一、使用和调用

1.第1次（n次）故障后停止

```
pytest -x  #第1次失败后停止
pytest --maxfail=2  #第2次失败后停止
```

2.指定测试/选择测试

- 在模块中运行测试

```
pytest test_one.py
```

- 在目录中运行测试

```
pytest directory/
```

- 按关键字运行测试

```
pytest -k "keyword"
```

- 按节点ID运行测试

```
pytest test_module.py::TestClass::test_one
```

- 按标记运行测试

```
pytest -m execute  #在需要运行的测试前用@pytest.mark.execute装饰器
```

- 查看指定配置下哪些测试会被运行

```
pytest --collect-only
```

3.修改Python回溯打印

```
pytest -l, --showlocals  #打印本地变量
pytest --tb=no  #不显示
pytest --tb=short  #简要显示
pytest --tb=line  #一行显示
```

4.总结报告

```
pytest -r  # 简短的测试报告（r后面跟可选参数）
f  # 失败的
E  # 错误的
s  # 跳过执行的
x  # 预期失败，实际也是失败的
X  # 预期失败，但实际成功的
p  # 通过的
P  # 通过且有输出的
a  # 除了pP
A  # 所有的
```

5.失败时加载PDB环境

```
pytest --pdb  # 加载python诊断器
```

6.分析测试执行时间

```
pytest --durations=10  # 执行最慢的10个测试用例
```

7.创建JUnitXML格式的测试报告

```
pytest --junitxml=/Users/weiqing/Desktop/test_1.xml test_one.py  # 在指定路径创建能被Jenkins或其他CI工具读取的XML测试报告
```

8.为测试报告提供URL链接

```
pytest --pastebin=failed  # 为失败案例创建链接
```

9.控制台打印print输出

```
pytest -s  # 打印输出
```

二、编写断言

1.使用assert编写断言

```
def func(x):
    return (x+1)

def test_sample():
    assert func(3) == 5, "断言失败的情况说明"
```

三、fixtures

1.fixture中，yield之前属于前置操作，yield之后属于后置操作

2.当测试用例同时使用多个fixture时，先传入的fixture先执行

```
import pytest

@pytest.fixture()
def login():
    print('登录操作!')
    yield
    print('退出登录!')

@pytest.fixture()
def log():
    print('打开日志功能! ')
    yield
    print('关闭日志功能! ')
```

```
class Test_Login():
    def test_01(self, login, log):
        print('需要用到登录！')

*****************输出结果****************

test_3.py::Test_Login::test_01 登录操作!

        SETUP    F login打开日志功能!

        SETUP    F log
        test_3.py::Test_Login::test_01 (fixtures used: log, login)需要用到登录!
PASSED关闭日志功能!

        TEARDOWN F log退出登录!

        TEARDOWN F login
```

3.fixture互相调用

```
import pytest

@pytest.fixture()
def login():
    print('\n登录操作')
    yield
    print('\n退出登录!')

@pytest.fixture()
def log(login):
    print('\n打开日志功能！')
    yield
    print('\n关闭日志功能！')

class Test_Login():
    def test_01(self, log):
        print('\n需要用到登录！')

*****************输出结果****************

test_4.py::Test_Login::test_01
登录操作

        SETUP    F login
```

```
打开日志功能!

        SETUP    F log (fixtures used: login)
        test_4.py::Test_Login::test_01 (fixtures used: log, login)
需要用到登录!
PASSED
关闭日志功能!

        TEARDOWN F log
退出登录!

        TEARDOWN F login
```

4.fixture参数详解

5个参数（name、scope、params、autouse、ids）

（1）name用于对fixture进行重命名，重命名后用以前的名字调用会报错

（2）scope控制fixture的作用域（function、class、module、session），执行顺序session>module>class>function

（3）autouse用于控制fixture在作用域内是否全部执行，默认False

（4）params参数化：fixture的参数request用于获取每种参数（request.param）

```
import pytest

@pytest.fixture(params=["用户1", "用户2", "用户3"])
def login(request):
    print('登录操作')
    yield request.param  # request.param用于获取参数
    print('退出登录')

class Test_01:
    def test_01(self, login):
        print('登录的用户名：%s' % login)

*************输出结果****************

test_1.py::Test_01::test_01[\u7528\u62371] 登录操作
登录的用户名：用户1
PASSED退出登录

test_1.py::Test_01::test_01[\u7528\u62372] 登录操作
```

```
登录的用户名：用户2
PASSED退出登录

test_1.py::Test_01::test_01[\u7528\u62373] 登录操作
登录的用户名：用户3
PASSED退出登录
```

（5）ids用于解释每种参数的意思，list类型

```
import pytest

@pytest.fixture(params=["用户1", "用户2", "用户3"], ids=["user1", "user2", "user3"])
def login(request):
    print('登录操作')
    yield request.param
    print('退出登录')

class Test_01:
    def test_01(self, login):
        print('登录的用户名：%s' % login)

*************输出结果****************

test_1.py::Test_01::test_01[user1] 登录操作
登录的用户名：用户1
PASSED退出登录

test_1.py::Test_01::test_01[user2] 登录操作
登录的用户名：用户2
PASSED退出登录

test_1.py::Test_01::test_01[user3] 登录操作
登录的用户名：用户3
PASSED退出登录
```

5.fixture实例化顺序

（1）高级别作用域（session、module）先于低级别作用域（class、function）进行实例化

（2）相同作用域的，按照在测试用例中声明的顺序（即形参的顺序）进行实例化

（3）autouse=True的fixture，先于同级别的其他fixture进行实例化

6.fixture的清理操作

（1）yield：yield之前的在测试用例之前执行，yield之后的在测试用例之后执行，yield返回值在测试用例中用fixture函数名代替

（2）with：

（3）addfinalizer：结合fixture的request参数来用，request.addfinalizer(fin)添加清理函数

```
@pytest.fixture()
def smtp_connection_fin(request):
    smtp_connection = smtplib.SMTP("smtp.163.com", 25, timeout=5)

    def fin():
        smtp_connection.close()

    request.addfinalizer(fin)
    return smtp_connection
```

7.在类、模块、会话级别用fixture

（1）类：在类前加上 @pytest.mark.usefixtures("fixture函数名")，效果是类中每个测试用例都调用fixture

（2）模块：在.py文件里加上 pytestmark = pytest.mark.usefixtures("fixture函数名")，表示这个模块里的测试用例都可以调用fixture，但还要由fixture作用域决定

（3）会话：在pytest.ini文件中加上 usefixtures=fixture函数名

8.关于调用fixture的思考

（1）首先看作用域，function、class、module、session，作用域决定了fixture会被调用几次

（2）再看调用方式，显式调用还是隐式调用。显式调用分为函数、类、模块、会话级调用；隐式调用通过设置autouse=True让fixture在作用域内被调用

四、临时目录和文件

1.tmp_path：用例级的fixture，返回一个唯一的临时目录对象

```
def test1(tmp_path):
    print(tmp_path)

*************输出结果****************

test_1.py::test1 /private/var/folders/mp/dw75vyxs0_vd6qy38dmrt88w0000gn/T/pytest-of-weiqing/pytest-3/test10
PASSED
```

2.tmp_path_factory：会话级的fixture，返回一个临时目录

```
def test2(tmp_path_factory):
    tmp = tmp_path_factory.mktemp("ww")
    print(tmp)

*************输出结果****************

test_1.py::test2 /private/var/folders/mp/dw75vyxs0_vd6qy38dmrt88w0000gn/T/pytest-of-weiqing/pytest-27/ww0
PASSED
```

3.tmpdir：用例级的fixture，返回一个临时目录

```
def test3(tmpdir):
    tmp = tmpdir.mkdir("sub")
    print(tmp)

*************输出结果****************

test_1.py::test3 /private/var/folders/mp/dw75vyxs0_vd6qy38dmrt88w0000gn/T/pytest-of-weiqing/pytest-28/test30/sub
PASSED
```

4.tmpdir_factory：会话级的fixture，返回一个临时目录

```
def test4(tmpdir_factory):
    tmp = tmpdir_factory.mktemp("sub")
    print(tmp)

*************输出结果****************

test_1.py::test4 /private/var/folders/mp/dw75vyxs0_vd6qy38dmrt88w0000gn/T/pytest-of-weiqing/pytest-30/sub0
PASSED
```

五、捕获告警信息

```
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

def test_one():
    assert api_v1() == 1
```

1.忽略某种类型的告警信息（告警变成功）

```
pytest -W ignore::UserWarning test_1.py
```

2.告警转换为异常（告警转失败）

```
pytest -W error::UserWarning test_1.py
```

六、skip和xfail标记

1.跳过执行

（1）@pytest.mark.skip装饰器

```
import pytest
@pytest.mark.skip(reason="dd")
def test_1():
    ...
```

（2）pytest.skip(reason="dd")强制跳过后续步骤，既可以在测试用例中使用，又可以在module中使用（加上allow_module_level=True）

```
*************在测试用例中使用*************
def test_1():
    if not valid_config():
        pytest.skip(reason="qq")

*************在module中使用*************
import sys
import pytest

if not sys.platform.startswith("win"):
    pytest.skip("skipping windows-only tests", allow_module_level=True)
```

（3）有条件跳过@pytest.mark.skipif装饰器

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

（4）跳过测试类

```
@pytest.mark.skip("作用于类中的每一个用例，所以 pytest 共收集到两个 SKIPPED 的用例。")
class TestMyClass():
    def test_one(self):
        assert True

    def test_two(self):
        assert True
```

（5）跳过测试模块

使用pytestmark变量 pytestmark = pytest.mark.skip() 或

直接使用 pytest.skip(reason=, allow_module_level=True) 跳过剩余部分

（6）跳过指定文件或目录

在conftest.py中配置 collect_ignore_glob 变量

2.标记用例为预期失败的

（1）xfail=pytest.mark.xfail

	@xfail()	@xfail(strict=True)	@xfail(raise=IndexError)	@xfail(strict=True, raise=IndexError)	@xfail(run=False)
执行成功	XPASS	FAILED	XPASS	FAILED	XFAIL
执行失败,上报AssertionError	XFAIL	XFAIL	FAILED	FAILED	XFAIL
执行失败,上报IndexError	XFAIL	XFAIL	XFAIL	XFAIL	XFAIL

（2）去使能xfail标记

通过命令行选项 pytest --runxfail 去使能xfail标记，把这些用例变成正常的用例执行

七、测试的参数化

1.@pytest.mark.parametrize标记

```
@pytest.mark.parametrize('input1, input2', [(1, 2), (3, 4)])
def test_sample(input1, input2):
    assert input1 == input2
```

（1）argnames：需要参数化的参数，用字符串、列表或元祖表示

（2）argvalues：对argnames参数的赋值（利用 pytest.param 或封装 ParameterSet 对象进行参数化）

```
@pytest.mark.parametrize(
    ('n', 'expected'),
    [(2, 1),
    pytest.param(2, 1, marks=pytest.mark.xfail(), id='XPASS')])
def test_params(n, expected):
    assert 2 / n == expected


@pytest.mark.parametrize(
    ('n', 'expected'),
    [(2, 1),
    ParameterSet(values=(1,2), marks=[],id=None)])
def test_params(n, expected):
    assert 2 / n == expected
```

（3）indirect

（4）ids：生成测试ID，理解为用例标识，用列表表示

```
@pytest.mark.parametrize(["a", "b"], [(1, 2), (3, 4)], ids=["first", "second"])
def test_one(a, b):
    return (a + b)

*************输出结果****************

test_1.py::test_one[first] PASSED
test_1.py::test_one[second] PASSED
```

（5）scope：argnames参数的作用域

2.多个标记组合

```
@pytest.mark.parametrize("arg2, arg3", [(4,5), (6,7), (8,9)])
@pytest.mark.parametrize("arg1", [1,2,3])
def test_one(arg1, arg2, arg3):
    return (arg1 + arg2 + arg3)

*************输出结果****************

test_1.py::test_one[1-4-5] PASSED
test_1.py::test_one[1-6-7] PASSED
test_1.py::test_one[1-8-9] PASSED
test_1.py::test_one[2-4-5] PASSED
test_1.py::test_one[2-6-7] PASSED
test_1.py::test_one[2-8-9] PASSED
test_1.py::test_one[3-4-5] PASSED
test_1.py::test_one[3-6-7] PASSED
test_1.py::test_one[3-8-9] PASSED
```

八、用缓存记录执行状态

1.pytest会将本轮测试的执行状态记录到.pytest_cache文件夹中，这个功能由自带的插件cacheprovider实现

2.--lf 或 --last-failed：只执行上一轮失败的用例

3.--ff 或 --failed-first：先执行上一轮失败的用例，再执行其他的

4.--nf 或 --new-first：先执行新加或修改的用例，再执行其他的
5.--sw 或 --stepwise：在第一个失败的用例退出执行，下次还是从这个用例开始执行