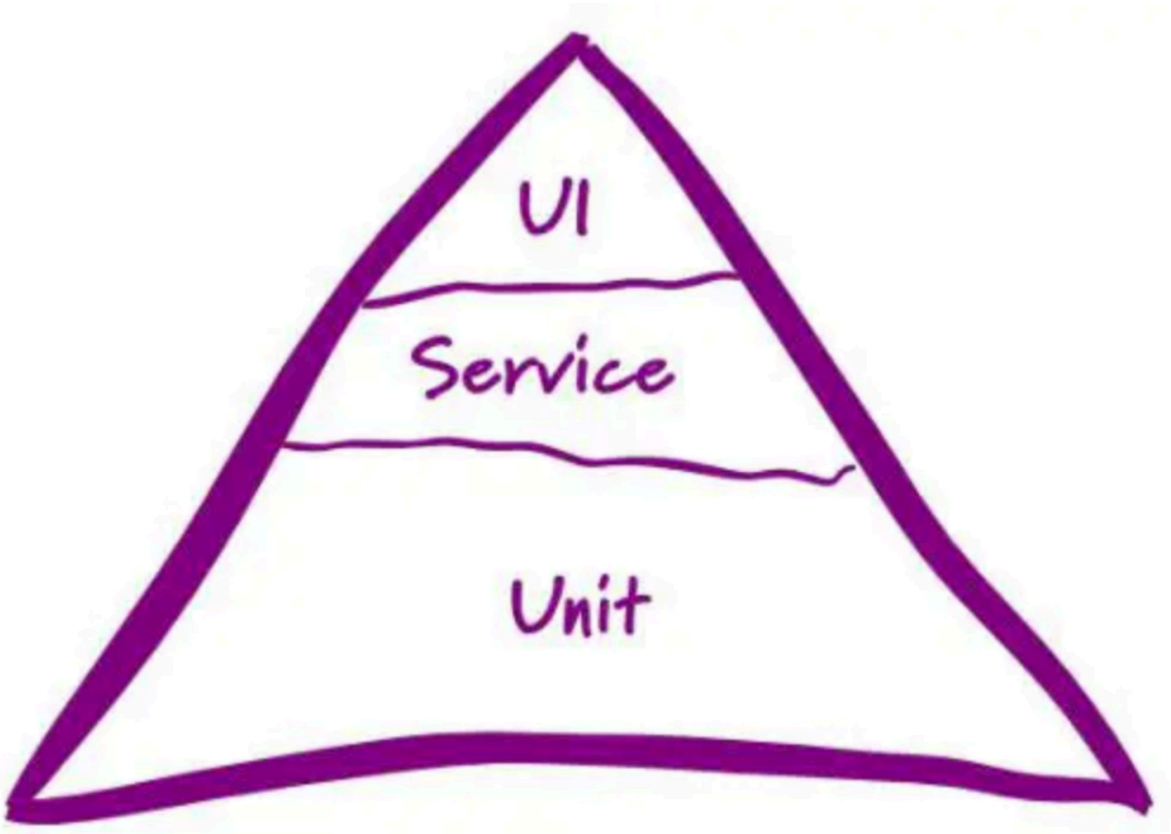
Python单元测试框架（无涯）

单元测试概述

Python测试框架

从软件架构的角度来说，测试最重要的步骤是在软件开发的时候界入比较好，所以在早期测试的界入，从软件经济学的角度上来说，发现的问题解决成本低，投入的资源比较少。因此，对一个测试的系统，开始最佳的测试就是源代码级别的测试，也就是单元测试阶段，这个过程也被成为白盒测试。单元测试是最基本也是最底层的测试类型，单元测试应用于最基本的软件代码，如类，函数。方法等，单元测试通过可执行的断言检查被测单元的输出是否满足预期结果。在测试金字塔的理论上来说，越往下的测试投入资源越高，得到的回报率越大，见测试金字塔模型：

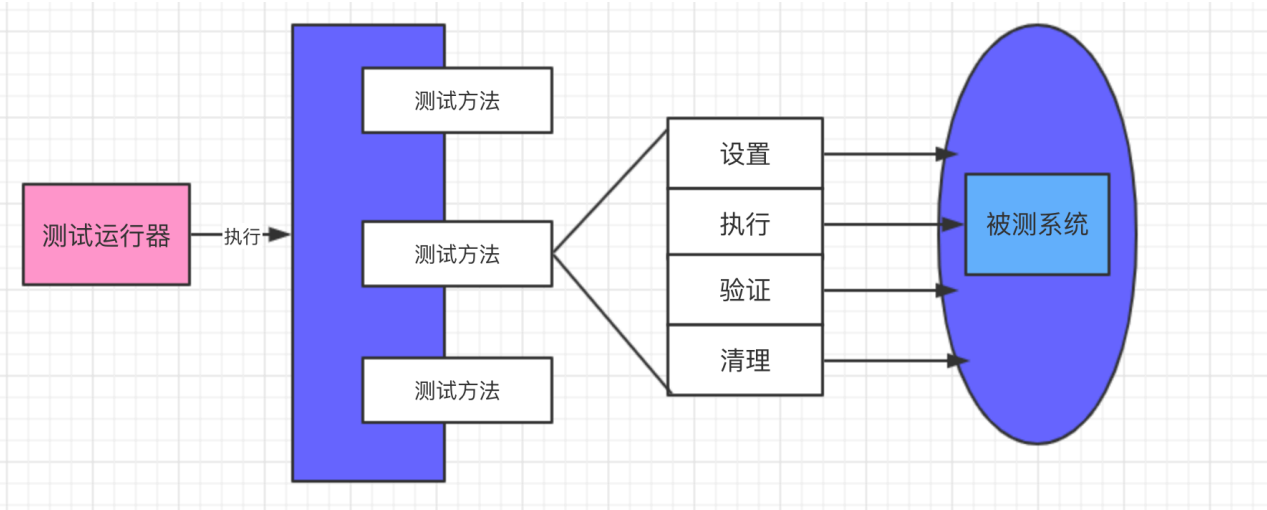


抛开软件架构的层面，在自动化测试的体系中，单元测试框架以及单元测试的知识体系是必须要掌握的技能之一，单元测试的知识体系是自动化测试工程师以及测试开发工程师的知识体系之一，而且是必须具备的知识之一。在Python语言中应用最广泛的单元测试框架是unittest和pytest,unittest属于标准库，只要安装了Python解释器后就可以直接导入使用了,pytest是第三方的库，需要单独的安装。单元测试框架的知识体系就围绕unittest和pytest来讲解。

白盒测试原理

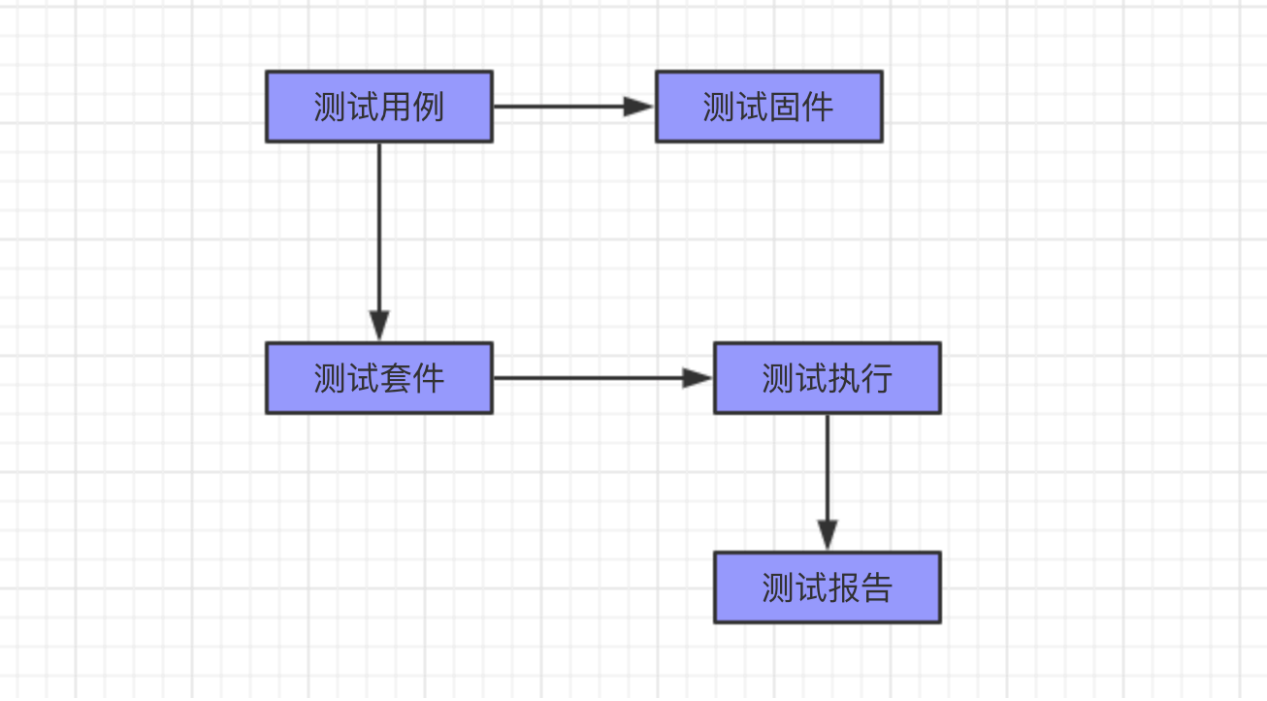
在软件架构的层面来说，测试最核心的步骤就是在软件开发过程中。就软件本身而言，软件的行为或者功能是软件细节实现的产物，这些最终是交付给用户的东西。所以在早期执行测试的系统有可能是一个可测试和健壮的系统，它会带来为用户提供的功能往往是让人满意的结果。因此给予这样的角度，开始执行测试的最佳方法是来自源代码，也就是软件编写的地方以及开发人员。由于源代码是对开发人员是可见的，这样的一个测试过程我们可以称为白盒测试。

自动化测试用例编写



unittest实战

unittest各个组件的介绍



测试用例：

测试类继承unittest模块中的TestCase类后，依据继承的这个类来设置一个新的测试用例类和测试方法，案例代码：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
class ApiTest(unittest.TestCase):
    def test_001(self):
        pass
```

测试固件：

测试固件表示一个测试用例或者多个测试以及清理工作所需要的设置或者准备，见案例代码：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
class ApiTest(unittest.TestCase):
    def setUp(self):
        pass
    def tearDown(self):
        pass
```

测试套件：

测试套件顾名思义就是相关测试用例的集合。在unittest中主要通过TestSuite类提供对测试套件的支持，见案例带代码：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
class ApiTest(unittest.TestCase):
    def test_001(self):
        pass
    def test_002(self):
        pass
if __name__ == '__main__':
    suite=unittest.TestSuite()
    suite.addTest('test_001')
    unittest.TextTestRunner(verbosity=2).run(suite)
```

测试运行:

管理和运行测试用例的对象。

测试断言：

对所测试的对象依据返回的实际结果与期望结果进行断言校验

测试结果：

测试结果类管理着测试结果的输出，测试结果呈现给最终的用户来反馈本次测试执行的结果信息。

unittest测试固件详解

在unittest中测试固件依据方法可以分为两种执行方式，一种是测试固件只执行一次，另外一种是测试固件每次都执行，下面依据具体的案例来讲解二者。

1、测试固件每次均执行

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
from selenium import  webdriver
class ApiTest(unittest.TestCase):
    def setUp(self):
        self.driver=webdriver.Chrome()
        self.driver.maximize_window()
        self.driver.get('http://www.baidu.com')
        self.driver.implicitly_wait(30)

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

    def test_baidu_title(self):
        self.assertEqual(self.driver.title,'百度一下，你就知道')

    def test_baidu_url(self):
        self.assertEqual(self.driver.current_url,'https://www.baidu.com/')

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

执行如上的代码后，它的顺序为：

测试固件-->测试用例，测试固件-->测试用例

2、测试固件只执行一次

使用的是类方法,这样测试固件只会执行一次的，见案例的代码：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
from selenium import  webdriver
class ApiTest(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.driver=webdriver.Chrome()
        cls.driver.maximize_window()
        cls.driver.get('http://www.baidu.com')
```

```
        cls.driver.implicitly_wait(30)

        @classmethod
        def tearDownClass(cls):
            cls.driver.quit()

        def test_baidu_title(self):
            self.assertEqual(self.driver.title,'百度一下，你就知道')

        def test_baidu_url(self):
           self.assertEqual(self.driver.current_url,'https://www.baidu.com/')

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

测试用例执行顺序详解

在unittest中，测试点的执行顺序是依据ascill码来执行的，也就是说根据ASCII码的顺序加载，数字与字母的顺序为：0-9，A-Z，a-z，所以以A开头的测试用例方法会优先执行，以a开头会后执行。也就是根据数字的大小从小到大执行的，切记数字的大小值的是不包含test，*值的是test*后面的测试点的数字大小，见案例代码：

```
#!/usr/bin/python3
#coding:utf-8



import  unittest


class Api(unittest.TestCase):
   def test_001(self):
      pass

   def test_002(self):
      pass

if __name__ == '__main__':
   unittest.main(verbosity=2)
```

执行的顺序为:test_001,下来时test_002

当然测试点不会单纯是数字的，也有字符串的，在python中，字符串与数字的转换为：

chr():数字转为字母

ord():字母转为数字

见测试点的名称是字符串的执行顺序：

```
#!/usr/bin/python3
#coding:utf-8



import  unittest


class Api(unittest.TestCase):
   def test_abc(self):
      pass

   def test_acb(self):
      pass

if __name__ == '__main__':
   unittest.main(verbosity=2)
```

执行的顺序为：test_abc,下来是test_acb

再看字符串与数字的比较：

```
#!/usr/bin/python3
#coding:utf-8



import  unittest


class Api(unittest.TestCase):
   def test_api_0a0(self):
      pass

   def test_api_00a(self):
      pass

if __name__ == '__main__':
   unittest.main(verbosity=2)
```

执行的顺序为：test_api_00a,下来是test_api_0a0

verbosity的详解

在unittest中,执行的时候默认是1,也就是unittest.main(verbosity=1),在verbosity有三个参数，分别是0，1，2，代表的意思具体如下：

0（静默模式）：仅仅获取总的测试用例数以及总的结果

1（默认模式）

2(详细模式):测试结果会显示每个测试用例的所有相关信息，下面分别演示几个在错误情况下的显示信息：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import  unittest
from selenium import  webdriver
class Baidu(unittest.TestCase):
    def setUp(self):
        self.driver=webdriver.Chrome()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()
        self.driver.get('http://www.baidu.com')

    def test_baidu_title(self):
        self.assertEqual(self.driver.title,'百度一下你就知道')

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

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

见默认情况的错误信息显示：

Ran 1 test in 59.437s

FAILED (failures=1)

百度一下你就知道 != 百度一下，你就知道

Expected :百度一下，你就知道 Actual :百度一下你就知道

Traceback (most recent call last): File "D:\Program Files\JetBrains\PyCharm 2018.3.5\helpers\pycharm\teamcity\diff_tools.py", line 32, in *patched_equals old(self, first, second, msg) File "C:\Python37\lib\unittest\case.py", line 839, in assertEqual assertion_func(first, second, msg=msg) File "C:\Python37\lib\unittest\case.py", line 1220, in assertMultiLineEqual self.fail(self.*formatMessage(msg, standardMsg)) File "C:\Python37\lib\unittest\case.py", line 680, in fail raise self.failureException(msg) AssertionError: '百度一下，你就知道' != '百度一下你就知道'

- 百度一下，你就知道？-
- 百度一下你就知道

During handling of the above exception, another exception occurred:

Traceback (most recent call last): File "C:\Python37\lib\unittest\case.py", line 59, in testPartExecutor yield File "C:\Python37\lib\unittest\case.py", line 615, in run testMethod() File "D:\git\GITHUB\pyUnit\unit01.py", line 16, in test_baidu_title self.assertEqual(self.driver.title,'百度一下你就知道')

编写测试用例注意事项

测试用例注意事项如下：

1、在一个测试类里面，每一个测试方法都是以test开头的，test不能是中间或者尾部，必须是开头，建议test_

2、每一个测试用例方法都应该有注释信息，这样在测试报告就会显示具体的测试点的检查点

3、在自动化测试中，每个测试用例都必须得有断言，无断言的自动化测试用例是无效的

4、最好一个测试用例方法对应一个业务测试点，不要多个业务检查点写一个测试用例

5、如果涉及到业务逻辑的处理，最好把业务逻辑的处理方法放在断言前面，这样做的目的是不要因为业务逻辑执行错误导致断言也是失败

6、测试用例名称最好规范，有约束

7、是否先写自动化测试的测试代码，在使用自动化测试方式写，本人觉得没必要，毕竟能够做自动化测试的都具备了功能测试的基本水平，所以没必要把一个业务的检查点写多次，浪费时间和人力成本。见案例代码：

```
def test_baidu_title(self):
    '''验证:验证百度首页的title是否正确'''
    self.assertEqual(self.driver.title,'百度一下，你就知道')
```

下面就以具体的案例演示第五点的情况，见测试的源码：

```
#!/usr/bin/env python
# -*-coding:utf-8 -*-
import requests
import  unittest
import  os
class ApiTest(unittest.TestCase):
    def writeBookID(self,bookID):
        with open(os.path.join(os.path.dirname(__file__),'bookID'),'w') as f:
            f.write(str(bookID))

    @property
    def readBookID(self):
        with open(os.path.join(os.path.dirname(__file__), 'bookID'), 'r') as f:
            return f.read()

    def creeteBook(self):
```

```
        dict1={'author':'无涯','name':'Python自动化测试实战','done':True}
        r=requests.post(url='http://localhost:5000/v1/api/books',json=dict1)
        self.writeBookID(r.json()[0]['datas']['id'])
        return r

    def test_api_book(self):
        '''API测试:查询书籍'''
        self.creeteBook()
        r=requests.get(url='http://localhost:5000/v1/api/book/{0}'.
                       format(self.readBookID))
        self.delBook()
        self.assertEqual(r.json()['datas'][0]['name'],'Python自动化测试实战')
        self.assertEqual(r.json()['datas'][0]['id'],int(self.readBookID))

    def delBook(self):
        r=requests.delete(url='http://localhost:5000/v1/api/book/{0}'.
                          format(self.readBookID))

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

命令行执行信息

当然编写的测试点也是可以依据命令行来进行执行的，unit03.py的源码如下：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest

class UiTest(unittest.TestCase):
   def test_001(self):
      pass

   def test_002(self):
      pass

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

执行的命令行命令为：

python3 -m unittest unit03.py —>按模块执行

python3 -m unittest unit03.UiTest —>按测试类来执行

python3 -m unittest unit03.UiTest.test_001 —>按每个测试用例来执行

python3 -m unittest -v unit03 —>显示详细的信息

python3 -m unittest discover -p 'unit03.py' —>测试发现

测试套件详解

按测试用例执行

只添加需要执行的测试用例

```
suite=unittest.TestSuite()suite.addTest(Api('test_001'))
unittest.TextTestRunner(verbosity=2).run(suite)
```

按测试类执行

```
import  unittest
class Api(unittest.TestCase):
    def test_001(self):
        pass

    def test_002(self):
        pass

if __name__ == '__main__':
    suite=unittest.TestLoader().loadTestsFromTestCase(Api)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

见测试类的另外一种执行的方式，也就是makeSuite()类，见源码：

```
import  unittest
class ApiTestCase(unittest.TestCase):
    def test_001(self):
        pass

    def test_002(self):
        pass

class AppTestCase(unittest.TestCase):
    def test_001(self):
        pass

    def test_002(self):
        pass

if __name__ == '__main__':
    apiSuite=unittest.makeSuite(ApiTestCase,'test')
    appSuite=unittest.makeSuite(AppTestCase,'test')
    suite=unittest.TestSuite(apiSuite,appSuite)
```

```
    unittest.TextTestRunner(verbosity=2).run(suite)
```

按测试模块来执行

```
import  unittest
class ApiTestCase(unittest.TestCase):
    def test_001(self):
        pass

    def test_002(self):
        pass

class AppTestCase(unittest.TestCase):
    def test_003(self):
        pass

    def test_004(self):
        pass
if __name__ == '__main__':
    suite=unittest.TestLoader().loadTestsFromModule('unit02.py')
    unittest.TextTestRunner(verbosity=2).run(suite)
```

自定义测试套件

见自定义测试套件的源码：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest

class UiTest(unittest.TestCase):
   def test_001(self):
      pass

   def test_002(self):
      pass

   @staticmethod
   def suite():
      suite=unittest.TestSuite()
      suite.addTest(UiTest('test_001'))
      return suite


if __name__ == '__main__':
   unittest.TextTestRunner().run(UiTest.suite())
```

优化测试套件

把测试套件分离成一个方法，然后直接调用这个方法就可以了，见实现的代码：

```
import  unittest
from selenium import  webdriver
class Baidu(unittest.TestCase):
    def setUp(self):
        self.driver=webdriver.Chrome()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()
        self.driver.get('http://www.baidu.com')

    def test_baidu_title(self):
        '''验证:验证百度首页的title是否正确'''
        self.assertEqual(self.driver.title,'百度一下，你就知道')

    def test_baidu_url(self):
        '''验证:验证百度首页的url是否正确'''
        self.assertEqual(self.driver.current_url,'https://www.baidu.com/')

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

   @staticmethod
   def suite():
        tests=unittest.TestLoader().loadTestsFromTestCase(Baidu)
        return tests

if __name__ == '__main__':
    unittest.TextTestRunner(verbosity=2).run(Baidu.suite())
```

分离测试套件

利用继承的思想分离出测试套件,测试类只需要继承分离的类就可以了，见init.py的源码：

```
import  unittest
from selenium import  webdriver
class Init(unittest.TestCase):
    def setUp(self):
        self.driver=webdriver.Chrome()
        self.driver.implicitly_wait(30)
        self.driver.maximize_window()
        self.driver.get('http://www.baidu.com')

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

见测试类的源码,只需要继承Init类就可以了，见源码：

```
import unittest
from init import  Init
class Baidu(Init):
    def test_baidu_title(self):
        '''验证:验证百度首页的title是否正确'''
        self.assertEqual(self.driver.title,'百度一下，你就知道')

    def test_baidu_url(self):
        '''验证:验证百度首页的url是否正确'''
        self.assertEqual(self.driver.current_url,'https://www.baidu.com/')

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

unittest的参数化

parameterized实战

在unittest的测试框架中，可以结合ddt的模块来达到参数化的应用，当然关于ddt库的应用在数据驱动方面有很详细的解释，这里就直接说另外的一个第三方的库parameterized,安装的命令为：

pip3 install parameterized

安装成功后，这里就以一个两个数相加的案例来演示它的应用，实现的源码如下：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest
from parameterized import  parameterized,param


def add(a,b):
```

```
    return a+b

class AddTest(unittest.TestCase):
    def setUp(self) -> None:
        pass

    def tearDown(self) -> None:
        pass

    @parameterized.expand([
        param(1,1,2),
        param(2,2,4),
        param(1.0,1.0,2.0),
        param('hi',' wuya','hi wuya')
    ])
    def test_add(self,a,b,result):
        self.assertEqual(add(a,b),result)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

执行后它的输出信息为：

```
test_add_0 (__main__.AddTest) ... ok
test_add_1 (__main__.AddTest) ... ok
test_add_2 (__main__.AddTest) ... ok
test_add_3_hi (__main__.AddTest) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

在UI自动化测试中，parameterized也是特别的有用，如针对一个登录案例的测试，针对登录就会有很多的测试案例的，主要是用户名和密码的input表单的验证以及错误信息的验证，下面就结合具体的案例来看它在UI自动化测试中的应用，案例源码为：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import  unittest
from selenium import  webdriver
from parameterized import  parameterized,param
import  time as t
```

```
class UiTest(unittest.TestCase):
    def setUp(self) -> None:
        self.driver=webdriver.Chrome()
        self.driver.maximize_window()
        self.driver.get('https://mail.sina.com.cn/')
        self.driver.implicitly_wait(30)

    def tearDown(self) -> None:
        self.driver.quit()

    @parameterized.expand([
        param('','','请输入邮箱名'),
        param('sadfdgrh@sina.com','asdfgh','登录名或密码错误'),
        param('esrtyu','dsgftyj','您输入的邮箱名格式不正确'),
        param('dsf','','您输入的邮箱名格式不正确')
    ])
    def test_sina_login(self,username,password,result):
        '''sina邮箱首页的登录验证'''
        self.driver.find_element_by_id('freename').send_keys(username)
        self.driver.find_element_by_id('freepassword').send_keys(password)
        self.driver.find_element_by_link_text('登录').click()
        t.sleep(3)
        divText=self.driver.find_element_by_xpath('
   /html/body/div[1]/div/div[2]/div/div/div[4]/div[1]/div[1]/div[1]/span[1]').te
xt
        self.assertTrue(divText,result)

if __name__ == '__main__':
     unittest.main(verbosity=2)
```

执行如上的代码后，输出的结果信息为：

```
collecting ... collected 4 items

f2.py::UiTest::test_sina_login_0_
f2.py::UiTest::test_sina_login_1_sadfdgrh_sina_com
f2.py::UiTest::test_sina_login_2_esrtyu
f2.py::UiTest::test_sina_login_3_dsf

==================== 4 passed in 23.39 seconds ====================
```

其实parameterized在执行的时候，它的原理和核心思想是在执行的过程中，它把列表里面的值首先进行循环，然后把param元组里面的值赋值给测试方法里面对应的参数username,password,result，如第一个元组，它的实际赋值也就是为:username='',password='',result='请输入邮箱名',具体可以打断点（切记debug的模式执行），看具体赋值的过程，见如下显示的截图信息(thinkpad的电脑操作截图信息)：

ddt实战

ddt是第三方的库，需要单独的安装，安装的命令为：

pip3 install ddt

ddt它能够实现在一个同样的测试步骤中，使用一个测试点的测试代码，实现多个测试场景的验证，这里就以sina的邮箱登录为案例，来演示它在UI自动化测试中的应用，具体代码如下：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest
from selenium import  webdriver
import  time as t
import  ddt

@ddt.ddt
class SinaTest(unittest.TestCase):
   def setUp(self) -> None:
      self.driver=webdriver.Chrome()
      self.driver.maximize_window()
      self.driver.get('https://mail.sina.com.cn/')
      self.driver.implicitly_wait(30)

   def tearDown(self) -> None:
      self.driver.quit()

   @ddt.data(
      ('','','请输入邮箱名'),
      ('admin','admin','您输入的邮箱名格式不正确'),
      ('wuya1303@sina.com','asdfghjk','登录名或密码错误'))
   @ddt.unpack
   def test_sina_login(self,username,password,result):
      self.driver.find_element_by_id('freename').send_keys(username)
      self.driver.find_element_by_id('freepassword').send_keys(password)
      self.driver.find_element_by_link_text('登录').click()
      t.sleep(3)

 divText=self.driver.find_element_by_xpath('/html/body/div[1]/div/div[2]/div/div/div[4]/div[1]/div[1]/div[1]/span[1]').text
      self.assertEqual(divText,result)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

断言的详解

在unit test的单元测试框架中，涉及到的断言主要如下，首先来看源码里面的断言方法：

```
def assertListEqual(self, list1, list2, msg=None):
    """A list-specific equality assertion.

    Args:
        list1: The first list to compare.
        list2: The second list to compare.
        msg: Optional message to use on failure instead of a list of
                differences.

    """
    self.assertSequenceEqual(list1, list2, msg, seq_type=list)

def assertTupleEqual(self, tuple1, tuple2, msg=None):
    """A tuple-specific equality assertion.

    Args:
        tuple1: The first tuple to compare.
        tuple2: The second tuple to compare.
        msg: Optional message to use on failure instead of a list of
                differences.
    """
    self.assertSequenceEqual(tuple1, tuple2, msg, seq_type=tuple)

def assertSetEqual(self, set1, set2, msg=None):
    """A set-specific equality assertion.

    Args:
        set1: The first set to compare.
        set2: The second set to compare.
        msg: Optional message to use on failure instead of a list of
                differences.

    assertSetEqual uses ducktyping to support different types of sets, and
    is optimized for sets specifically (parameters must support a
    difference method).
    """
    try:
        difference1 = set1.difference(set2)
    except TypeError as e:
        self.fail('invalid type when attempting set difference: %s' % e)
    except AttributeError as e:
        self.fail('first argument does not support set difference: %s' % e)

    try:
```

```
            difference2 = set2.difference(set1)
        except TypeError as e:
            self.fail('invalid type when attempting set difference: %s' % e)
        except AttributeError as e:
            self.fail('second argument does not support set difference: %s' % e)

        if not (difference1 or difference2):
            return

        lines = []
        if difference1:
            lines.append('Items in the first set but not the second:')
            for item in difference1:
                lines.append(repr(item))
        if difference2:
            lines.append('Items in the second set but not the first:')
            for item in difference2:
                lines.append(repr(item))

        standardMsg = '\n'.join(lines)
        self.fail(self._formatMessage(msg, standardMsg))

    def assertIn(self, member, container, msg=None):
        """Just like self.assertTrue(a in b), but with a nicer default message."""
        if member not in container:
            standardMsg = '%s not found in %s' % (safe_repr(member),
                                                  safe_repr(container))
            self.fail(self._formatMessage(msg, standardMsg))

    def assertNotIn(self, member, container, msg=None):
        """Just like self.assertTrue(a not in b), but with a nicer default
    message."""
        if member in container:
            standardMsg = '%s unexpectedly found in %s' % (safe_repr(member),
                                                           safe_repr(container))
            self.fail(self._formatMessage(msg, standardMsg))

    def assertIs(self, expr1, expr2, msg=None):
        """Just like self.assertTrue(a is b), but with a nicer default message."""
        if expr1 is not expr2:
            standardMsg = '%s is not %s' % (safe_repr(expr1),
                                            safe_repr(expr2))
            self.fail(self._formatMessage(msg, standardMsg))

    def assertIsNot(self, expr1, expr2, msg=None):
        """Just like self.assertTrue(a is not b), but with a nicer default
    message."""
        if expr1 is expr2:
            standardMsg = 'unexpectedly identical: %s' % (safe_repr(expr1),)
```

```
        self.fail(self._formatMessage(msg, standardMsg))

def assertDictEqual(self, d1, d2, msg=None):
    self.assertIsInstance(d1, dict, 'First argument is not a dictionary')
    self.assertIsInstance(d2, dict, 'Second argument is not a dictionary')

    if d1 != d2:
        standardMsg = '%s != %s' % _common_shorten_repr(d1, d2)
        diff = ('\n' + '\n'.join(difflib.ndiff(
                       pprint.pformat(d1).splitlines(),
                       pprint.pformat(d2).splitlines())))
        standardMsg = self._truncateMessage(standardMsg, diff)
        self.fail(self._formatMessage(msg, standardMsg))

def assertDictContainsSubset(self, subset, dictionary, msg=None):
    """Checks whether dictionary is a superset of subset."""
    warnings.warn('assertDictContainsSubset is deprecated',
                  DeprecationWarning)
    missing = []
    mismatched = []
    for key, value in subset.items():
        if key not in dictionary:
            missing.append(key)
        elif value != dictionary[key]:
            mismatched.append('%s, expected: %s, actual: %s' %
                              (safe_repr(key), safe_repr(value),
                               safe_repr(dictionary[key])))

    if not (missing or mismatched):
        return

    standardMsg = ''
    if missing:
        standardMsg = 'Missing: %s' % ','.join(safe_repr(m) for m in
                                               missing)
    if mismatched:
        if standardMsg:
            standardMsg += '; '
        standardMsg += 'Mismatched values: %s' % ','.join(mismatched)

    self.fail(self._formatMessage(msg, standardMsg))


def assertCountEqual(self, first, second, msg=None):
    """An unordered sequence comparison asserting that the same elements,
    regardless of order.  If the same element occurs more than once,
    it verifies that the elements occur the same number of times.

        self.assertEqual(Counter(list(first)),
```

```
                        Counter(list(second)))

     Example:
        - [0, 1, 1] and [1, 0, 1] compare equal.
        - [0, 0, 1] and [0, 1] compare unequal.

    """
    first_seq, second_seq = list(first), list(second)
    try:
        first = collections.Counter(first_seq)
        second = collections.Counter(second_seq)
    except TypeError:
        # Handle case with unhashable elements
        differences = _count_diff_all_purpose(first_seq, second_seq)
    else:
        if first == second:
            return
        differences = _count_diff_hashable(first_seq, second_seq)

    if differences:
        standardMsg = 'Element counts were not equal:\n'
        lines = ['First has %d, Second has %d:  %r' % diff for diff in
differences]
        diffMsg = '\n'.join(lines)
        standardMsg = self._truncateMessage(standardMsg, diffMsg)
        msg = self._formatMessage(msg, standardMsg)
        self.fail(msg)

def assertMultiLineEqual(self, first, second, msg=None):
    """Assert that two multi-line strings are equal."""
    self.assertIsInstance(first, str, 'First argument is not a string')
    self.assertIsInstance(second, str, 'Second argument is not a string')

    if first != second:
        # don't use difflib if the strings are too long
        if (len(first) > self._diffThreshold or
            len(second) > self._diffThreshold):
            self._baseAssertEqual(first, second, msg)
        firstlines = first.splitlines(keepends=True)
        secondlines = second.splitlines(keepends=True)
        if len(firstlines) == 1 and first.strip('\r\n') == first:
            firstlines = [first + '\n']
            secondlines = [second + '\n']
        standardMsg = '%s != %s' % _common_shorten_repr(first, second)
        diff = '\n' + ''.join(difflib.ndiff(firstlines, secondlines))
        standardMsg = self._truncateMessage(standardMsg, diff)
        self.fail(self._formatMessage(msg, standardMsg))

def assertLess(self, a, b, msg=None):
```

```
    """Just like self.assertTrue(a < b), but with a nicer default message."""
    if not a < b:
        standardMsg = '%s not less than %s' % (safe_repr(a), safe_repr(b))
        self.fail(self._formatMessage(msg, standardMsg))

def assertLessEqual(self, a, b, msg=None):
    """Just like self.assertTrue(a <= b), but with a nicer default message."""
    if not a <= b:
        standardMsg = '%s not less than or equal to %s' % (safe_repr(a),
safe_repr(b))
        self.fail(self._formatMessage(msg, standardMsg))

def assertGreater(self, a, b, msg=None):
    """Just like self.assertTrue(a > b), but with a nicer default message."""
    if not a > b:
        standardMsg = '%s not greater than %s' % (safe_repr(a), safe_repr(b))
        self.fail(self._formatMessage(msg, standardMsg))

def assertGreaterEqual(self, a, b, msg=None):
    """Just like self.assertTrue(a >= b), but with a nicer default message."""
    if not a >= b:
        standardMsg = '%s not greater than or equal to %s' % (safe_repr(a),
safe_repr(b))
        self.fail(self._formatMessage(msg, standardMsg))

def assertIsNone(self, obj, msg=None):
    """Same as self.assertTrue(obj is None), with a nicer default message."""
    if obj is not None:
        standardMsg = '%s is not None' % (safe_repr(obj),)
        self.fail(self._formatMessage(msg, standardMsg))

def assertIsNotNone(self, obj, msg=None):
    """Included for symmetry with assertIsNone."""
    if obj is None:
        standardMsg = 'unexpectedly None'
        self.fail(self._formatMessage(msg, standardMsg))

def assertIsInstance(self, obj, cls, msg=None):
    """Same as self.assertTrue(isinstance(obj, cls)), with a nicer
    default message."""
    if not isinstance(obj, cls):
        standardMsg = '%s is not an instance of %r' % (safe_repr(obj), cls)
        self.fail(self._formatMessage(msg, standardMsg))

def assertNotIsInstance(self, obj, cls, msg=None):
    """Included for symmetry with assertIsInstance."""
    if isinstance(obj, cls):
        standardMsg = '%s is an instance of %r' % (safe_repr(obj), cls)
        self.fail(self._formatMessage(msg, standardMsg))
```

```
def assertRaisesRegex(self, expected_exception, expected_regex,
                      *args, **kwargs):
    """Asserts that the message in a raised exception matches a regex.

    Args:
        expected_exception: Exception class expected to be raised.
        expected_regex: Regex (re.Pattern object or string) expected
                to be found in error message.
        args: Function to be called and extra positional args.
        kwargs: Extra kwargs.
        msg: Optional message used in case of failure. Can only be used
                when assertRaisesRegex is used as a context manager.
    """
    context = _AssertRaisesContext(expected_exception, self, expected_regex)
    return context.handle('assertRaisesRegex', args, kwargs)

def assertWarnsRegex(self, expected_warning, expected_regex,
                     *args, **kwargs):
    """Asserts that the message in a triggered warning matches a regexp.
    Basic functioning is similar to assertWarns() with the addition
    that only warnings whose messages also match the regular expression
    are considered successful matches.

    Args:
        expected_warning: Warning class expected to be triggered.
        expected_regex: Regex (re.Pattern object or string) expected
                to be found in error message.
        args: Function to be called and extra positional args.
        kwargs: Extra kwargs.
        msg: Optional message used in case of failure. Can only be used
                when assertWarnsRegex is used as a context manager.
    """
    context = _AssertWarnsContext(expected_warning, self, expected_regex)
    return context.handle('assertWarnsRegex', args, kwargs)

def assertRegex(self, text, expected_regex, msg=None):
    """Fail the test unless the text matches the regular expression."""
    if isinstance(expected_regex, (str, bytes)):
        assert expected_regex, "expected_regex must not be empty."
        expected_regex = re.compile(expected_regex)
    if not expected_regex.search(text):
        standardMsg = "Regex didn't match: %r not found in %r" % (
            expected_regex.pattern, text)
        # _formatMessage ensures the longMessage option is respected
        msg = self._formatMessage(msg, standardMsg)
        raise self.failureException(msg)

def assertNotRegex(self, text, unexpected_regex, msg=None):
```

```
    """Fail the test if the text matches the regular expression."""
    if isinstance(unexpected_regex, (str, bytes)):
        unexpected_regex = re.compile(unexpected_regex)
    match = unexpected_regex.search(text)
    if match:
        standardMsg = 'Regex matched: %r matches %r in %r' % (
            text[match.start() : match.end()],
            unexpected_regex.pattern,
            text)
        # _formatMessage ensures the longMessage option is respected
        msg = self._formatMessage(msg, standardMsg)
        raise self.failureException(msg)


def _deprecate(original_func):
    def deprecated_func(*args, **kwargs):
        warnings.warn(
            'Please use {0} instead.'.format(original_func.__name__),
            DeprecationWarning, 2)
        return original_func(*args, **kwargs)
    return deprecated_func

# see #9424
failUnlessEqual = assertEquals = _deprecate(assertEqual)
failIfEqual = assertNotEquals = _deprecate(assertNotEqual)
failUnlessAlmostEqual = assertAlmostEquals = _deprecate(assertAlmostEqual)
failIfAlmostEqual = assertNotAlmostEquals = _deprecate(assertNotAlmostEqual)
failUnless = assert_ = _deprecate(assertTrue)
failUnlessRaises = _deprecate(assertRaises)
failIf = _deprecate(assertFalse)
assertRaisesRegexp = _deprecate(assertRaisesRegex)
assertRegexpMatches = _deprecate(assertRegex)
assertNotRegexpMatches = _deprecate(assertNotRegex)
```

具体详解如下：

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

assertEqual

assertEqual()是验证两个人相等，值的是数据类型与内容也是相等的，见案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

class BaiduUI(unittest.TestCase):
   def setUp(self):
      self.driver=webdriver.Chrome()
      self.driver.maximize_window()
      self.driver.implicitly_wait(30)
      self.driver.get('http://www.baidu.com/')

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


   def test_baidu_title(self):
      self.assertEqual(self.driver.title,'百度一下，你就知道')

if __name__ == '__main__':
```

```
    unittest.main(verbosity=2)
```

再看内容一致，但是数据类型不一致的案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

class BaiduUI(unittest.TestCase):
   def setUp(self):
      self.driver=webdriver.Chrome()
      self.driver.maximize_window()
      self.driver.implicitly_wait(30)
      self.driver.get('http://www.baidu.com/')

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


   def test_baidu_title(self):
      self.assertEqual(self.driver.title,'百度一下，你就知道'.encode('utf-8'))

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

执行如上的代码，会显示失败

assertTrue

返回的是bool类型，也就是对被测试的对象进行验证，如果返回的是boolean类型并且是true，那么结果验证通过，那么方法assertFlase()验证的是被测试对象返回的内容是false,见案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

def login(username,password):
   if username=='wuya' and password=='admin':
      return True
   else:
      return False


class LoginTest(unittest.TestCase):
```

```
    def test_login_true(self):
        self.assertTrue(login('wuya','admin'))

    def test_login_notTrue(self):
        self.assertTrue(login('wuya','weike'))

    def test_login_false(self):
        self.assertFalse(login('wuya','weike'))


if __name__ == '__main__':
    unittest.main(verbosity=2)
```

assertIn

assertIn()值的是一个值是否包含在另外一个值里面，在这里特别的强调一下，在assertIn()的方法里面，有两个参数，那么值的包含其实就是第二个实际参数包含第一个实际参数。与之相反的方法是assergNotIn(),见案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

def login(username,password):
    if username=='wuya' and password=='admin':
        return '无涯课堂'
    else:
        return '请登录'


class LoginTest(unittest.TestCase):
    def test_login_in(self):
        self.assertIn(login('wuya','admin'),'无涯课堂')

    def test_login_in_second(self):
        self.assertIn(login('wuya','admin'),'无涯课堂专注于测试技术的提升和培训')

    def test_login_not_in(self):
        self.assertNotIn(login('wuya','weike'),'无涯课堂')

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

下面我结合UI自动化测试来具体演示它的实际应用，如测试百度首页的url，那么依据该断言的方法，可以编写日下的几个测试方法（测试点只有一个，主要是演示测试断言），案例源码如下：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import  unittest
from selenium import  webdriver


class UiTest(unittest.TestCase):
   def setUp(self) -> None:
      self.driver=webdriver.Chrome()
      self.driver.maximize_window()
      self.driver.get('http://www.baidu.com')
      self.driver.implicitly_wait(30)

   def tearDown(self) -> None:
      self.driver.quit()

   def test_baidu_title_001(self):
      self.assertIn('百度',self.driver.title)

   def test_baidu_title_002(self):
      self.assertIn('百度一下，你就知道',self.driver.title)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

在UI自动化测试中，断言相对来说很好处理，在API的自动化测试中，很多时候返回很多的响应数据，但是一般需要断言好几个的字段，这个时候assertIn()方法就可以使用了，如接口返回的响应数据为：

```
{
  "datas": [
    {
      "author": "无涯",
      "done": true,
      "id": 1,
      "name": "Python自动化测试实战"
    }
  ],
  "msg": "ok",
  "status": 0
}
```

对 这样的响应数据需要断言，如name字段的值以及author需要断，可以使用JSON的方式或者assertIn()的方法，实现API的源码为：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


from flask import   jsonify,Flask
from flask_restful import  Api,Resource

app=Flask(__name__)
api=Api(app=app)

books=[
    {
        'id':1,
        'author':'无涯',
        'name':'Python自动化测试实战',
        "done":True
    }
]

@app.route('/v1/books',methods=['GET'])
def get_books():
   return jsonify({'status':0,'msg':'ok','datas':books})

if __name__ == '__main__':
    app.run(debug=True)
```

具体测试代码为：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import  unittest
import  requests

class Api(unittest.TestCase):
   def test_api_books(self):
      '''获取所有书的信息'''
      r=requests.get(url='http://localhost:5000/v1/books')
      self.assertIn('Python自动化测试',r.json()['datas'][0]['name'])

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

断言中的注意事项

在自动化测试的应用中，测试的结果只有一个，那就是通过或者是不通过，不能存在可能通过或者可能不通过，测试结果必须是权威的，确定性的。

不正确的使用if应用

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

def login(username,password):
   if username=='wuya' and password=='admin':
      return '无涯课堂'
   else:
      return '请登录'


class LoginTest(unittest.TestCase):
   def test_login_in(self):
      text=login('wuya','admin')
      if text=='无涯课堂':
         print('pass')
      else:
         print('fail')
```

不正确的使用异常应用

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
from selenium import  webdriver

def login(username,password):
   if username=='wuya' and password=='admin':
      return '无涯课堂'
   else:
      return '请登录'


class LoginTest(unittest.TestCase):
   def test_login_in(self):
      text=login('wuya','adain')
      try:
         self.assertIn(text,'无涯课堂')
```

```
        except:
            print('fail')
```

批量执行所有的测试用例

在自动化测试中，尽量建议把所有的测试模块放在一个tests的包下，不要分放在多个包下，那样会对批量执行带来一定的困难。当然在实际的应用中，我们都是批量执行所有的测试模块，以及测试模块里面所有的testcase，不可能说单个模块的执行或者是单个testcase的执行。自动化的目的就是应用，在批量执行的时候，首先需要解决的是:**获取到tests包下所有测试模块下的testcase**。下面就结合具体的案例来说明，在tests包下有三个测试模块,分别是test_login.py,test_logout.py和test_user.py，源码分别是如下：

test_login.py的源码：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest

class Login(unittest.TestCase):
    def test_login_001(self):
        pass


    def test_login_002(self):
        pass
```

test_user.py的源码：

```
#!/usr/bin/python3
#coding:utf-8


import  unittest

class User(unittest.TestCase):
    def test_user_001(self):
        pass


    def test_user_002(self):
        pass
```

test_logout.py的源码:

```
#!/usr/bin/python3
#coding:utf-8


import  unittest

class Logout(unittest.TestCase):
   def test_logout_001(self):
      pass


   def test_logout_002(self):
      pass
```

获取tests包下所有的测试模块里面的testcase，使用到的方法是discover()，见实现allTests.py源码：

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
import  os


def getSuite():
   tests=unittest.defaultTestLoader.discover(

 start_dir=os.path.join(os.path.dirname(os.path.dirname(__file__)),'tests'),
      pattern='test_*.py',
      top_level_dir=None
   )
   return tests


if __name__ == '__main__':
    print(getSuite())
```

见执行后输出的结果信息：

```
< unittest.suite.TestSuite
tests = [ < unittest.suite.TestSuite
tests = [ < unittest.suite.TestSuite
tests = [ < test_baidu.LoginTest
testMethod = test_login_in >] >] >, < unittest.suite.TestSuite
tests = [ < unittest.suite.TestSuite
tests = [ < test_login.Login
testMethod = test_login_001 >, < test_login.Login
testMethod = test_login_002 >] >] >, < unittest.suite.TestSuite
tests = [ < unittest.suite.TestSuite
```

```
tests = [ < test_logout.Logout
testMethod = test_logout_001 >, < test_logout.Logout
testMethod = test_logout_002 >] >] >, < unittest.suite.TestSuite
tests = [ < unittest.suite.TestSuite
tests = [ < test_user.User
testMethod = test_user_001 >, < test_user.User
testMethod = test_user_002 >] >] >] >
```

依据如上的输出信息，可以看到它的类型为：<class 'unittest.suite.TestSuite'>,转为list类型再输出，见源码:

```
#!/usr/bin/python3
#coding:utf-8

import  unittest
import  os


def getSuite():
   tests=unittest.defaultTestLoader.discover(

 start_dir=os.path.join(os.path.dirname(os.path.dirname(__file__)),'tests'),
      pattern='test_*.py',
      top_level_dir=None
   )
   return tests


if __name__ == '__main__':
    for modules in list(getSuite()):
       for suites in modules:
          for testCase in suites:
             print(testCase)
```

执行后，见输出的结果信息：

test_login_001 (test_login.Login) test_login_002 (test_login.Login) test_logout_001 (test_logout.Logout) test_logout_002 (test_logout.Logout) test_user_001 (test_user.User) test_user_002 (test_user.User)

unittest中测试报告的详解

在unittest的框架中，生成测试报告需要使用到HTMLTestRunner，它的下载地址为:

https://github.com/tungwaiyip/HTMLTestRunner

由于早期是python2的，针对python3它的最新源码需要做修改。

```
#!/usr/bin/python3
```

```
#coding:utf-8

import  unittest
import  os
import  time
import HTMLTestRunner



def getTests():
   tests=unittest.defaultTestLoader.discover(

 start_dir=os.path.join(os.path.dirname(os.path.dirname(__file__)),'tests'),
      pattern='test_*.py',
      top_level_dir=None
   )
   return tests

def getNowTime():
   return time.strftime('%y-%m-%d %h_%m_%s',time.localtime(time.time()))


def run():
   filename=getNowTime()+'report.html'
   fp=open(filename,'wb')
   runner=HTMLTestRunner.HTMLTestRunner(
      stream=fp,
      title='',
      description=''
   )
   runner.run(getTests())

if __name__ == '__main__':
    run()
```

执行如上的代码后，就会生成基于html的测试报告，以及测试报告显示的内容，见官方的报告案例截图：

http://tungwaiyip.info/software/sample_test_report.html，在该链接里面就能够看到案例的测试报告地址。

Pytest实战

Pytest的应用

Pytest初步应用

Pytest是基于Python语言的单元测试框架，也是一个命令行的工具，可以自动的找到测试用例执行和反馈反馈测试结果信息，在编写测试点方面比较自由，可以使用函数式的编程等语言，也可以使用面向对象的编程语言。而且它的断言使用的是Python原生的assert关键字，同时Pytest测试框架可以很完美的和unittest整合应用到一起，能够很好的进行兼容。Pytest比起unittest来说比较自由，使用unittest首先要继承TestCase的类，但是pytest是不需要的，安装成功后，直接编写测试函数或者测试方法就可以使用了。下面具体来说Pytest测试框架与unittest测试框架的区别。

unittest	**Pytest**
测试类里面需要继承unittest.TestCase类	不需要继承，可以是一个函数，也可以是类
参数化需要依赖第三方的库	不需要依赖，直接使用内部的parametrize
测试报告HTMLTestRunner	Pytest-html或者是allure
没有插件	有很丰富的插件
不支持失败重试	支持失败重试

安装的命令为：

Pip3 install pytest

安装成功后，就可以直接的使用。在pytest中，它会首先寻找以test*开头或者以*test结尾的测试模块，然后执行模块里面**以test*开头或者是以*test结尾的测试代码**，这里依据这个要去，编写测试模块，见案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import pytest

def add(a,b):
   return a+b

def test_add_integer():
   assert  add(2,3)==5


def test_add_str():
   assert  add('hi',' wuya')=='hi wuya'
```

执行命令pytest，见输出的信息:

```
Test session starts (platform: darwin, Python 3.7.4, pytest 4.0.2, pytest-sugar
0.9.2)
rootdir: /Applications/code/stack/xunit, inifile:
plugins: xdist-1.29.0, forked-1.0.2, sugar-0.9.2, html-1.22.0, cov-2.7.1,
allure-adaptor-1.7.10, metadata-1.8.0
collecting ...
 test_pytest01.py ✓✓                                                  100%
██████████

Results (0.05s):
       2 passed
```

其实在一个模块里面，不仅仅是包含了函数，也包含了类，那么看一个模块包含了类，函数，它的执行结果：

```
#!/usr/bin/python3
#coding:utf-8

import pytest

def add(a,b):
   return a+b

def test_add_integer():
   assert  add(2,3)==5


def test_add_str():
   assert  add('hi',' wuya')=='hi wuya'

class TestAdd(object):
   def test_add_integer(self):
      assert add(2, 3) == 5

   def test_add_str(self):
      assert add('hi', ' wuya') == 'hi wuya'
```

见执行后输出的信息：

```
Test session starts (platform: darwin, Python 3.7.4, pytest 4.0.2, pytest-sugar
0.9.2)
rootdir: /Applications/code/stack/xunit, inifile:
plugins: xdist-1.29.0, forked-1.0.2, sugar-0.9.2, html-1.22.0, cov-2.7.1,
allure-adaptor-1.7.10, metadata-1.8.0
collecting ...
 test_pytest01.py ✓✓✓✓                                              100%
██████████

Results (0.05s):
       4 passed
```

当然也可以看到它的详细的信息，执行的命令为：pytest -v ，见执行后输出的信息：

```
Test session starts (platform: darwin, Python 3.7.4, pytest 4.0.2, pytest-sugar
0.9.2)
cachedir: .pytest_cache
metadata: {'Python': '3.7.4', 'Platform': 'Darwin-18.7.0-x86_64-i386-64bit',
'Packages': {'pytest': '4.0.2', 'py': '1.8.0', 'pluggy': '0.12.0'}, 'Plugins':
{'xdist': '1.29.0', 'forked': '1.0.2', 'sugar': '0.9.2', 'html': '1.22.0',
'cov': '2.7.1', 'allure-adaptor': '1.7.10', 'metadata': '1.8.0'}, 'JAVA_HOME':
'/Library/Java/JavaVirtualMachines/jdk-12.0.2.jdk/Contents/Home'}
rootdir: /Applications/code/stack/xunit, inifile:
plugins: xdist-1.29.0, forked-1.0.2, sugar-0.9.2, html-1.22.0, cov-2.7.1,
allure-adaptor-1.7.10, metadata-1.8.0
collecting ...
 test_pytest01.py::test_add_integer ✓                               25% ██
 test_pytest01.py::test_add_str ✓                                   50% ████
 test_pytest01.py::TestAdd.test_add_integer ✓                       75% ██████

 test_pytest01.py::TestAdd.test_add_str ✓                          100%
██████████

Results (0.06s):
       4 passed
```

依据如上的信息可以得出，pytest的使用规范总结如下：

1、要执行的测试模块必须是test开头的

2、要执行的测试函数必须是test开头或者是test为结尾的

3、如果是一个类要被pytest执行，那么该类的名称首字母必须是Test，否则不会被执行,这部分见案例的代码：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest
```

```
def add(a,b):
   return a+b

class TestAdd(object):
   def test_add_int(self):
      assert  add(2,3)==5

   def test_add_str(self):
      assert  add('hi',' 无涯')=='hi 无涯'

   def test_add_float(self):
      add(1.0,2.0)==3.0
```

如上的代码执行后的输出信息为：

collecting ...

test_pytest01.py::TestAdd.test_add_int ✓ 33% ████

test_pytest01.py::TestAdd.test_add_str ✓ 67% ████████

test_pytest01.py::TestAdd.test_add_float ✓ 100% ████████████

Results (0.03s):

3 passed

如果把类的名称修改为：AddTest()，修改后的源码为：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest

def add(a,b):
   return a+b

class AddTest(object):
   def test_add_int(self):
      assert  add(2,3)==5

   def test_add_str(self):
      assert  add('hi',' 无涯')=='hi 无涯'

   def test_add_float(self):
      add(1.0,2.0)==3.0
```

再次执行pytest -v ，输出的信息显示执行测试用例为0，这点需要特别的注意。当然很多时候，为了测试的需要，建议把测试的模块放在tests的测试包里面，测试模块建议都使用test_模块名称这样的方式来命名更加好点。

在Pytest中，执行成功表示为PASSED,执行失败表示为FAILED，源码为：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest

def add(a,b):
   return a+b

class TestAdd(object):
   def test_add_int(self):
      assert  add(2,3)==5

   def test_add_str(self):
      assert  add('hi',' 无涯')=='hi无涯'

   def test_add_float(self):
      add(1.0,2.0)==3.0
```

pytest -v执行如上的测试类后，输出的信息为：

collecting ...

test_pytest01.py::TestAdd.test_add_int ✓ 33% ████

———————————————————————————— TestAdd.test_add_str ————————————————————————————

self = <xunit.test_pytest01.TestAdd object at 0x108e10b10>

def test_add_str(self):

> assert add('hi',' 无涯')=='hi无涯'

E AssertionError: assert 'hi 无涯' == 'hi无涯'

E - hi 无涯

E ? -

E + hi无涯

test_pytest01.py:14: AssertionError

test_pytest01.py::TestAdd.test_add_str × 67% ████████▌

test_pytest01.py::TestAdd.test_add_float ✓ 100% ██████████████

Results (0.09s):

2 passed

1 failed

- test_pytest01.py:13 TestAdd.test_add_str

依据如上的信息，就能够很准确的看出执行成功的和执行失败的，以及执行失败的给出的错误信息。

结果信息

在Pytest的测试框架中，执行测试模块或者测试函数以及测试方法后，就会显示出每个测试用例的结果信息，在Ui常用的结果信息主要包含如下：

Failure：失败

Error:异常

Skip:跳过

xFail:预期失败

xPass:预期成功

执行结果后，分别显示为：F,E,s,x,X，使用命令-v可以查看详细的信息，能够展示出来,具体见如下的案例代码:

```
#!/usr/bin/env python
#!coding:utf-8

import pytest


def test_passing():
   assert  (1,2,3)==(1,2,3)


def test_failing():
   assert  1==2

@pytest.mark.skip()
def test_skip():
   assert 1==1
```

```python
@pytest.mark.xfail()
def test_xfail():
    assert 1==2


@pytest.mark.xpass()
def test_xpass():
    assert 1==1
```

执行如上后，展示信息如下：

```
collecting ...
 test_one.py ✓                                                      20% ██

---------------------------------- test_failing ----------------------------------

    def test_failing():
>       assert  1==2
E    assert 1 == 2

test_one.py:12: AssertionError

 test_one.py xsx✓                                                  100% ███████
============================== warnings summary ===============================
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/_p
ytest/mark/structures.py:324
  /Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages/
_pytest/mark/structures.py:324: PytestUnknownMarkWarning: Unknown pytest.mark.xp
ass - is this a typo?  You can register custom marks to avoid this warning - for
 details, see https://docs.pytest.org/en/latest/mark.html
    PytestUnknownMarkWarning,

-- Docs: https://docs.pytest.org/en/latest/warnings.html

Results (0.20s):
       2 passed
       1 failed
         - test_one.py:11 test_failing
       1 xfailed
       1 skipped
```

测试搜索

测试搜索指的是在pytest的测试框架中，如果没有指定目录，它默认是会搜索一个项目下所有可执行的测试模块以及测试模块里面的测试用例来进行的，并不在乎测试用例是在哪个package那个模块的，这样的一个过程成为“测试搜索”，只要符合它的规则的它都是会被执行的。所以一般建议把需要执行的测试模块都放在tests的包下面，每个测试模块都已test_这样的方式开头。在具体执行的过程中，只需要进入到tests的包里面后，执行pytest -v，它会搜索到符合规则的测试模块中的测试函数或者测试方法，然后按顺序的执行。

Pytest测试用例执行顺序

在unittest的单元测试框架中，测试用例执行的顺序是依据ascil码来执行的，在pytest的测试框架它是根据编写的测试用例顺序执行的，见案例的代码：

```
#!/usr/bin/python3
#coding:utf-8


def test_add():
    pass

def test_aaa():
    pass

def test_abc():
    pass

def test_001():
    pass

class TestAscill():
    def test_001(self):
        pass

    def test_add(self):
        pass

    def test_aaa(self):
        pass

    def test_abc(self):
        pass
```

执行该模块后，测试用例执行的顺序为从上到下执行，具体见执行后的输出信息：

```
**collecting ...**

 test_ascill.py::test_add ✓                    12% ▉

 test_ascill.py::test_aaa ✓                    25% ▉▉

 test_ascill.py::test_abc ✓                    38% ▉▉▉

 test_ascill.py::test_001 ✓                    50% ▉▉▉▉

 test_ascill.py::TestAscill.test_001 ✓              62% ▉▉▉▉▉▉
```

```
 test_ascill.py::TestAscill.test_add ✓              75% ███████

 test_ascill.py::TestAscill.test_aaa ✓              88% ████████

 test_ascill.py::TestAscill.test_abc ✓             100% ██████████

Results (0.03s):

    8 passed
```

Pytest执行命令

在Pytest的测试框架执行中会使用到很多的命令，具体在下面逐步的介绍这些命令的使用。在执行中，有这么几个需求，分别是全量执行，单独执行某一个模块的测试代码，或者执行一个模块里面的测试函数，或者是执行一个测试类里面的测试方法，具体执行的命令如下：

全量执行

pytest -v :执行测试包下所有的测试模块

执行某一个测试模块

pytest -v 测试模块名称

执行模块下的测试函数

pytest -v 测试模块::测试函数

pytest -v test_ascill.py::test_001,执行后，输出的信息为：

```
**collecting ...**

 test_ascill.py::test_001 ✓                   100% ██████████

Results (0.02s):

    1 passed
```

执行模块下测试类里面的测试方法

pytest -v 测试模块::测试类::测试方法

pytest -v test_ascill.py::TestAscill::test_001,执行后，输出的结果信息：

```
**collecting ...**

 test_ascill.py::TestAscill.test_001 ✓             100% ██████████

Results (0.02s):

    1 passed
```

执行一个包下所有的测试点

pytest -v tests/ ,输出结果为：

```
 tests/test_ascill.py ✓✓✓                                    50% ████▌
 tests/test_pytest01.py ✓                                    67% ██████▋

 tests/test_pytest01.py ×✓                                  100%
██████████

Results (0.09s):
       5 passed
       1 failed
         - xunit/test_pytest01.py:13 TestAdd.test_add_str
```

按分类执行测试用例

在编写测试点的时候，由于业务的需求，会编写不同分类的测试点，比如在一个测试模块中会编写登录的测试点，也会编写退出的测试点，当然还有其他的测试点，具体见案例代码：

```
#!/usr/bin/python3
#coding:utf-8


def test_login_001():
   pass

def test_login_002():
   pass

def test_login_003():
   pass

def test_logout_001():
   pass

def test_logout_002():
   pass

def test_profile_001():
   pass

def test_profile_002():
   pass
```

如果只想执行login和profile的测试代码，那么需要使用到-k的命令，它主要是允许使用

表达式指定希望运行的测试用例，或者多个前缀或者是后缀的测试用例名称相同，--collect-only主要的应用于筛选

pytest -v -k "login or profile" --collect-only，见执行命令后的输出信息：

```
**collecting ...** <Package '/Applications/code/stack/xunit'>

 <Module 'test_ascill.py'>

  <Function 'test_login_001'>

  <Function 'test_login_002'>

  <Function 'test_login_003'>

  <Function 'test_profile_001'>

  <Function 'test_profile_002'>


Results (0.03s):

    5 deselected
```

选择出来后，就可以执行这些测试点了，具体执行的命令为:

pytest -v -k "login or profile",见执行后的输出信息:

collecting ...

test_ascill.py::test_login_001 ✓ 20% ███

test_ascill.py::test_login_002 ✓ 40% ██████

test_ascill.py::test_login_003 ✓ 60% ████████

test_ascill.py::test_profile_001 ✓ 80% ██████████

test_ascill.py::test_profile_002 ✓ 100% ████████████

Results (0.03s):

5 passed

5 deselected

快速找到分组

在上面的案例中，说是分类，不是说是分组，命令-m可以快速的找到分组并且执行，对代码进行修改，增加装饰器,案例代码为：

```
#!/usr/bin/python3
#coding:utf-8
```

```
import  pytest

@pytest.mark.login
def test_login_001():
   pass

@pytest.mark.login
def test_login_002():
   pass

@pytest.mark.login
def test_login_003():
   pass

@pytest.mark.logout
def test_logout_001():
   pass

@pytest.mark.logout
def test_logout_002():
   pass

@pytest.mark.profile
def test_profile_001():
   pass

@pytest.mark.profile
def test_profile_002():
   pass
```

只执行login的分组，命令为：

pytest -v -m login

collecting ...

test_ascill.py::test_login_001 ✓ 33% ████▌

test_ascill.py::test_login_002 ✓ 67% ████████▌

test_ascill.py::test_login_003 ✓ 100% ████████████

Results (0.06s):

3 passed

7 deselected

当然也可以执行多个分组，之间是关系就是or的关系，具体命令为：

pytest -v -m "login or logout"，见执行后输出的结果信息：

collecting ...

test_ascill.py::test_login_001 ✓ 20% ███

test_ascill.py::test_login_002 ✓ 40% ██████

test_ascill.py::test_login_003 ✓ 60% ████████

test_ascill.py::test_logout_001 ✓ 80% ██████████

test_ascill.py::test_logout_002 ✓ 100% ████████████

Results (0.03s):

5 passed

5 deselected

失败立刻停止：-x

我们希望失败立刻停止，见具体的案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest

@pytest.mark.login
def test_login_001():
   assert  1==2

@pytest.mark.login
def test_login_002():
   pass

@pytest.mark.login
def test_login_003():
   pass

@pytest.mark.logout
def test_logout_001():
   pass

@pytest.mark.logout
def test_logout_002():
   pass

@pytest.mark.profile
def test_profile_001():
   pass
```

```
@pytest.mark.profile
def test_profile_002():
   pass
```

执行命令为：

pytest -v -x -m login

collecting ...

———————————————————————————————— test_login_001 ————————————————————————————————

@pytest.mark.login

def test_login_001():

> assert 1==2

E assert 1 == 2

test_ascill.py:8: AssertionError

test_ascill.py::test_login_001 × 33% ████▌

Results (0.08s):

1 failed

- test_ascill.py:6 test_login_001

7 deselected

指定失败次数:--maxfail

当然也可以指定失败的次数，比如最大能够忍耐的失败次数，比如2，或者0，案例就为2，命令为：

pytest -v -x --maxfail=2 -m login ,见执行后输出的结果信息：

```
collecting ...

———————————————————————————— test_login_001 ————————————————————————————

    @pytest.mark.login
    def test_login_001():
>       assert  1==2
E    assert 1 == 2
```

```
test_ascill.py:8: AssertionError

 test_ascill.py::test_login_001 ×                                 33% ███
 test_ascill.py::test_login_002 ✓                                 67% ██████
 test_ascill.py::test_login_003 ✓                                100% ████████

Results (0.06s):
       2 passed
       1 failed
         - test_ascill.py:6 test_login_001
       7 deselected
```

输出信息展示

--tb=no:关闭信息

--tb=short:只输出assert的错误信息

--tb=line:一行展示所有的信息，具体分别看如下的信息：

pytest -v -x -m login --tb=no，输出结果：

```
collecting ...

 test_ascill.py::test_login_001 ×                                 33% ███

Results (0.08s):
       1 failed
         - test_ascill.py:6 test_login_001
       7 deselected
```

pytest -v -x -m login --tb=short,输出的信息：

```
collecting ...

_____________________________ test_login_001 ______________________________
test_ascill.py:8: in test_login_001
    assert  1==2
E   assert 1 == 2

 test_ascill.py::test_login_001 ×                                 33% ███

Results (0.06s):
       1 failed
         - test_ascill.py:6 test_login_001
       7 deselected
```

pytest -v -x -m login --tb=line，输出的信息：

```
collecting ...
/Applications/code/stack/xunit/test_ascill.py:8: assert 1 == 2

 test_ascill.py::test_login_001 ×                                  33% ███

Results (0.06s):
       1 failed
         - test_ascill.py:6 test_login_001
       7 deselected
```

定位错误:--lf

在执行的时候出错，我们希望能够快速定位出具体是哪一行出错了，就会使用到--lf，命令为：

pytest --lf -m login，输出结果信息：

```
collecting ... run-last-failure: rerun previous 2 failures


_____________________________ test_login_001 ______________________________

    @pytest.mark.login
    def test_login_001():
>       assert  1==2
E     assert 1 == 2

test_ascill.py:8: AssertionError

 test_ascill.py ×                                                  100% ████████

Results (0.06s):
       1 failed
         - test_ascill.py:6 test_login_001
       9 deselected
```

遇到错误继续执行:--ff

-x是遇到错误立刻停止，我们还是希望遇到错误后，继续执行，不要因为一个错误而导致不执行，命令：

pytest --ff -m login

```
collecting ... run-last-failure: rerun previous 2 failures first


_____________________________ test_login_001 ______________________________

    @pytest.mark.login
    def test_login_001():
```

```
>       assert  1==2
E    assert 1 == 2

test_ascill.py:8: AssertionError

 test_ascill.py ×✓✓                                                  100% ██████████

Results (0.06s):
       2 passed
       1 failed
         - test_ascill.py:6 test_login_001
       7 deselected
```

测试用例执行速度:--duration=0

这里以测试几个主流的搜索引擎公司为案例，来看测试用例的执行速度，案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest
import  requests


def test_baidu_speed():
   r=requests.get('http://www.baidu.com/')
   assert  r.status_code==200

def test_bing_speed():
   r=requests.get('http://www.bing.com/')
   assert  r.status_code==200

def test_sina_speed():
   r=requests.get('http://www.sina.com/')
   assert  r.status_code==200
```

执行命令：

pytest -v --duration=0 test_ascill.py,执行命令为输出的结果信息：

```
collecting ...
 test_ascill.py::test_baidu_speed ✓                                  33% ███
 test_ascill.py::test_bing_speed ✓                                   67% ██████
 test_ascill.py::test_sina_speed ✓                                  100% ██████████
============================ slowest test durations =============================
0.20s call     test_ascill.py::test_bing_speed
0.12s call     test_ascill.py::test_sina_speed
0.09s call     test_ascill.py::test_baidu_speed

(0.00 durations hidden.  Use -vv to show these durations.)

Results (0.52s):
```

忽略执行: -rs

在编写的测试点中，有的测试用例我们是忽略执行的，先来看案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest

@pytest.mark.login
def test_login_001():
   pass

@pytest.mark.skip(reason='忽略执行该测试点')
def test_login_002():
   pass
```

执行命令:

pytest -rs，执行后，输出的信息就会展示出忽略执行的具体原因，输出结果为：

```
collecting ...
 test_ascill.py::test_login_001 ✓                                    50% █████
 test_ascill.py::test_login_002 s                                   100%
██████████
============================ short test summary info
============================
SKIP [1] test_ascill.py:10: 忽略执行该测试点

Results (0.02s):
       1 passed
       1 skipped
```

如果没有rs，就不会显示出详细的信息，见没-rs输出的结果信息：

```
collecting ...
 test_ascill.py::test_login_001 ✓                                    50% ████▌
 test_ascill.py::test_login_002 s                                   100%
████████▌

Results (0.03s):
       1 passed
       1 skipped
```

分组执行:-m -k

前面介绍过对测试点的分组执行，k主要是应用测试搜索的方式，而m的方式是使用mark分组的方式。
案例代码：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest

@pytest.mark.login
def test_login_001():
   pass

@pytest.mark.login
def test_login_002():
   pass

@pytest.mark.login
def test_logout_001():
   pass
```

pytest -k "login",输出的结果信息为：

```
collecting ...
 test_ascill.py::test_login_001 ✓                                    33% ███▎

 test_ascill.py::test_login_002 ✓                                    67% ██████▋
 test_ascill.py::test_logout_001 ✓                                  100%
████████▌

Results (0.03s):
       3 passed
```

pytest -m login，输出的结果信息：

```
collecting ...
 test_ascill.py::test_login_001 ✓                                  33% ███
 test_ascill.py::test_login_002 ✓                                  67% ██████
 test_ascill.py::test_logout_001 ✓                                100% ██████████

Results (0.02s):
       3 passed
```

测试执行速度:--duration=N

使用该选项，可以加速测试用例的执行,当然有一点需要记住的是它不关心测试内部是怎么实现的，只统计测试过程中那个阶段执行速度是最慢的，如果参数是--duration=0，它就会按所有阶段耗时从长到短进行排序，见如下的测试代码：

```
#!/usr/bin/env python
#!coding:utf-8

import pytest
import requests


def test_baidu():
   r=requests.get(url='http://www.baidu.com/')
   assert r.status_code==200


def test_taobao():
   r=requests.get(url='http://www.taobao.com/')
   assert r.status_code==200


def test_360():
   r=requests.get(url='http://www.360.com/')
   assert r.status_code==200
```

执行命令后的结果信息，具体如下图所示：

```
collecting ...
 test_one.py::test_baidu ✓                                      33%
 test_one.py::test_taobao ✓                                     67%
 test_one.py::test_360 ✓                                       100%
============================ slowest test durations ============================
0.23s call     test_one.py::test_360
0.23s call     test_one.py::test_taobao
0.08s call     test_one.py::test_baidu

(0.00 durations hidden.  Use -vv to show these durations.)

Results (0.59s):
       3 passed
```

Pytest与WebDriver

首先安装对应的库，安装的命令为：

pip install pytest-selenium

selenium是主流的基于web应用程序的测试框架，它支持主流的浏览器，当然主流的开发语言也支持该测试框架在UI自动化测试中的应用。在pytest的测试框架中，需要特别强调的是，在一个测试函数或者测试方法中，selenium是固定参数，指的是driver，而什么是driver,driver是webdriver实例化后的对象。这里以测试百度首页的请求地址和title的测试点为案例，来说明pytest-selenium的应用，具体案例代码如下：

```
#!/usr/bin/python3
#coding:utf-8

import  pytest


def test_baidu_title(selenium):
    '''验证：百度首页的title是否正确'''
    selenium.get('https://www.baidu.com/')
    assert selenium.title=='百度一下，你就知道'

def test_baidu_url(selenium):
    '''验证：百度首页的url是否正确'''
    selenium.get('http://www.baidu.com/')
    assert  selenium.current_url=='https://www.baidu.com/'


def test_baidu_soKeyword(selenium):
    '''验证：百度搜索输入框输入的关键字'''
    selenium.get('http://www.baidu.com/')
    so=selenium.find_element_by_id('kw')
    so.send_keys('无涯课堂')
    assert  so.get_attribute('value')=='无涯课堂'
```

下面使用命令执行如下的代码，执行的命令为：

pytest -v 测试模块 --driver Chrome(指定测试的浏览器)，具体执行命令为：

pytest -v test_ascill.py --driver Chrome，执行后输出的结果信息：

```
collecting ...
 test_ascill.py::test_baidu_title ✓
33% ███
 test_ascill.py::test_baidu_url ✓
67% ██████
 test_ascill.py::test_baidu_soKeyword ✓
 100% ██████████

Results (7.86s):
       3 passed
```

依据如上的信息可以看出，执行后的结果信息。

我们继续编写，使用pytest-selenium页面对象设计模式，如果对页面对象设计模式有不清楚的同学后面在页面对象设计模式有详细的介绍，这里就不说明，直接案例代码来演示和说明，见案例代码：

Pytest中参数化实战

参数化实战

熟悉ddt的同学，该库它在测试中的应用还是很有价值的，相同的执行步骤，不同的期望结果，我们可以结合ddt的库来实现一个测试用例的代码实现多个测试点的测试，当然ddt也是存在缺陷的，出错后不好定位问题。但是优点大于缺陷的。在pytest的测试框架中，我们也是可以通过parametrize来实现参数化的测试，达到刚才说的测试点，比如写一个add()的函数，那么现在来测试两个数的相加，会存在很多的测试场景的，具体还是依据被测试的函数以及测试案例代码，源码部分为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest


def add(a,b):
   return a+b

@pytest.mark.parametrize('a,b,result',[
   (1,2,3),
   (2,3,5),
   ('Hi,','无涯','Hi,无涯'),
   (1.0,1.0,2.0),
   (1,1.0,2.0),
   (0,0,0)])
```

```
def test_add_many(a,b,result):
    assert add(a,b)==result
```

备注：在如上的代码中，在装饰器的@pytest.mark.parametrize中，第一个参数里面三个指，分别是函数的形式参数，以及函数返回值的结果，后面列表里面的元组，就是测试需要使用到的数据，也就是参数化部分，在执行的过程中，会对列表里面的元组依次循环并且赋值给a,b,result。执行如上的代码后，输出的结果信息为：

```
collecting ...
 test_ascill.py::test_add_many[1-2-3] ✓
17% ██
 test_ascill.py::test_add_many[2-3-5] ✓
33% ████
 test_ascill.py::test_add_many[Hi,-\u65e0\u6daf-Hi,\u65e0\u6daf] ✓
 50% ██████
 test_ascill.py::test_add_many[1.0-1.0-2.0] ✓
67% ████████
 test_ascill.py::test_add_many[1-1.0-2.0] ✓
83% ██████████
 test_ascill.py::test_add_many[0-0-0] ✓
 100% ████████████

Results (0.06s):
       6 passed
```

依据如上的信息可以看到，执行的结果信息都成功，我们故意修改代码，让其中错误一个场景，看错误信息的提示，再次执行，输出的结果信息：

```
collecting ...
 test_ascill.py::test_add_many[1-2-3] ✓
17% ██
 test_ascill.py::test_add_many[2-3-5] ✓
33% ████
 test_ascill.py::test_add_many[Hi,-\u65e0\u6daf-Hi,\u65e0\u6daf] ✓
 50% ██████
 test_ascill.py::test_add_many[1.0-1.0-2.0] ✓
67% ████████
 test_ascill.py::test_add_many[1-1.0-2.0] ✓
83% ██████████

_____________________________________ test_add_many[0-0-1]
_____________________________________

a = 0, b = 0, result = 1

    @pytest.mark.parametrize('a,b,result',[
       (1,2,3),
       (2,3,5),
```

```
        ('Hi,','无涯','Hi,无涯'),
        (1.0,1.0,2.0),
        (1,1.0,2.0),
        (0,0,1)])
    def test_add_many(a,b,result):
>       assert add(a,b)==result
E    assert 0 == 1
E     +  where 0 = add(0, 0)

test_ascill.py:19: AssertionError

 test_ascill.py::test_add_many[0-0-1] ×
 100% ██████████

Results (0.13s):
       5 passed
       1 failed
         - test_ascill.py:11 test_add_many[0-0-1]
```

依据如上的信息可以看到，即使其中有一个测试场景错误了，提示的信息也是很明晰的，parametrize在测试某些场景方面是非常具备优势的，可以写最少的代码，实现最大的测试场景。在前面说到，测试函数执行的过程本质上其实就是对列表的元组进行循环不断赋值的过程，那么可以对代码进行完善，完全把参数化的部分放到列表中，完善后的代码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest

def add(a,b):
   return a+b

def listParam():
   list1=[
      (1, 2, 3),
      (2, 3, 5),
      ('Hi,', '无涯', 'Hi,无涯'),
      (1.0, 1.0, 2.0),
      (1, 1.0, 2.0),
      (0, 0, 0)
   ]
   return list1

@pytest.mark.parametrize('a,b,result',listParam())
def test_add_many(a,b,result):
   assert add(a,b)==result
```

对参数化的分离到了一个列表中，其他执行的本质上不会有改变的，执行命令后输出的结果信息为：

```
collecting ...
 test_ascill.py::test_add_many[1-2-3] ✓                                  17% ██

 test_ascill.py::test_add_many[2-3-5] ✓                                  33%
███
 test_ascill.py::test_add_many[Hi,-\u65e0\u6daf-Hi,\u65e0\u6daf] ✓       50%
████
 test_ascill.py::test_add_many[1.0-1.0-2.0] ✓                            67%
██████
 test_ascill.py::test_add_many[1-1.0-2.0] ✓                              83%
███████
 test_ascill.py::test_add_many[0-0-0] ✓                                 100%
█████████

Results (0.07s):
       6 passed
```

固件request

在Pytest的测试框架中，参数化也会使用到pytest内置的固件request，通过request.param来获取参数，对上面的案例代码进行修改.fixture参数列表中的request也是内建的fixture,代表了fixture的调用状态。完善后的代码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest

def add(a,b):
   return a+b

def listParam():
   list1=[
      {'a':1,'b':2,'result':3},
      {'a': 'Hi ', 'b': '无涯', 'result': 'Hi 无涯'},
      {'a':1.0,'b':1.0,'result':2.0}
   ]
   return list1

@pytest.fixture(params=listParam())
def param(request):
   return request.param


def test_add_many(param):
```

```
    assert add(param['a'],param['b'])==param['result']
```

如上是修改后的代码，执行如上的代码后，输出的结果为：

```
collecting ...
 test_ascill.py::test_add_many[param0] ✓                                  33%
███
 test_ascill.py::test_add_many[param1] ✓                                  67%
██████
 test_ascill.py::test_add_many[param2] ✓                                 100%
█████████

Results (0.14s):
       3 passed
```

参数化场景

参数化的场景我们可以把它应用在对一个点测试，但是可以有很多不同的数据来进行，并且它的步骤是一样的，比如有这样的一个业务，我们就可以使用参数化的方式来编写测试点。添加不同类型的用户，用户添加后需要验证添加的用户名称是否正确，那么假设添加不同类型的用户有几十个，按照常规的思维我们会编写几十个步骤一样的测试用例，但是参数化的思想，就可以使用很少的代码来实现刚的部分，下面还是通过具体的案例来说明这部分的案例应用。案例源码：

```
#!/usr/bin/env python
#!coding:utf-8


from flask import  Flask,jsonify
from flask_restful import  Api,Resource,reqparse


app=Flask(__name__)
api=Api(app)

class UserView(Resource):

   def post(self):
      parser=reqparse.RequestParser()
      parser.add_argument('username', type=str, required=True, help='用户名不能为空')
      parser.add_argument('password',type=str,required=True,help='账户密码不能为空')
      parser.add_argument('auth',type=str,required=True,help='用户权限不能为空')
      args=parser.parse_args()
      return jsonify({'status':0,'msg':'','datas':args})
```

```
api.add_resource(UserView,'/user',endpoint='user')

if __name__ == '__main__':
    app.run(debug=True)
```

添加不同的用户，来验证它不同的权限，结合参数化的思路，来实现这个测试用例的编写：

```
#!/usr/bin/env python
#!coding:utf-8


import  pytest
import requests
import json


def datas():
    return [
        ('wuya','admin','超级管理员'),
        ('weike','admin','管理员')
    ]


@pytest.mark.parametrize('data',datas())
def test_user_auth_api(data):
    dict1={"username":data[0],"password":data[1],"auth":data[2]}
    r=requests.post(
        url='http://127.0.0.1:5000/user',
        json=dict1)
    assert r.json()['datas']['auth']==data[2]

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

Pytest的fixture

在单元测试的组件中，主要分为测试用例，测试固件，测试套件，测试执行以及测试报告，看过我书的

同学对这些应该很清晰。测试固件也是不难理解，也就是在测试用例执行前需要做的动作和测试执行后需要

做的事情。比如在UI的自动化测试中，我们更加关注的是对页面的操作，而不是关心打开浏览器和关闭浏览器，

在数据库的操作中，更加关注的是对MySQL的基本操作，而不怎么关心连接数据库和数据库断开连接这部分。

所以打开浏览器和关闭浏览器，连接数据库和关闭数据库部分，可以让测试固件去干，测试用例的层面更加

关心测试用例的执行结果以及断言结果。在pytest的测试框架中，测试固件有各种形式的表现，比如除了刚才

说的初始化与清理外，还有它强大的参数化的部分。下面还是通过具体的案例来说明这部分的应用。

fixture返回值

fixture是在测试函数运行前后，首先来看fixture在返回值中的案例应用，如一个系统登录成功后，返回token，需要访问个人主页，前提是在登录成功的前提下。来看fixture在该案例中的应用，案例代码：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest


@pytest.fixture()
def login(username='wuya',password='admin'):
   if username=='wuya' and password=='admin':
      return 'sdrtyuds344dfgsdfgh345'
   else:
      raise  '用户名或者密码错误，请核对'

def test_profile(login):
   '''验证：访问个人主页'''
   assert  login=='sdrtyuds344dfgsdfgh345'
```

执行后，见输出的结果信息：

```
collecting ...
 test_ascill.py::test_profile ✓
 100% ██████████

Results (0.06s):
       1 passed
```

装饰器@pytest.fixture(),它是声明一个函数是fixture，如果测试函数的参数列表中包含了fixture名，那么pytest执行的时候，就会检测到，并且在测试函数运行之前执行该fixture，fixture可以完成任务，也可以返回数据给测试函数。

依据返回值的思想，看fixture在UI自动化测试中的应用，案例代码如下：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest
```

```
from selenium import  webdriver

@pytest.fixture()
def driver():
   return  webdriver.Chrome()

def test_baidu_title(driver):
   driver.maximize_window()
   driver.get('http://www.baidu.com')
   assert  driver.title=='百度一下，你就知道'
```

初始化与清理

UI的案例应用

接着来看另外的案例，分别是UI自动化测试的应用，首先不管测试什么样的测试场景，都是需要打开浏览器，那么完全可以使用fixtuer来处理，在被测试的函数前打开浏览器，然后具体测试需要测试的点，案例代码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest

from selenium import  webdriver

@pytest.fixture()
def init(selenium):
   selenium.maximize_window()
   selenium.get('https://www.baidu.com/')
   selenium.implicitly_wait(30)
   yield
   selenium.quit()


def test_baidu_title(init,selenium):
   '''验证：百度首页的title'''
   assert  selenium.title=='百度一下，你就知道'

def test_baidu_url(init,selenium):
   '''验证：百度首页的URL'''
   assert  selenium.current_url=='https://www.baidu.com/'

def test_baidu_soKeyword(init,selenium):
   '''验证:搜索输入框的内容是否正确'''
   so=selenium.find_element_by_id('kw')
   so.send_keys('无涯您好')
   assert  so.get_attribute('value')=='无涯您好'
```

在如上的案例代码中，使用fixture完全把UI自动化测试的初始化操作和清理分离了出来，这样只需要编写测试的函数就可以了，执行如上的代码，输出结果为：

```
collecting ...
 test_ascill.py::test_baidu_title ✓
                      33% ███
 test_ascill.py::test_baidu_url ✓
                      67% ██████
 test_ascill.py::test_baidu_soKeyword ✓
                     100% ██████████

Results (8.27s):
       3 passed
```

对如上的代码修改下，使用面向对象的方式来实现，完善后的代码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest

class TestUi(object):

   @pytest.fixture()
   def init(self,selenium):
      selenium.maximize_window()
      selenium.get('https://www.baidu.com/')
      selenium.implicitly_wait(30)
      yield
      selenium.quit()

   def test_baidu_title(self,init,selenium):
      '''验证：百度首页的title'''
      assert  selenium.title=='百度一下，你就知道'

   def test_baidu_url(self,init,selenium):
      '''验证：百度首页的URL'''
      assert  selenium.current_url=='https://www.baidu.com/'

   def test_baidu_soKeyword(self,init,selenium):
      '''验证:搜索输入框的内容是否正确'''
      so=selenium.find_element_by_id('kw')
      so.send_keys('无涯您好')
      assert  so.get_attribute('value')=='无涯您好'
```

执行后输出的结果信息为：

```
collecting ...
 test_ascill.py::TestUi.test_baidu_title ✓                                 33%
███
 test_ascill.py::TestUi.test_baidu_url ✓                                   67%
██████
 test_ascill.py::TestUi.test_baidu_soKeyword ✓                            100%
██████████

Results (8.29s):
       3 passed
```

API案例应用

下面来看fixture在API自动化测试中的应用，在API的自动化测试中，该部分主要演示初始化的操作和清理部分，假设需要测试一个书籍的查看信息，前提是需要添加书籍信息，查看书籍信息完成后，删除书籍信息，这样的一个步骤是一个完整的过程，要不下次执行代码会受到影响的。案例代码为：

```
from flask import  Flask,request,jsonify,abort,make_response
from flask_restful import  Resource,Api


app=Flask(__name__)
api=Api(app=app)




books=[
    {
        'id':1,
        'author':'无涯',
        'name':'Python自动化测试实战',
        "done":True
    }
]

class BooksApi(Resource):

    def get(self):
        return jsonify(books)

    def post(self):
        if not request.json or not 'author' in request.json:
            abort(400)
        book = {
            'id': books[-1]['id'] + 1,
            'author': request.json.get('author'),
            'name': request.json.get('name'),
```

```
            'done': False
        }
        books.append(book)
        return jsonify({'status':1002,'msg':'添加成功','datas':book},201)

class BookApi(Resource):
    def get(self,book_id):
        book=list(filter(lambda t:t['id']==book_id,books))
        print(book)
        if len(book)==0:
            abort(400)
        else:
            return jsonify(book)

    def delete(self,book_id):
        book = list(filter(lambda t: t['id'] == book_id, books))
        if len(book)==0:
            abort(404)
        books.remove(book[0])
        return jsonify({'status':1001,'msg':'删除成功'})



api.add_resource(BooksApi,'/v1/api/books',endpoint='/v1/api/books')
api.add_resource(BookApi,'/v1/api/book/<int:book_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

继续编写测试的案例代码：

```
import  pytest
import  requests


def addBook():
    '''添加书籍'''
    dict1={"author":"无涯","name":"无涯课堂","done":True}
    r=requests.post(url='http://127.0.0.1:5000/v1/api/books',json=dict1)
    with open('bookID','w') as f:
        f.write(str(r.json()[0]['datas']['id']))

def getBookID():
    with open('bookID', 'r') as f:
        return f.read()


def delBook():
```

```
    '''删除书籍'''

 r=requests.delete(url='http://127.0.0.1:5000/v1/api/book/{0}'.format(getBookID
()))
    print(r.json())

@pytest.fixture()
def api():
    addBook()
    yield
    delBook()

def test_query_book(api):

 r=requests.get(url='http://127.0.0.1:5000/v1/api/book/{0}'.format(getBookID())
)
    assert  r.json()[0]['id']==int(getBookID())
```

Conftest.py

通过conftest.py可以共享fixture，fixture可以很好的解决测试固件的案例应用，依据它的思想可以编写多个案例，再本质的说就是通过conftest.py文件来达到共享fixture。conftest.py虽然是一个Python模块的文件，但是它是不能导入的，这点需要特别的注意。代码如下：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest

from selenium import  webdriver


@pytest.fixture()
def driver():
   return webdriver.Chrome()

def test_baidu_url(driver):
   driver.get('http://www.baidu.com/')
   assert driver.current_url=='https://www.baidu.com/'


@pytest.fixture()
def login(username='wuya',password='admin'):
   if username=='wuya' and password=='admin':
      return 'eghdsfghj478dfghjfg'
   else:
      return False
```

```
def test_cms_profile(login):
    if login=='eghdsfghj478dfghjfg':
        print('欢迎访问无涯课堂个人主页')
```

执行后输出的结果信息：

```
collecting ...
 test_ascill.py::test_baidu_url ✓                                          50%
■■■■
 test_ascill.py::test_cms_profile ✓                                        100%
■■■■■■■■■

Results (2.94s):
       2 passed
```

在如上的代码中，可以看到编写了两个fixture，但是不能共享给其他的模块来使用，只能在自己本模块的范围内使用，如果其他模块使用，又需要在其他的模块中编写fixture，这样导致的结果就是编写了重复的代码。如果能够共享fixture这部分，那么就不需要编写重复的代码了，在pytest中通过conftest.py来共享fixture，如果希望多个测试文件共同使用一个fixture时候，可以在该目录下创建conftest.py文件，但是切记该文件绝对不能倒入使用，这点一定要注意，创建conftest.py文件后，把需要的fixture加入到里面，就可以使用了。对上面的案例代码进行优化和完善，把使用到的fixture分离到conftest.py的文件中，完善后的代码为：

conftest.py的源码：

```
#!/usr/bin/python3
#coding:utf-8


from selenium import  webdriver
import pytest


@pytest.fixture()
def driver():
    return webdriver.Chrome()

@pytest.fixture()
def login(username='wuya',password='admin'):
    if username=='wuya' and password=='admin':
        return 'eghdsfghj478dfghjfg'
    else:
        return False
```

测试模块的源码为：

```
#!/usr/bin/python3
#coding:utf-8
```

```
def test_baidu_url(driver):
   driver.get('http://www.baidu.com/')
   assert driver.current_url=='https://www.baidu.com/'

def test_baidu_title(driver):
   driver.get('http://www.baidu.com/')
   assert  driver.title=='百度一下，你就知道'


def test_cms_profile(login):
   if login=='eghdsfghj478dfghjfg':
      print('欢迎访问无涯课堂个人主页')
```

执行代码后输出的结果信息为：

```
collecting ...
 test_ascill.py::test_baidu_url ✓
 33% ███
 test_ascill.py::test_baidu_title ✓
 67% ██████
 test_ascill.py::test_cms_profile ✓
100% ██████████

Results (5.19s):
       3 passed
```

回溯这个过程，使用到的命令是--setup-show，执行后输出的结果信息为：

```
collecting ...
SETUP    S base_url
SETUP    S _verify_url (fixtures used: base_url)
SETUP    S sensitive_url (fixtures used: base_url)
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        SETUP    F driver
        test_ascill.py::test_baidu_url (fixtures used: _skip_sensitive,
_verify_url, base_url, driver, sensitive_url)

        TEARDOWN F driver
 test_ascill.py::test_baidu_url ✓
 33% ███
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        SETUP    F driver
        test_ascill.py::test_baidu_title (fixtures used: _skip_sensitive,
_verify_url, base_url, driver, sensitive_url)

        TEARDOWN F driver
```

```
 test_ascill.py::test_baidu_title ✓
 67% ██████
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        SETUP    F login
        test_ascill.py::test_cms_profile (fixtures used: _skip_sensitive,
_verify_url, base_url, login, sensitive_url)

        TEARDOWN F login
        TEARDOWN F _skip_sensitive
TEARDOWN S sensitive_url
TEARDOWN S _verify_url
 test_ascill.py::test_cms_profile ✓
100% ██████████

Results (5.65s):
       3 passed
```

使用conftest.py有这个几点需要特别的注意：

1、conftest.py文件绝对不能当一般的模块到人，这点需要特别的注意

2、conftest.py很多时候被当作pytest测试框架的一个本地插件库来执行

3、在一个测试包（目录）下面，conftest.py被看成是该目录下所有测试用例使用的fixture仓库

Conftest.py的范围

在Conftest.py文件中主要使用scope的参数来控制fixture执行配置和销毁逻辑，它里面的四个值分别代表了它的范围，分别是：

Session:会话级别:针对一个项目中所有的模块，类，以及测试函数

Module：模块级别,主要是针对一个模块的范围

Class:针对类级别，主要是针对一个类的范围

Function:函数级别，主要是针对一个函数的

function级别

依据如上的级别范围，依次来演示它的应用范围，首先来看function级别的应用,它的默认级别也是函数级别的，这里为了演示的需要，主要是以webdriver结合来演示。

Conftest.py的源码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest
from selenium import  webdriver
```

```
@pytest.fixture()
def driver():
    return webdriver.Chrome()


@pytest.fixture(scope='function')
def init(driver):
    driver.maximize_window()
    driver.get('http://www.baidu.com')
    driver.implicitly_wait(30)
    yield
    driver.quit()
```

测试模块的代码为：

```
#!/usr/bin/python3
#coding:utf-8


def test_baidu_url(driver,init):
    assert  driver.title=='百度一下，你就知道'
```

执行后输出的信息：

```
collecting ...
 test_ascill.py::test_baidu_url ✓                                        100%
██████████

Results (2.98s):
       1 passed
```

class级别

类级别主要应用于类，对如上的代码进行完善和优化，conftest.py的源码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest
from selenium import  webdriver


@pytest.fixture(scope='class')
def driver():
    return webdriver.Chrome()
```

```
@pytest.fixture(scope='class')
def init(driver):
   driver.maximize_window()
   driver.get('http://www.baidu.com')
   driver.implicitly_wait(30)
   yield
   driver.quit()
```

测试模块的源码为：

```
#!/usr/bin/python3
#coding:utf-8


def test_baidu_url(driver,init):
   assert  driver.current_url=='https://www.baidu.com/'


class TestUi(object):
   def test_baidu_title(self,driver,init):
      assert  driver.title=='百度一下，你就知道'
```

执行后输出的信息为：

```
collecting ...
 test_ascill.py::test_baidu_url ✓                                      50% ████▌

 test_ascill.py::TestUi.test_baidu_title ✓                            100%
██████████

Results (5.96s):
       2 passed
```

module级别

module级别主要是应用于模块级别的，增加N个模块应用conftest.py里面的fixture，需要再次完善代码。conftest.py的源码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest
from selenium import  webdriver


@pytest.fixture(scope='module')
def driver():
```

```
    return webdriver.Chrome()


@pytest.fixture(scope='module')
def init(driver):
    driver.maximize_window()
    driver.get('http://www.baidu.com')
    driver.implicitly_wait(30)
    yield
    driver.quit()
```

模块代码分别为：

```
#!/usr/bin/python3
#coding:utf-8


def test_baidu_url(driver,init):
    assert  driver.current_url=='https://www.baidu.com/'


class TestUi(object):
    def test_baidu_title(self,driver,init):
        assert  driver.title=='百度一下，你就知道'
```

```
#!/usr/bin/python3
#coding:utf-8


def test_baidu_so(driver,init):
    so=driver.find_element_by_id('kw')
    so.send_keys('无涯课堂')
    assert  so.get_attribute('value')=='无涯课堂'
```

执行代码后，输出的信息为：

```
collecting ...
SETUP    S base_url
SETUP    S _verify_url (fixtures used: base_url)
SETUP    S sensitive_url (fixtures used: base_url)
    SETUP    M driver
    SETUP    M init (fixtures used: driver)
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ascill.py::test_baidu_url (fixtures used: _skip_sensitive,
_verify_url, base_url, driver, init, sensitive_url)
```

```
 test_ascill.py::test_baidu_url ✓                                    33% ███

        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ascill.py::TestUi::test_baidu_title (fixtures used:
_skip_sensitive, _verify_url, base_url, driver, init, sensitive_url)

        TEARDOWN F _skip_sensitive
    TEARDOWN M init
 test_ascill.py::TestUi.test_baidu_title ✓                           67%
██████
    SETUP    M driver
    SETUP    M init (fixtures used: driver)
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ui.py::test_baidu_so (fixtures used: _skip_sensitive, _verify_url,
base_url, driver, init, sensitive_url)

        TEARDOWN F _skip_sensitive
    TEARDOWN M init
    TEARDOWN M driver
TEARDOWN S sensitive_url
TEARDOWN S _verify_url
 test_ui.py::test_baidu_so ✓                                         100%
████████

Results (5.79s):
       3 passed
```

session级别

session级别就是会话级别的，在一个工程中，一个pytest的执行就是一个会话级别的，所以在会话级别并不是一个具体的函数，类，或者模块，再次完善和修改代码。conftest.py的源码为：

```
#!/usr/bin/python3
#coding:utf-8


import  pytest
from selenium import  webdriver


@pytest.fixture(scope='session')
def driver():
   return webdriver.Chrome()


@pytest.fixture(scope='session')
def init(driver):
   driver.maximize_window()
   driver.get('http://www.baidu.com')
```

```
    driver.implicitly_wait(30)
    yield
    driver.quit()
```

测试模块的代码还是上面的代码，不需要修改，再次执行命令，输出的信息为：

```
collecting ...
SETUP    S base_url
SETUP    S _verify_url (fixtures used: base_url)
SETUP    S driver
SETUP    S init (fixtures used: driver)
SETUP    S sensitive_url (fixtures used: base_url)
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ascill.py::test_baidu_url (fixtures used: _skip_sensitive,
_verify_url, base_url, driver, init, sensitive_url)

 test_ascill.py::test_baidu_url ✓                                    33% ███

        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ascill.py::TestUi::test_baidu_title (fixtures used:
_skip_sensitive, _verify_url, base_url, driver, init, sensitive_url)

 test_ascill.py::TestUi.test_baidu_title ✓                           67%
██████
        SETUP    F _skip_sensitive (fixtures used: sensitive_url)
        test_ui.py::test_baidu_so (fixtures used: _skip_sensitive, _verify_url,
base_url, driver, init, sensitive_url)

        TEARDOWN F _skip_sensitive
TEARDOWN S sensitive_url
TEARDOWN S init
TEARDOWN S driver
TEARDOWN S _verify_url
 test_ui.py::test_baidu_so ✓                                        100%
██████████

Results (2.78s):
       3 passed
```

fixture的重命名

在Pytest的测试框架中，可以对fixture通过参数name来达到重命名的目的，如下案例：

```
#!/usr/bin/env python
#!coding:utf-8


import  pytest
```

```
@pytest.fixture(name="token")
def login(username='wuya',password='admin'):
   if username=='wuya' and password=='admin':
      return 'wertyusdfg4356'
   else:pass

def test_profile(token):
   if token=='wertyusdfg4356':
      print('欢迎你访问用户的个人主页')

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

执行后，输出的结果信息为：

```
collecting ... collected 1 item

test_one.py::test_profile 欢迎你访问用户的个人主页
PASSED

============================ 1 passed in 0.06 seconds ============================

Process finished with exit code 0
```

内置fixture

tmpdir

主要应用在读取文件和对文件的操作案例，这里我们还是以创建一个文件，然后在文件里面写入内容，最后读取文件的内容为验证的点，结合内置tempdir来演示该案例的应用：

```
#!/usr/bin/env python
#!coding:utf-8


import  pytest

def login(username,password):
   if username=='wuya' and password =='admin':
      return 'dfghjkerty45fdsgudf'

def profile(token):
   if token=='dfghjkerty45fdsgudf':
      return 'profile'
   else:
      print('请登录系统')
```

```
def test_order_tmpdir(tmpdir):
    obj=tmpdir.join('token.txt')
    obj.write(login('wuya','admin'))
    assert  profile(obj.read())=='profile'


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

在这样的一个过程中，事实上我们是需要临时文件来存储登录成功后的token的信息的，但是在IDE中我们并没有看到临时文件，这是因为Pytest内置的tmpdir内部进行了额自动处理。

Pytest的Hook函数

Pytest的配置

Pytest常用插件

pytest单元测试框架有很丰富的插件，下面分别介绍这些插件的应用。

Pytest-sugar

Pytest-sugar在执行的时候显示进度条，即使有失败的也会立刻显示出来，安装命令为：

```
pip3 install pytest-sugar
```

执行后在输出中就会显示进度条，即使错误的也会显示出来，见如下的信息：

```
collecting ...
 test_login.py::test_001 ✓                                                  50% ▌

---------------------------------------- test_002 ----------------------------------------

    def test_002():
>       assert  1==2
E    assert 1 == 2
E      -1
E      +2

test_login.py:9: AssertionError

 test_login.py::test_002 ⨯                                                 100% ▌

Results (0.10s):
       1 passed
       1 failed
         - test_login.py:8 test_002
```

Pytest-html

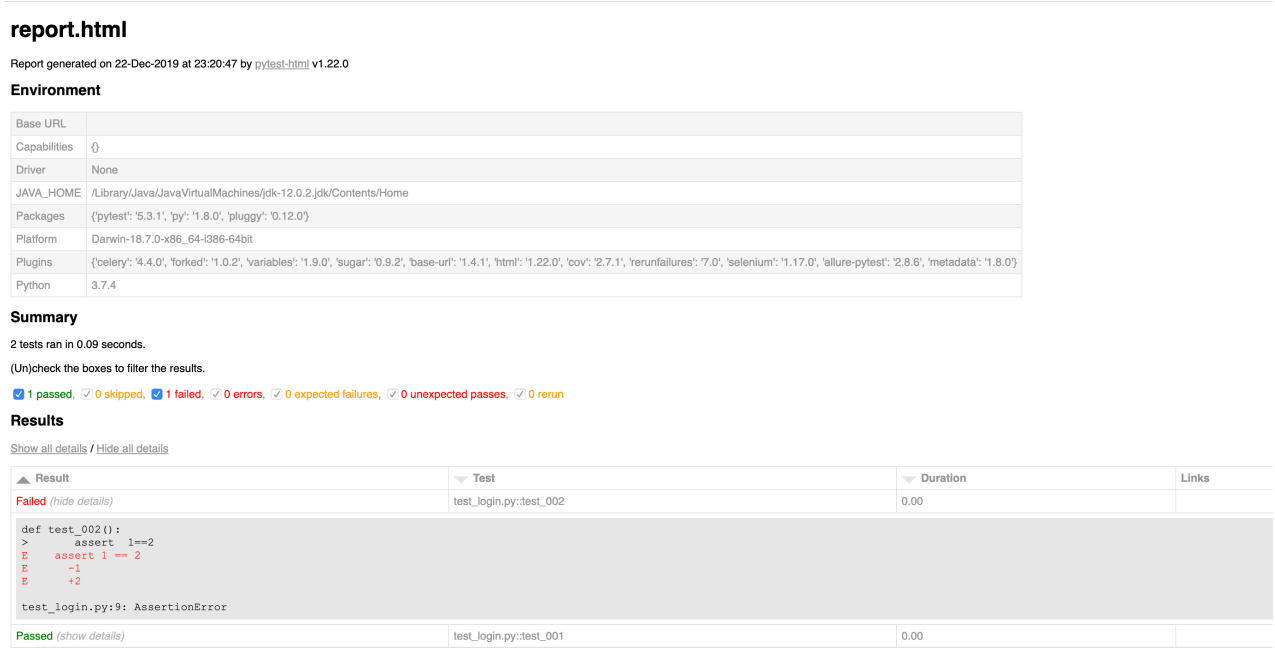
执行后生成给予html的测试报告，安装它的命令为：

```
pip3 install pytest-html
```

应用场景主要为：

```
pytest -v -s --html=report.html
```

执行后，会在当前目录下生成 一个report.html的文件，打开后会展示详细的测试报告，如下图所示：



report.html

Report generated on 22-Dec-2019 at 23:20:47 by pytest-html v1.22.0

Environment

Base URL	
Capabilities	{}
Driver	None
JAVA_HOME	/Library/Java/JavaVirtualMachines/jdk-12.0.2.jdk/Contents/Home
Packages	{'pytest': '5.3.1', 'py': '1.8.0', 'pluggy': '0.12.0'}
Platform	Darwin-18.7.0-x86_64-i386-64bit
Plugins	{'celery': '4.4.0', 'forked': '1.0.2', 'variables': '1.9.0', 'sugar': '0.9.2', 'base-url': '1.4.1', 'html': '1.22.0', 'cov': '2.7.1', 'rerunfailures': '7.0', 'selenium': '1.17.0', 'allure-pytest': '2.8.6', 'metadata': '1.8.0'}
Python	3.7.4

Summary

2 tests ran in 0.09 seconds.

(Un)check the boxes to filter the results.

1 passed, 0 skipped, 1 failed, 0 errors, 0 expected failures, 0 unexpected passes, 0 rerun

Results

Show all details / Hide all details

Result	Test	Duration	Links
Failed *(hide details)*	test_login.py::test_002	0.00	

```
def test_002():
>       assert  1==2
E       assert 1 == 2
E         -1
E         +2

test_login.py:9: AssertionError
```

Passed *(show details)*	test_login.py::test_001	0.00	

Pytest-rerunfailures

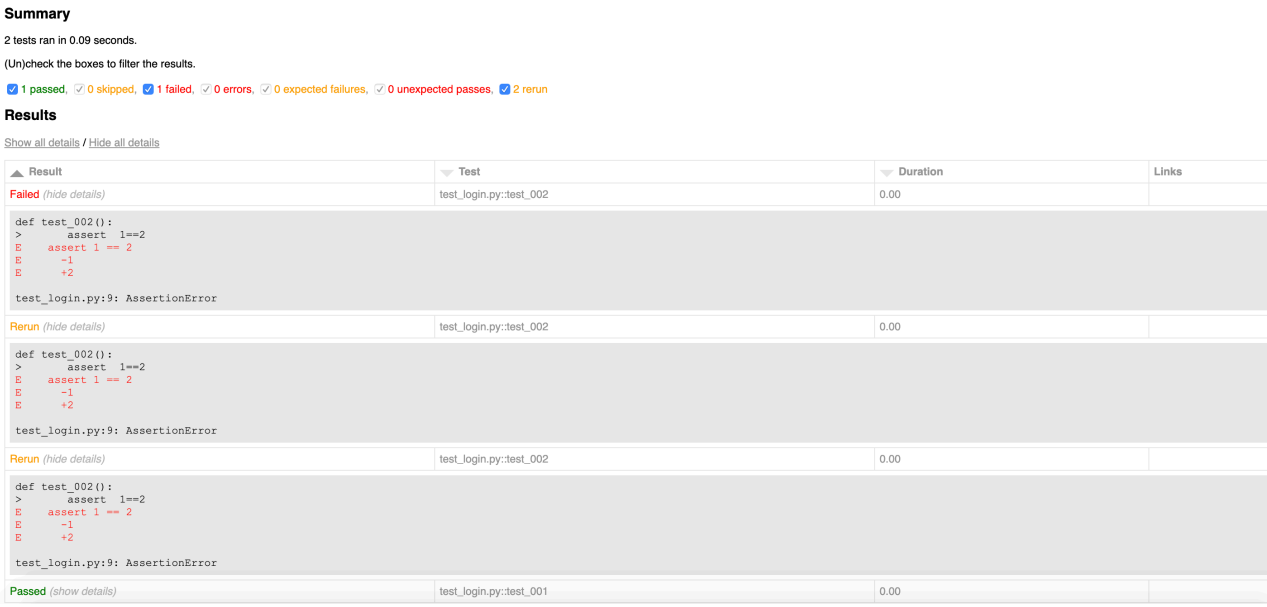
对执行失败的再次执行，安装的命令为：

```
pip3 install pytest-rerunfailures
```

使用的场景是:--reruns --N N就是重试的次数，使用命令如下：

```
pytest -v -s  --reruns 2  --html=report.html
```

打开报告，可以看到重试的次数，如下图所示：



Summary

2 tests ran in 0.09 seconds.

(Un)check the boxes to filter the results.

1 passed, 0 skipped, 1 failed, 0 errors, 0 expected failures, 0 unexpected passes, 2 rerun

Results

Show all details / Hide all details

Result	Test	Duration	Links
Failed *(hide details)*	test_login.py::test_002	0.00	

```
def test_002():
>       assert  1==2
E     assert 1 == 2
E       -1
E       +2

test_login.py:9: AssertionError
```

Rerun *(hide details)*	test_login.py::test_002	0.00	

```
def test_002():
>       assert  1==2
E     assert 1 == 2
E       -1
E       +2

test_login.py:9: AssertionError
```

Rerun *(hide details)*	test_login.py::test_002	0.00	

```
def test_002():
>       assert  1==2
E     assert 1 == 2
E       -1
E       +2

test_login.py:9: AssertionError
```

Passed *(show details)*	test_login.py::test_001	0.00	

Pytest-xdist

插件官方地址：https://pypi.org/project/pytest-xdist/

开启多个worker进程，同时执行多个测试用例，达到并发运行的效果，大大提升构建效率,安装命令为：

```
pip3 install  pytest-xdist
```

见测试的案例代码：

```
#!/usr/bin/python3
#coding:utf-8


import  requests


def test_baidu():
    r=requests.get(url='http://www.baidu.com/')
    assert  r.status_code==200

def test_jd():
    r=requests.get(url='http://www.jd.com/')
    assert  r.status_code==200

def test_taobao():
    r=requests.get(url='http://www.taobao.com/')
    assert  r.status_code==200

def test_sina():
    r=requests.get(url='http://www.sina.com/')
    assert  r.status_code==200
```

```
def test_qq():
    r=requests.get(url='http://www.qq.com/')
    assert  r.status_code==200

def test_huaban():
    r=requests.get(url='http://www.huaban.com/')
    assert  r.status_code==200

def test_360():
    r=requests.get(url='http://www.360.com/')
    assert  r.status_code==200
```

执行命令pytest -v 查看执行的时间：

执行如下命令并发执行:

```
pytest -v -n 4 test_login.py
```

备注:4指的是工作进程数，在如上的案例中，也就是使用4个工作进程数来执行测试套件

见执行后输出的结果信息：

```
[gw0] darwin Python 3.7.4 cwd: /Applications/code/stack/xunit
[gw1] darwin Python 3.7.4 cwd: /Applications/code/stack/xunit
[gw2] darwin Python 3.7.4 cwd: /Applications/code/stack/xunit
[gw3] darwin Python 3.7.4 cwd: /Applications/code/stack/xunit
[gw0] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw1] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw2] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw3] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
gw0 [7] / gw1 [7] / gw2 [7] / gw3 [7]
scheduling tests via LoadScheduling

 test_login.py::test_jd ✓                                                    14%
[gw1] PASSED test_login.py
 test_login.py::test_baidu ✓                                                 29%
[gw0] PASSED test_login.py
 test_login.py::test_sina ✓                                                  43%
[gw3] PASSED test_login.py
 test_login.py::test_taobao ✓                                                57%
[gw2] PASSED test_login.py
 test_login.py::test_qq ✓                                                    71%
[gw0] PASSED test_login.py
 test_login.py::test_360 ✓                                                   86%
[gw2] PASSED test_login.py
 test_login.py::test_huaban ✓                                               100%
[gw1] PASSED test_login.py

Results (1.59s):
```

如果你不你的CPU到底是几个，那么pytest-xdist它可以自动检测并且执行，执行的命令为：

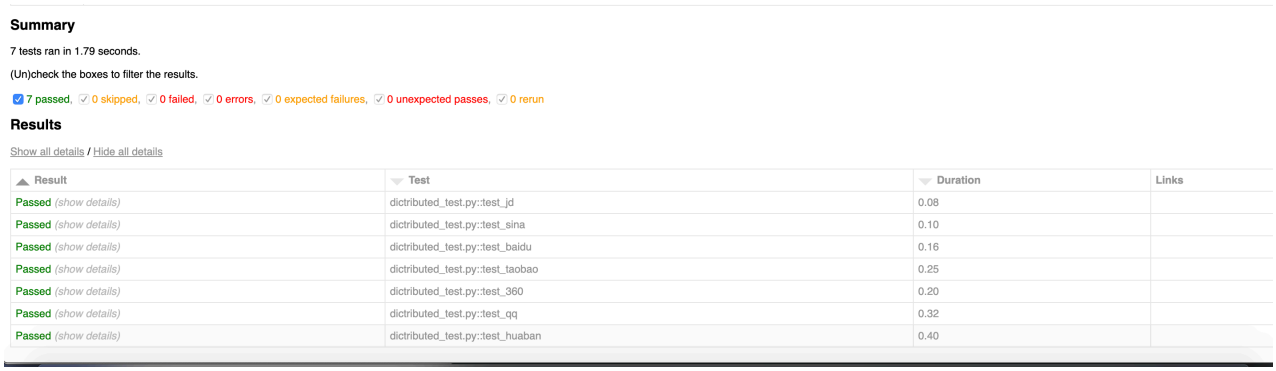
pytest -v -n auto test_*.py，执行后，输出的信息为：

```
[gw0] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw1] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw2] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw3] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw4] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw5] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw6] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw7] darwin Python 3.7.4 cwd: /Applications/code/TavernCi/PytestBasic
[gw0] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw1] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw2] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw3] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw4] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw5] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw6] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
[gw7] Python 3.7.4 (v3.7.4:e09359112e, Jul  8 2019, 14:54:52)  -- [Clang 6.0 (clang-600.0.57)]
gw0 [7] / gw1 [7] / gw2 [7] / gw3 [7] / gw4 [7] / gw5 [7] / gw6 [7] / gw7 [7]
scheduling tests via LoadScheduling

 dictributed_test.py::test_jd ✓                                              14%
[gw1] PASSED dictributed_test.py
 dictributed_test.py::test_sina ✓                                            29%
[gw3] PASSED dictributed_test.py
 dictributed_test.py::test_baidu ✓                                           43%
[gw0] PASSED dictributed_test.py
 dictributed_test.py::test_qq ✓                                              57%
[gw4] PASSED dictributed_test.py
 dictributed_test.py::test_taobao ✓                                          71%
[gw2] PASSED dictributed_test.py
 dictributed_test.py::test_huaban ✓                                          86%
[gw5] PASSED dictributed_test.py
 dictributed_test.py::test_360 ✓                                            100%
[gw6] PASSED dictributed_test.py
```

执行完成后测试报告的展示，执行的命令为：

pytest -v -n 4 --html=report.html --self-contained-html

执行如上的命令后，分布式的执行测试用例并且html的测试报告，输出信息如下：



Summary

7 tests ran in 1.79 seconds.

(Un)check the boxes to filter the results.

7 passed, 0 skipped, 0 failed, 0 errors, 0 expected failures, 0 unexpected passes, 0 rerun

Results

Show all details / Hide all details

Result	Test	Duration	Links
Passed *(show details)*	dictributed_test.py::test_jd	0.08	
Passed *(show details)*	dictributed_test.py::test_sina	0.10	
Passed *(show details)*	dictributed_test.py::test_baidu	0.16	
Passed *(show details)*	dictributed_test.py::test_taobao	0.25	
Passed *(show details)*	dictributed_test.py::test_360	0.20	
Passed *(show details)*	dictributed_test.py::test_qq	0.32	
Passed *(show details)*	dictributed_test.py::test_huaban	0.40	

分布式执行：

pytest -v --tx ssh=root@47.95.142.233//python=python3 --rsyncdir PytestBasic/

Pytest-mock

https://github.com/pytest-dev/pytest-mock

Allure测试报告

测试覆盖率

尽管测试先行编程（test-first programming）和单元测试已不能算是新概念，但测试驱动的开发仍然是过去 10 年中最重要的编程创新。最好的一些编程人员在过去半个世纪中一直在使用这些技术，不过，只是在最近几年，这些技术才被广泛地视为在时间及成本预算内开发健壮的无缺陷软件的关键所在。但是，测试驱动的开发不能超过测试所能达到的程度。测试改进了代码质量，但这也只是针对实际测试到的那部分代码而言的。您需要有一个工具告诉您程序的哪些部分没有测试到，这样就可以针对这些部分编写测试代码并找出更多 bug。

coverage

安装coverage

在python的单元测试框架unittest中，使用到的覆盖率工具是coverage，它属于第三方的库，安装的命令为：

pip3 install coverage

使用命令coverage --version，可以显示版本号

Coverage.py, version 4.5.4 with C extension

Documentation at https://coverage.readthedocs.io

coverage实战

coverage使用命令可以达到想要的目标，这些命令主要为coverage的参数，具体为：

```
run：运行Python程序并收集执行数据
report：报告覆盖范围
html：生成基于HTML的测试报告
xml：生成包含XML的测试报告
debug：获取调试信息
```

也可以使用命令coverage --help查看帮助信息，如展示的信息为：

Coverage.py, version 4.5.4 with C extension Measure, collect, and report on code coverage in Python programs.

usage: coverage [options] [args]

Commands: annotate Annotate source files with execution information. combine Combine a number of data files. erase Erase previously collected coverage data. help Get help on using coverage.py. html Create an HTML report. report Report coverage stats on modules. run Run a Python program and measure code execution. xml Create an XML report of coverage results.

Use "coverage help " for detailed help on any command. For full documentation, see https://coverage.readthedocs.io

下面还是根据具体的信息来看它的实战和案例应用,案例代码：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import unittest
from parameterized import parameterized,param


def add(a,b):
   try:
      return a+b
   except Exception as e:
      raise '执行异常'


class AddTest(unittest.TestCase):
   @parameterized.expand([
      param(1,1,2),
      param(1.0,1.0,2.0),
      param('hi ','wuya','hi wuya')
   ])
   def test_add(self,a,b,result):
      self.assertEqual(add(a,b),result)

if __name__ == '__main__':
```

```
    unittest.main(verbosity=2)
```

针对该测试模块执行,执行的命令为：

python3 -m coverage run xx.py，执行后，输出的结果信息为：

```
test_add_0 (__main__.AddTest) ... ok
test_add_1 (__main__.AddTest) ... ok
test_add_2_hi_ (__main__.AddTest) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

执行命令查看覆盖率，命令为：

coverage report -m ,执行后，输出的信息为：

```
Name    Stmts   Miss  Cover   Missing
---------------------------------------
f1.py      12      2    83%   14-15
```

生成基于HTML的测试报告，命令为：

coverage html，执行后，会在当前目录下生成htmlcov的文件夹，在该文件夹里面，打开index.html,就会显示

覆盖率的情况，具体为：

Coverage report: 83%

Module ↓	*statements*	*missing*	*excluded*	*coverage*
f1.py	12	2	0	83%
Total	**12**	**2**	**0**	**83%**

coverage.py v4.5.4, created at 2019-11-23 16:26

点击f1.py，就会显示出代码覆盖率的具体情况，包含了覆盖到的还未覆盖到的，具体为：

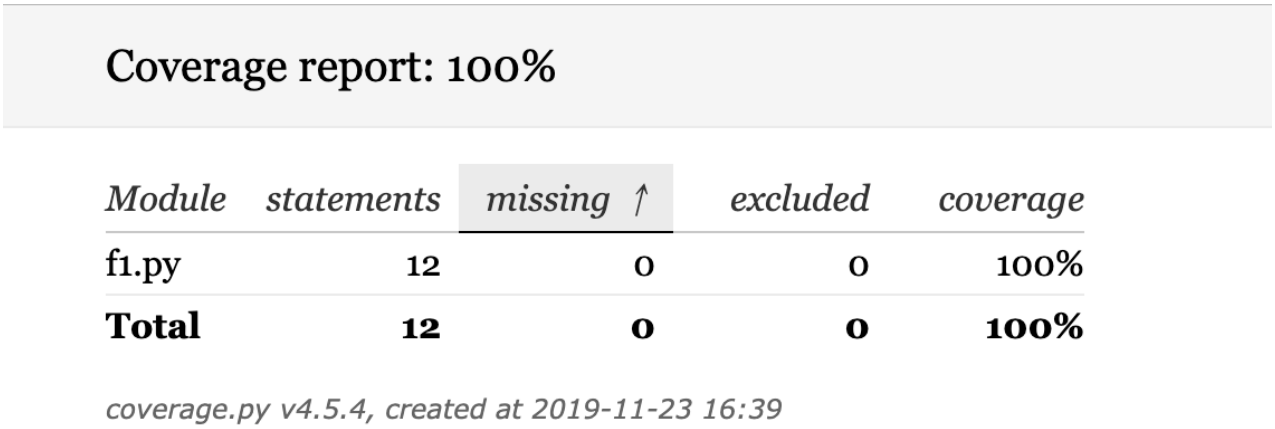
```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import unittest
from parameterized import parameterized,param


def add(a,b):
        try:
                return a+b
        except Exception as e:
                raise '执行异常'


class AddTest(unittest.TestCase):
        @parameterized.expand([
                param(1,1,2),
                param(1.0,1.0,2.0),
                param('hi ','wuya','hi wuya')
        ])
        def test_add(self,a,b,result):
                self.assertEqual(add(a,b),result)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

红色的部分表示没有覆盖到的，这说明了什么？说明了虽然设计了测试点，但是没有考虑全的，也就是说存在漏测的情况，测试点考虑的还是不全，至少没有考虑到两个数相加的情况，那么就得需要再次设计测试场景。依据这样的结果，我们再次完善我们的测试点，完善后的代码为：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00


import unittest
from parameterized import parameterized,param


def add(a,b):
   try:
      return a+b
   except Exception as e:
      return e.args[0]


class AddTest(unittest.TestCase):
   @parameterized.expand([
      param(1,1,2),
```

```
        param(1.0,1.0,2.0),
        param(1, 1.0, 2.0),
        param(1,0,1),
        param('','',''),
        param('hi ','wuya','hi wuya'),
        param(0, '', "unsupported operand type(s) for +: 'int' and 'str'"),
        param(1,'hi',"unsupported operand type(s) for +: 'int' and 'str'"),
         param(1.0,'wuya',"unsupported operand type(s) for +: 'float' and
'str'"),
    ])
    def test_add(self,a,b,result):
        self.assertEqual(add(a,b),result)

if __name__ == '__main__':
     unittest.main(verbosity=2)
```

执行命令和之前的一模一样，就不再演示，看最新的基于HTML的测试覆盖率报告文件，如下图所示：



Coverage report: 100%

Module	statements	missing ↑	excluded	coverage
f1.py	12	0	0	100%
Total	**12**	**0**	**0**	**100%**

coverage.py v4.5.4, created at 2019-11-23 16:39

可以看到没有漏测的情况，再次点击f1.py的文件，然后点击里面的run按钮，就会显示出覆盖到的代码，所下图所示：

```
#!/usr/bin/python3
#coding:utf-8
#author:无涯
#time:2019-11-23 14:00:00
```

```
import unittest
from parameterized import parameterized,param
```

```
def add(a,b):
        try:
                return a+b
        except Exception as e:
                return e.args[0]
```

```
class AddTest(unittest.TestCase):
        @parameterized.expand([
                param(1,1,2),
                param(1.0,1.0,2.0),
                param(1, 1.0, 2.0),
                param(1,0,1),
                param('','',''),
                param('hi ','wuya','hi wuya'),
                param(0, '', "unsupported operand type(s) for +: 'int' and 'str'"),
                param(1,'hi',"unsupported operand type(s) for +: 'int' and 'str'"),
            param(1.0,'wuya',"unsupported operand type(s) for +: 'float' and 'str'"),
        ])
        def test_add(self,a,b,result):
                self.assertEqual(add(a,b),result)
```

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

可以看到，函数add()它存在吃的测试出现都已考虑到为，利用coverage它可以很方便的帮助我们检测那些场景没考虑到位。

当然如上的操作都是基于命令行来操作的，下面具体说下在Pycharm的IDE中操作，在要执行的Python的模块文件中，右键点击Run 'unittest in xx.py' coverage，点击后，最下面会显示出执行的结果信息，最右边会显示出覆盖了的情况，先具体看最下面输出的情况，如下图所示：

```
f1.py::AddTest::test_add_0 PASSED                                        [ 11%]
f1.py::AddTest::test_add_1 PASSED                                        [ 22%]
f1.py::AddTest::test_add_2 PASSED                                        [ 33%]
f1.py::AddTest::test_add_3 PASSED                                        [ 44%]
f1.py::AddTest::test_add_4_ PASSED                                       [ 55%]
f1.py::AddTest::test_add_5_hi_ PASSED                                    [ 66%]
f1.py::AddTest::test_add_6 PASSED                                        [ 77%]
f1.py::AddTest::test_add_7 PASSED                                        [ 88%]
f1.py::AddTest::test_add_8 PASSED                                        [100%]

============================ 9 passed in 0.19 seconds ============================

Process finished with exit code 0
```

覆盖率在IDE的代码中的显示，如下图所示：

```python
import unittest
from parameterized import parameterized,param


def add(a,b):
    try:
        return a+b
    except Exception as e:
        return e.args[0]


class AddTest(unittest.TestCase):
    @parameterized.expand([
        param(1,1,2),
        param(1.0,1.0,2.0),
        param(1, 1.0, 2.0),
        param(1,0,1),
        param('','',''),
        param('hi ','wuya','hi wuya'),
        param(0, '', "unsupported operand type(s) for +: 'int' and 'str'"),
        param(1,'hi',"unsupported operand type(s) for +: 'int' and 'str'"),
        param(1.0,'wuya',"unsupported operand type(s) for +: 'float' and 'str'"),
    ])
    def test_add(self,a,b,result):
        self.assertEqual(add(a,b),result)

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

鼠标到最左边的部分，可以看到提示是否隐藏覆盖率，如下图所示：

点击后，覆盖率就会被隐藏。

coverage原理

coverage主要分为三个阶段来进行，分别是执行，分析，报告，下面具体做以说明，主要为：

执行：coverage.py运行被运行的代码，进行检测执行了多少行

分析：coverage.py检查代码已确定可以执行那些行

报告：coverage.py结合执行和分析的结果最后以报告的形式呈现（HTML或者其他形式）

Saas化的验证