因，必传参数	

参数	含义	使用方法
		@pytest.mark.xfail(condition, reason="xx")

代码演示：

```

1
2 import pytest
3 class Test_ABC:
4     def setup_class(self):
5         print("----->setup_class")
6     def teardown_class(self):
7         print("----->teardown_class")
8     def test_a(self):
9         print("----->test_a")
10        assert 1
11    @pytest.mark.xfail(2 > 1, reason="标注为预期失败") # 标记为预期失败函数test_b
12        def test_b(self):
13            print("----->test_b")
14            assert 0

```

执行结果：

```

1 ----->setup_class
2 ----->test_a
3 PASSED----->test_b
4 XFAIL (标注为预期失败)

```

失败后重试

```

1 class TestClass:
2     def test_case(self):
3         print("第一个测试用例：计算3+3的和")
4         sum = 3+3
5         assert 9 == sum

```

执行效果：

```

1 第一个测试用例：计算3+3的和
2 RERUN
3 test_demo3.py::TestClass::test_case 第一个测试用例：计算3+3的和
4 RERUN
5 test_demo3.py::TestClass::test_case 第一个测试用例：计算3+3的和
6 RERUN
7 test_demo3.py::TestClass::test_case 第一个测试用例：计算3+3的和

```

十四、执行顺序

需要安装pytest-ordering插件

代码演示：

```

1 import pytest

```

```

2
3 class TestOrder:
4     def test_case01(self):
5         print("第一条测试用例")
6         @pytest.mark.run(order=-1)
7     def test_case02(self):
8         print("第二条测试用例")
9         @pytest.mark.run(order=1)
10    def test_case03(self):
11        print("第三条测试用例")
12        @pytest.mark.run(order=0)
13    def test_case04(self):
14        print("第四条测试用例")
15        @pytest.mark.run(order=-2)
16    def test_case05(self):
17        print("第五条测试用例")
18
19 if __name__ == '__main__':
20     pytest.main(["-s", "test_demo3.py"])

```

执行效果：

```

1 test_demo3.py::TestOrder::test_case04 第四条测试用例
2 PASSED
3 test_demo3.py::TestOrder::test_case03 第三条测试用例
4 PASSED
5 test_demo3.py::TestOrder::test_case01 第一条测试用例
6 PASSED
7 test_demo3.py::TestOrder::test_case05 第五条测试用例
8 PASSED
9 test_demo3.py::TestOrder::test_case02 第二条测试用例
10 PASSED

```

PO设计模式

简要介绍：页面为类，元素为属性，提高代码的复用性|可维护性，PO模式会造成测试代码结构比较复杂

三层架构：

- base：封装每个页面的属性和方法 如：元素，输入，点击
- pageobject：每个页面定义一个类。类中包含元素和操作元素的方法
- testcase：包含业务流程及测试数据

例如初始化启动页面类：


```

1 from selenium import webdriver
2
3 class DriverManager():
4
5     def __init__(self, name='chrome'):
6         if name == 'chrome':
7             self.d = webdriver.Chrome()
8         elif name == 'firefox':
9             self.d = webdriver.Firefox()
10
11
12 # 初始化driver对象
13 driver = DriverManager().d

```

初始化元素选择操作类：

```

1 from selenium.webdriver import ActionChains
2 from selenium.webdriver.common.by import By
3
4 from drivers.driver_manager import driver
5
6
7 class BasePage():
8
9     # 打开指定url的网页
10    def open(self, url):
11        driver.get(url)
12
13    # 定位元素
14    def find_element(self, type, value):
15        if type == 'id':
16            el = driver.find_element(By.ID, value)
17        elif type == 'name':
18            el = driver.find_element(By.NAME, value)
19        elif type == 'tag':
20            el = driver.find_element(By.TAG_NAME, value)
21        elif type == 'class':
22            el = driver.find_element(By.CLASS_NAME, value)
23        elif type == 'link':
24            el = driver.find_element(By.LINK_TEXT, value)
25        elif type == 'plink':
26            el = driver.find_element(By.PARTIAL_LINK_TEXT, value)
27        elif type == 'css':
28            el = driver.find_element(By.CSS_SELECTOR, value)
29        elif type == 'xpath':
30            el = driver.find_element(By.XPATH, value)
31        else:
32            print('输入的定位方式有误')
33        return el
34
35    # 点击目标元素
36    def click_element(self, type, value):
37        self.find_element(type, value).click()
38
39    # 向目标元素输入文本

```

```

40     def input_data(self, type, value, text):
41         self.find_element(type, value).send_keys(text)
42
43     # 鼠标移动到目标元素上悬停
44     def move_mouse_to_element(self, type, value):
45
46         ActionChains(driver).move_to_element(self.find_element(type, value)).pause(2
47         ).perform()
48
49     # 切换框架
50     def switch_to_frame(self, id_name_or_element):
51         driver.switch_to.frame(id_name_or_element)
52
53     # 切换窗口
54     def switch_to_window(self, title):
55         handles = driver.window_handles
56         for handle in handles:
57             driver.switch_to.window(handle)
58             if driver.title == title:
59                 break

```

之后在类中调用对应的方法从而实现PO模式的转变：

```

1  from pages.base_page import BasePage
2  from pages.reg_page import RegPage
3  from pages.login_page import LoginPage
4
5  class MainPage(BasePage):
6      url = 'http://192.168.32.137/opencart/'
7
8      会员中心 = {'type': 'link', 'value': '会员中心'}
9      会员注册 = {'type': 'link', 'value': '会员注册'}
10     会员登录 = {'type': 'link', 'value': '会员登录'}
11     搜索框 = {'type': 'name', 'value': 'search'}
12     搜索按钮 = {'type': 'xpath', 'value': '//*[@id="search"]/button'}
13
14     def open_main_page(self):
15         self.open(self.url)
16
17     def go_to_reg(self):
18         self.move_mouse_to_element(self.会员中心['type'], self.会员中心
19         ['value'])
20         self.click_element(self.会员注册['type'], self.会员注册['value'])
21         return RegPage()
22
23     def go_to_login(self):
24         self.move_mouse_to_element(self.会员中心['type'], self.会员中心
25         ['value'])
26         self.click_element(self.会员登录['type'], self.会员登录['value'])
27         return LoginPage()
28
29     def search(self):
30         self.input_data(self.搜索框['type'], self.搜索框['value'], 'HAXA')
31         self.click_element(self.搜索按钮['type'], self.搜索按钮['value'])

```

主要的意思为：

- 建立对应的工具类，简化代码，
- 然后将代码分为：对象层 -> 操作层 -> 业务层

几个层级关系与含义：

（一）对象库层

Base（基类）：封装page 页面一些公共的方法，如初始化方法、查找元素方法、点击元素方法、输入方法、获取文本方法、截图方法等

（二）操作层

page(页面对象)：封装对元素的操作，一个页面封装成一个对象；

思路：继承 Base;

模块名：实际操作模块名称 如：page_login.py

页面对象类名：以大驼峰方法将模块名抄进来，有下划线去掉下划线，如class PageLogin(Base);

方法：涉及元素，将每个元素操作单独封装成一个操作方法；

组装：根据需求组装以上操作步骤。

（三）业务层

scripts（业务层）：将一个或多个操作组合起来完成一个业务功能。比如登录：需要输入帐号、密码、点击登录三个操作。

思路：导包调用 page 页面

模块：test+实际操作模块名称 如：test_login.py

测试业务类名：以大驼峰方法将模块名抄进来，有下划线去掉下划线,如class TestLogin(unittest.TestCase):

方法：

① 初始化方法 setUp() 注：在 unittest 框架中不能使用 def init()初始化方法;

#实例化 页面对象

#前置操作 如：打开网址等

② 结束方法 tearDown

#关闭驱动

③ 测试方法

#根据要操作的业务来实现

代码的变化：

1.普通Py代码+selenium:

```
1
2 """
3 登录功能-账号不存在
4 """
5 from selenium import webdriver
6 # 创建浏览器驱动对象，并完成初始化操作
7 driver = webdriver.Firefox()
8 driver.maximize_window()
9 driver.implicitly_wait(10)
10 driver.get("http://localhost")
11
12 # 点击首页的‘登录’链接，进入登录页面
13 # 输入用户名
14 driver.find_element_by_id("username").send_keys("11234567843")
15 # 输入密码
```

```

16 driver.find_element_by_name("password").send_keys("23615115")
17 # 点击登录按钮
18 driver.find_element_by_name("sbtbutton").click()
19 # 获取提示信息
20 msg = driver.find_element_by_class_name(".layui-layer-padding").text
21 print("msg=", msg)
22 expect_result = "账号不存在!"
23 try:
24     # 断言
25     assert expect_result==msg
26     print("result:", result)
27 except AssertionError:
28     # 截图
29
30     driver.get_screenshot_as_file("./{}.png".format(time.strftime("%Y_%m_%d_%H_%M_%S")))
31     print("result:", result)
32     raise
33 # 关闭驱动对象
34 driver.quit()

```

2.unittest框架加入

```

1  # -*-coding:utf-8 -*-
2  # Auothor:yue_luo
3  import time
4  import unittest
5  from time import sleep
6  from selenium import webdriver
7  class TestLogin(unittest.TestCase):
8      """
9      对登录模块的功能进行测试
10     """
11     driver = None
12     @classmethod
13     def setUpClass(cls):
14         cls.driver = webdriver.Chrome()
15         cls.driver.maximize_window()
16         cls.driver.implicitly_wait(10)
17         cls.driver.get("http://www.tpshop.com")
18         cls.driver.find_element_by_link_text("登录").click()
19     @classmethod
20     def tearDownClass(cls):
21         cls.driver.quit()
22     # 账号不存在
23     def test_login_username_is_error(self):
24         # 输入用户名
25         username =self.driver.find_element_by_id("username")
26         username.clear()
27         username.send_keys("13099999999")
28         # 输入密码
29         password=self.driver.find_element_by_id("password")
30         password.clear()
31         password.send_keys("123456")
32         # 输入验证码

```

```
33     verify_code = self.driver.find_element_by_id("verify_code")
34     verify_code.clear()
35     verify_code.send_keys("8888")
36     # 点击‘登录’
37     self.driver.find_element_by_name("sbbutton").click()
38     # 断言提示信息
39     msg = self.driver.find_element_by_class_name("layui-layer-
content").text
40     print("msg=", msg)
41     try:
42         self.assertIn("账号不存在!", msg)
43         sleep(3)
44     except AssertionError:
45
46         self.driver.get_screenshot_as_file("./{}.png".format(time.strftime("%Y_%m_%
d_%H_%M_%S")))
47         raise
48     finally:
49         self.driver.find_element_by_css_selector(".layui-layer-
btn0").click()
50
51     def test_login_password_is_error(self):
52         # 输入用户名
53         username = self.driver.find_element_by_id("username")
54         username.clear()
55         username.send_keys("17864307785")
56         # 输入密码
57         password = self.driver.find_element_by_id("password")
58         password.clear()
59         password.send_keys("error")
60         # 输入验证码
61         verify_code = self.driver.find_element_by_id("verify_code")
62         verify_code.clear()
63         verify_code.send_keys("8888")
64         # 点击‘登录’
65         self.driver.find_element_by_name("sbbutton").click()
66         # 断言提示信息
67         msg = self.driver.find_element_by_class_name("layui-layer-
content").text
68         print("msg=", msg)
69         try:
70             self.assertIn("密码错误!", msg)
71             sleep(3)
72         except AssertionError:
73
74             self.driver.get_screenshot_as_file("./{}.png".format(time.strftime("%Y_%m_%
d_%H_%M_%S")))
75             raise
76         finally:
77             self.driver.find_element_by_css_selector(".layui-layer-
btn0").click()
```

3.PO模式的加入

-对象库层: base.py

```
1  """base.py"""
2  import time
3  from selenium import webdriver
4  from selenium.webdriver.support.wait import WebDriverWait
5
6  class Base:
7
8      # 初始化
9      def __init__(self, driver):
10         self.driver = driver
11
12     # 查找元素方法
13     def base_find_element(self, loc, timeout=30, poll_frequency=0.5):
14         return
15         WebDriverWait(driver=self.driver, timeout=timeout, poll_frequency=poll_frequency).until(lambda x: x.find_element(*loc))
16
17     # 点击方法
18     def base_click(self, loc):
19         self.base_find_element(loc).click()
20
21     # 输入方法
22     def base_input(self, loc, value):
23         element = self.base_find_element(loc)
24         element.clear()
25         element.send_keys(value)
26
27     # 获取文本方法
28     def base_get_text(self, loc):
29         msg = self.base_find_element(loc).text
30         return msg
31
32     # 截图
33     def base_get_image(self, ):
34         self.driver.get_screenshot_as_file("./{}.png".format(time.strftime("%Y_%m_%d_%H_%M_%S")))
35
```

-操作层: __init__.py 和 page_login.py

__init__.py

```
1  """__init__.py"""
2  # -*-coding:utf-8 -*-
3  # Auothor:yue_luo
4  from selenium.webdriver.common.by import By
```

```
5
6 """以下为服务器域名配置地址"""
7 url = "http://www.tpshop.com"
8
9 """以下为登录页面配置信息"""
10 # 登录链接
11 login_link = By.PARTIAL_LINK_TEXT , "登录"
12 # 用户名
13 login_username = By.ID, "username"
14 # 密码
15 login_pwd = By.NAME, "password"
16 # 验证码
17 login_verify_code = By.CSS_SELECTOR, "#verify_code"
18 # 登录按钮
19 login_btn = By.NAME, "sbtbutton"
20 # 获取异常文本信息
21 login_err_info = By.CLASS_NAME, "layui-layer-content"
22 # 异常提示框 确定按钮
23 login_err_btn_ok = By.CSS_SELECTOR, ".layui-layer-btn0"
```

page_login.py

```
1 """page_login.py"""
2 from v3 import page
3 from v3.base.base import Base
4
5 class PageLogin(Base):
6
7     # 点击登录链接
8     def page_click_login_link(self):
9         self.base_click(page.login_link)
10
11     # 输入用户名
12     def page_input_username(self, username):
13         self.base_input(page.login_username, username)
14
15     # 输入密码
16     def page_input_passwd(self, pwd):
17         self.base_input(page.login_pwd, pwd)
18
19     # 输入验证码
20     def page_input_verify_code(self, code):
21         self.base_input(page.login_verify_code, code)
22
23     # 点击登录
24     def page_click_login_btn(self):
25         self.base_click(page.login_btn)
26
27     # 获取异常提示信息
28     def page_get_error_info(self):
29         return self.base_get_text(page.login_err_info)
```

```

30
31     # 点击异常信息框 确定
32     def page_click_error_btn_ok(self):
33         self.base_click(page.login_err_btn_ok)
34
35     # 截图
36     def page_get_screenshot(self):
37         self.base_get_image()
38
39     # 组合业务方法
40     def page_login(self, username, pwd, code):
41         self.page_input_username(username)
42         self.page_input_passwd(pwd)
43         self.page_input_verify_code(code)
44         self.page_click_login_btn()

```

-业务层: test_login.py

```

1  """test_login.py"""
2  # 导包
3  import unittest
4  from time import sleep
5  from parameterized import parameterized
6
7  from v3.base.get_driver import GetDriver
8  from v3.page.page_login import PageLogin
9
10 # 参数化
11 def get_data():
12     return [("13099999999", "123456", "8888", "账号不存在!"),
13            ("17864307785", "error", "8888", "密码错误!")]
14
15 # 新建测试类并继承
16 class TestLogin(unittest.TestCase):
17
18     # setUp
19     @classmethod
20     def setUpClass(cls):
21         # 实例化driver并获取
22         cls.driver = GetDriver().get_driver()
23         # 实例化 获取登录对象
24         cls.login = PageLogin(cls.driver)
25         # 点击登录链接
26         cls.login.page_click_login_link()
27
28     # tearDown
29     @classmethod
30     def tearDownClass(cls):
31         # 关闭driver浏览器驱动对象
32         GetDriver().quit_driver()
33
34     # 登录测试方法
35     @parameterized.expand(get_data())
36     def test_login(self, username, pwd, code, expect_result):

```



```

37         # 调用登录方法
38         self.login.page_login(username,pwd,code)
39         # 获取登录提示信息
40         result = self.login.page_get_error_info()
41         print("result: ",result)
42         sleep(1)
43         #断言
44         try:
45             self.assertIn(result,expect_result)
46             sleep(1)
47         except AssertionError:
48             #截图
49             self.login.page_get_screenshot()
50             raise
51         finally:
52             # 点击提示框确定按钮
53             self.login.page_click_error_btn_ok()
54             sleep(1)

```

4.PO模式+数据驱动

-对象库层：base.py和get_driver.py

```

1  """base.py"""
2  # -*-coding:utf-8 -*-
3  # Author:yue_luo
4  from selenium.webdriver.support.wait import WebDriverWait
5  import time
6
7
8  class Base:
9      def __init__(self,driver):
10         self.driver =driver
11
12         def base_find_element(self,loc,timeout=30,poll=0.5):
13             return
14             WebDriverWait(self.driver,timeout=timeout,poll_frequency=poll).until(lambda
15             x:x.find_element(*loc))
16
17         def base_click(self,loc):
18             self.base_find_element(loc).click()
19
20         # 获取value属性方法
21         def base_get_value(self,loc):
22             return self.base_find_element(loc).get_attribute("value")
23
24         def base_get_img(self):
25
26         self.driver.get_screenshot_as_file("./{}.png".format(time.strftime("%Y_%m_%
27         d_%H_%M_%S")))

```

get_driver.py

```

1  """get_driver.py"""
2  # -*-coding:utf-8 -*-
3  # Auothor:yue_luo
4  from selenium import webdriver
5  from PO import page
6
7  class GetDriver:
8
9      driver = None
10     @classmethod
11     def get_driver(cls):
12         if cls.driver == None:
13             cls.driver = webdriver.Chrome()
14             cls.driver.maximize_window()
15             cls.driver.get(page.url)
16             return cls.driver
17
18     @classmethod
19     def quit_driver(cls):
20         if cls.driver:
21             cls.driver.quit()
22             """置空,一定要置空"""
23             cls.driver = None

```

-操作层: __init__.py 和 page_login.py

```

1  """__init__.py"""
2  # -*-coding:utf-8 -*-
3  # Auothor:yue_luo
4  from selenium.webdriver.common.by import By
5
6  """以下为服务器域名配置地址"""
7  url = "http://cal.apple886.com/"
8
9  """以下为登录页面配置信息"""
10 # cal_num = By.CSS_SELECTOR, "#simple"
11 cal_add = By.CSS_SELECTOR, "#simpleAdd"
12 cal_eq = By.CSS_SELECTOR, "#simpleEqual"
13 cal_result = By.CSS_SELECTOR, "#resultIpt"
14 cal_clear = By.CSS_SELECTOR, "#simpleClearAllBtn"

```

```

1  """page_cal.py"""
2  # -*-coding:utf-8 -*-
3  # Auothor:yue_luo
4  from time import sleep
5
6  from PO import page
7  from PO.base.base import Base
8  from selenium.webdriver.common.by import By
9
10 class PageCal(Base):

```

```

11     def page_click_num(self,num):
12         s = By.CSS_SELECTOR , "#simple2"
13         self.base_find_element(s)
14         for n in str(num):
15             loc = By.CSS_SELECTOR, "#simple{}".format(n)
16             self.base_click(loc)
17
18     def page_click_add(self):
19         self.base_click(page.cal_add)
20
21     def page_click_eq(self):
22         self.base_click(page.cal_eq)
23     def page_get_value(self):
24         return self.base_get_value(page.cal_result)
25     def page_click_clear(self):
26         self.base_click(page.cal_clear)
27     def page_get_img(self):
28         self.base_get_img()
29
30     def page_add_cal(self,a,b):
31         self.page_click_num(a)
32         sleep(0.5)
33         self.page_click_add()
34         self.page_click_num(b)
35         sleep(0.5)
36         self.page_click_eq()
37         sleep(0.5)

```

- 业务层: test_login.py

```

1  """test_cal.py"""
2  # -*-coding:utf-8 -*-
3  # Auothor:yue_luo
4  import unittest
5
6  from PO.base.get_driver import GetDriver
7  from PO.page.page_cal import PageCal
8  from parameterized import parameterized
9
10 from PO.tool.read_json import read_json
11
12
13 def get_data():
14     data = read_json("cal.json")
15     # print(data)
16     """
17     期望数据格式: [(1234, 1234, 2468), (1, 2, 3), (123445, 223265, 32626)]
18     """
19     list = []
20     for n in data.values():
21         list.append((n["a"], n["b"], n["expect_result"]))
22     return list
23
24 class TestCal(unittest.TestCase):

```

```

25     # driver = None
26     @classmethod
27     def setUpClass(cls):
28         cls.driver = GetDriver().get_driver()
29         cls.cal = PageCal(cls.driver)
30
31     @classmethod
32     def tearDownClass(cls):
33         GetDriver().quit_driver()
34     @parameterized.expand(get_data())
35     def test_add_cal(self, a, b, expect_result):
36         self.cal.page_add_cal(a, b)
37         result = self.cal.page_get_value()
38         print("result:", result)
39         print("ex_result:", expect_result)
40         #136465446
41         try:
42             self.assertEqual(result, str(expect_result))
43         except:
44             self.cal.page_get_img()
45             raise
46         finally:
47             self.cal.page_click_clear()

```

-数据 (cal.json)

```

1  {
2      "cal_001":{"a":1,"b":2,"expect_result":3},
3      "cal_004":{"a":1,"b":22,"expect_result":23},
4      "cal_005":{"a":1,"b":23,"expect_result":24},
5      "cal_006":{"a":1,"b":42,"expect_result":43},
6      "cal_008":{"a":1,"b":52,"expect_result":53},
7      "cal_007":{"a":1,"b":522,"expect_result":523},
8      "cal_002":{"a":123445,"b":223265,"expect_result":32626},
9      "cal_003":{"a":1234,"b":1234,"expect_result":2468}
10 }

```

-工具类 (read_json.py)

```

1  import json
2
3  def read_json(filename):
4      filepath = "../data/" + filename
5      with open(filepath, "r", encoding="utf8") as f:
6          return json.load(f)

```