Web自动化测试

1.元素的交互

1)点击

```
element.click()
```

2)清空

```
element.clear()
```

3)输入

```
element.send_keys("输入的内容")
```

2.元素的定位

1)css选择器

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

2）id选择器

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

3）class选择器

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

4）标签选择器

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

5）文本选择器

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

6）xPhan选择器

```
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

```
element = browser.find_element_by_xpath("提交")
time.sleep(2)
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

```
element = browser.find_element_by_xpath("提交")
#多少秒内找到元素立刻执行，找不到则报错
element.implicitly_wait(10)
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

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

5.日期/评分交互

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

6.上传交互

```
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

```
#关闭浏览器的当前页
driver.close()
#关闭浏览器
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
sccreenshot_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
locatio_once_scrolled_into_view	滚动可见

8.三大等待

```
#强制等待
time.sleep(2)
#隐式等待
driver.implicitly_wait(10)
#显示等待
ele = WebDriverWait(driver,10).until(
    lambda _:driver.find_element(
        By.XPATH,"/...路径")
)
```

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");**

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

```
webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL,"A");
        sleep(1000);
        // ctrl + X

webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL,"X");
        sleep(1000);
        // ctrl + V

webDriver.findElement(By.cssSelector("#kw")).sendKeys(Keys.CONTROL,"V");
        sleep(1000);

    }
```

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.默认的测试类：

```
import unittest


class MyTestCase(unittest.TestCase):
    def test_something(self):
        self.assertEqual(True, False)  # add assertion here


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

2.第一个测试用例

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

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
```

```python
class TestDemo2(unittest.TestCase):
    def test_case2(self):
        print("test_case2")


if __name__ == '__main__':
    suite = unittest.TestSuite()
    suite.addTest(TestDemo("test_case1"))
    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

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

注意:

```python
# 1、导包
# 2、实例化（创建对象）套件对象
# 3、使用套件对象添加用例方法
# 4、实例化对象运行
# 5、使用运行对象去执行套件对象

import unittest


class TestDemo(unittest.TestCase):
    def test_case1(self):
        print("test_case1")

    def test_case2(self):
        print("test_case2")


class Demo(unittest.TestCase):
    def test_case3(self):
        print("test_case3")

    def test_case4(self):
        print("test_case4")


suite = unittest.TestSuite()

# 将一个测试类中的所有方法进行添加
# 套件对象.addTest(unittest.makeSuite(测试类名))
suite.addTest(TestDemo())
suite.addTest(Demo())
if __name__ == '__main__':
    # 4、实例化运行对象
```

```
    runner = unittest.TextTestRunner()
    # 5、使用运行对象去执行套件对象
    # 运行对象.run(套件对象)
    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. 导包
# 2. 实例化测试加载对象并添加用例 ---> 得到的是 suite 对象
# 3. 实例化 运行对象
# 4. 运行对象执行套件对象

import unittest

# 实例化测试加载对象并添加用例 ---> 得到的是 suite 对象
# unittest.defaultTestLoader.discover('用例所在的路径', '用例的代码文件名')
# 测试路径：相对路径
# 测试文件名：可以使用 * 通配符，可以重复使用
suite = unittest.defaultTestLoader.discover('./Case', 'cs*.py')
runner = unittest.TextTestRunner()
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：

```
TestSuite与TestLoader区别：
    共同点：都是测试套件
    不同点：实现方式不同
        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test开头的方法
        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）

```
# 初始化
def setUp(self):
    # 每个测试方法执行之前执行的函数

# 释放
def tearDown(self):
    # 每个测试方法执行之后执行的函数
```

代码补充完整：

```
import unittest


class Demo(unittest.TestCase):
    def test_case3(self):
        print("Test1:test_case3")

    def test_case4(self):
        print("Test1:test_case4")

    # 初始化
    def setUp(self):
        # 每个测试方法执行之前执行的函数
        print("Demo:setUp")

    # 释放
    def tearDown(self):
        # 每个测试方法执行之后执行的函数
        print("Demo:tearDown")
```

运行结果为：

```
    Demo:setUp
    Test1:test_case3
    Demo:tearDown
    Demo:setUp
    Test1:test_case4
    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

```
import unittest


class Demo(unittest.TestCase):
```

```
    def test_case3(self):
        print("Test1:test_case3")

    def test_case4(self):
        print("Test1:test_case4")

    @classmethod
    def setUpClass(cls):
        print('-----------类加载前')

    @classmethod
    def tearDownClass(cls):
        print('------------类加载后')
```

运行结果：

```
-----------类加载前
Test1:test_case3
Test1:test_case4
------------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：

```
self.assertEqual(ex1, ex2) # 判断ex1 是否和ex2 相等
self.assertIn(ex1, ex2) #  ex2是否包含 ex1   注意：所谓的包含不能跳字符
self.assertTrue(ex) #  判断ex是否为True

重点讲前两个assertEqual 和 assertIn
方法：
assertEqual：self.assertEqual(预期结果，实际结果） 判断的是预期是否相等实际
assertIn：self.assertIn(预期结果，实际结果） 判断的是预期是否包含实际中
assertIn('admin', 'admin') # 包含
assertIn('admin', 'adminnnnnnnn') # 包含
assertIn('admin', 'aaaaaadmin') # 包含
assertIn('admin', 'aaaaaadminnnnnnn') # 包含
assertIn('admin', 'adddddddmin') # 不是包含
```

断言的使用：

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

assertEqual：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

```python
import unittest
from HTMLTestRunner import HTMLTestRunner

#当前路径
test_dir = './'

```

```
#iot_*.py表示iot_开头的所有测试用例
discover = unittest.defaultTestLoader.discover(test_dir, pattern='*.py')

#报告存放的路径
fp = open("./iot.html","wb")
runner = HTMLTestRunner(stream=fp, title='测试报告',description='测试用例情况:')
runner.run(discover)
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：

```
class TestDemo:
    def test_demo1(self):
        print("测试用例1")

    def test_demo2(self):
        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

```
[pytest]
#参数
addopts = -vs
# 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
testpaths = ./testcases
# 文件命名规则
```

```
python_files = test_*.py
# 类名命名规则
python_classes = Test*
# Case命名规则
python_functions = test_*

# 标记
markers =
# 冒烟规则
smoke:冒烟用例
product_manage:商品管理
```

三、执行pytest

命令行执行：

```
# -vs：  -v输出详细信息 -s输出调试信息
pytest -vs

# -n：  多线程运行（前提安装插件：pytest-xdist）
pytest -vs -n=2

# --reruns num: 失败重跑（前提安装插件：pytest-rerunfailres）
pytest -vs --reruns=2

# -x: 出现一个用例失败则停止测试
pytest -vs -x

# --maxfail: 出现几个失败才终止
pytest -vs --maxfail=2

# --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）
pytest -vs --html ./reports/result.html

# -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
pytest -vs -k "qiuluo"
pytest -vs -k "qiuluo or weiliang"
pytest -vs -k "qiuluo and weiliang"

# -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
pytest -vs -m user_manage
```

添加注解： @pytest.mark.user_manage

命令行执行：pytest -vs -m user_manage

```
class TestDemo:

    # 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
    # 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
    @pytest.mark.user_manage
    def test_demo1(self):
        print("user_manage_test1")

```

```
    @pytest.mark.product_manage
    def test_demo2(self):
        print("product_manage_test1")

    @pytest.mark.user_manage
    @pytest.mark.product_manage
    def test_demo3(self):
        print("manage_test1")

```

Main函数执行：

添加mian：使用 `main` 方法执行

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

if __name__ == '__main__':
    pytest.main(["-vs"])
```

完整代码：

```
import  pytest

def test_01():
    print("啥也没有")

if __name__=='__main__':
    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=repot.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

```
[pytest]
addopts=-vs -m slow --html=./report/report.html
testpaths=testcase
test_files=test_*.py
test_classes=Test*
test_functions=test_*
makerers=
    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	改变默认的类搜索规则
python_functions	改变默认的测试用例的搜索规则
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

```
#配置pytest命令行运行参数
   [pytest]
    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
    testpaths = ./scripts  # 当前目录下的scripts文件夹 -可自定义
#配置测试搜索的文件名称
    python_files = test*.py
#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
    python_classes = Test_*

   #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

    python_functions = test_*

```

```
#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：

```
@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

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

```
    return a

@pytest.fixture()
def logout():
    print("关闭浏览器")

class TestLogin:
    #传入lonin fixture
    def test_001(self, login):
        print("001传入了loging fixture")
        assert login == "account"

    #传入logout fixture
    def test_002(self, logout):
        print("002传入了logout fixture")

    def test_003(self, login, logout):
        print("003传入了两个fixture")

    def test_004(self):
        print("004未传入仍何fixture哦")
```

执行结果：

```
============================= test session starts ==============================
collecting ... collected 4 items

scripts/test_demo1.py::TestLogin::test_001 打开浏览器
001传入了loging fixture
PASSED
scripts/test_demo1.py::TestLogin::test_002 关闭浏览器
002传入了logout fixture
PASSED
scripts/test_demo1.py::TestLogin::test_003 打开浏览器
关闭浏览器
003传入了两个fixture
PASSED
scripts/test_demo1.py::TestLogin::test_004 004未传入仍何fixture哦
PASSED

============================== 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

```
import pytest


# fixtrue作为参数，互相调用传入
```

```python
@pytest.fixture()
def account():
    a = "account"
    print("第一层fixture")
    return a


# Fixture的相互调用一定是要在测试类里调用这层fixture才会生次，普通函数单独调用是不生效的
@pytest.fixture()
def login(account):
    print("第二层fixture")


class TestLogin:
    def test_1(self, login):
        print("直接使用第二层fixture,返回值为{}".format(login))

    def test_2(self, account):
        print("只调用account fixture,返回值为{}".format(account))
```

执行结果：

```
============================= test session starts ==============================
collecting ... collected 2 items

scripts/test_demo1.py::TestLogin::test_1 第一层fixture
第二层fixture
直接使用第二层fixture,返回值为None
PASSED
scripts/test_demo1.py::TestLogin::test_2 第一层fixture
只调用account fixture,返回值为account
PASSED

============================== 2 passed in 0.01s ===============================
```

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

5、scope = “class”

代码如下：

```python
import pytest
# fixture作用域 scope = 'class'
@pytest.fixture(scope='class')
def login():
    print("scope为class")


class TestLogin:
    def test_1(self, login):
        print("用例1")

    def test_2(self, login):
```

```
        print("用例2")

    def test_3(self, login):
        print("用例3")


if __name__ == '__main__':
    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：

```
import pytest
@pytest.fixture(scope='class')
def login():
    a = '123'
    print("输入账号密码登陆")

class TestLogin:
    def test_1(self):
        print("用例1")

    def test_2(self, login):
        print("用例2")

    def test_3(self, login):
        print("用例3")

    def test_4(self):
        print("用例4")

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

6.scope = “module”：与class相同，

只从.py文件开始引用fixture的位置生效

```
import pytest
# fixture scope = 'module'
@pytest.fixture(scope='module')
def login():
    print("fixture范围为module")


def test_01():
    print("用例01")


def test_02(login):
    print("用例02")


```

```python
class TestLogin():
    def test_1(self):
        print("用例1")

    def test_2(self):
        print("用例2")

    def test_3(self):
        print("用例3")

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

7.fixtrue参数详解-autouse

> 默认False
> 若为True，刚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，无需传仍何参数，作用范围跟着scope走（谨慎使用）
> 让我们来看一下，当autouse=ture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：

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

运行命令：

```
pytest.main()
```

生成好json代码

运行生成html命令

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

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

```
import pytest


@pytest.fixture(autouse=True)
def login():
    print("====登录====")


def test_case01():
    print("我是测试用例11111")


@pytest.mark.skip(reason="不执行该用例！！因为没写好！！")
def test_case02():
    print("我是测试用例22222")


class Test1:

    def test_1(self):
        print("%% 我是类测试用例1111 %%")

    @pytest.mark.skip(reason="不想执行")
    def test_2(self):
        print("%% 我是类测试用例2222 %%")


@pytest.mark.skip(reason="类也可以跳过不执行")
class TestSkip:
    def test_1(self):
        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：

```
====登录====
我是测试用例11111
PASSEDSKIPPED (不执行该用例！！因为没写好！！)
Skipped: 不执行该用例！！因为没写好！！
====登录====
%% 我是类测试用例1111 %%
PASSEDSKIPPED (不想执行)
Skipped: 不想执行
SKIPPED (类也可以跳过不执行)
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

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

执行结果为：

```
这是我第1条用例
这是我第2条用例
这是我第3条用例
这是我第4条用例
SKIPPED (我跑五次了不跑了)
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

```
import sys
import pytest

if sys.platform.startswith("win"):
    pytest.skip("跳过仅限 Windows 的测试", allow_module_level=True)


@pytest.fixture(autouse=True)
def login():
    print("====登录====")


def test_case01():
    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：

```
import pytest
class Test_ABC:
    def setup_class(self):
        print("------->setup_class")
    def teardown_class(self):
        print("------->teardown_class")
    def test_a(self):
        print("------->test_a")
        assert 1
    @pytest.mark.skipif(condition=2>1,reason = "跳过该函数") # 跳过测试函数test_b
    def test_b(self):
        print("------->test_b")
            assert 0
```

执行结果：

```
------->setup_class
------->test_a
PASSEDSKIPPED (跳过该函数)
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：

```
# 标记
skipmark = pytest.mark.skip(reason="不能在window上运行=====")
skipifmark = pytest.mark.skipif(sys.platform == 'win32', reason="不能在window上运行啦啦啦=====")


@skipmark
class TestSkip_Mark(object):

    @skipifmark
    def test_function(self):
        print("测试标记")

    def test_def(self):
        print("测试标记")


@skipmark
def test_skip():
    print("测试标记")

```

执行效果：

```
SKIPPED
Skipped: 不能在window上运行啦啦啦=====
SKIPPED (不能在window上运行=====)
Skipped: 不能在window上运行=====
SKIPPED (不能在window上运行=====)
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：

```
pexpect = pytest.importorskip("pexpect", minversion="0.3")
@pexpect
def test_import():
    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：

```

import pytest

@pytest.mark.model
def test_model_a():
    print("执行test_model_a")

@pytest.mark.regular
def test_regular_a():
    print("test_regular_a")

@pytest.mark.model
def test_model_b():
    print("test_model_b")

@pytest.mark.regular
```

```
class TestClass:
    def test_method(self):
        print("test_method")

def testnoMark():
    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：

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

运行结果为：

```

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

```
import pytest

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

执行结果：

```
test_demo3.py::Test_ABC::test_a[3] ------->setup_class
test data:a=3
PASSED
test_demo3.py::Test_ABC::test_a[6] test data:a=6
PASSED------->teardown_class
```

多个参数

代码演示：

```
import pytest
class Test_demo1:
    def setup_class(self):
        print("------->setup_class")
    def teardown_class(self):
        print("------->teardown_class")
    @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
    def test_a(self,a,b): # 参数必须和parametrize里面的参数一致
        print("test data:a=%d,b=%d"%(a,b))
        assert a+b == 3
```

执行结果：

```

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

```

函数返回值作为参数

代码演示：

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

执行结果：

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

“笛卡尔积”，多个参数化装饰器

代码演示：

```python

import pytest

# 笛卡尔积，组合数据
data_1 = [1, 2, 3]
data_2 = ['a', 'b']


@pytest.mark.parametrize('a', data_1)
@pytest.mark.parametrize('b', data_2)
def test_parametrize_1(a, b):
    print(f'笛卡尔积 测试数据为 ： {a}，{b}')
```

执行结果：

```
test_demo3.py::test_parametrize_1[a-1] 笛卡尔积 测试数据为 ： 1，a
PASSED
test_demo3.py::test_parametrize_1[a-2] 笛卡尔积 测试数据为 ： 2，a
PASSED
test_demo3.py::test_parametrize_1[a-3] 笛卡尔积 测试数据为 ： 3，a
PASSED
test_demo3.py::test_parametrize_1[b-1] 笛卡尔积 测试数据为 ： 1，b
PASSED
test_demo3.py::test_parametrize_1[b-2] 笛卡尔积 测试数据为 ： 2，b
PASSED
test_demo3.py::test_parametrize_1[b-3] 笛卡尔积 测试数据为 ： 3，b
PASSED
```

参数化，标记数据

代码演示：

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

```
第一个测试用例：计算3+3的和
RERUN
test_demo3.py::TestClass::test_case 第一个测试用例：计算3+3的和
RERUN
test_demo3.py::TestClass::test_case 第一个测试用例：计算3+3的和
RERUN
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：

```python
import pytest

class TestOrder:
    def test_case01(self):
        print("第一条测试用例")
    @pytest.mark.run(order=-1)
    def test_case02(self):
        print("第二条测试用例")
    @pytest.mark.run(order=1)
    def test_case03(self):
        print("第三条测试用例")
    @pytest.mark.run(order=0)
    def test_case04(self):
        print("第四条测试用例")
    @pytest.mark.run(order=-2)
    def test_case05(self):
        print("第五条测试用例")

if __name__ == '__main__':
    pytest.main(["-s", "test_demo3.py"])
```

执行效果：

```
test_demo3.py::TestOrder::test_case04 第四条测试用例
PASSED
test_demo3.py::TestOrder::test_case03 第三条测试用例
PASSED
test_demo3.py::TestOrder::test_case01 第一条测试用例
PASSED
test_demo3.py::TestOrder::test_case05 第五条测试用例
PASSED
test_demo3.py::TestOrder::test_case02 第二条测试用例
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