《python 自动化框架 pytest》

作者：上海-悠悠

QQ 号：283340479

作者简介

个人博客：https://www.cnblogs.com/yoyoketang

微信公众号：yoyoketang（扫二维码关注）

QQ 交流群：874033608

目录

Pytest 简介

pytest 是 python 的一种单元测试框架，与 python 自带的 unittest 测试框架类似，但是比 unittest 框架使用起来更简洁，效率更高。根据 pytest 的官方网站介绍，它具有如下特点：

- 非常容易上手，入门简单，文档丰富，文档中有很多实例可以参考
- 能够支持简单的单元测试和复杂的功能测试
- 支持参数化 parametrize,比 unittest 的 ddt 更简单
- 执行测试过程中可以将某些测试 skip 跳过，或者对某些预期失败的 case 标记成失败
- 强大的 fixture 自定义功能，这个是框架的核心亮点功能
- pytest-rerunfailures（失败 case 重复执行）
- pytest-html（完美 html 测试报告生成，失败截图展示）
- allure2 漂亮的 html 报告展示
- 方便的和 jenkins 持续集成工具集成
- 支持运行由 nose，unittest， doctest 框架编写的测试 case
- 可以用来做 web 和 app 自动化（pytest+selenium/appnium）、接口（pytest+requests）

可以用来做 pytest+selenium/appnium 等自动化测试、接口自动化测试（pytest+requests）

第 1 章 pytest 框架介绍

首先说下为什么要学 pytest，在此之前相信大家已经掌握了 python 里面的 unittest 单元测试框架，那再学一个框架肯定是需要学习时间成本的。

刚开始我的内心是拒绝的，我想我用 unittest 也能完成自动化测试，干嘛要去学 pytest 呢？最近看到越来越多的招聘要求会 pytest 框架了，也有小伙伴出去面试说会 unittest 框架被鄙视的。

所以学此框架应该至少有以下 2 个理由，第一条已经足够：

- 学会了可以装逼
- 可以避免被面试官鄙视

1.1 环境准备与入门

前言

pytest 是 python2 默认自带的，python3 的版本 pytest 框架独立出来了，需用 pip 安装。本书以 python3.6 版本为教学。

pytest 简介

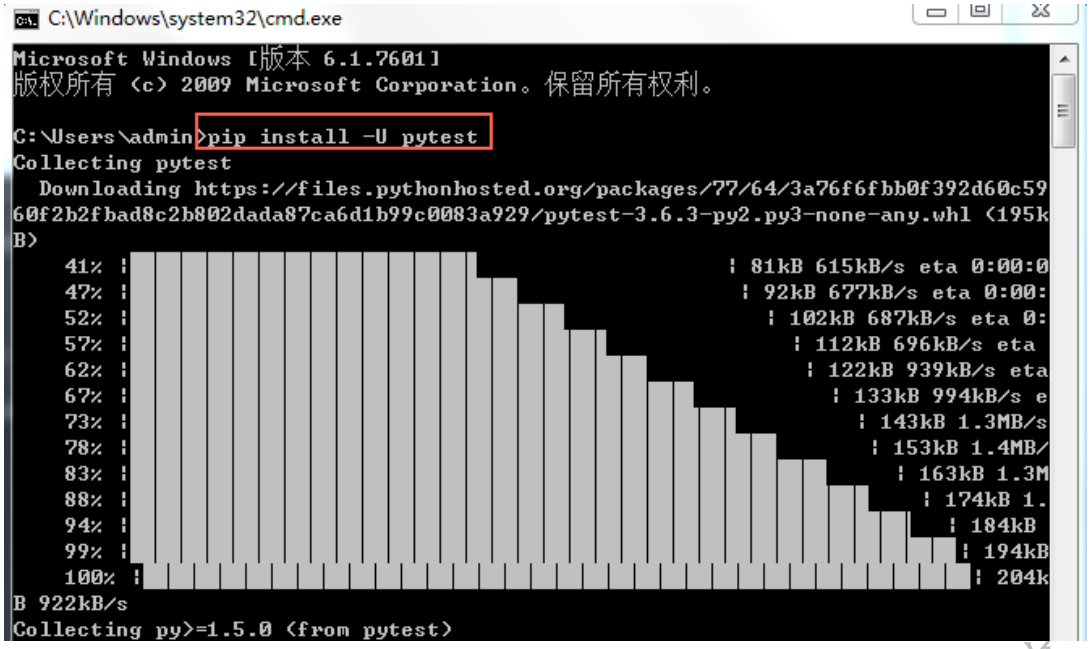
pytest 是 python 的一种单元测试框架，与 python 自带的 unittest 测试框架类似，但是比 unittest 框架使用起来更简洁，效率更高。根据 pytest 的官方网站介绍，它具有如下特点：

- 非常容易上手，入门简单，文档丰富，文档中有很多实例可以参考
- 能够支持简单的单元测试和复杂的功能测试
- 支持参数化
- 执行测试过程中可以将某些测试跳过（skip），或者对某些预期失败的 case 标记成失败
- 支持重复执行(rerun)失败的 case
- 支持运行由 nose, unittest 编写的测试 case
- 可生成 html 报告
- 方便的和持续集成工具 jenkins 集成
- 可支持执行部分用例
- 具有很多第三方插件，并且可以自定义扩展

安装 pytest

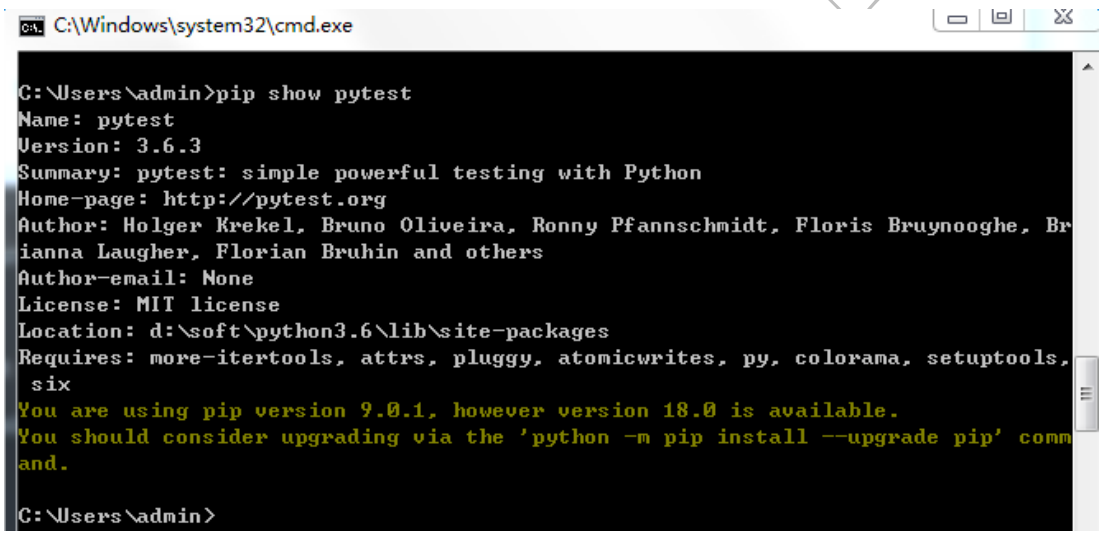
使用 pip 直接安装

```
> pip install -U pytest
```



pip show pytest 查看安装版本

```
> pip show pytest
```



也可以 pytest --version 查看安装的版本

```
> pytest --version

This is pytest version 3.6.3, imported from

d:\soft\python3.6\lib\site-packages\pytest.py
```

快速开始

新建一个 test_sample.py 文件，写以下代码

```
# content of test_sample.py
def func(x):
    return x +1


def test_answer():
    assert func(3)==5
```

打开 test_sample.py 所在的文件夹，cmd 窗口输入：pytest（或者输入 py.test 也可以）

```
D:\YOYO>pytest
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\YOYO, inifile:
collected 1 item

test_sample.py F

================================== FAILURES ====================
_____________________________________ test_answer _____________

    def test_answer():
>       assert func(3)==5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================== 1 failed in 0.19 seconds ============
```

pytest 运行规则：**查找当前目录及其子目录下以 test_*.py 或 *_test.py 文件，找到文件后，在文件中找到以 test 开头函数并执行。**

写个测试类

前面是写的一个 test 开头的测试函数，当用例用多个的时候，写函数就不太合适了。这时可以把多个测试用例，写到一个测试类里

```
# test_class.py


class TestClass:
    def test_one(self):
        x = "this"
        assert 'h' in x
```

```
def test_two(self):
    x = "hello"
    assert hasattr(x, 'check')
```

.pytest 会找到符合规则（test_.py 和_test.py）所有测试，因此它发现两个 test_前缀功能。 如果只想运行其中一个，可以指定传递文件名 test_class.py 来运行模块：

备注： -q, --quiet decrease verbosity(显示简单结果)


```
> py.test -q test_class.py
```

```
D:\YOYO>py.test -q test_class.py
.F
================================== FAILURES ===================
_____________________________ TestClass.test_two _____________

self = <test_class.TestClass object at 0x00000000039F1828>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:11: AssertionError
1 failed, 1 passed in 0.04 seconds
```

第一次测试通过，第二次测试失败。 您可以在断言中轻松查看失败的原因。

pytest 用例规则

- 测试文件以 test_开头（以_test 结尾也可以）

- 测试类以 Test 开头，并且不能带有 init 方法
- 测试函数以 test_开头
- 断言使用 assert

1.2-用例运行规则

前言

当我们使用 pytest 框架写用例的时候，一定要按它的命名规范去写用例，这样框架才能找到哪些是用例需要执行，哪些不是用例不需要执行。

用例设计原则

- 文件名以 test_*.py 文件和*_test.py
- 以 test_开头的函数
- 以 Test 开头的类
- 以 test_开头的方法
- 所有的包 pakege 必须要有__init__.py 文件

help 帮助

查看 pytest 命令行参数，可以用 pytest -h 或 pytest —help 查看

```
C:\Users\admin>pytest -h
usage: pytest [options] [file_or_dir] [file_or_dir] [...]

positional arguments:
  file_or_dir

general:
  -k EXPRESSION         only run tests which match the given sub
                        expression. An expression is a python ev
                        expression where all names are substring
                        against test names and their parent clas
                        -k 'test_method or test_other' matches a
                        functions and classes whose name contain
                        'test_method' or 'test_other', while -k
                        test_method' matches those that don't co
                        'test_method' in their names. Additional
                        are matched to classes and functions con
                        names in their 'extra_keyword_matches' s
                        functions which have names assigned dire
  -m MARKEXPR           only run tests matching given mark expre
                        example: -m 'mark1 and not mark2'.
  --markers             show markers (builtin, plugin and per-pr
  -x, --exitfirst       exit instantly on first error or failed

reporting:
  -v, --verbose         increase verbosity.
  -q, --quiet           decrease verbosity.
  --verbosity=VERBOSE   set verbosity
```

只贴了一部分

按以下目录写用例

D:YOYO\

__init__.py

test_class.py

```
#  content of  test_class.py

class TestClass:

    def test_one(self):

        x = "this"

        assert 'h' in x


    def test_two(self):

        x = "hello"

        assert hasattr(x, 'check')


    def test_three(self):

        a = "hello"

        b = "hello world"

        assert a in b
```

test_sample.py

```
#  content of  test_sample.py

def func(x):

    return x +1



def test_answer():

    assert func(3)==5
```

python -m

cmd 执行 pytest 用例有三种方法,以下三种方法都可以，一般推荐第一个

- pytest
- py.test
- python -m pytest

如果不带参数，在某个文件夹下执行时，它会查找该文件夹下所有的符合条件的用例（查看用例设计原则）

执行用例规则

1.某个目录下所有的用例

```
> pytest 文件名/
```

2.执行某一个 py 文件下用例

```
> pytest 脚本名称.py
```

3.-k 按关键字匹配

```
> pytest -k "MyClass and not method"
```

行包含与给定字符串表达式匹配的名称的测试，其中包括 Python 使用文件名，类名和函数名作为变量的运算符。 上面的例子将运行 TestMyClass.test_something 但不运行 TestMyClass.test_method_simple

4.按节点运行

每个收集的测试都分配了一个唯一的 nodeid，它由模块文件名和后跟说明符组成

来自参数化的类名，函数名和参数，由:: characters 分隔。

运行.py 模块里面的某个函数

```
> pytest test_mod.py::test_func
```

运行.py 模块里面,测试类里面的某个方法

```
> pytest test_mod.py::TestClass::test_method
```

5.标记表达式

```
> pytest -m slow
```

将运行用@ pytest.mark.slow 装饰器修饰的所有测试。后面章节会讲自定义标记 mark 的功能

6.从包里面运行

```
> pytest --pyargs pkg.testing
```

这将导入 pkg.testing 并使用其文件系统位置来查找和运行测试。

-x 遇到错误时停止测试

```
> pytest -x test_class.py
```

```
D:\YOYO>pytest -x test_class.py
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\YOYO, inifile:
collected 3 items

test_class.py .F

================================== FAILURES ====================
_____________________________ TestClass.test_two _______________

self = <YOYO.test_class.TestClass object at 0x0000000003A29780>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:11: AssertionError
==================== 1 failed, 1 passed in 0.05 seconds ========
```

从运行结果可以看出，本来有 3 个用例，第二个用例失败后就没继续往下执行了

—maxfail=num

当用例错误个数达到指定数量时，停止测试

> ytest –maxfail=1

```
D:\YOYO>pytest --maxfail=1
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\YOYO, inifile:
collected 4 items

test_class.py .F

================================== FAILURES ====================
_____________________________ TestClass.test_two _______________

self = <YOYO.test_class.TestClass object at 0x0000000003A3D080>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:11: AssertionError
==================== 1 failed, 1 passed in 0.06 seconds ========
```

1.3-pycharm 运行 pytest

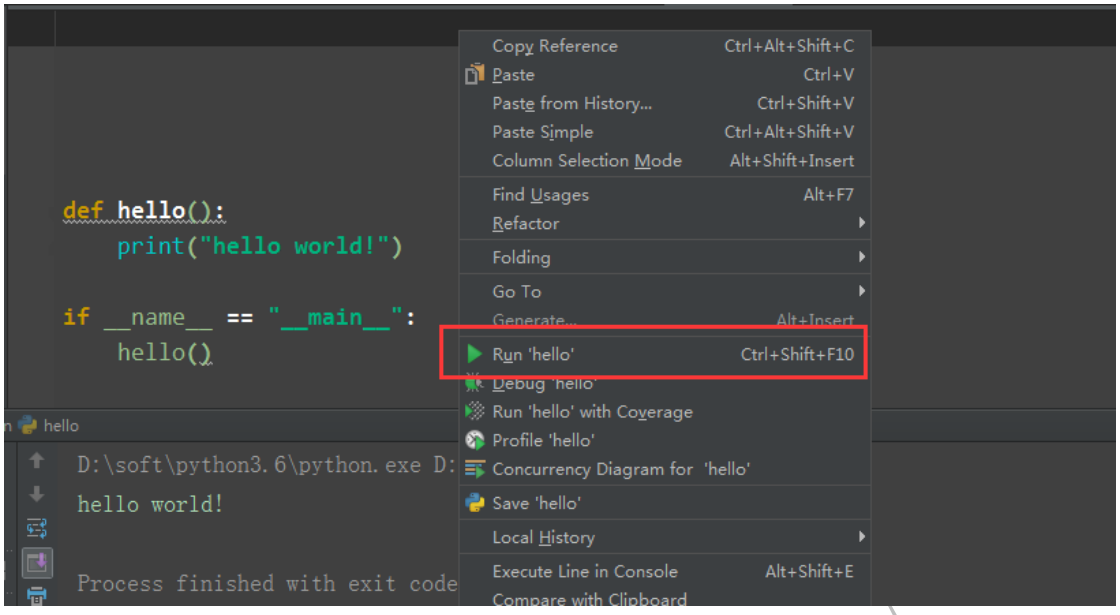
前言

上一篇已经介绍了如何在 cmd 执行 pytest 用例，平常我们写代码在 pycharm 比较多

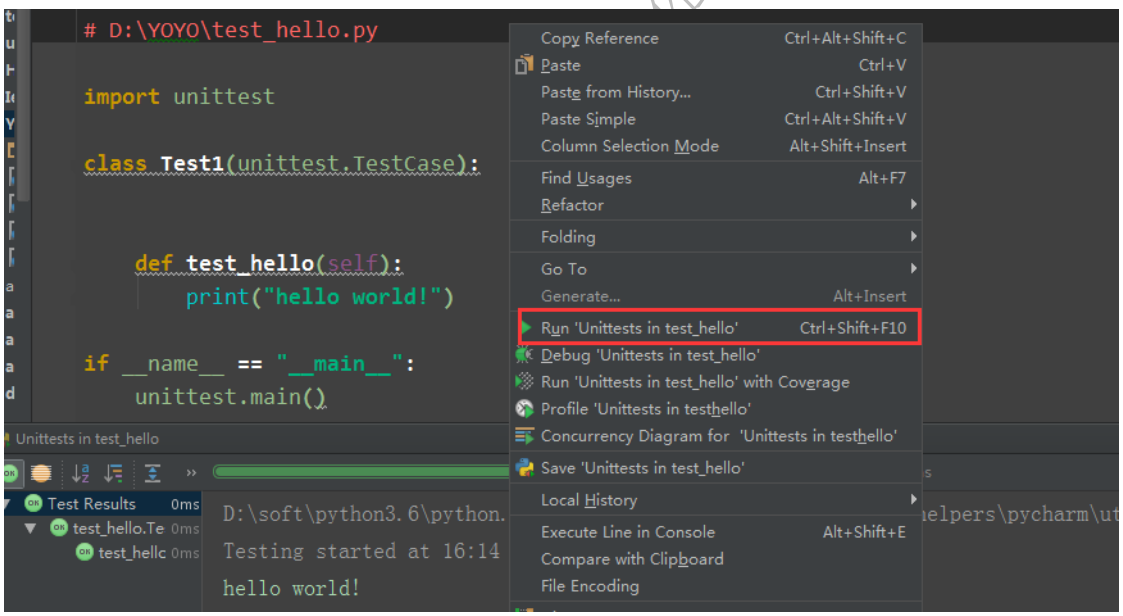
写完用例之后，需要调试看看，是不是能正常运行，如果每次跑去 cmd 执行，太麻烦，所以很有必要学习如何在 pycharm 里面运行 pytest 用例

pycharm 运行三种方式

1.以 xx.py 脚本方式直接执行，当写的代码里面没用到 unittest 和 pytest 框架时，并且脚本名称不是以 test_开头命名的，此时 pycharm 会以 xx.py 脚本方式运行



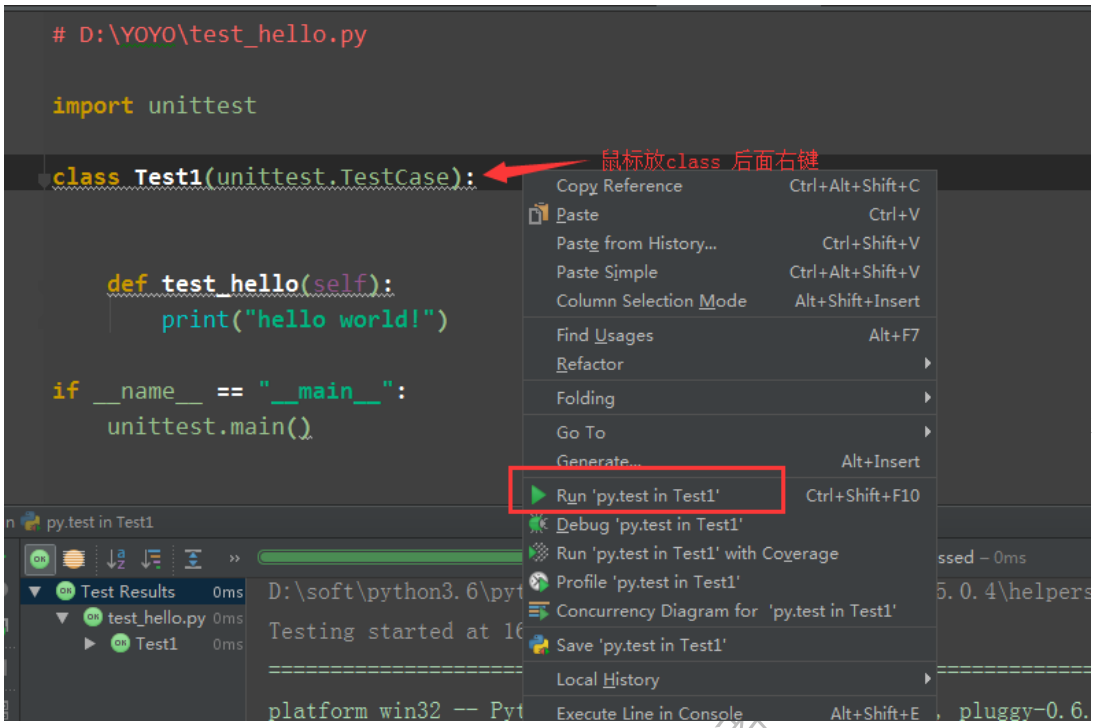
2.当脚本命名为 test_xx.py 时，用到 unittest 框架，此时运行代码，pycharm 会自动识别到以 unittest 方式运行



3.以 pytest 方式运行，需要改该工程设置默认的运行器：

file->Setting->Tools->Python Integrated Tools->项目名称

->Default test runner->选择 py.test



pycharm 写 pytest 代码

1.在 pycharm 里面写 pytest 用例，先导入 pytest

```
# D:/YOYO/test_class.py


** 作者：上海-悠悠 QQ 交流群：874033608**


import pytest


class TestClass:
        def test_one(self):
```

```
            x = "this"
            assert 'h' in x

        def test_two(self):
            x = "hello"
            assert hasattr(x, 'check')

        def test_three(self):
            a = "hello"
            b = "hello world"
            assert a in b

if __name__ == "__main__":
    pytest.main('-q test_class.py')
```

运行结果

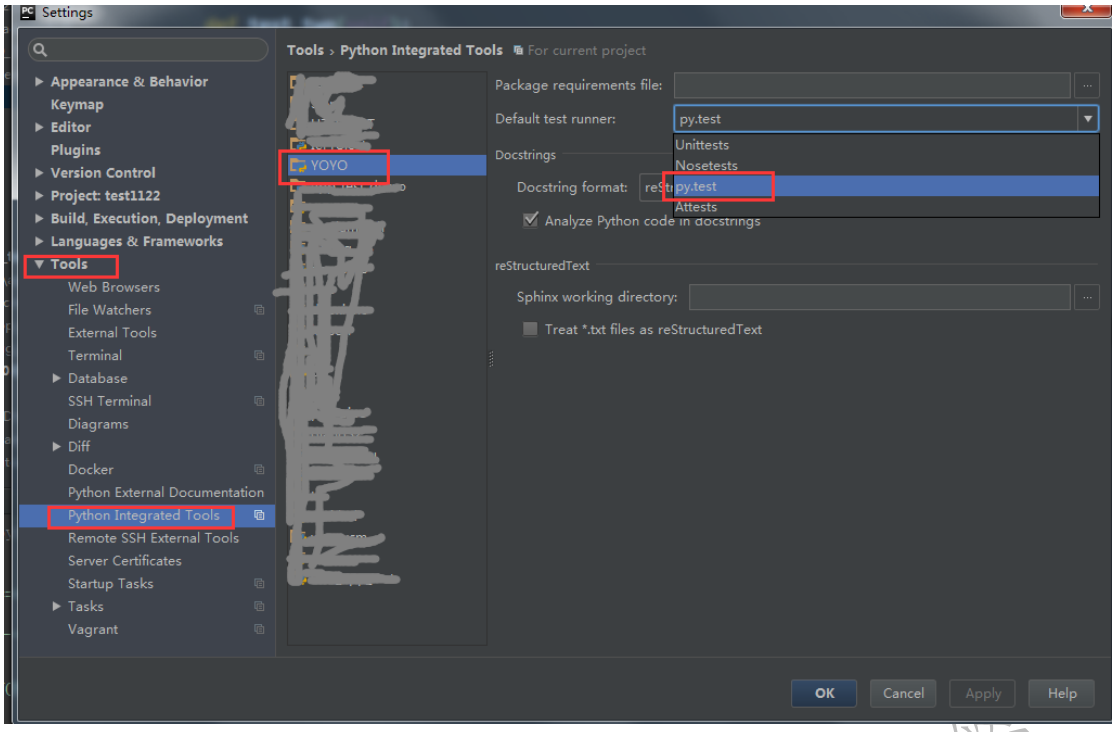
```
.F.
================================== FAILURES ===================================
_____________________________ TestClass.test_two ______________________________

self = <YOYO.test_class.TestClass object at 0x00000000039F9080>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:11: AssertionError
============================== warnings summary ===============================
<undetermined location>
  passing a string to pytest.main() is deprecated, pass a list o

-- Docs: http://doc.pytest.org/en/latest/warnings.html
1 failed, 2 passed, 1 warnings in 0.06 seconds
```

2.运行结果“.F.” 点是代表测试通过，F 是 Fail 的意思，1 warnings 是用于 pytest.main(‘-q test_class.py’)里面参数需要传 list，多个参数放 list 就不会有警告了

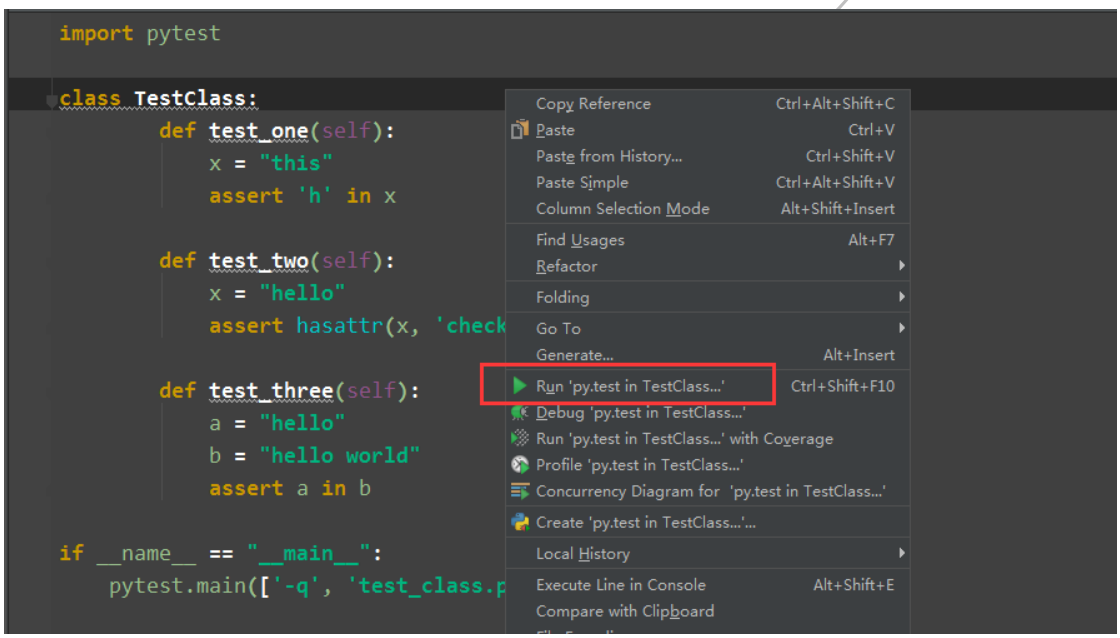
```
pytest.main(['-q', 'test_class.py'])
```

pycharm 设置 pytest

1.新建一个工程后，左上角 file->Setting->Tools->Python Integrated Tools->项目名称->Default test runner->选择 py.test



2.改完之后，再重新建个脚本（注意是先改项目运行方式，再写代码才能出来），接下来右键运行就能出来 pytest 运行了



3.pytest 是可以兼容 unittest 脚本的，之前写的 unittest 用例也能用 pytest 框架去运行

1.4-测试用例 setup 和 teardown

前言

学过 unittest 的都知道里面用前置和后置 setup 和 teardown 非常好用，在每次用例开始前和结束后都去执行一次。

当然还有更高级一点的 setupClass 和 teardownClass，需配合 @classmethod 装饰器一起使用，在做 selenium 自动化的时候，它的效率尤为突然，可以只启动一次浏览器执行多个用例。

pytest 框架也有类似于 setup 和 teardown 的语法，并且还不止这四个

用例运行级别

- 模块级（setup_module/teardown_module）开始于模块始末，全局的
- 函数级（setup_function/teardown_function）只对函数用例生效（不在类中）
- 类级（setup_class/teardown_class）只在类中前后运行一次(在类中)
- 方法级（setup_method/teardown_method）开始于方法始末（在类中）
- 类里面的（setup/teardown）运行在调用方法的前后

函数式

setup_function/teardown_function

pytest 框架支持函数和类两种用例方式，先看函数里面的前置与后置用法:

setup_function/teardown_function 每个用例开始和结束调用一次

```
# test_fixt.py


# coding:utf-8
import pytest
# 函数式


# ** 作者 上海-悠悠 QQ 交流群 874033608**


def setup_function():
    print("setup_function：每个用例开始前都会执行")


def teardown_function():
    print("teardown_function：每个用例结束后都会执行")


def test_one():
    print("正在执行----test_one")
    x = "this"
    assert 'h' in x
```

```
def test_two():
    print("正在执行----test_two")
    x = "hello"
    assert hasattr(x, 'check')


def test_three():
    print("正在执行----test_three")
    a = "hello"
    b = "hello world"
    assert a in b


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

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: E:\YOYO, inifile:
collected 3 items

test_fixt.py setup_function：每个用例开始前都会执行
正在执行----test_one
.teardown_function：每个用例结束后都会执行
setup_function：每个用例开始前都会执行
正在执行----test_two
Fteardown_function：每个用例结束后都会执行
setup_function：每个用例开始前都会执行
正在执行----test_three
.teardown_function：每个用例结束后都会执行


================================== FAILURES ====================
_____________________________ test_two _____________________

    def test_two():
        print("正在执行----test_two")
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_fixt.py:19: AssertionError
==================== 1 failed, 2 passed in 0.03 seconds ========
```

从结果可以看出用例执行顺序：setup_function》用例 1》teardown_function，setup_function》用例 2》teardown_function，setup_function》用例 3》teardown_function

setup_function/teardown_function

setup_module 是所有用例开始前只执行一次，teardown_module 是所有用例结束后只执行一次

```
# test_fixt.py


# coding:utf-8
import pytest
# 函数式


# ** 作者：上海-悠悠 QQ 交流群：874033608**


def setup_module():
    print("setup_module：整个.py 模块只执行一次")
    print("比如：所有用例开始前只打开一次浏览器")


def teardown_module():
    print("teardown_module：整个.py 模块只执行一次")
    print("比如：所有用例结束只最后关闭浏览器")


def setup_function():
    print("setup_function：每个用例开始前都会执行")


def teardown_function():
    print("teardown_function：每个用例结束前都会执行")
```

```
def test_one():
    print("正在执行----test_one")
    x = "this"
    assert 'h' in x


def test_two():
    print("正在执行----test_two")
    x = "hello"
    assert hasattr(x, 'check')


def test_three():
    print("正在执行----test_three")
    a = "hello"
    b = "hello world"
    assert a in b


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

从运行结果可以看到 setup_module 和 teardown_module 只执行了一次

test_fixt.py setup_module：整个.py 模块只执行一次

比如：所有用例开始前只打开一次浏览器

setup_function：每个用例开始前都会执行

正在执行----test_one

.teardown_function：每个用例结束前都会执行

setup_function：每个用例开始前都会执行

正在执行----test_two

Fteardown_function：每个用例结束前都会执行

setup_function：每个用例开始前都会执行

正在执行----test_three

.teardown_function：每个用例结束前都会执行

teardown_module：整个.py 模块只执行一次

比如：所有用例结束只最好关闭浏览器

类和方法

1.setup/teardown 和 unittest 里面的 setup/teardown 是一样的功能，setup_class 和 teardown_class 等价于 unittest 里面的 setupClass 和 teardownClass

```
#test_fixtclass.py


# coding:utf-8
import pytest
# 类和方法


# ** 作者：上海-悠悠 QQ 交流群：874033608**
```

```
class TestCase():

    def setup(self):
        print("setup: 每个用例开始前执行")

    def teardown(self):
        print("teardown: 每个用例结束后执行")

    def setup_class(self):
        print("setup_class：所有用例执行之前")

    def teardown_class(self):
        print("teardown_class：所有用例执行之前")

    def setup_method(self):
        print("setup_method:  每个用例开始前执行")

    def teardown_method(self):
        print("teardown_method:  每个用例结束后执行")

    def test_one(self):
```

```
        print("正在执行----test_one")
        x = "this"
        assert 'h' in x

    def test_two(self):
        print("正在执行----test_two")
        x = "hello"
        assert hasattr(x, 'check')

    def test_three(self):
        print("正在执行----test_three")
        a = "hello"
        b = "hello world"
        assert a in b

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

运行结果:

test_fixtclass.py setup_class：所有用例执行之前

setup_method： 每个用例开始前执行

setup：每个用例开始前执行

正在执行----test_one

.teardown: 每个用例结束后执行

teardown_method: 每个用例结束后执行

setup_method: 每个用例开始前执行

setup: 每个用例开始前执行

正在执行----test_two

Fteardown: 每个用例结束后执行

teardown_method: 每个用例结束后执行

setup_method: 每个用例开始前执行

setup: 每个用例开始前执行

正在执行----test_three

.teardown: 每个用例结束后执行

teardown_method: 每个用例结束后执行

teardown_class：所有用例执行之前

2.从结果看出，运行的优先级：setup_class》setup_method》setup 》用例》teardown》teardown_method》teardown_class

函数和类混合

1.如果一个.py 的文件里面既有函数用例又有类和方法用例，运行顺序又是怎样的呢?

```
# coding:utf-8
import pytest
# 类和方法
```

```
# ** 作者：上海-悠悠 QQ 交流群：874033608**

def setup_module():
    print("setup_module：整个.py 模块只执行一次")
    print("比如：所有用例开始前只打开一次浏览器")

def teardown_module():
    print("teardown_module：整个.py 模块只执行一次")
    print("比如：所有用例结束只最后关闭浏览器")

def setup_function():
    print("setup_function：每个用例开始前都会执行")

def teardown_function():
    print("teardown_function：每个用例结束前都会执行")

def test_one():
    print("正在执行----test_one")
    x = "this"
    assert 'h' in x
```

```
def test_two():
    print("正在执行----test_two")
    x = "hello"
    assert hasattr(x, 'check')


class TestCase():


    def setup_class(self):
        print("setup_class：所有用例执行之前")


    def teardown_class(self):
        print("teardown_class：所有用例执行之前")


    def test_three(self):
        print("正在执行----test_three")
        x = "this"
        assert 'h' in x


    def test_four(self):
        print("正在执行----test_four")
        x = "hello"
        assert hasattr(x, 'check')
```

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

运行结果：

test_fixtclass.py setup_module：整个.py 模块只执行一次

比如：所有用例开始前只打开一次浏览器

setup_function：每个用例开始前都会执行

正在执行----test_one

.teardown_function：每个用例结束前都会执行

setup_function：每个用例开始前都会执行

正在执行----test_two

Fteardown_function：每个用例结束前都会执行

setup_class：所有用例执行之前

正在执行----test_three

.正在执行----test_four

Fteardown_class：所有用例执行之前

teardown_module：整个.py 模块只执行一次

比如：所有用例结束只最后关闭浏览器

2.从运行结果看出，setup_module/teardown_module 的优先级是最大的，然后函数里面用到的 setup_function/teardown_function 与类里面的 setup_class/teardown_class 互不干涉

1.5-fixture 之 conftest.py

前言

前面一篇讲到用例加 setup 和 teardown 可以实现在测试用例之前或之后加入一些操作，但这种是整个脚本全局生效的，如果我想实现以下场景：

用例 1 需要先登录，用例 2 不需要登录，用例 3 需要先登录。很显然这就无法用 setup 和 teardown 来实现了。这就是本篇学习的目的，自定义测试用例的预置条件

fixture 优势

firture 相对于 setup 和 teardown 来说应该有以下几点优势

- 命名方式灵活，不局限于 setup 和 teardown 这几个命名
- conftest.py 配置里可以实现数据共享，不需要 import 就能自动找到一些配置
- scope="module" 可以实现多个.py 跨文件共享前置
- scope="session" 以实现多个.py 跨文件使用一个 session 来完成多个用例


```
fixture(scope="function", params=None, autouse=False,
ids=None, name=None):

    """使用装饰器标记 fixture 的功能
```

可以使用此装饰器 (带或不带参数) 来定义 fixture 功能。 fixture 功能的名称可以在以后使用

引用它会在运行测试之前调用它：test 模块或类可以使用 pytest.mark.usefixtures（fixturename 标记。

测试功能可以直接使用 fixture 名称作为输入参数，在这种情况下，夹具实例从 fixture 返回功能将被注入。

:arg scope: scope 有四个级别参数 "function" (默认), "class", "module" or "session".

:arg params: 一个可选的参数列表，它将导致多个参数调用 fixture 功能和所有测试使用它

:arg autouse: 如果为 True，则为所有测试激活 fixture func 可以看到它。 如果为 False（默认值）则显式需要参考来激活 fixture

:arg ids: 每个字符串 id 的列表，每个字符串对应于 params 这样他们就是测试 ID 的一部分。 如果没有提供 ID 它们将从 params 自动生成

:arg name: fixture 的名称。 这默认为装饰函数的名称。 如果 fixture 在定义它的同一模块中使用，夹具的功能名称将被请求夹具的功能 arg 遮蔽; 解决这个问题的一种方法是将装饰函数命名

fixture_ <fixturename>”然后使用”@ pytest.fixture（name ='<fixturename>'） “” 。

Fixtures 可以选择使用 yield 语句为测试函数提供它们的值，而不是 return。 在这种情况下，yield 语句之后的代码块作为拆卸代码执行，而不管测试结果如何。fixture 功能必须只产生一次

fixture 参数传入（scope=”function”）

1.实现场景：用例 1 需要先登录，用例 2 不需要登录，用例 3 需要先登录

```
# 新建一个文件 test_fix.py
# coding:utf-8
#** 作者：上海-悠悠 QQ 交流群：874033608**
import pytest


# 不带参数时默认 scope="function"
@pytest.fixture()
def login():
    print("输入账号，密码先登录")


def test_s1(login):
    print("用例 1：登录之后其它动作 111")


def test_s2():  # 不传 login
```

```
    print("用例 2：不需要登录，操作 222")


def test_s3(login):
    print("用例 3：登录之后其它动作 333")


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

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: E:\YOYO, inifile:
collected 3 items

test_fix.py 输入账号，密码先登录
用例1：登录之后其它动作111
.用例2：不需要登录，操作222
.输入账号，密码先登录
用例3：登录之后其它动作333
.

========================== 3 passed in 0.06 seconds ============
```

2.如果@pytest.fixture()里面没有参数，那么默认 scope="function"，也就是此时的级别的 function，针对函数有效

conftest.py 配置

1.上面一个案例是在同一个.py 文件中，多个用例调用一个登陆功能，如果有多个.py 的文件都需要调用这个登陆功能的话，那就不能把登陆写到用例里面去了。

此时应该要有一个配置文件，单独管理一些预置的操作场景，pytest 里面默认读取 conftest.py 里面的配置

conftest.py 配置需要注意以下点：

- conftest.py 配置脚本名称是固定的，不能改名称
- conftest.py 与运行的用例要在同一个 pakage 下，并且有 __init__.py 文件
- 不需要 import 导入 conftest.py，pytest 用例会自动查找

2.参考脚本代码设计如下

__init__.py

conftest.py


```
# coding:utf-8
import pytest


@pytest.fixture()
def login():
    print("输入账号，密码先登录")
```

test_fix1.py

```
# coding:utf-8
import pytest


def test_s1(login):
```

```
    print("用例 1：登录之后其它动作 111")


def test_s2():  # 不传 login
    print("用例 2：不需要登录，操作 222")


def test_s3(login):
    print("用例 3：登录之后其它动作 333")


if __name__ == "__main__":
    pytest.main(["-s", "test_fix1.py"])
```

test_fix2.py

```
# coding:utf-8
import pytest


def test_s4(login):
    print("用例 4：登录之后其它动作 111")


def test_s5():  # 不传 login
    print("用例 5：不需要登录，操作 222")


if __name__ == "__main__":
    pytest.main(["-s", "test_fix2.py"])
```

** 作者：上海-悠悠 QQ 交流群：874033608**

3.单独运行 test_fix1.py 和 test_fix2.py 都能调用到 login()方法，这样就能实现一些公共的操作可以单独拿出来了

1.6-fixture 之 yield 实现 teardown

前言

上一篇讲到 fixture 通过 scope 参数控制 setup 级别，既然有 setup 作为用例之前前的操作，用例执行完之后那肯定也有 teardown 操作。这里用到 fixture 的 teardown 操作并不是独立的函数，用 yield 关键字呼唤 teardown 操作

scope=”module”

1.fixture 参数 scope=”module”，module 作用是整个.py 文件都会生效，用例调用时，参数写上函数名称就行

```
# 新建一个文件 test_f1.py
# coding:utf-8
import pytest
'''
** 作者：上海-悠悠 QQ 交流群：874033608**
'''

@pytest.fixture(scope="module")
```

```
def open():
    print("打开浏览器，并且打开百度首页")


def test_s1(open):
    print("用例 1：搜索 python-1")


def test_s2(open):
    print("用例 2：搜索 python-2")


def test_s3(open):
    print("用例 3：搜索 python-3")


if __name__ == "__main__":
    pytest.main(["-s", "test_f1.py"])
```

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
collected 3 items

..\..\..\..\..\..\YOYO\test_f1.py 打开浏览器，并且打开百度首页
用例1：搜索python-1
.用例2：搜索python-2
.用例3：搜索python-3
.

========================== 3 passed in 0.01 seconds ============
```

从结果看出，虽然 test_s1,test_s2,test_s3 三个地方都调用了 open 函数，但是它只会在第一个用例前执行一次

2.如果 test_s1 不调用,test_s2（调用 open）,test_s3 不调用，运行顺序会是怎样的?

```
# 新建一个文件 test_f1.py
# coding:utf-8
import pytest
'''
** 作者：上海-悠悠 QQ 交流群：874033608**
'''

@pytest.fixture(scope="module")
def open():
    print("打开浏览器，并且打开百度首页")

def test_s1():
    print("用例 1：搜索 python-1")

def test_s2(open):
    print("用例 2：搜索 python-2")

def test_s3():
```

```
    print("用例 3：搜索 python-3")


if __name__ == "__main__":
    pytest.main(["-s", "test_f1.py"])
```

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
collected 3 items

..\..\..\..\..\..\YOYO\test_f1.py 用例1：搜索python-1
.打开浏览器，并且打开百度首页
用例2：搜索python-2
.用例3：搜索python-3
.

========================== 3 passed in 0.01 seconds ============
```

从结果看出，module 级别的 fixture 在当前.py 模块里，只会在用例（test_s2）第一次调用前执行一次

yield 执行 teardown

1.前面讲的是在用例前加前置条件，相当于 setup,既然有 setup 那就有 teardown,fixture 里面的 teardown 用 yield 来唤醒 teardown 的执行

```
# 新建一个文件 test_f1.py
# coding:utf-8
import pytest
'''
```

```
** 作者：上海-悠悠 QQ 交流群：874033608**
'''


@pytest.fixture(scope="module")
def open():
    print("打开浏览器，并且打开百度首页")

    yield
    print("执行 teardown!")
    print("最后关闭浏览器")

def test_s1(open):
    print("用例 1：搜索 python-1")

def test_s2(open):
    print("用例 2：搜索 python-2")

def test_s3(open):
    print("用例 3：搜索 python-3")

if __name__ == "__main__":
    pytest.main(["-s", "test_f1.py"])
```

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
collected 3 items

..\..\..\..\..\..\YOYO\test_f1.py 打开浏览器，并且打开百度首页
用例1：搜索python-1
.用例2：搜索python-2
.用例3：搜索python-3
.执行teardown!
最后关闭浏览器


========================== 3 passed in 0.01 seconds ============
```

yield 遇到异常

1.如果其中一个用例出现异常，不影响 yield 后面的 teardown 执行,运行结果互不影响，并且全部用例执行完之后，yield 呼唤 teardown 操作

```
# 新建一个文件 test_f1.py
# coding:utf-8
import pytest
'''
** 作者：上海-悠悠 QQ 交流群：874033608**
'''


@pytest.fixture(scope="module")
def open():
```

```
    print("打开浏览器，并且打开百度首页")
    yield
    print("执行 teardown!")
    print("最后关闭浏览器")

def test_s1(open):
    print("用例 1：搜索 python-1")

    # 如果第一个用例异常了，不影响其他的用例执行
    raise NameError  # 模拟异常

def test_s2(open):
    print("用例 2：搜索 python-2")

def test_s3(open):
    print("用例 3：搜索 python-3")

if __name__ == "__main__":
    pytest.main(["-s", "test_f1.py"])
```

运行结果：

```
\YOYO\test_f1.py 打开浏览器，并且打开百度首页
用例1：搜索python-1
F
open = None

    def test_s1(open):
        print("用例1：搜索python-1")

        # 如果第一个用例异常了，不影响其他的用例执行
>       raise NameError  # 模拟异常
E       NameError

D:\YOYO\test_f1.py:16: NameError
用例2：搜索python-2
.用例3：搜索python-3
.执行teardown!
最后关闭浏览器
```

2.如果在 setup 就异常了，那么是不会去执行 yield 后面的 teardown 内容了

3.yield 也可以配合 with 语句使用，以下是官方文档给的案例

```
# 官方文档案例
# content of test_yield2.py


import smtplib
import pytest
'''
** 作者：上海-悠悠 QQ 交流群：588402570**
'''
```

```
@pytest.fixture(scope="module")
def smtp():
    with smtplib.SMTP("smtp.gmail.com") as smtp:
        yield smtp  # provide the fixture value
```

addfinalizer 终结函数

1.除了 yield 可以实现 teardown,在 request-context 对象中注册 addfinalizer 方法也可以实现终结函数。

```
# 官方案例


# content of conftest.py
import smtplib
import pytest


@pytest.fixture(scope="module")
def smtp_connection(request):
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587,
timeout=5)
    def fin():
        print("teardown smtp_connection")
        smtp_connection.close()
```

```
    request.addfinalizer(fin)
    return smtp_connection  # provide the fixture value
```

2.yield 和 addfinalizer 方法都是在测试完成后呼叫相应的代码。但是 addfinalizer 不同的是：

- 他可以注册多个终结函数。
- 这些终结方法总是会被执行，无论在之前的 setup code 有没有抛出错误。这个方法对于正确关闭所有的 fixture 创建的资源非常便利，即使其一在创建或获取时失败

1.7-fixture 之 autouse=True

前言

平常写自动化用例会写一些前置的 fixture 操作，用例需要用到就直接传该函数的参数名称就行了。当用例很多的时候，每次都传这个参数，会比较麻烦。

fixture 里面有个参数 autouse，默认是 Fasle 没开启的，可以设置为 True 开启自动使用 fixture 功能，这样用例就不用每次都去传参了

调用 fixture 三种方法

- 1.函数或类里面方法直接传 fixture 的函数参数名称
- 2.使用装饰器@pytest.mark.usefixtures()修饰
- 3.autouse=True 自动使用

用例传 fixture 参数

方法一：先定义 start 功能，用例全部传 start 参数，调用该功能

```
# content of test_06.py
import time
import pytest

# ** 作者：上海-悠悠 QQ 交流群：874033608**

@pytest.fixture(scope="function")
def start(request):
    print('\n-----开始执行 function----')

def test_a(start):
    print("-------用例 a 执行-------")

class Test_aaa():
   def test_01(self, start):
        print('------------用例 01--------------')

    def test_02(self, start):
        print('------------用例 02-------------')
```

```
if __name__ == "__main__":
    pytest.main(["-s", "test_06.py"])
```

装饰器 usefixtures

方法二：使用装饰器@pytest.mark.usefixtures()修饰需要运行的用例

```
# content of test_07.py
import time
import pytest

# ** 作者：上海-悠悠 QQ 交流群：874033608**

@pytest.fixture(scope="function")
def start(request):
    print('\n-----开始执行 function----')


@pytest.mark.usefixtures("start")
def test_a():
    print("-------用例 a 执行-------")

@pytest.mark.usefixtures("start")
```

```
class Test_aaa():


    def test_01(self):
        print('------------用例 01---------------')


    def test_02(self):
        print('------------用例 02-------------')


if __name__ == "__main__":
    pytest.main(["-s", "test_07.py"])
```

设置 autouse=True

方法三、autouse 设置为 True，自动调用 fixture 功能

- start 设置 scope 为 module 级别，在当前.py 用例模块只执行一次，autouse=True 自动使用
- open_home 设置 scope 为 function 级别，每个用例前都调用一次，自动使用

```
# content of test_08.py
import time
import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**
```

```
@pytest.fixture(scope="module", autouse=True)
def start(request):
    print('\n-----开始执行 moule----')
    print('module       : %s' % request.module.__name__)
    print('----------启动浏览器---------')
    yield
    print("------------结束测试 end!-----------")

@pytest.fixture(scope="function", autouse=True)
def open_home(request):
    print("function：%s \n--------回到首页--------" %
request.function.__name__)

def test_01():
    print('----------用例 01--------------')

def test_02():
    print('----------用例 02-----------')

if __name__ == "__main__":
    pytest.main(["-s", "test_08.py"])
```

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 2 items

..\..\..\..\..\..\YOYO\peizhi\test_08.py
-----开始执行moule----
module       : YOYO.peizhi.test_08
----------启动浏览器---------
function：test_01
--------回到首页--------
-----------用例01--------------
.function：test_02
--------回到首页--------
-----------用例02------------
.------------结束测试-----------


========================== 2 passed in 0.01 seconds =============
```

上面是函数去实现用例，写的 class 里也是一样可以的

```
# content of test_09.py

import time

import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**


@pytest.fixture(scope="module", autouse=True)

def start(request):
```

```
    print('\n-----开始执行 moule----')
    print('module       : %s' % request.module.__name__)
    print('----------启动浏览器----------')
    yield
    print("-------------结束测试 end!------------")


class Test_aaa():
    @pytest.fixture(scope="function", autouse=True)
    def open_home(self, request):
        print("function: %s \n--------回到首页--------" %
request.function.__name__)


    def test_01(self):
        print('----------用例 01-------------')


    def test_02(self):
        print('----------用例 02-----------')


if __name__ == "__main__":
    pytest.main(["-s", "test_09.py"])
```

1.8-参数化 parametrize

前言

pytest.mark.parametrize 装饰器可以实现测试用例参数化。

parametrizing

1.这里是一个实现检查一定的输入和期望输出测试功能的典型例子

```
# content of test_expectation.py
# coding:utf-8
import pytest
@pytest.mark.parametrize("test_input,expected",
                         [ ("3+5", 8),
                           ("2+4", 6),
                           ("6 * 9", 42),
                         ])
def test_eval(test_input, expected):
    assert eval(test_input) == expected


if __name__ == "__main__":
    pytest.main(["-s", "test_canshu1.py"])
```

运行结果

```
================================== FAILURES ===================================
_____________________________ test_eval[6 * 9-42] _____________________________

test_input = '6 * 9', expected = 42

    @pytest.mark.parametrize("test_input,expected",
                             [ ("3+5", 8),
                               ("2+4", 6),
                               ("6 * 9", 42),
                             ])
    def test_eval(test_input, expected):
>       assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E        +  where 54 = eval('6 * 9')

test_canshu1.py:11: AssertionError
==================== 1 failed, 2 passed in 1.98 seconds ========
```

在这个例子中设计的，只有一条输入/输出值的简单测试功能。和往常一样

函数的参数，你可以在运行结果看到在输入和输出值

2.它也可以标记单个测试实例在参数化，例如使用内置的 mark.xfail

```
# content of test_expectation.py
import pytest
@pytest.mark.parametrize("test_input,expected", [
                         ("3+5", 8),
                         ("2+4", 6),
                         pytest.param("6 * 9", 42,
marks=pytest.mark.xfail),
                         ])
```

```
def test_eval(test_input, expected):
    print("-------开始用例------")
    assert eval(test_input) == expected




if __name__ == "__main__":
    pytest.main(["-s", "test_canshu1.py"])
```


运行结果：

test_canshu1.py -------开始用例------

.-------开始用例------

.-------开始用例------

x

========== 2 passed, 1 xfailed in 1.84 seconds ==========

标记为失败的用例就不运行了，直接跳过显示 xfailed

参数组合

1.若要获得多个参数化参数的所有组合，可以堆叠参数化装饰器

```
import pytest
@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
```

```
def test_foo(x, y):
    print("测试数据组合：x->%s, y->%s" % (x, y))



if __name__ == "__main__":
    pytest.main(["-s", "test_canshu1.py"])
```

运行结果

test_canshu1.py 测试数据组合：x->0, y->2

.测试数据组合：x->1, y->2

.测试数据组合：x->0, y->3

.测试数据组合：x->1, y->3

.

========== 4 passed in 1.75 seconds ==========

这将运行测试，参数设置为 x = 0 / y = 2，x = 1 / y = 2，x = 0 / y = 3，x = 1 / y = 3 组合参数。

1.9-assert 断言

前言

断言是写自动化测试基本最重要的一步，一个用例没有断言，就失去了自动化测试的意义了。什么是断言呢?

简单来讲就是实际结果和期望结果去对比，符合预期那就测试 pass，不符合预期那就测试 failed

assert

pytest 允许您使用标准 Python 断言来验证 Python 测试中的期望和值。例如，你可以写下

```
# content of test_assert1.py
def f():
    return 3
def test_function():
    assert f() == 4
```

断言 f()函数的返回值,接下来会看到断言失败,因为返回的值是 3，判断等于 4，所以失败了

```
$ pytest test_assert1.py
============================ test session starts ================
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0
rootdir: $REGENDOC_TMPDIR, inifile:
collected 1 item
test_assert1.py F [100%]
================================= FAILURES ======================
_____________________________ test_function _____________________
def test_function():
> assert f() == 4
E assert 3 == 4
E + where 3 = f()
test_assert1.py:5: AssertionError
========================== 1 failed in 0.12 seconds =============
```

从报错信息可以看到断言失败原因：E assert 3 == 4

异常信息

接下来再看一个案例，如果想在异常的时候，输出一些提示信息，这样报错后，就方便查看是什么原因了

```
def f():
    return 3


def test_function():


    a = f()
    assert a % 2 == 0, "判断 a 为偶数，当前 a 的值为：%s"%a
```

运行结果

```
================================== FAILURES ===================================
_____________________________ test_function ______________________________

    def test_function():

        a = f()
>       assert a % 2 == 0, "判断a为偶数，当前a的值为：%s"%a
E       AssertionError: 判断a为偶数，当前a的值为：3
E       assert (3 % 2) == 0

test_03.py:9: AssertionError
=========================== 1 failed in 0.18 seconds ==========
```

这样当断言失败的时候，会给出自己写的失败原因了

E AssertionError: 判断 a 为偶数，当前 a 的值为：3

异常断言

为了写关于引发异常的断言，可以使用 pytest.raises 作为上下文管理器，如下

```
# content of test_assert1.py


import pytest
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0.
rootdir: D:\YOYO\canshuhua, inifile:
plugins: metadata-1.7.0, html-1.19.0
collected 1 item

test_assert1.py.

========================== 1 passed in 0.31 seconds =============
```

如果我们要断言它抛的异常是不是预期的，比如执行：1/0,预期结果是抛异常：ZeroDivisionError: division by zero，那我们要断言这个异常，通常是断言异常的 type 和 value 值了。

这里 1/0 的异常类型是 ZeroDivisionError，异常的 value 值是 division by zero，于是用例可以这样设计

```
# content of test_assert1.py
# ** 作者：上海-悠悠 QQ 交流群：874033608**
```

```
import pytest
def test_zero_division():
    '''断言异常'''
    with pytest.raises(ZeroDivisionError) as excinfo:
        1 / 0

    # 断言异常类型 type
    assert excinfo.type == ZeroDivisionError
    # 断言异常 value 值
    assert "division by zero" in str(excinfo.value)
```


excinfo 是一个异常信息实例，它是围绕实际引发的异常的包装器。主要属性是.type、 .value 和 .traceback

注意：断言 type 的时候，异常类型是不需要加引号的，断言 value 值的时候需转 str

在上下文管理器窗体中，可以使用关键字参数消息指定自定义失败消息：

常用断言

pytest 里面断言实际上就是 python 里面的 assert 断言方法，常用的有以下几种

- assert xx 判断 xx 为真
- assert not xx 判断 xx 不为真

- assert a in b 判断 b 包含 a
- assert a == b 判断 a 等于 b
- assert a != b 判断 a 不等于 b

```
import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**


def is_true(a):
    if a > 0:
        return True
    else:
        return False


def test_01():
    '''断言 xx 为真'''
    a = 5
    b = -1
    assert is_true(a)
    assert not is_true(b)


def test_02():
    '''断言 b 包含 a'''
```

```
    a = "hello"
    b = "hello world"
    assert a in b


def test_03():
    '''断言相等'''
    a = "yoyo"
    b = "yoyo"
    assert a == b


def test_04():
    '''断言不等于'''
    a = 5
    b = 6
    assert a != b


if __name__ == "__main__":
    pytest.main(["-s", "test_01.py"])
```

1.10-skip 跳过用例

前言

pytest.mark.skip 可以标记无法在某些平台上运行的测试功能，或者您希望失败的测试功能

skip 意味着只有在满足某些条件时才希望测试通过，否则 pytest 应该跳过运行测试。 常见示例是在非 Windows 平台上跳过仅限 Windows 的测试，或跳过测试依赖于当前不可用的外部资源（例如数据库）。

xfail 意味着您希望测试由于某种原因而失败。 一个常见的例子是对功能的测试尚未实施，或尚未修复的错误。 当测试通过时尽管预计会失败（标有 pytest.mark.xfail），它是一个 xpass，将在测试摘要中报告。

pytest 计数并分别列出 skip 和 xfail 测试。 未显示有关跳过/xfailed 测试的详细信息默认情况下，以避免混乱输出。 您可以使用-r 选项查看与“short”字母对应的详细信息显示在测试进度中

```
> pytest -rxXs # show extra info on xfailed, xpassed, and skipped tests
```

有关-r 选项的更多详细信息，请运行 pytest -h

skip

跳过测试函数的最简单方法是使用跳过装饰器标记它，可以传递一个可选的原因

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
```

...

或者，也可以通过调用来在测试执行或设置期间强制跳过 pytest.skip（reason）功能：

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

也可以使用 pytest.skip（reason，allow_module_level = True）跳过整个模块级别：

```
import pytest
if not pytest.config.getoption("--custom-flag"):
    pytest.skip("--custom-flag is missing, skipping tests", allow_module_level=True)
```

当在导入时间内无法评估跳过条件时，命令性方法很有用。

skipif

如果您希望有条件地跳过某些内容，则可以使用 skipif 代替。 这是标记测试的示例在 Python3.6 之前的解释器上运行时要跳过的函数

```
import sys
@pytest.mark.skipif(sys.version_info < (3,6),
reason="requires python3.6 or higher")
def test_function():
    ...
```

如果条件在收集期间评估为 True，则将跳过测试函数，具有指定的原因使用-rs 时出现在摘要中。

```
# content of test_mymodule.py
import mymodule
minversion = pytest.mark.skipif(mymodule.__versioninfo__ <
(1,1),
reason="at least mymodule-1.1 required")
@minversion
def test_function():
    ...
```

您可以在模块之间共享 skipif 标记。参考以下案例

```
# test_myothermodule.py
from test_mymodule import minversion
@minversion
def test_anotherfunction():
    ...
```

您可以导入标记并在另一个测试模块中重复使用它：

对于较大的测试套件，通常最好有一个文件来定义标记，然后一致适用于整个测试套件。

或者，您可以使用条件字符串而不是布尔值，但它们之间不能轻易共享它们支持它们主要是出于向后兼容的原因

skip 类或模块

您可以在类上使用 skipif 标记（与任何其他标记一样）：

```
@pytest.mark.skipif(sys.platform == 'win32',
reason="does not run on windows")
class TestPosixCalls(object):
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

如果条件为 True，则此标记将为该类的每个测试方法生成跳过结果

如果要跳过模块的所有测试功能，可以在全局级别使用 pytestmark 名称

test_module.py

pytestmark = pytest.mark.skipif(...)

如果将多个 skipif 装饰器应用于测试函数，则如果任何跳过条件为真，则将跳过它

skip 文件或目录

有时您可能需要跳过整个文件或目录，例如，如果测试依赖于特定于 Python 的版本功能或包含您不希望 pytest 运行的代码。 在这种情况下，您必须排除文件和目录来自收藏。 有关更多信息，请参阅自定义测试集合。

skip 缺少导入依赖项

您可以在模块级别或测试或测试设置功能中使用以下帮助程序

```
docutils = pytest.importorskip("docutils")
```

如果无法在此处导入 docutils，则会导致测试跳过结果。 你也可以跳过库的版本号

```
docutils = pytest.importorskip("docutils",
minversion="0.3")
```

将从指定模块的属性中读取版本。

概要

这是一个快速指南，介绍如何在不同情况下跳过模块中的测试

1.无条件地跳过模块中的所有测试：

```
pytestmark = pytest.mark.skip("all tests still WIP")
```

2.根据某些条件跳过模块中的所有测试

```
pytestmark = pytest.mark.skipif(sys.platform == "win32",
"tests for linux
    ➟ only"
```

3.如果缺少某些导入，则跳过模块中的所有测试

```
pexpect = pytest.importorskip("pexpect")
```


1.11-使用自定义标记 mark

前言

pytest 可以支持自定义标记，自定义标记可以把一个 web 项目划分多个模块，然后指定模块名称执行。app 自动化的时候，如果想

android 和 ios 公用一套代码时，

也可以使用标记功能，标明哪些是 ios 用例，哪些是 android 的，运行代码时候指定 mark 名称运行就可以

mark 标记

1.以下用例，标记 test_send_http()为 webtest

```
# content of test_server.py


import pytest


@pytest.mark.webtest
def test_send_http():
    pass # perform some webtest test for your app


def test_something_quick():
    pass


def test_another():
    pass


class TestClass:
    def test_method(self):
        pass
```

```
if __name__ == "__main__":
    pytest.main(["-s", "test_server.py", "-m=webtest"])
```

只运行用 webtest 标记的测试，cmd 运行的时候，加个-m 参数，指定参数值 webtest

```
> pytest -v -m webtest
```

```
============================= test session starts ================
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0.
rootdir: E:\YOYO\se, inifile:
plugins: metadata-1.7.0, html-1.19.0
collected 4 items / 3 deselected

test_server.py .

==================== 1 passed, 3 deselected in 0.10 seconds =======
```

如果不想执行标记 webtest 的用例，那就用”not webtest”

```
> pytest -v -m "not webtest"
```

```
import pytest


@pytest.mark.webtest
def test_send_http():
    pass # perform some webtest test for your app
def test_something_quick():
    pass
```

```
def test_another():
    pass
class TestClass:
    def test_method(self):
        pass


if __name__ == "__main__":
    pytest.main(["-s", "test_server.py", "-m='not webtest'"])
```

运行结果

============================= test session starts ==============================

platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0.6.0

rootdir: E:\YOYO\se, inifile:

plugins: metadata-1.7.0, html-1.19.0

collected 4 items

test_server.py ....

========================== 4 passed in 0.06 seconds ===========================

-v 指定的函数节点 id

如果想指定运行某个.py 模块下，类里面的一个用例，如：TestClass 里面 test 开头(或_test 结尾)的用例，函数(或方法)的名称就是用例的节点 id，指定节点 id 运行用-v 参数

```
> pytest -v test_server.py::TestClass::test_method
```

pycharm 运行代码

```
if __name__ == "__main__":

    pytest.main(["-v",
"test_server.py::TestClass::test_method"])
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
cachedir: .pytest_cache
metadata: {'Python': '3.6.0', 'Platform': 'Windows-10-10.0.17134
rootdir: E:\YOYO\se, inifile:
plugins: metadata-1.7.0, html-1.19.0
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED

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

当然也能选择运行整个 class

```
> pytest -v test_server.py::TestClass
```

也能选择多个节点运行，多个节点中间空格隔开

```
> pytest -v test_server.py::TestClass
test_server.py::test_send_http
```

pycharm 运行参考

```
if __name__ == "__main__":

    pytest.main(["-v", "test_server.py::TestClass",
"test_server.py::test_send_http"])
```

-k 匹配用例名称

可以使用-k 命令行选项指定在匹配用例名称的表达式

```
> pytest -v -k http
```

```
$ pytest -v -k http # running with the above defined example modu
============================ test session starts ==================
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.
cachedir: .pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items / 3 deselected
test_server.py::test_send_http PASSED [100%]
=================== 1 passed, 3 deselected in 0.12 seconds ========
```

您也可以运行所有的测试，根据用例名称排除掉某些用例：

```
> pytest -k "not send_http" -v
```

```
============================ test session starts ==================
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0
cachedir: .pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items / 1 deselected
test_server.py::test_something_quick PASSED [ 33%]
test_server.py::test_another PASSED [ 66%]
test_server.py::TestClass::test_method PASSED [100%]
=================== 3 passed, 1 deselected in 0.12 seconds ========
```

也可以同时选择匹配 “http” 和“quick”

```
> pytest -k "http or quick" -v
```

```
============================= test session starts ==============
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.
cachedir: .pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile:
collecting ... collected 4 items / 2 deselected
test_server.py::test_send_http PASSED [ 50%]
test_server.py::test_something_quick PASSED [100%]
================== 2 passed, 2 deselected in 0.12 seconds =======
```

1.12-用例 a 失败，跳过测试用例 b 和 c 并标记失败 xfail

前言

当用例 a 失败的时候，如果用例 b 和用例 c 都是依赖于第一个用例的结果，那可以直接跳过用例 b 和 c 的测试，直接给他标记失败 xfail 用到的场景，登录是第一个用例，登录之后的操作 b 是第二个用例，登录之后操作 c 是第三个用例，很明显三个用例都会走到登录。

如果登录都失败了，那后面 2 个用例就没测试必要了，直接跳过，并且标记为失败用例，这样可以节省用例时间。

用例设计

1.pytest 里面用 xfail 标记用例为失败的用例，可以直接跳过。实现基本思路

- 把登录写为前置操作

- 对登录的账户和密码参数化，参数用 canshu = [{"user":"amdin", "psw":"111"}]表示
- 多个用例放到一个 Test_xx 的 class 里
- test_01，test_02， test_03 全部调用 fixture 里面的 login 功能
- test_01 测试登录用例
- test_02 和 test_03 执行前用 if 判断登录的结果，登录失败就执行，pytest.xfail("登录不成功，标记为 xfail")

```
# content of test_05.py


# coding:utf-8
import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**


canshu = [{"user":"amdin", "psw":"111"}]


@pytest.fixture(scope="module")
def login(request):
    user = request.param["user"]
    psw = request.param["psw"]
    print("正在操作登录，账号：%s, 密码：%s" % (user, psw))
    if psw:
```

```
        return True
    else:
        return False


@pytest.mark.parametrize("login", canshu, indirect=True)
class Test_xx():

    def test_01(self, login):
        '''用例 1 登录'''
        result = login
        print("用例 1：%s" % result)
        assert result == True


    def test_02(self, login):
        result = login
        print("用例 3,登录结果：%s" % result)
        if not result:
            pytest.xfail("登录不成功，标记为 xfail")

        assert 1 == 1
```

```
    def test_03(self, login):
        result = login
        print("用例 3,登录结果：%s" %result)
        if not result:
            pytest.xfail("登录不成功, 标记为 xfail")

        assert 1 == 1


if __name__ == "__main__":
    pytest.main(["-s", "test_05.py"])
```

上面传的登录参数是登录成功的案例，三个用例全部通过

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 3 items

..\..\..\..\..\..\YOYO\peizhi\test_05.py 正在操作登录，账号：amdin,
用例1：True
.用例3,登录结果：True
.用例3,登录结果：True
.

========================== 3 passed in 0.02 seconds ============
```

标记为 xfail

1.再看看登录失败情况的用例,修改登录的参数

```
# content of test_05.py
# coding:utf-8
import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**


canshu = [{"user":"amdin", "psw":""}]


@pytest.fixture(scope="module")
def login(request):
    user = request.param["user"]
    psw = request.param["psw"]
    print("正在操作登录，账号：%s, 密码：%s" % (user, psw))
    if psw:
        return True
    else:
        return False



@pytest.mark.parametrize("login", canshu, indirect=True)
```

```
class Test_xx():


    def test_01(self, login):
        '''用例 1 登录'''
        result = login
        print("用例 1：%s" % result)
        assert result == True



    def test_02(self, login):
        result = login
        print("用例 3,登录结果：%s" % result)
        if not result:
            pytest.xfail("登录不成功，标记为 xfail")


        assert 1 == 1


    def test_03(self, login):
        result = login
        print("用例 3,登录结果：%s" %result)
        if not result:
            pytest.xfail("登录不成功，标记为 xfail")
```

```
        assert 1 == 1



if __name__ == "__main__":
    pytest.main(["-s", "test_05.py"])
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 3 items

..\..\..\..\..\..\YOYO\peizhi\test_05.py 正在操作登录，账号：amdin,
用例1：False
F
self = <YOYO.peizhi.test_05.Test_xx object at 0x00000000045ACF98>

    def test_01(self, login):
        '''用例1登录'''
        result = login
        print("用例1：%s" % result)
>       assert result == True
E       assert False == True

D:\YOYO\peizhi\test_05.py:24: AssertionError
用例3,登录结果：False
x
Test ignored.用例3,登录结果：False
x
Test ignored.
```

```
================================== FAILURES ===================================
_____________________________ Test_xx.test_01[login0] _____________

self = <YOYO.peizhi.test_05.Test_xx object at 0x00000000045ACF98>

    def test_01(self, login):
        '''用例1登录'''
        result = login
        print("用例1: %s" % result)
>       assert result == True
E       assert False == True

D:\YOYO\peizhi\test_05.py:24: AssertionError
==================== 1 failed, 2 xfailed in 0.06 seconds ======
```

从结果可以看出用例 1 失败了，用例 2 和 3 没执行，直接标记为 xfail 了

1.13-函数传参和 firture 传参数 request

前言

为了提高代码的复用性，我们在写用例的时候，会用到函数，然后不同的用例去调用这个函数。比如登录操作，大部分的用例都会先登录，那就需要把登录单独抽出来写个函数，其它用例全部的调用这个登陆函数就行。

但是登录的账号不能写死，有时候我想用账号 1 去登录，执行用例 1，用账号 2 去登录执行用例 2，所以需要对函数传参。

登录函数传参

把登录单独成立，写一个函数，传 2 个参数 user 和 psw，写用例的时候调用登录函数，输入几组 user,psw 参数化登录用例

测试用例传参需要用装饰器@pytest.mark.parametrize，里面写两个参数

- 第一个参数是字符串，多个参数中间用逗号隔开
- 第二个参数是 list,多组数据用元祖类型

```
# test_01.py


# coding:utf-8
import pytest


# ** 作者：上海-悠悠 QQ 交流群：874033608**


# 测试登录数据
test_login_data = [("admin", "111111"),  ("admin", "")]


def login(user, psw):
    '''普通登录函数'''
    print("登录账户：%s"%user)
    print("登录密码：%s"%psw)
    if psw:
        return True
    else:
        return False
```

```
@pytest.mark.parametrize("user, psw", test_login_data)
def test_login(user, psw):
    '''登录用例'''
    result = login(user, psw)
    assert result == True, "失败原因：密码为空"


if __name__ == "__main__":
    pytest.main(["-s", "test_01.py"])
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 2 items

..\..\..\..\..\..\YOYO\marktest\test_01.py 登录账户：admin
登录密码：111111
.登录账户：admin
登录密码：
F
user = 'admin', psw = ''

    @pytest.mark.parametrize("user, psw", [("admin", "111111"),
    def test_01(user, psw):
        result = login(user, psw)
>       assert result == True
E       assert False == True

D:\YOYO\marktest\test_01.py:18: AssertionError
```

```
================================== FAILURES ===================================
_____________________________ test_01[admin-] _____________________________

user = 'admin', psw = ''

    @pytest.mark.parametrize("user, psw", [("admin", "111111"),
    def test_01(user, psw):
        result = login(user, psw)
>       assert result == True
E       assert False == True

D:\YOYO\marktest\test_01.py:18: AssertionError
==================== 1 failed, 1 passed in 0.05 seconds ========
```

从结果可以看出，有 2 个用例，一个测试通过，一个测试失败了，互不影响

request 参数

如果想把登录操作放到前置操作里，也就是用到@pytest.fixture 装饰器，传参就用默认的 request 参数

user = request.param 这一步是接收传入的参数，本案例是传一个参数情况

```
# test_02.py
# coding:utf-8
import pytest


#** 作者：上海-悠悠 QQ 交流群：874033608**


# 测试账号数据
test_user_data = ["admin1", "admin2"]
```

```
@pytest.fixture(scope="module")
def login(request):
    user = request.param
    print("登录账户：%s"%user)
    return user

@pytest.mark.parametrize("login", test_user_data,
indirect=True)
def test_login(login):
    '''登录用例'''
    a = login
    print("测试用例中 login 的返回值:%s" % a)
    assert a != ""

if __name__ == "__main__":
    pytest.main(["-s", "test_02.py"])
```

运行结果：

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 2 items

..\..\..\..\..\..\YOYO\marktest\test_02.py 登录账户：admin1
测试用例中login的返回值:admin1
.登录账户：admin2
测试用例中login的返回值:admin2
.

========================== 2 passed in 0.01 seconds ============
```

添加 indirect=True 参数是为了把 login 当成一个函数去执行，而不是一个参数

request 传 2 个参数

如果用到@pytest.fixture，里面用 2 个参数情况，可以把多个参数用一个字典去存储，这样最终还是只传一个参数

不同的参数再从字典里面取对应 key 值就行，如： user = request.param[“user”]

```
# test_03.py
# coding:utf-8
import pytest


# ** 作者 上海-悠悠 QQ 交流群 874033608**


# 测试账号数据
test_user_data = [{"user": "admin1", "psw": "111111"},
```

```
                {"user": "admin1", "psw": ""}]


@pytest.fixture(scope="module")
def login(request):
    user = request.param["user"]
    psw = request.param["psw"]
    print("登录账户：%s" % user)
    print("登录密码：%s" % psw)
    if psw:
        return True
    else:
        return False


# indirect=True 声明 login 是个函数
@pytest.mark.parametrize("login", test_user_data,
indirect=True)
def test_login(login):
    '''登录用例'''
    a = login
    print("测试用例中 login 的返回值:%s" % a)
    assert a, "失败原因：密码为空"
```

```
if __name__ == "__main__":
    pytest.main(["-s", "test_03.py"])
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: D:\, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 2 items

..\..\..\..\..\..\YOYO\marktest\test_03.py 登录账户：admin1
登录密码：111111
测试用例中login的返回值:True
.登录账户：admin1
登录密码：
测试用例中login的返回值:False
F
login = False

    @pytest.mark.parametrize("login", test_user_data, indirect=T
    def test_login(login):
        '''登录用例'''
        a = login
        print("测试用例中login的返回值:%s" % a)
>       assert a, "失败原因：密码为空"
E       AssertionError: 失败原因：密码为空
E       assert False
```

```
================================== FAILURES ===================
_____________________________ test_login[login1] _____________

login = False

    @pytest.mark.parametrize("login", test_user_data, indirect=T
    def test_login(login):
        '''登录用例'''
        a = login
        print("测试用例中login的返回值:%s" % a)
>       assert a, "失败原因：密码为空"
E       AssertionError: 失败原因：密码为空
E       assert False

D:\YOYO\marktest\test_03.py:25: AssertionError
==================== 1 failed, 1 passed in 0.05 seconds =======
```

如果要用到 login 里面的返回值，def test_login(login)时，传入 login 参数，函数返回值就是 login 了

多个 fixtrue

用例上面是可以同时放多个 fixture 的，也就是多个前置操作，可以支持装饰器叠加，使用 parametrize 装饰器叠加时，用例组合是 2 个参数个数相乘

```
# test_04.py
# ** 作者 上海-悠悠 QQ 交流群 874033608**
# coding:utf-8
import pytest


# 测试账号数据
test_user = ["admin1", "admin2"]
```

```
test_psw = ["11111", "22222"]



@pytest.fixture(scope="module")
def input_user(request):
    user = request.param
    print("登录账户：%s" % user)
    return user



@pytest.fixture(scope="module")
def input_psw(request):
    psw = request.param
    print("登录密码：%s" % psw)
    return psw




@pytest.mark.parametrize("input_user", test_user,
indirect=True)
@pytest.mark.parametrize("input_psw", test_psw, indirect=True)
def test_login(input_user, input_psw):
```

```
    '''登录用例'''
    a = input_user
    b = input_psw
    print("测试数据 a-> %s,  b-> %s" % (a,b))
    assert b


if __name__ == "__main__":
    pytest.main(["-s", "test_04.py"])
```

运行结果

```
<----============================== test session starts ==========
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
rootdir: E:\YOYO\par, inifile:
plugins: metadata-1.7.0, html-1.19.0
collected 4 items

test_04.py 登录账户：admin1
登录密码：11111
测试数据a-> admin1,  b-> 11111
.登录账户：admin2
测试数据a-> admin2,  b-> 11111
.登录密码：22222
测试数据a-> admin2,  b-> 22222
.登录账户：admin1
测试数据a-> admin1,  b-> 22222
.

========================== 4 passed in 0.05 seconds ============
```

如果参数 user 有 2 个数据，参数 psw 有 2 个数据，那么组合起来的案例是两个相乘，也就是组合 2*2 = 4 个用例

1.14-命令行参数

前言

命令行参数是根据命令行选项将不同的值传递给测试函数，比如平常在 cmd 执行”pytest —html=report.html”,这里面的”—html=report.html“就是从命令行传入的参数

对应的参数名称是 html，参数值是 report.html

conftest 配置参数

1.首先需要在 conftest.py 添加命令行选项,命令行传入参数”—cmdopt“，用例如果需要用到从命令行传入的参数，就调用 cmdopt 函数：

```
# content of conftest.py
import pytest



def pytest_addoption(parser):
    parser.addoption(
        "--cmdopt", action="store", default="type1", help="my
option: type1 or type2"
    )


@pytest.fixture
```

```
def cmdopt(request):
    return request.config.getoption("--cmdopt")
```

2.测试用例编写案例

```
# content of test_sample.py
import pytest
def test_answer(cmdopt):
    if cmdopt == "type1":
        print("first")
    elif cmdopt == "type2":
        print("second")
    assert 0  # to see what was printed


if __name__ == "__main__":
    pytest.main(["-s", "test_case1.py"])
```

cmd 打开，输入指令启动，也可以在 pycharm 里面右键执行上面代码

```
> pytest -s test_sample.py
```

运行结果：

```
test_sample.py second
F

================================== FAILURES ===================================
_____________________________________ test_answer _____________________________

cmdopt = 'type2'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>       assert 0  # to see what was printed
E       assert 0

test_case1.py:8: AssertionError
========================== 1 failed in 0.05 seconds ===========
```

带参数启动

1.如果不带参数执行，那么传默认的 default=" type1" ，接下来在命令行带上参数去执行

```
> pytest -s test_sample.py --cmdopt=type2
```

```
test_sample.py second
F

================================== FAILURES ===================================
_____________________________________ test_answer _____________________________

cmdopt = 'type2'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>       assert 0  # to see what was printed
E       assert 0

test_case1.py:8: AssertionError
========================== 1 failed in 0.05 seconds ===========
```

2.命令行传参数有两种写法，还有一种分成 2 个参数也可以的,参数和名称用空格隔开

```
> pytest -s test_case1.py --cmdopt type2
```

1.15-配置文件 pytest.ini

前言

pytest 配置文件可以改变 pytest 的运行方式，它是一个固定的文件 pytest.ini 文件，读取配置信息，按指定的方式去运行。

ini 配置文件

pytest 里面有些文件是非 test 文件

- pytest.ini pytest 的主配置文件，可以改变 pytest 的默认行为
- conftest.py 测试用例的一些 fixture 配置

- __init__.py 识别该文件夹为 python 的 package 包
- tox.ini 与 pytest.ini 类似，用 tox 工具时候才有用
- setup.cfg 也是 ini 格式文件，影响 setup.py 的行为

ini 文件基本格式

```
# 保存为 pytest.ini 文件


[pytest]


addopts = -rsxX

xfail_strict = true
```

用 pytest —help 指令可以查看 pytest.ini 的设置选项

```
[pytest] ini-options in the first pytest.ini|tox.ini|setup.cfg file

  markers (linelist)       markers for test functions
  empty_parameter_set_mark (string) default marker for empty parame
  norecursedirs (args)     directory patterns to avoid for recursic
  testpaths (args)         directories to search for tests when no

  console_output_style (string) console output: classic or with add

  usefixtures (args)       list of default fixtures to be used with

  python_files (args)      glob-style file patterns for Python test

  python_classes (args)    prefixes or glob names for Python test c

  python_functions (args)  prefixes or glob names for Python test f

  xfail_strict (bool)      default for the strict parameter of
  addopts (args)           extra command line options
  minversion (string)      minimally required pytest version
```

--rsxX 表示 pytest 报告所有测试用例被跳过、预计失败、预计失败但实际被通过的原因

mark 标记

如下案例，使用了 2 个标签：webtest 和 hello,使用 mark 标记功能对于以后分类测试非常有用处

```
# content of test_mark.py
import pytest


@pytest.mark.webtest
def test_send_http():
    print("mark web test")


def test_something_quick():
    pass


def test_another():
    pass


@pytest.mark.hello
class TestClass:
    def test_01(self):
        print("hello :")
```

```
    def test_02(self):
        print("hello world!")


if __name__ == "__main__":
    pytest.main(["-v", "test_mark.py", "-m=hello"])
```

运行结果

```
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
cachedir: .pytest_cache
metadata: {'Python': '3.6.0', 'Platform': 'Windows-7-6.1.7601-SP
rootdir: D:\YOYO, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collecting ... collected 5 items / 3 deselected

test_mark.py::TestClass::test_01 PASSED
test_mark.py::TestClass::test_02 PASSED

=================== 2 passed, 3 deselected in 0.11 seconds =====
```

有时候标签多了，不容易记住，为了方便后续执行指令的时候能准确使用 mark 的标签，可以写入到 pytest.ini 文件

```
# pytest.ini
[pytest]

markers =
  webtest:  Run the webtest case
  hello: Run the hello case
```

标记好之后，可以使用 pytest —markers 查看到

```
> pytest --markers
```

D:\YOYO>pytest --markers

@pytest.mark.webtest: Run the webtest case

@pytest.mark.hello: Run the hello case

@pytest.mark.skip(reason=None): skip the given test function with an optional re

ason. Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition)

results in a True value. Evaluation happens within the module global context. E

xample: skipif('sys.platform == "win32"') skips the test if we are on the win32

platform. see http://pytest.org/latest/skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False):

mark the test function as an expected failure if eval(condition) has a True val

ue. Optionally specify a reason for better reporting and run=False if you don't

even want to execute the test function. If only specific exception(s) are expect

ed, you can list them in raises, and if the test fails in other ways, it will be

reported as a true failure. See http://pytest.org/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple tim

es passing in different arguments in turn. argvalues generally needs to be a lis

t of values if argnames specifies only one name or a list of tuples of values if

argnames specifies multiple names. Example:

@parametrize('arg1', [1,2]) would l

ead to two calls of the decorated test function, one with arg1=1 and another wit

h arg1=2.see http://pytest.org/latest/parametrize.html for more info and example

s.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing

all of the specified fixtures. see

http://pytest.org/latest/fixture.html#usefix

tures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin

machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin m

achinery will try to call it last/as late as possible.

最上面两个就是刚才写入到 pytest.ini 的配置了

禁用 xpass

设置 xfail_strict = true 可以让那些标记为@pytest.mark.xfail 但实际通过的测试用例被报告为失败

```
# content of test_xpass.py
import pytest
# ** 作者 上海-悠悠 QQ 交流群 874033608**
def test_hello():
    print("hello world!")
    assert 1


@pytest.mark.xfail()
def test_yoyo1():
    a = "hello"
    b = "hello world"
    assert a == b


@pytest.mark.xfail()
def test_yoyo2():
    a = "hello"
    b = "hello world"
    assert a != b
```

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

什么叫标记为@pytest.mark.xfail 但实际通过，这个比较绕脑，看以下案例

测试结果

collecting ... collected 3 items

test_xpass.py::test_hello PASSED [33%]

test_xpass.py::test_yoyo1 xfail [66%]

test_xpass.py::test_yoyo2 XPASS [100%]

========== 1 passed, 1 xfailed, 1 xpassed in 0.27 seconds ==========

test_yoyo1 和 test_yoyo2 这 2 个用例一个是 a == b 一个是 a != b,两个都标记失败了，我们希望两个用例不用执行全部显示 xfail。实际上最后一个却显示 xpass.为了让两个都显示 xfail，那就加个配置 xfail_strict = true

```
# pytest.ini
[pytest]

markers =
  webtest:  Run the webtest case
```

```
  hello: Run the hello case


xfail_strict = true
```

再次运行，结果就变成

```
collecting ... collected 3 items

test_xpass.py::test_hello PASSED        [ 33%]
test_xpass.py::test_yoyo1 xfail         [ 66%]
test_xpass.py::test_yoyo2 FAILED        [100%]

================================== FAILURES ===================
_____________________________ test_yoyo2 ____________________
[XPASS(strict)]
================ 1 failed, 1 passed, 1 xfailed in 0.05 seconds =
```

这样标记为 xpass 的就被强制性变成 failed 的结果

配置文件如何放

一般一个工程下方一个 pytest.ini 文件就可以了，放到顶层文件夹下

addopts

addopts 参数可以更改默认命令行选项，这个当我们在 cmd 输入指令去执行用例的时候，会用到，比如我想测试完生成报告，指令比较长

```
> pytest -v --rerun 1 --html=report.html
--self-contained-html
```

每次输入这么多，不太好记住，于是可以加到 pytest.ini 里

```
# pytest.ini
[pytest]

markers =
  webtest:  Run the webtest case
  hello: Run the hello case

 xfail_strict = true

 addopts = -v --rerun 1 --html=report.html --self-contained-html
```

这样我下次打开 cmd，直接输入 pytest，它就能默认带上这些参数了

(备注：--html=report.html 是生成 html 报告，看第二章 pytest-html)

1.16-doctest 框架

前言

doctest 从字面意思上看，那就是文档测试。doctest 是 python 里面自带的一个模块，它实际上是单元测试的一种。

官方解释：doctest 模块会搜索那些看起来像交互式会话的 Python 代码片段，然后尝试执行并验证结果

doctest 测试用例可以放在两个地方

- 函数或者方法下的注释里面
- 模块的开头

案例

先看第一个案例，将需要测试的片段,标准格式，需要运行的代码前面加>>> ,相当于进入 cmd 这种交互环境执行，期望的结果前面不需要加>>>

```
>>> multiply(4, 3)
    12
>>> multiply('a', 3)
    'aaa'
```

放到 multiply 函数的注释里

```
def multiply(a, b):
    """
    fuction: 两个数相乘
    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
    """
```

```
    return a * b
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

运行结果

Trying:

multiply(4, 3)

Expecting:

12

ok

Trying:

multiply('a', 3)

Expecting:

'aaa'

ok

1 items had no tests:

__main__

1 items passed all tests:

2 tests in __main__.multiply

2 tests in 2 items.

2 passed and 0 failed.

Test passed.

从运行的结果可以看出，虽然函数下方的注释里面有其它内容“fuction: 两个数相乘”，但不会去执行，只识别“>>>”这种符号。2 个测试用例都是通过的，实际的结果与期望的结果一致。

失败案例

doctest 的内容放到.py 模块的开头也是可以识别到的

```
# 保存为 xxx.py
'''
fuction: 两个数相乘
>>> multiply(4, 8)
12
>>> multiply('a', 5)
'aaa'
'''


def multiply(a, b):
```

```
    """
    fuction: 两个数相乘
    """
    return a * b
if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

运行结果 2 个都失败

****

1 items had failures:

2 of 2 in __main__

2 tests in 2 items.

0 passed and 2 failed.

Test Failed 2 failures.

verbose 参数，设置为 True 则在执行测试的时候会输出详细信息

cmd 执行

以上案例是在编辑器直接运行的，如果在 cmd 里面，也可以用指令去执行

```
> python -m doctest -v xxx.py
```

- m 参数指定运行方式 doctest
- -v 参数是 verbose，带上-v 参数相当于 verbose=True

```
D:\test1122\a>python -m doctest -v xxx.py
Trying:
    multiply(4, 8)
Expecting:
    12
**********************************************************************
File "D:\test1122\a\xxx.py", line 3, in xxx
Failed example:
    multiply(4, 8)
Expected:
    12
Got:
    32
Trying:
    multiply('a', 5)
Expecting:
    'aaa'
**********************************************************************
File "D:\test1122\a\xxx.py", line 5, in xxx
Failed example:
    multiply('a', 5)
Expected:
    'aaa'
Got:
    'aaaaa'
1 items had no tests:
    xxx.multiply
**********************************************************************
1 items had failures:
   2 of   2 in xxx
2 tests in 2 items.
0 passed and 2 failed.
***Test Failed*** 2 failures.

D:\test1122\a>
```

pytest 运行

pytest 框架是可以兼容 doctest 用例，执行的时候加个参数 —doctest-modules ,这样它就能自动搜索到 doctest 的用例

```
> pytest -v --doctest-modules xxx.py
```

```
D:\test1122\a>pytest -v --doctest-modules  xxx.py
============================= test session starts =============================
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0.6.0 -- d:\soft\
python3.6\python.exe
cachedir: .pytest_cache
metadata: {'Python': '3.6.0', 'Platform': 'Windows-7-6.1.7601-SP1', 'Packages':
{'pytest': '3.6.3', 'py': '1.5.4', 'pluggy': '0.6.0'}, 'Plugins': {'metadata': '
1.7.0', 'html': '1.19.0', 'allure-adaptor': '1.7.10'}, 'JAVA_HOME': 'D:\\soft\\j
dk18\\jdk18v'}
rootdir: D:\test1122\a, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 1 item

xxx.py::xxx.multiply FAILED                                              [100%]

================================== FAILURES ===================================
_____________________________ [doctest] xxx.multiply _____________________________
004
005     fuction: 两个数相乘
006     >>> multiply(4, 3)
007     12
008     >>> multiply('a', 5)
Expected:
    'aaa'
Got:
    'aaaaa'

D:\test1122\a\xxx.py:8: DocTestFailure
========================== 1 failed in 0.16 seconds ===========================
```

如下是函数下的文档

fuction: 两个数相乘

>>> multiply(4, 3)

12

>>> multiply('a', 5)

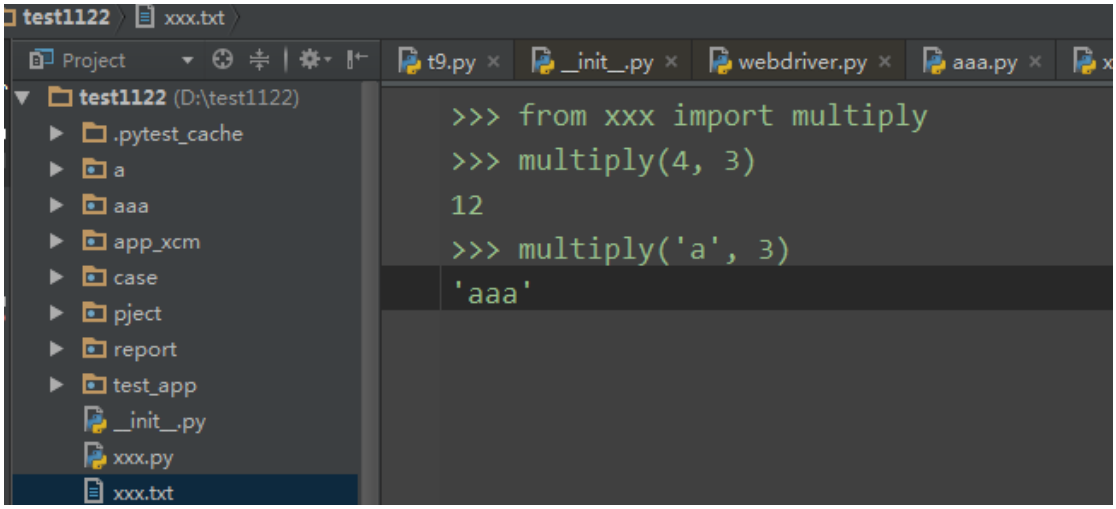
'aaa'

运行结果

```
D:\test1122\a>pytest -v --doctest-modules  xxx.py
============================= test session starts ==============
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0
python3.6\python.exe
cachedir: .pytest_cache
metadata: {'Python': '3.6.0', 'Platform': 'Windows-7-6.1.7601-SP
{'pytest': '3.6.3', 'py': '1.5.4', 'pluggy': '0.6.0'}, 'Plugins'
1.7.0', 'html': '1.19.0', 'allure-adaptor': '1.7.10'}, 'JAVA_HOM
dk18\\jdk18v'}
rootdir: D:\test1122\a, inifile:
plugins: metadata-1.7.0, html-1.19.0, allure-adaptor-1.7.10
collected 1 item

xxx.py::xxx.multiply FAILED

================================== FAILURES ====================
_____________________________ [doctest] xxx.multiply _____________
004
005     fuction: 两个数相乘
006     >>> multiply(4, 3)
007     12
008     >>> multiply('a', 5)
Expected:
    'aaa'
Got:
    'aaaaa'

D:\test1122\a\xxx.py:8: DocTestFailure
========================== 1 failed in 0.16 seconds ============
```

结果可以看出，文档里面的每一行都被执行了，当遇到测试不通过的用例时，就不会继续往下执行了

doctest 独立文件



doctest 内容也可以和代码抽离开，单独用一个.txt 文件保存

在当前 xxx.py 同一目录新建一个 xxx.txt 文件，写入测试的文档，要先导入该功能，导入代码前面也要加>>>

```
>>> from xxx import multiply
>>> multiply(4, 3)
12
>>> multiply('a', 3)
'aaa'
```

cmd 执行“python -m doctest -v xxx.txt”测试结果

```
D:\test1122>python -m doctest -v  xxx.txt
Trying:
    from xxx import multiply
Expecting nothing
ok
Trying:
    multiply(4, 3)
Expecting:
    12
ok
Trying:
    multiply('a', 3)
Expecting:
    'aaa'
ok
1 items passed all tests:
   3 tests in xxx.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```


第 2 章 HTML 报告生成

生成 html 报告，这里介绍了 2 个框架

1.pytest-HTML

2.allure2

2.1-pytest-html 生成 html 报告

前言

pytest-HTML 是一个插件，pytest 用于生成测试结果的 HTML 报告。兼容 Python 2.7,3.6

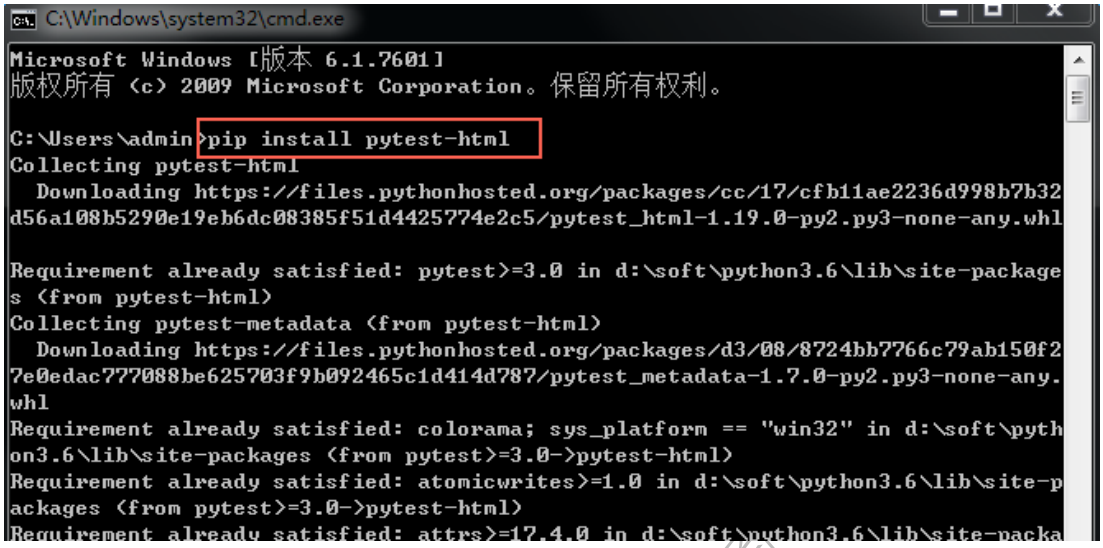
pytest-html

1.github 上源码地址

【https://github.com/pytest-dev/pytest-html】

2.pip 安装

```
> pip install pytest-html
```

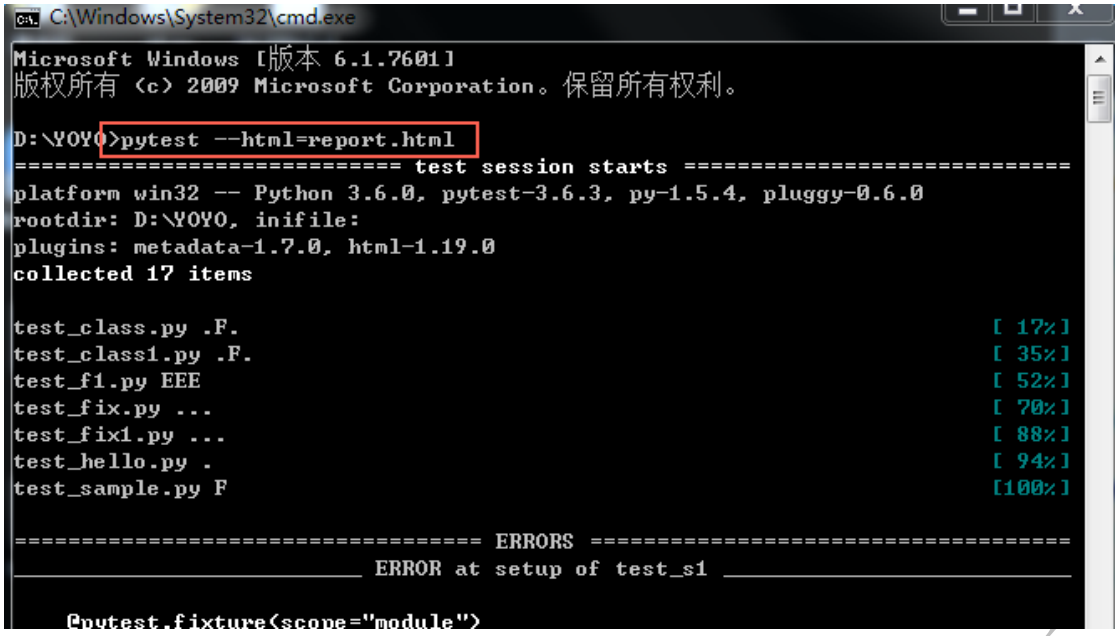


3.执行方法

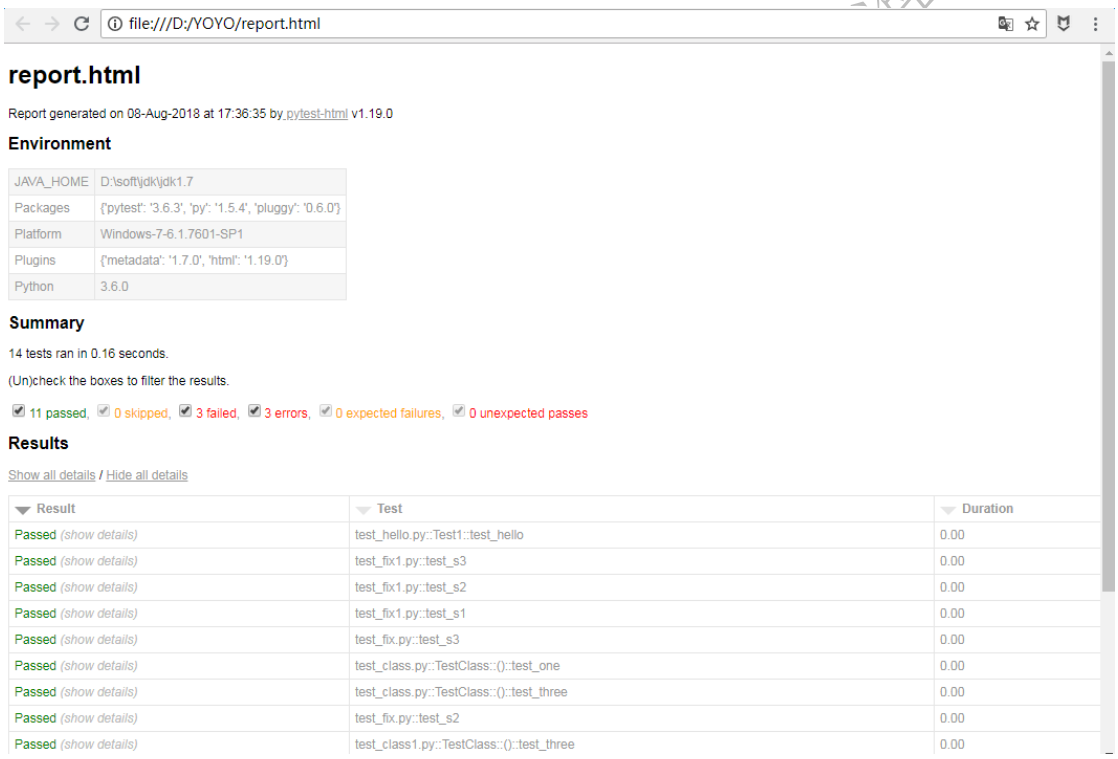
```
>  pytest --html=report.html
```

html 报告

1.打开 cmd，cd 到需要执行 pytest 用例的目录，执行指令：pytest —html=report.html



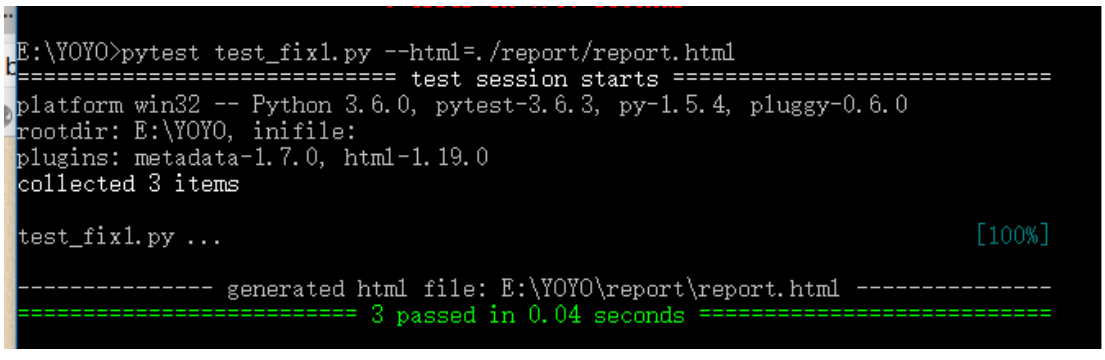
2.执行完之后，在当前目录会生成一个 report.html 的报告文件，显示效果如下



指定报告路径

1.直接执行”pytest —html=report.html”生成的报告会在当前脚本的同一路径，如果想指定报告的存放位置，放到当前脚本的同一目录下的 report 文件夹里

2.如果想指定执行某个.py 文件用例或者某个文件夹里面的所有用例，需加个参数。具体规则参考【pytest 文档 2-用例运行规则】



报告独立显示

上面方法生成的报告，css 是独立的，分享报告的时候样式会丢失，为了更好的分享发邮件展示报告，可以把 css 样式合并到 html 里

```
> pytest --html=report.html --self-contained-html
```

显示选项

默认情况下，“结果”表中的所有行都将被展开，但具测试通过的行除外 Passed。

可以使用查询参数自定义此行为：?collapsed=Passed,XFailed,Skipped。

更多功能

1.更多功能查看官方文档【https://github.com/pytest-dev/pytest-html】

2.2-html 报告报错截图+失败重跑

前言

做 web 自动化的小伙伴应该都希望在 html 报告中展示失败后的截图，提升报告的档次，pytest-html 也可以生成带截图的报告。

conftest.py

1.失败截图可以写到 conftest.py 文件里，这样用例运行时，只要检测到用例实例，就调用截图的方法，并且把截图存到 html 报告上

2.用例部分如下：

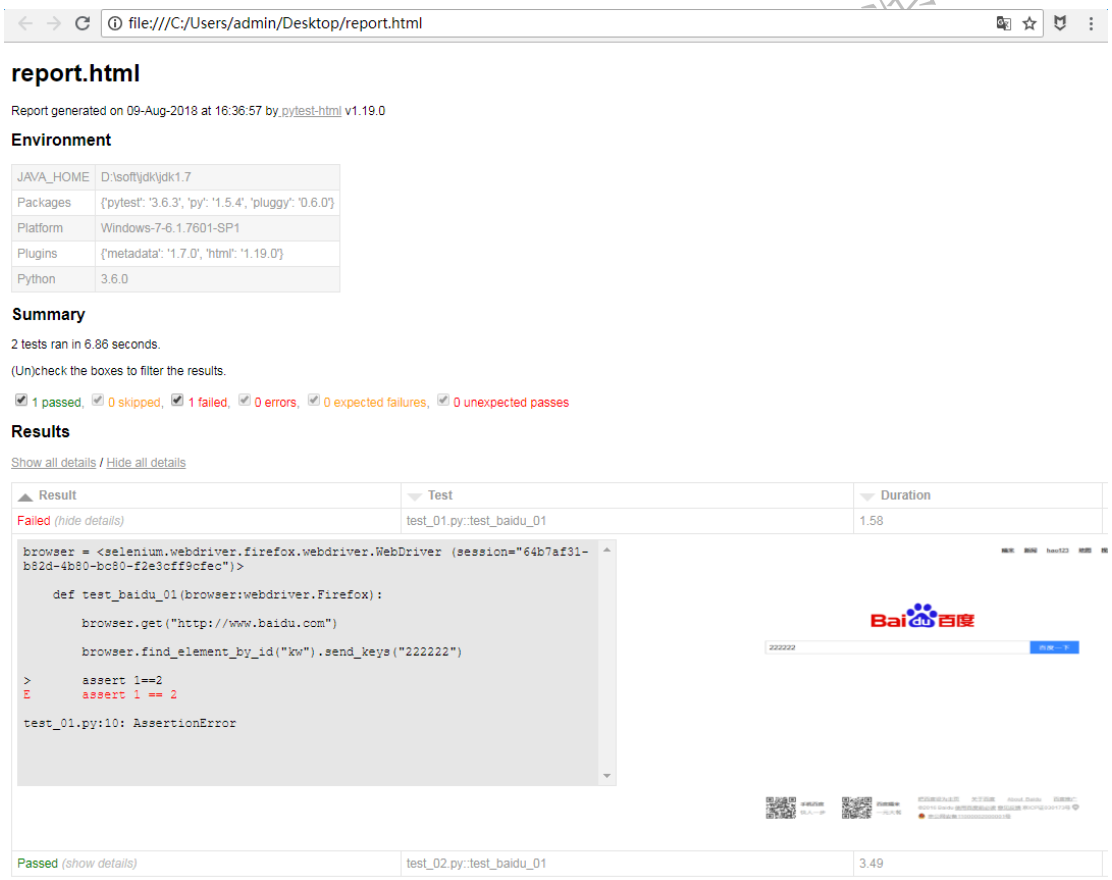
报告展示

cmd 打开，cd 到用例的目录，执行指令

```
> pytest --html=report.html --self-contained-html
```


C:\Windows\System32\cmd.exe
D:\YOYO\case>pytest --html=report.html
test session starts
platform win32 -- Python 3.6.0, pytest-3.6.3, py-1.5.4, pluggy-0.6.0
rootdir: D:\YOYO\case, inifile:
plugins: metadata-1.7.0, html-1.19.0
collected 2 items
test_01.py F [50%]
test_02.py . [100%]
FAILURES
test_yoyo_01
browser = <selenium.webdriver.firefox.webdriver.WebDriver (session="3d3e91a8-f625-428e-b69a-dab9d3ea290f")>
def test_yoyo_01(browser:webdriver.Firefox):
browser.get("https://www.cnblogs.com/yoyoketang/")
time.sleep(2)
t = browser.title
> assert t == "上海-悠悠"
E AssertionError: assert '上海-悠悠 - 博客园' == '上海-悠悠'

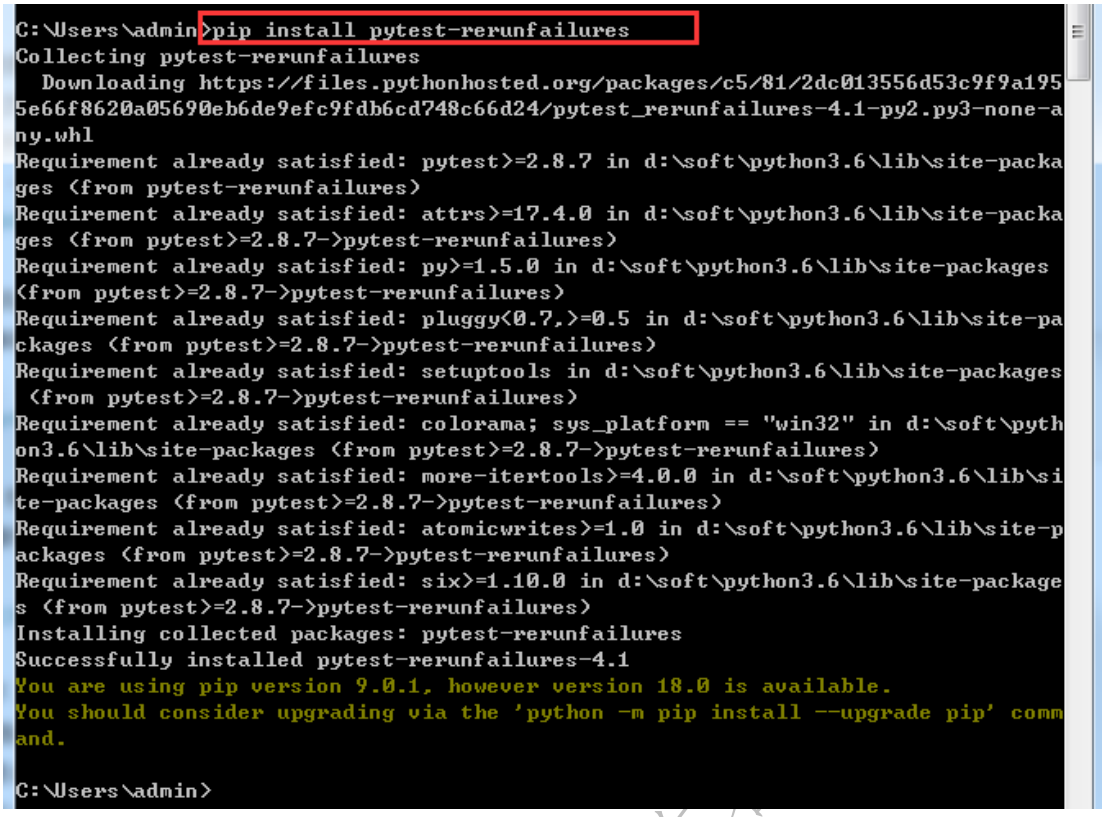
生成报告如下


file:///C:/Users/admin/Desktop/report.html
report.html
Report generated on 09-Aug-2018 at 16:36:57 by pytest-html v1.19.0
Environment
JAVA_HOME D:\soft\jdk\jdk1.7
Packages {'pytest': '3.6.3', 'py': '1.5.4', 'pluggy': '0.6.0'}
Platform Windows-7-6.1.7601-SP1
Plugins {'metadata': '1.7.0', 'html': '1.19.0'}
Python 3.6.0
Summary
2 tests ran in 6.86 seconds.
(Un)check the boxes to filter the results.
1 passed, 0 skipped, 1 failed, 0 errors, 0 expected failures, 0 unexpected passes
Results
Show all details / Hide all details
Result Test Duration
Failed (hide details) test_01.py::test_baidu_01 1.58
browser = <selenium.webdriver.firefox.webdriver.WebDriver (session="64b7af31-b82d-4b80-bc80-f2e3cff9cfec")>
def test_baidu_01(browser:webdriver.Firefox):
browser.get("http://www.baidu.com")
browser.find_element_by_id("kw").send_keys("222222")
> assert 1==2
E assert 1 == 2
test_01.py:10: AssertionError
Passed (show details) test_02.py::test_baidu_01 3.49

失败重试

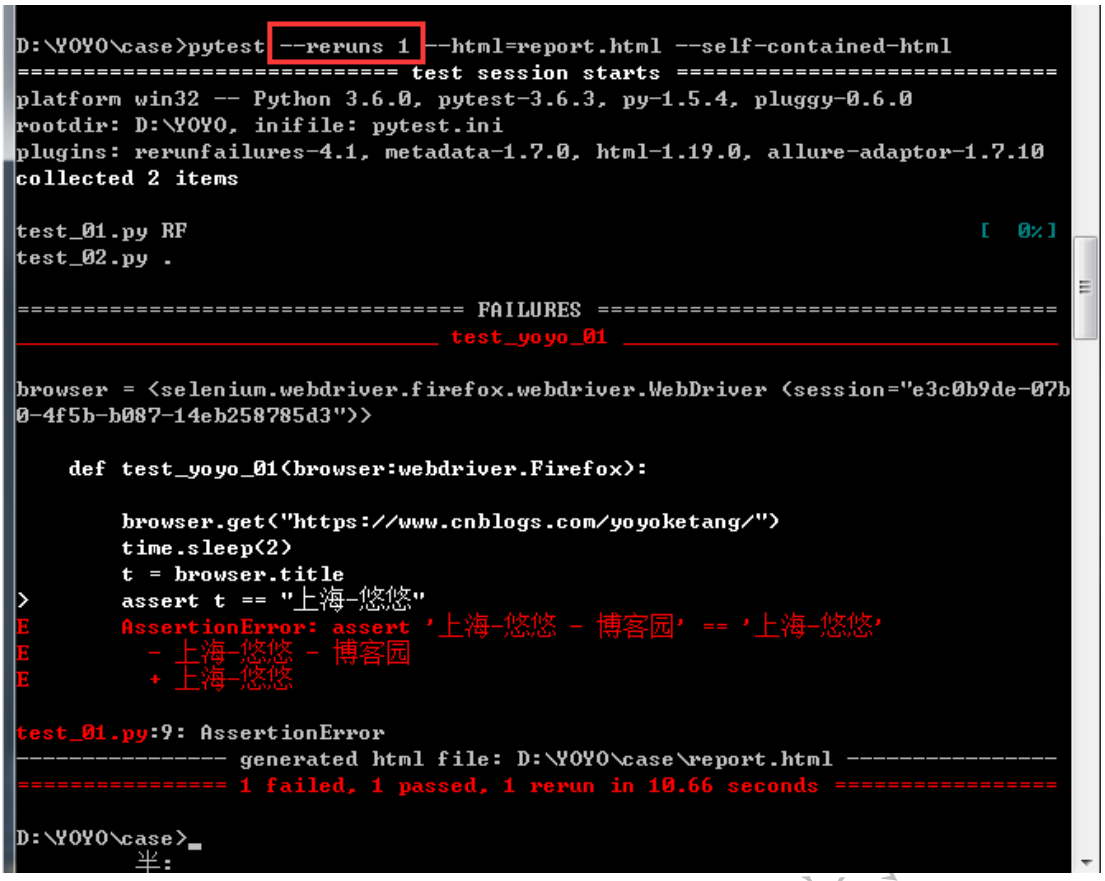
失败重跑需要依赖 pytest-rerunfailures 插件，使用 pip 安装就行

```
> pip install pytest-rerunfailures
```



用例失败再重跑 1 次,命令行加个参数--reruns 就行了

```
> py.test --reruns 1 --html=report.html --self-contained-html
```



关于 reruns 参数的 2 个用法

--reruns=RERUNS RERUNS 参数是失败重跑的次数，默认为 0

--reruns-delay=RERUNS_DELAY RERUNS_DELAY 是失败后间隔多少 s 重新执行，时间单位是 s

2.3-allure2 生成 html 报告(史上最详细)

前言

allure 是一个 report 框架,支持 java 的 Junit/testng 等框架,当然也可以支持 python 的 pytest 框架，也可以集成到 Jenkins 上展示高大上的报告界面。

环境准备

- 1.python3.6
- 2.windows 环境
- 3.pycharm
- 4.pytest-allure-adaptor
- 5.allure2.7.0
- 6.java1.8

pytest-allure-adaptor 下载

pip 安装 pytest-allure-adaptor,github 地址

https://github.com/allure-framework/allure-pytest

```
> pip3 install pytest-allure-adaptor
```

生成 xml 报告

```
> pytest -s -q --alluredir report
```

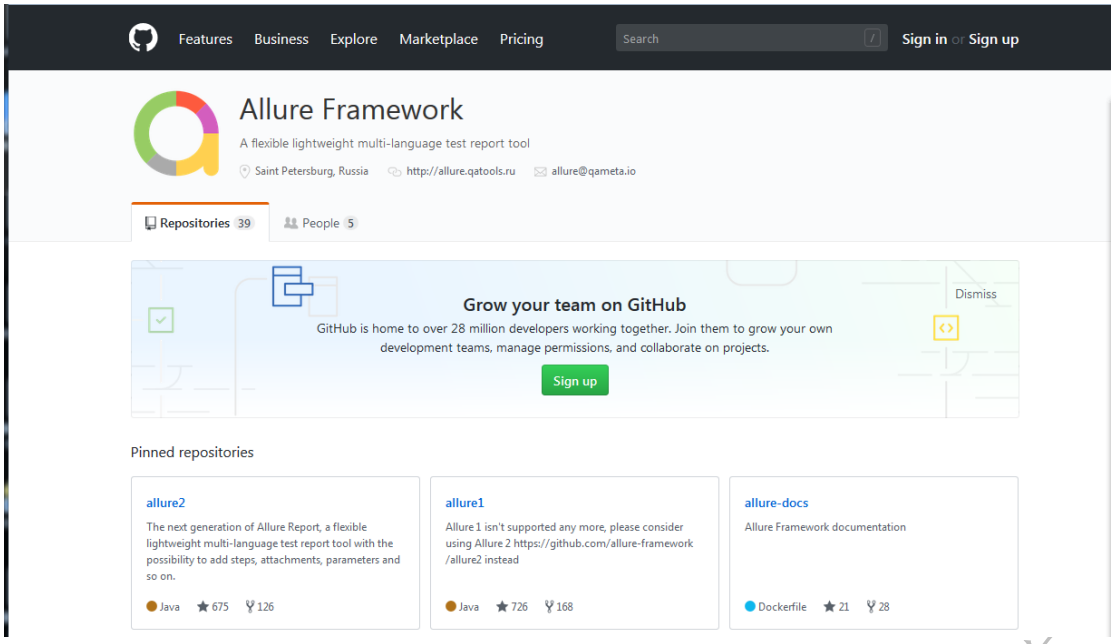
如果不指定路径，默认在当前目录下新建一个 report 目录，当然也可以指定路径

```
> pytest -s -q --alluredir 指定 report 路径
```

执行完之后打开 report 文件夹,会自动生成 xml 格式的报告

安装 Command Tool

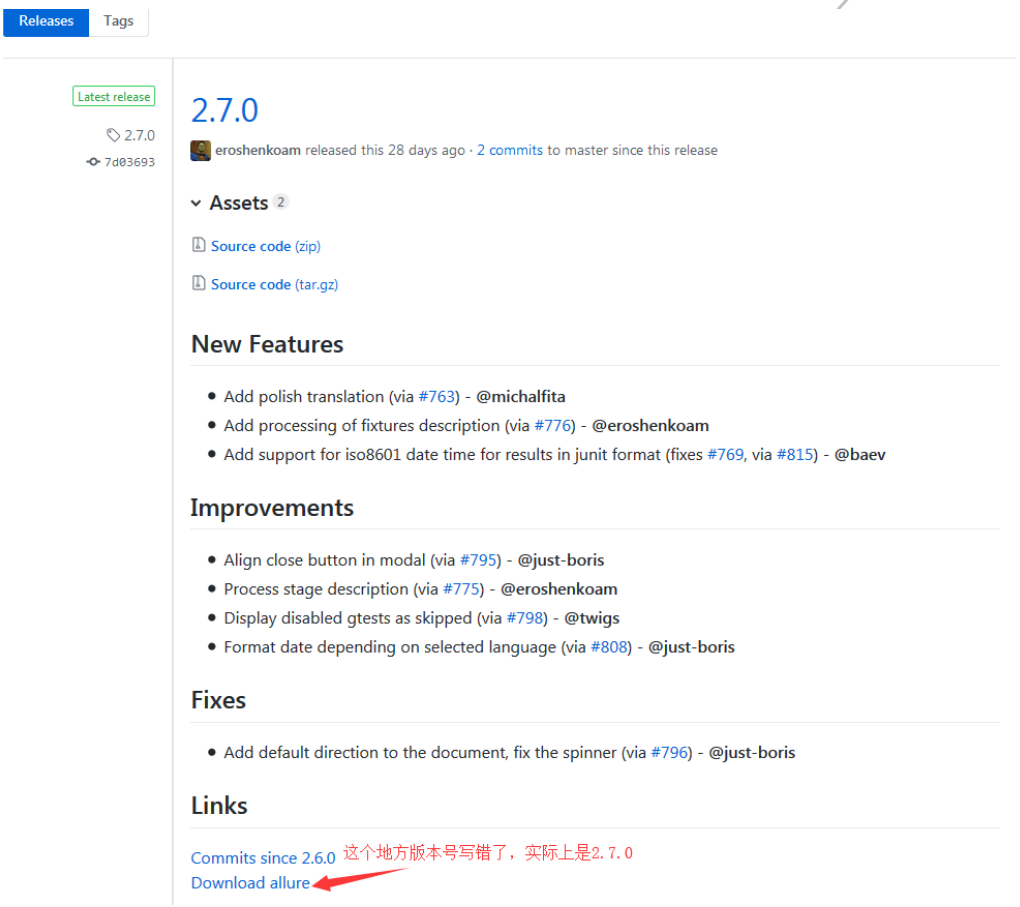
allure 的版本目前有 2 个，从 github 上看，allure1 不再被支持，请考虑使用 allure2 https://github.com/allure-framework/allure2 替代



[allure-commandline releases 版本](https://github.com/allure-framework/allure2/releases)

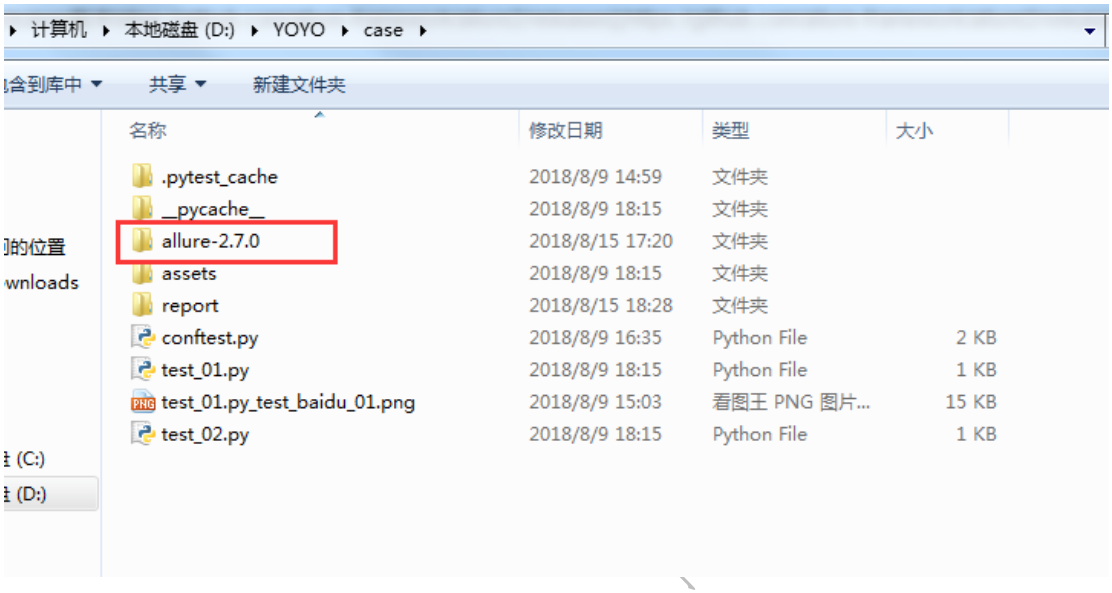
https://github.com/allure-framework/allure2/releases

下载最新的 Download allure2.7.0 版本

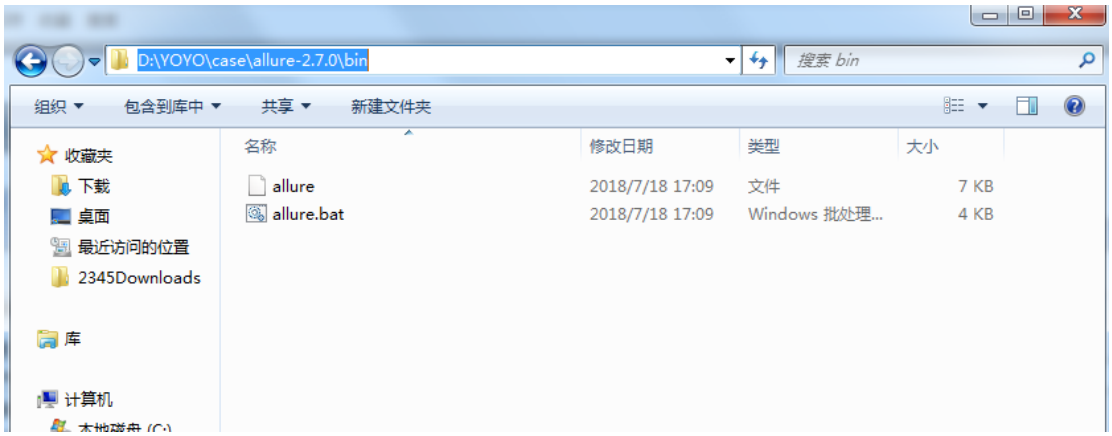


[下载 Download allure2.7.0 地址：
https://dl.bintray.com/qameta/generic/io/qameta/allure/allure/2.7.0/allure-2.7.0.zip]
(https://dl.bintray.com/qameta/generic/io/qameta/allure/allure/2.7.0/allure-2.7.0.zip)

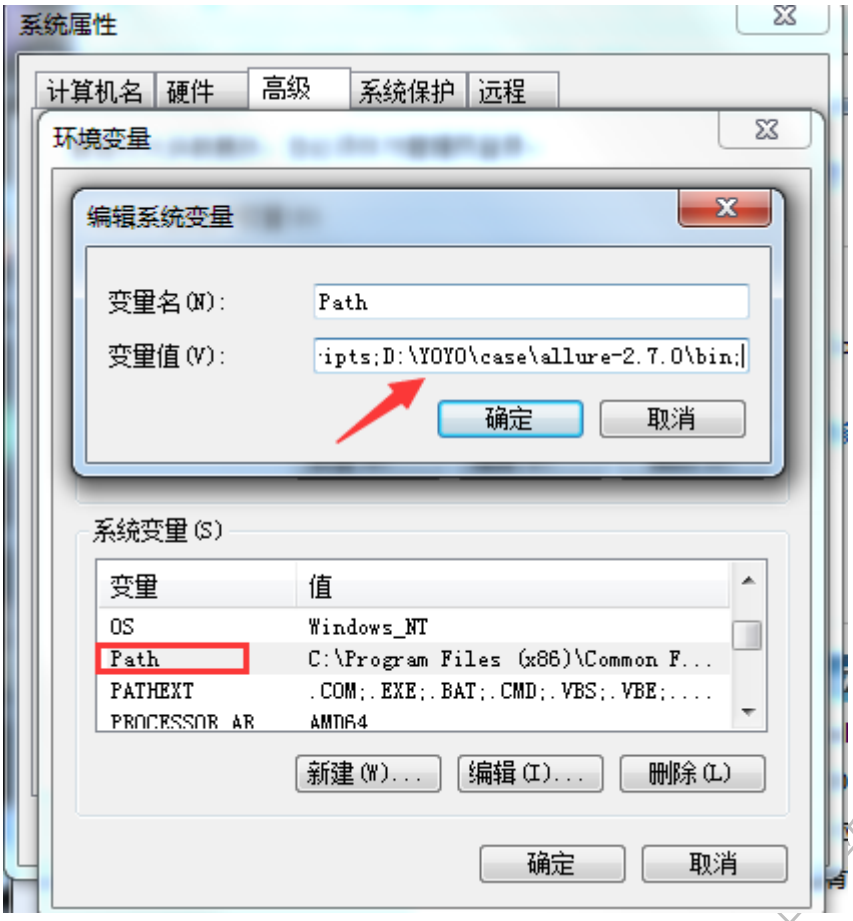
下载好之后，解压到运行 pytest 的目录下



打开\allure-2.7.0\bin 文件夹，会看到 allure.bat 文件，讲此路径设置为系统环境变量 path 下，这样 cmd 任意目录都能执行了



比如我的路径：D:\YOYO\case\allure-2.7.0\bin



运行 allure2

前面 pytest -s -q —alluredir 这一步已经生产了 xml 格式的报告，放到了 report 目录下，接着执行以下命令格式

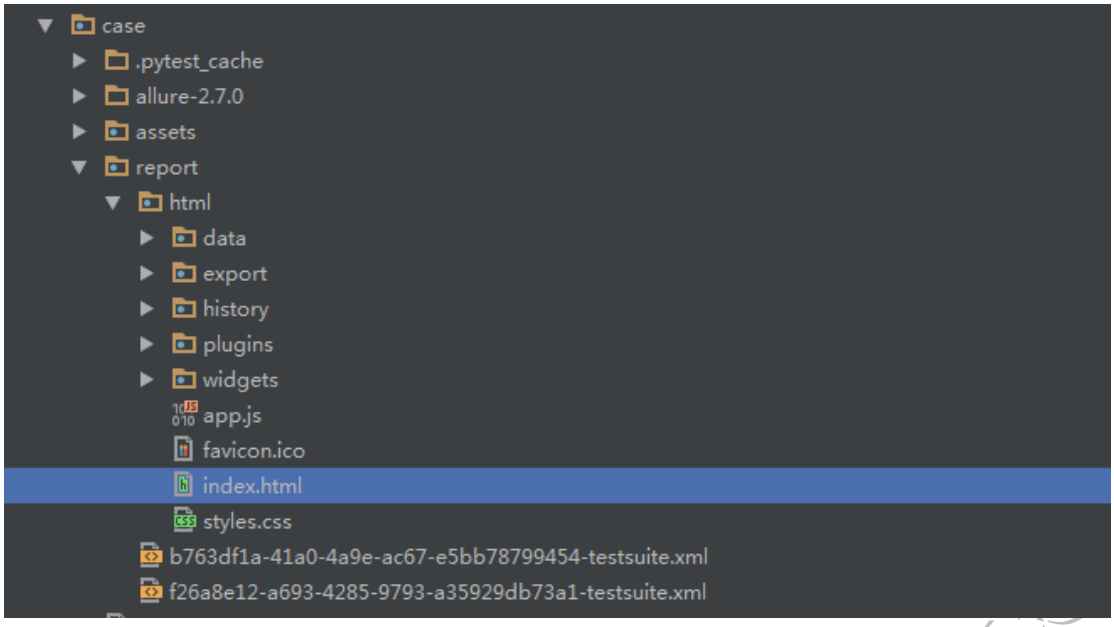
```
> allure generate directory-with-results/ -o
directory-with-report
```

directory-with-results 是 alluredir 生成的 xml 目录，directory-with-report 是最终生成 html 的目录

allure.bat 已经加到环境变量了，所以可以用相对路径去生成 html 报告

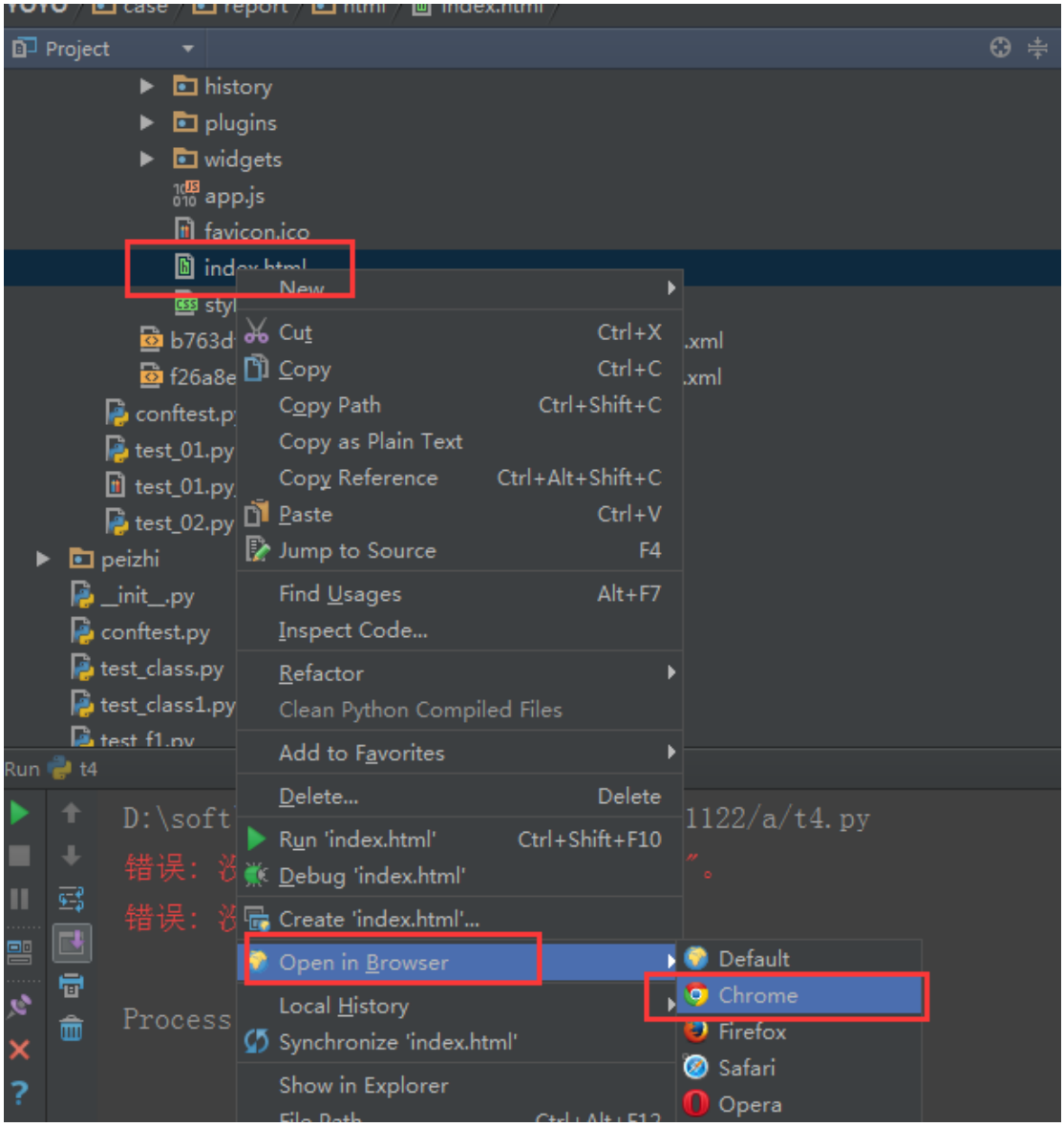
```
> allure generate report/ -o report/html
```

执行完之后目录结构如下：

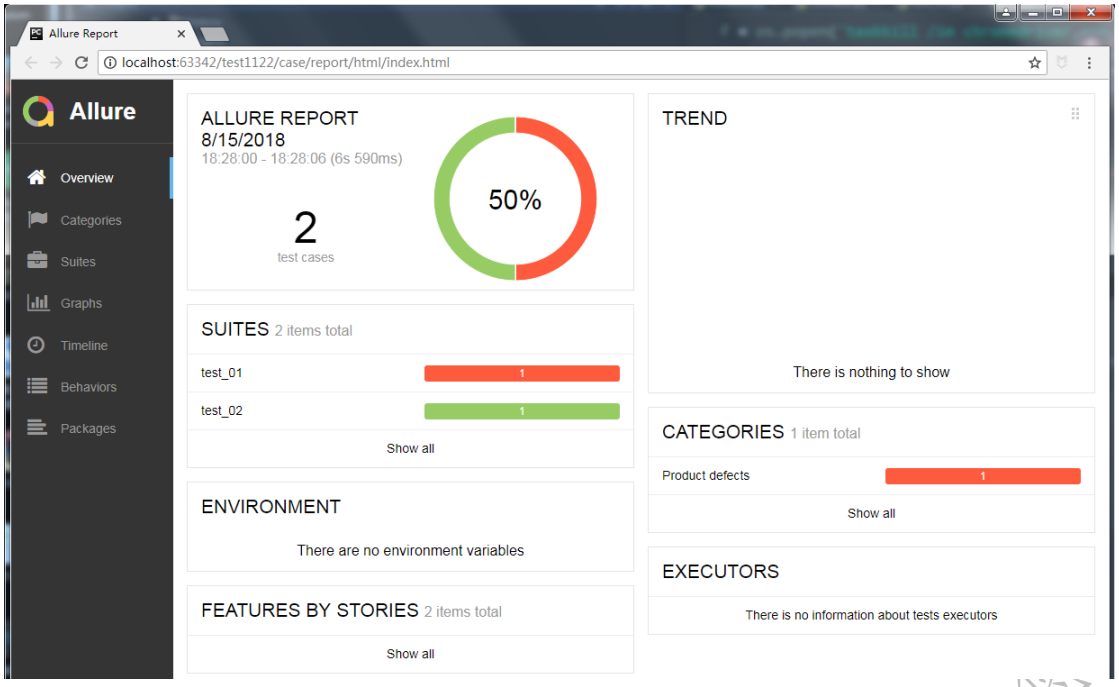


打开报告

直接找到 report/html 打开 index.html 会显示一个空的报告，这里用 pycharm 去打开


Project
history
plugins
widgets
app.js
favicon.ico
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path Ctrl+Shift+C
Copy as Plain Text
Copy Reference Ctrl+Alt+Shift+C
Paste Ctrl+V
Jump to Source F4
Find Usages Alt+F7
Inspect Code...
Refactor
Clean Python Compiled Files
Add to Favorites
Delete... Delete
Run 'index.html' Ctrl+Shift+F10
Debug 'index.html'
Create 'index.html'...
Open in Browser
Local History
Synchronize 'index.html'
Show in Explorer
Default
Chrome
Firefox
Safari
Opera
conftest.py
test_01.py
test_02.py
peizhi
__init__.py
conftest.py
test_class.py
test_class1.py
Run t4
Process

打开后报告展示



依赖 java 环境

之前用的 jdk1.7 版本，执行 allure 时候报错：Unsupported major.minor VERSION 51.0

由于 allure2 是 java 开发的，需要依赖 java 环境，解决办法：jdk 版本用 1.8 就可以了

第 3 章 selenium+pytest 项目案例

前面两章的学习对 pytest 框架用了初步的了解，接下来把 pytest 框架和 selenium 结合起来，运用到项目中。

之前学过 unittest 框架的同学应该知道，unittest 框架用个痛点，用例与用例之间的独立的，不能跨脚本进行数据共享。比如 test_01.py 的 driver，不能共享到 test_02.py 上。这样会导致一个问题，每次运行一个.py 的脚本用例，会重复打开浏览器，浪费比较多的时间。

pytest 框架就能很好的解决这个问题，只需在 conftest.py 设置一个前置的 fixture 功能，设置为 session 级别，这样所有的用例之间都能共享 driver 了。

3.1-本地项目环境搭建

项目环境，selenium 实战章节在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.2-二次封装 selenium 基本操作

项目环境，selenium 实战章节在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.3-登陆案例

项目环境，selenium 实战章节在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.4-参数化 parametrize

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.5-driver 全局调用(session)

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.6-drive 在不同 fixture 之间传递调用

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.7-登陆作为用例前准备

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.8-mark 功能使用

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.9-skipif 失败时候跳过(xfail)

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.10-一套代码 firefox 与 chrome 切换

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

3.11-多线程跑 firefox 和 chrome 并行执行

项目环境，selenium 实战章节需在百度阅读上在线看

购买地址：https://yuedu.baidu.com/ebook/902224ab27fff705cc1755270722192e4536582b

作者其它书籍

《selenium webdriver 基于 Python 源码案例》

本书是非常适合小白入门的，是接地气，符合人类阅读习惯的，书中大量的实际案例分析，图文详解（selenium python 读书 QQ 群 372471871，凭订单号加入可获取一份对应 PDF 文档）

小编是个实在的人，没太多的理论道理，简单粗暴方式直接上源码， 前面三章学完可以搭建简单框架输出报告了，第四章开始参数化，二次封装，后面涉及到 page object 设计模式，加入 logging 日志，多线程等深入学习内容

购买地址：

https://yuedu.baidu.com/ebook/0f6a093b7dd184254b35eefdc8d376eeaeaa17e3

《Python 接口自动化测试》

本书是非常适合小白入门接口自动化的（购买此书（活动赠送的无 PDF）联系作者领取一份对应 PDF 版 QQ 交流群：226296743）。本书涉及内容：fiddler 抓包、http 协议详解、json、python+requests 写接口请求、unittest 单元测试框架、python 爬虫利器 beautifulsoup 等内容、jenkins 持续集成环境搭建。新增 excel+ddt 接口自动化数据驱动

购买地址：
https://yuedu.baidu.com/ebook/585ab168302b3169a45177232f60ddccda38e695

《Appium 自动化入门级（图文教程）-python》

appium+python 环境搭建初级入门教程，本教程只是初级入门，从环境搭建到一个简单的 demo。前面三章内容已整理出 pdf 文件，可以点最后一章节直接下载，另外 Appium 自动化测试 QQ 群：330467341 可以交流讨论

购买地址：
https://yuedu.baidu.com/ebook/7d75728ca0c7aa00b52acfc789eb172ded63991c