Pytest框架

- 讲师：Pansir

自动化鄙视链：pytest -> unittest -> robotframework

pytest优势

1. 完全基于python语法规则
2. 支持失败重试
3. 支持多线程执行
4. 很方便和jenkins集成
5. 很多第三方插件，而且可以自定义扩展
6. 社区很活跃，有问题方便解决
7. 完全兼容unittest用例，之前写过的用例，不用修改，直接用pytest执行即可

安装

命令`pip install pytest`

配置pycharm启动器

perference -> tools -> python intergrated tools -> {你自己的项目} -> Default Test Runner 选择pytest即可。注意：更改执行器后，需要重新新建文件才可生效

Pytest

运行规则

1. 查找指定目录下及其子目录下所有的`test_*.py`或`*_test.py`文件。注意：文件夹中必须包含`__init__.py`
2. 找到文件中所有符合`test*`的函数
3. 将这些函数当作测试用例，全部执行

```
import sys
# !{sys.executable} -m pip install pytest
# !{sys.executable} -m pip install ipytest

import ipytest.magics
import pytest

# Filename has to be set explicitly for ipytest
__file__ = '19pytest框架.ipynb'
```

单个函数

```
# ipython使用，请忽略
%%run_pytest[clean]

# 断言通过
def test_1():
    assert 1 + 1 == 2

# 断言失败，会显示对应的哪一行出错，不需要自己输出断言失败结果
def test_2():
    assert 1 + 1 == 1
```

一组用例

用例规则：

1. 测试类必须以`Test`开头，且不能带有`__init__`方法
2. 测试方法以`test_`开头
3. 断言使用`assert`即可

```
%%run_pytest[clean]

class Testmany():
    def test_01(self):
        assert 'hh' == 'hh'

    def test_02(self):
        assert 1 == 2
```

用例状态

用例执行状态有三种：

1. passed：用例执行通过
2. failed：断言失败
3. error：代码错误，注意区分failed

前置与后置

函数级

只针对函数，实现前置与后置

```
%%run_pytest[clean]
import logging
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
def setup_module():
    logging.info("setup_module 开始")

def teardown_module():
    logging.info("teardown_module 开始")

def setup_function():
    logging.info("setup_function 开始")

def teardown_function():
    logging.info("teardown_function 开始")

def test_01():
    logging.info("用例开始")
    x = "apple"
    assert 'a' in x
```

class级

针对class类

```
%%run_pytest[clean]
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
class TestCase():

    def setup(self):
        logging.info("setup 开始")

    def teardown(self):
        logging.info("teardown 开始")
    # setUpClass unittest
    def setup_class(self):
        logging.info("setup_class 开始")

    def teardown_class(self):
        logging.info("teardown_class 开始")

    def setup_method(self):
        logging.info("setup_method 开始")

    def teardown_method(self):
        logging.info("teardown_method 开始")

    def test_01(self):
```

```
    logging.info("用例开始")
    x = "apple"
    assert 'a' in x
```

- 解释说明：
    1. setup_method 和 setup 效果是一样的，都是用例前置，使用的时候选择一个即可。如果混用，method优先级更高
    2. 前置执行顺序：module > class > method > setup。后置则反过来
    3. 函数级和class级，会共用 module 前后置，moudule优先级最高，写用例的时候需要留意

fixture自定义前后置

测试夹具

问题引入

我们测试的时候写入前置和后置的时候发现，我们的前置和后置，并不是所有用例都需要。例如：测试正常的审核接口，前置需要先创建一个签名，再进行审核。但是测试异常场景，例如：审核时传入密码错误，则不需要先创建签名。但是我们目前的前置仍然会执行，执行了一些不必要的操作。pytest提供了fixture可以很好的解决这个问题

简单应用

1. 通过`@pytest.fixture()`指定前置，默认传参数`scope=function`函数前置
2. 用例函数传入被装饰的那个前置即可

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# # 通过fixture指定login为前置
@pytest.fixture(scope="session")
def login():
    logging.info("自定义的前置")

# 函数当作参数传入
def test_01(login):
    logging.info("test_01执行")
    x = "apple"
    assert 'a' in x

# # 不传login则没有前置
def test_02(login):
    logging.info("test_02执行")
    assert 1 == 1
```

- 练习：用fixture封装一个前置：创建一个签名。写一个查询签名用例，引用这个前置

```
import pytest
from pytest_commons.tsms_base import Tsmstest

ts = Tsmstest()


@pytest.fixture()
def create_ran_sign():
    ts.req_post('sign', ts.sign_data)


def test_01(create_ran_sign):
    ts.tsms_get('sign', user='root', passwd=123)
```

conftest统一管理

上面是一个文件中使用，考虑到如果多个文件需要使用一个共同的前置，则需要引入`conftest.py`

1. `conftest.py`名称固定，不能改
2. `conftest.py`必须与用例在同一个目录下
3. 该目录下必须存在`__init__.py`
4. 不需要导入`conftest.py`，pytest会自动检索

```
# conftest.py
import pytest
import logging
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

@pytest.fixture()
def login():
    logging.info("自定义前置：执行登录")
```

```
# 用例test_case_1.py中引用 login前置
def test_2(login):
    logging.info("执行测试")
    assert 1 + 1 == 2
```

- 解释：conftest.py的作用域为当前目录下及子目录的用例文件，但是每个目录也可以有自己的`conftest.py`，但是各自`conftest.py`中的方法，不能共用
- 练习：在conftest中实现登录流程，在用例中引用该前置

```
@pytest.fixture(scope="session")
def login_tsms():
    logging.info("开始执行登录")
    tb = TsmsWeb()
    tb.login_c('dcs', 123)
    assert tb.is_login()
```

scope参数说明

scope参数支持4种级别

1. function(默认)：函数级别，即用例级别的前置/后置。每一个函数和方法都会调用
2. class：类级别，即套件级别的前置/后置。每个类只调用一次
3. moudule：模块级别，即文件的前置/后置。每个py文件(module)只调用一次，注意：文件中可以有多个类+多个函数
4. session：全局，即总前置/后置。多个文件只调用一次

function前置

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')

# 通过fixture指定login为前置
@pytest.fixture(scope="function")
def login():
    logging.info("function前置")

# 函数使用function前置
def test_fun1(login):
    logging.info("test_fun1执行")
    x = "apple"
    assert 'a' in x

# 类中的方法使用function前置
class TestFun():
    def test_fun2(self, login):
        logging.info("test_fun2执行")
        assert 1 == 1
```

class前置

- class级别的前置可以被函数使用，但是不推荐这样用
- 类中每个方法都可以使用class前置，但是只有第一个运行的会生效

```
%%run_pytest[clean]
import logging, pytest
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture(scope="class")
def login():
    logging.info("class前置")

@pytest.fixture(scope="function")
def login1():
    logging.info("class前置1")

# # 函数是运行使用class前置的，但是建议这样使用！！
def test_function1(login1):
    logging.info("test_function1 执行")
    x = "apple"
    assert 'a' in x

# 类中多个地方使用class前置，但只有第一个使用的方法会生效
class TestCls():
    def test_class2(self, login):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self, login):
        logging.info("test_class3 执行")
        assert 1 == 1
```

module前置

- 整个文件中第一个用例执行前运行
- 一般来说，运行顺序是从上往下运行，但是class的执行优先级要高于函数级

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture(scope="module")
def login():
    logging.info("module 前置")

# 这个不会运行，因为class优先运行
def test_module1(login):
    logging.info("test_module1 执行")
    x = "apple"
    assert 'a' in x
```

```
# 类中的第一个用例优先运行
class TestMod():
    # 仅这条用例会运行一次前置
    def test_module2(self, login):
        logging.info("test_module2 执行")
        assert 1 == 1

    def test_module3(self, login):
        logging.info("test_module3 执行")
        assert 1 == 1
```

session前置

- 多个文件前只执行一次
- 需要把这个前置写到`conftest.py`文件里，如果放到项目的根目录则全局有效，如果放到某个包下，则该只在该包内有效

```
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture(scope="session")
def login():
    logging.info("session 前置")
```

```
# 文件1
def test_session1(login):
    logging.info("test_session1 执行")
    x = "apple"
    assert 'a' in x
```

```
# 文件2
class TestSe():
    # 仅这条用例会运行一次前置
    def test_session2(self, login):
        logging.info("test_session2 执行")
        assert 1 == 1

    def test_session3(self, login):
        logging.info("test_session3 执行")
        assert 1 == 1
```

练习：使用fixture定义三个前置：function前置/module前置/session前置，然后在多个用例文件中使用

fixture自定义后置

后置是通过yield实现

函数级后置

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture()
def login():
    logging.info("登录过程")
    yield
    logging.info("退出登录")

# 只要后置
@pytest.fixture()
def claer():
    yield
    logging.info("仅仅是后置")

# def test_01(login):
#     logging.info("test_01执行")
#     x = "apple"
#     assert 'a' in x

# def test_02(login):
#     logging.info("test_02执行")
#     assert 1 == 1

def test_03(claer):
    logging.info("test_03执行")
    assert 1 == 1
```

class级后置

- class级前后置可以被函数使用，效果等同于函数级，即每个函数都会生效
- 类中使用class级前后置，则只会生效一次

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

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

```
def login():
    logging.info("自定义的前置")
    yield
    logging.info("自定义的后置")

def test_class1(login):
    logging.info("test_class1 执行")
    x = "apple"
    assert 'a' in x

def test_class0(login):
    logging.info("test_class0 执行")
    x = "apple"
    assert 'a' in x

class TestCls():
    def test_class2(self, login):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self, login):
        logging.info("test_class3 执行")
        assert 1 == 1
```

模块级后置

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture(scope="module")
def login():
    logging.info("module 前置")
    yield
    logging.info("module 后置")

# 这个不会运行，因为class优先运行
def test_module1(login):
    logging.info("test_module1 执行")
    x = "apple"
    assert 'a' in x

# 类中的第一个用例优先运行
class TestMod():
    # 仅这条用例会运行一次前置
    def test_module2(self, login):
```

```
        logging.info("test_module2 执行")
        assert 1 == 1

    def test_module3(self, login):
        logging.info("test_module3 执行")
        assert 1 == 1
```

文件级别后置

在所有py文件的用例都执行完毕后，最后执行

yield异常说明

执行顺序：前置-> 用例 -> 后置

用例异常

用例出现异常时，不会影响后置的运行

前置异常

- 如果前置执行出现异常，则不会执行yield后面的后置内容
- 必须确保前置不会出错，否则用例将不执行

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
@pytest.fixture()
def login():
    a = int("a")
    logging.info("自定义的前置")
    yield
    logging.info("自定义的后置")

# 直接阻塞，用例和后置都不执行
def test_01(login):
    logging.info("test_01执行")
    x = "apple"
    assert 'a' in x

class TestCls():
    def test_class2(self, login):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self, login):
        logging.info("test_class3 执行")
```

```
        assert 1 == 1
```

强制后置

- 使用addfinalizer方法实现强制后置，如果前置出错了，用例不会执行，但后置依然会执行
- addfinalizer可以注册多个终结函数

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 这里后置必须传入一个参数：request，否则会报错
@pytest.fixture(scope='function')
def login(request):
    def teardown_function1():
        logging.info("强制后置1，无论你的前置是否中断")
    def teardown_function2():
        logging.info("强制后置2，无论你的前置是否中断")
    # 此内嵌函数做teardown工作
    request.addfinalizer(teardown_function2)
    request.addfinalizer(teardown_function1)  # 下面的先执行

    # 这是前置
#     a = int("a")
    logging.info("这是前置，尽管我放到后面")

def test_01(login):
    logging.info("test_01执行")
    x = "apple"
    assert 'a' in x

class TestCls():
    def test_class2(self, login):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self, login):
        logging.info("test_class3 执行")
        assert 1 == 1
```

- 预留练习：在所用用例完成后，检查cache中是否有sign_id，若有，则进行删除

自动前置

上面使用前置必须要传入`login`，下面提供两种自动引用的方法

userfixtures装饰器

当一个类中有多个方法都需要用前置时，对类使用userfixtures装饰器：`@pytest.mark.usefixtures("login")`。注意`"login"`是函数名的字符串值

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 通过fixture指定login为前置
# @pytest.fixture(scope="class")
@pytest.fixture(scope="function")
def login():
    logging.info("自定义的前置")
    yield
    logging.info("自定义的后置")

@pytest.mark.usefixtures("login")
class TestCls():
    def test_class2(self):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self):
        logging.info("test_class3 执行")
        assert 1 == 1
```

使用装饰器后，前置/后置对整个类生效

autouse自动生效

对前置设置`autouse=True`后，该前/后置不需要指定就可以生效

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 指定autouse自动引用
@pytest.fixture(scope="class", autouse=True)
def login():
    logging.info("自定义的前置")
    yield
    logging.info("自定义的后置")

class TestCls():
    @pytest.fixture(scope="function", autouse=True)
    def create_sign(self):
        logging.info("创建签名")
```

```
        yield
        logging.info("删除签名")

    def test_class2(self):
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self):
        logging.info("test_class3 执行")
        assert 1 == 1
```

使用`autouse`后，无论放到哪里，都会自动引用

fixture传递变量

变量可以返回给用例

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# @pytest.fixture(scope="class", autouse=True)
@pytest.fixture(scope="function")
def login():
    logging.info("自定义的前置")
    return "随便返回个值"


class TestCls():

    def test_class2(self, login):
        # 获取返回值
        a = login
        logging.info(a)
        logging.info("test_class2 执行")
        assert 1 == 1

    def test_class3(self,login):
        a = login
        logging.info("test_class3 执行")
        assert 1 == 1
```

- fixture互相调用，参数也可以进行透传

```
%%run_pytest[clean]
import logging, pytest
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

@pytest.fixture(scope="function")
def login():
    logging.info("自定义的前置")
    return 200

@pytest.fixture()
def open_1(login):
    a = login
    logging.info("fixture传参为: {}".format(a))
    return a, "ok"

class TestCls():

    def test_class2(self, open_1):
        # 获取返回值
        a, b = open_1
        logging.info("获取返回值为: {} {} ".format(a, b))
        assert 1 == 1

    def test_class3(self):
        logging.info("test_class3 执行")
        assert 1 == 1
```

装饰器参数化(类数据驱动)

parametrize基本用法

`@pytest.mark.parametrize("user, psw", test_login_data)`

1. 第一个参数：指定要传入方法的多个参数名
2. 第二个参数，传入一个列表，每个元素为一个测试数据组

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 测试登录数据
test_login_data = [
    ("admin", "admin"),    # 元素可以有多个，而且不局限类型
    ("root", "123"),
    ("dcs", "123"),
]

def login(user, psw):
```

```
    logging.info(user)
    logging.info(psw)

# 注意：这里的user/psw 一定要和 test_login的变量名称一致
@pytest.mark.parametrize("user, psw", test_login_data)
def test_login(user, psw):
    result = login(user, psw)
    logging.info("ok")
```

结合fixture

1. `@pytest.mark.parametrize("login", test_login_data, indirect=True)` 接收三个参数：
    - 字符串，和前置函数的名称一致
    - 测试数据组
    - indirect=True，标识第一个参数是函数名，而不是字符串
2. `user, psw = request.param` 接收parametrize传过去的列表`test_login_data`，逐个解析出来

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 测试登录数据
test_login_data = [
    ("admin", "admin"),
    ("root", "123"),
    {"name": "dcs", "age": 18}
]
# 登录是模块前置
@pytest.fixture(scope="module")
def login(request):
    logging.info(request.param)
    user, psw = request.param
    logging.info(user)
    logging.info(psw)

@pytest.mark.parametrize("login", test_login_data, indirect=True)
def test_login(login):
    logging.info("ok")
```

parametrize支持传字典

```
%%run_pytest[clean]
import logging, pytest
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 测试登录数据
test_login_data = [{"user":"dcs", "psw": "123"}, {"user":"root", "psw":
"123"}]
# 登录是模块前置
@pytest.fixture(scope="module")
def login(request):
    logging.info(request.param)
    user = request.param["user"]
    psw = request.param["psw"]
    logging.info(user)
    logging.info(psw)

@pytest.mark.parametrize("login", test_login_data, indirect=True)
def test_login(login):
    logging.info("ok")
```

- 扩展思维：可以利用records库 + pytest做数据驱动

练习

使用`mark.parametrize`装饰器实现数据驱动，完成多组数据计算：`[ ("1+1", 2), ("2*3", 6), ("3-1", 1),]` 和断言

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

a = "test_input,expected"
b =  [
    ("1+1", 2),
    ("2*3", 6),
    ("3-1", 1),
]

@pytest.mark.parametrize(a, b)
def test_eval(test_input, expected):
    logging.info("测试用例执行")
    assert eval(test_input) == expected
```

标记失败

通过`marks=pytest.mark.xfail`标记失败，则该场景不运行

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

a = "test_input,expected"
b =  [
    ("1+1", 2),
    ("2*3", 6),
    #
    pytest.param("3-0", 3, marks=pytest.mark.xfail),
]


@pytest.mark.parametrize(a, b)
def test_eval(test_input, expected):
    logging.info("测试用例执行")
    assert eval(test_input) == expected
    logging.info(test_input)
    logging.info(expected)
```

参数组合

利用嵌套装饰器可以实现参数组合

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')


@pytest.mark.parametrize('a', [ None, 1, "a"])
@pytest.mark.parametrize('b', [None, 3, "b"])
def test_01(a, b):
    logging.info("测试用例执行 {} {}".format(a, b))
    assert a == b
```

多重前置

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

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

```
def input_user(request):
    user = "dcs"
    logging.info(user)
    return user


@pytest.fixture(scope="module")
def input_psw(request):
    psw = "123"
    logging.info(psw)
    return psw
# 依次传入，则依次运行
def test_login(input_user, input_psw):
    a = input_user
    b = input_psw
    logging.info("数据组合 {} {} ".format(a, b))
```

多重前置+param

多重前置+多重param

```
%%run_pytest[clean]
import logging, pytest
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

test_user = ["admin", "root"]
test_psw = ["123", "456"]


@pytest.fixture(scope="module")
def input_user(request):
    user = request.param
    logging.info(user)
    return user


@pytest.fixture(scope="module")
def input_psw(request):
    psw = request.param
    logging.info(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):
    a = input_user
    b = input_psw
```

```
    logging.info("数据组合 {} {} ".format(a, b))
```

用例数为：2 * 2 = 4个

练习：需求：一般接口返回错误，说明服务端并没有对这次错误请求进行数据落地(包括redis/db的变动)，而这类异常场景又相对较多，所以可以使用数据驱动对这类场景进行统一管理，可以大大减少代码编写量。同时也方便统一管理

1. 只测最简单的接口返回，错误码 + 错误内容
2. 一个字典就是一条用例

断言与异常

一般断言

1. assert xx 判断xx为真
2. assert not xx 判断xx不为真
3. assert a in b 判断b包含a
4. assert a == b 判断a等于b
5. assert a != b 判断a不等于b

异常断言

- 如何去断言异常呢?

```
%%run_pytest[clean]
import logging, pytest, json
from json import JSONDecodeError
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

def test_json_error():
    '''触发异常'''
    a = '''{'name': 'dcs'}'''
    b = json.loads(a)

def test_json_error1():
    '''先捕获再断言'''
    a = '''{'name': 'dcs'}'''
#     a = '''{"name": "dcs"}'''
#     b = json.loads(a)
    try:
        a = int("a")
        b = json.loads(a)
    except Exception as e:
        logging.info("只要打印这行，则说明捕获到了异常")
        logging.info(e)
        assert e == "1"
```

问题：我们只是捕获了异常，但无法断言异常，因为无法拿到实际结果。可以通过pytest.raises获取到异常类型，和异常内容

```
%%run_pytest[clean]
import logging, pytest, json
from json import JSONDecodeError
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

def test_json_error1():
    a = '''{'name': 'dcs'}'''
    with pytest.raises(JSONDecodeError) as e:
        b = json.loads(a)
    # 断言异常类型type，从json库导入
    logging.info(e.type)
    assert e.type == JSONDecodeError
    # 断言异常value值，需要转换str
#     assert "Expecting property name enclosed in double quotes: line 1 column
2 (char 1)" in str(e.value)
```

小结：异常的捕获与断言，是单元测试或sdk测试必须学会的思路。测试的逻辑：主动去触发异常，然后断言异常是否符合预期

用例跳过

skip标记

当我们希望用例跳过时，我们可以用skip装饰器标记它。通常会用到跳过的场景：

1. 自动化用例已经写完了，但是开发的接口还没写完，先跳过
2. 本地跟内网不通，需要操作数据的场景，先跳过
3. 调试过程

装饰器标记

直接跳过

```
%%run_pytest[clean]
import logging, pytest, json
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 1. 装饰器标记
@pytest.mark.skip(reason="跳过该用例")
def test_skip1():
    assert 1 ==2

def test_skip4():
    assert 1 == 1
```

条件触发跳过

当满足某个条件后，执行跳过

```
%%run_pytest[clean]
import logging, pytest, json
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 2. 装饰器标记
def valid_config():
    logging.info("判断配置是否准确")
    return False

def test_skip2():
    if not valid_config():
        pytest.skip("配置错误")
    logging.info("如果没有跳过，会执行这行")
    assert 1 == 2

def test_skip4():
    assert 1 == 1
```

命令行跳过

写死到代码里，控制起来不方便，使用命令行参数才是最常用的做法

```
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
a = 2
# 通过条件来判断，是否跳过整个文件
```

```
if a == 1:
    pytest.skip("skipping windows-only tests", allow_module_level=True)


def test_skip5():
    logging.info("ok")
    assert 1 == 1
```

执行命令：`py.test test_skip_this.py`即可看到跳过

skipif条件跳过

通过条件判断，来决定是否跳过

```
%%run_pytest[clean]
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')


# 当python版本低于3.8则不执行该用例，我的版本为3.7 所以跳过该用例
@pytest.mark.skipif(sys.version_info < (3, 6), reason="python版本过低跳过")
def test_function():
    logging.info(sys.version_info)
    logging.info("hello")
    assert 1 == 1
```

- 可以通过定义一个标记，然后进行复用。注意不管你使不使用装饰器：`@limit_ver`，都会跳过

```
%%run_pytest[clean]
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

limit_ver = pytest.mark.skipif(sys.version_info < (3, 8), reason="python版本过低
跳过")

# 当python版本低于3.8则不执行该用例，我的版本为3.7 所以跳过该用例
@limit_ver
```

```
def test_function():
    logging.info(sys.version_info)
    logging.info("hello")
    assert 1 == 1
```

- 真实使用的时候，是统一用一个文件管理，需要使用的时候再导入。导入的skip则必须使用装饰器才生效

```
import sys
import pytest
import logging
import limits
from limits import limit_ver

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')


# 当python版本低于3.8则不执行该用例，我的版本为3.7 所以跳过该用例
@limit_ver
def test_function():
    logging.info(sys.version_info)
    logging.info("hello")
    assert 1 == 1


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

跳过类

同上，当前文件定义skip，则不实用装饰器也会跳过，如果导入的话，则需要使用装饰器修饰才会生效

```
%%run_pytest[clean]
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')

limit_ver = pytest.mark.skipif(sys.version_info < (3, 8), reason="python版本过低跳过")
# @limit_ver
class TestClsSkip(object):
    def test_1(self):
```

```
        logging.info("ok")

    def test_2(self):
        logging.info("okk")
```

缺少依赖跳过

如果python缺少某种依赖，也可以跳过

```
%%run_pytest[clean]
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')
# 如果导入jsons失败，则会跳过本文件
# rerp = pytest.importorskip("requests1")
# 如果版本
rerp = pytest.importorskip("requests", minversion='2.18.2')


def test_function1():
    logging.info("hello")
    assert 1 == 1
```

小结

1. `pytestmark = pytest.mark.skip("跳过所有用例")`
2. `pytestmark = pytest.mark.skipif(a == 1, "满足条件则跳过用例")`
3. `pexpect = pytest.importorskip("requests1")` 缺少某个依赖则跳过用例

xfail跳过

当用例互相有依赖时，比如：需要登录成功，才能进行相关的页面内容解析

```
%%run_pytest[clean]
import logging, pytest
import random
logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

# 测试登录数据
test_login_data = [
    {"user":"dcs", "psw": "123"},
    {"user":"root", "psw": "123"}
```

```
]
# 登录是模块前置
@pytest.fixture(scope="module")
def login(request):
    logging.info(request.param)
    user = request.param["user"]
    psw = request.param["psw"]
    logging.info(user)
    logging.info(psw)
    res = random.choice([True, False])
    logging.info("登录状态是：{}".format(res))
    return res

@pytest.mark.parametrize("login", test_login_data, indirect=True)
class TestMark():
    def test_login1(self, login):
        res = login
        if not res:
            pytest.xfail("登录失败，标记为xfail")
        logging.info("ok1")

#     def test_login2(self, login):
#         logging.info("ok2")
```

- 判断依赖条件是否满足，如果依赖条件不满足则标记为xfail
- 示例场景：
    1. 登录失败，则不需要进行后续的登录检查操作
    2. 创建签名的前置失败，则不需要测试删除该签名的操作，标记为xfail

用例标记

可以通过装饰器：`@pytest.mark.{tag}`给用例做标记，类似RF的tags

对函数/方法标记

```
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')


class TestClsMark(object):
    @pytest.mark.debug
    def test_1(self):
```

```
        logging.info("ok")

    @pytest.mark.ok
    @pytest.mark.debug
    def test_2(self):
        """可以打多个标签"""
        logging.info("okk")
```

执行命令：

1. 通过命令：`py.test -s test_mark.py -m debug` 执行，则只会执行标记的用例
2. 若想仅不执行该用例：`py.test -s test_mark.py -m 'not debug'` 执行，则会执行其他的用例
3. `pytest -m "level1 or critical" test_tsms_sign_api_create.py`

对类标记

```
import sys
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

@pytest.mark.hello
class TestClass:
    def test_01(self):
        logging.info("hello kitty")

    def test_02(self):
        logging.info("hello world")
```

mark使用小结

```
# 跳过测试
@pytest.mark.skip(reason=None)

# 满足某个条件时跳过该测试
@pytest.mark.skipif(condition)

# 预期该测试是失败的
@pytest.mark.xfail(condition, reason=None, run=True, raises=None,
strict=False)

# 数据驱动
@pytest.mark.parametrize(argnames, argvalues)
```

```
# 对统一使用
@pytest.mark.usefixtures(fixturename1, fixturename2, ...)

# 让测试尽早地被执行
@pytest.mark.tryfirst

# 让测试尽量晚执行
@pytest.mark.trylast

# 标记用例
@pytest.mark.level1
```

变量共享

```
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')


# 1. 直接通过pytestconfig获取
def test_var1(pytestconfig):
    host_address = pytestconfig.getoption('--host')
    startall = pytestconfig.getoption('--startall')
    if host_address:
        host = host_address
    else:
        host = '1.1.1.1'
    logging.info("使用的host是：{}".format(host))
    logging.info("使用的startall是：{}".format(startall))


# 2. 通过函数名：host_address1，获取到变量，介质是pytestconfig
def test_var2(host_address1):
    if host_address1:
        host = host_address1
    else:
        host = '1.1.1.2'
    logging.info("通过fixture装饰host_address1 + pytestconfig 获取到到host：
{}".format(host))


# 3. 通过函数名：host_address2，获取到变量，介质是request
def test_var3(host_address2):
    if host_address2:
        host = host_address2
    else:
```

```
        host = '1.1.1.3'
    logging.info("通过fixture装饰host_address2 + request 获取到到host：
{}".format(host))


# 4. 通过函数名：share_var，获取到变量，介质是pytest
def test_var4(share_var):
    logging.info(pytest.name)
    logging.info(pytest.passwd)


if __name__ == '__main__':
    pytest.main(['-s', '--host=127.0.0.1', __file__])
```

- pytest全局变量案例

```
@pytest.fixture(scope="session")
def clear_sign():
    pytest.sign_id = []
    yield
    sign_ids = pytest.sign_id
    logging.info("[经过一轮测试，新增的sign_id是]：{}".format(sign_ids))
    try:
        ts = Tsmstest()
        for sign_id in sign_ids:
            ts.req_delete("sign", {"sign_id": sign_id})
            if ts.status_code != 200:
                logging.error("[签名删除失败，签名id为]：{}".format(sign_id))
    except Exception as e:
        logging.error(e)
```

```
def test_01_cache(clear_sign):
    # 创建签名
    ts.req_post('sign', ts.sign_data)
    assert ts.status_code == 200
    logging.info(ts.json["sign_id"])
    # 把sign_id丢到列表缓存
    pytest.sign_id.append(ts.json["sign_id"])
```

命令模式

1. 执行命令有三种：`pytest`；`py.test`；`python -m pytest`；

2. 执行整个目录：`pytest {目录名称}`。如果不带目录名称，则执行当前目录下所有符合要求的用例
3. 执行某个py文件：`pytest {py文件名}`
4. 关键字匹配：`pytest -k "{文件名/类名/函数名} and not {文件名/类名/函数名}"`；按照指定规则来执行用例
5. 运行py文件中的某个用例：`pytest {py文件名}::{函数名}`
6. 运行py文件中测试类中的某个用例：`pytest {py文件名}::{类名}::{方法名}`
7. 遇到错误停止：`pytest -x {py文件名}`
8. 当错误数量到达限制则停止：`pytest --maxfail=2`
9. 运行上次执行失败的用例：`pytest --lf/--last-failed`
10. 执行上次失败的用例：`--lf/--last-failed`

常见命令参数：

1. `-q`：安静模式，不输出环境信息
2. `-v`：丰富信息模式
3. `-s`：显示 print/logging输出
4. `--resultlog=./test.log`：生成日志文件
5. `--junitxml=./test.xml`：生成xml报告
6. `--durations=N`：显示执行最慢的前N条用例
7. `--collect-only`：只收集用例不执行
8. `-h`：显示所有命令

python命令行

普通命令行如何使用，参考博客文章：http://birdgugu.com/2019/11/28/python-script-param/

pytest自定义命令行

```
# 定义命令行参数
# 1. 在用例中，可直接通过 pytestconfig.getoption('--host') 获取
def pytest_addoption(parser):
    # 定义命令行参数
    parser.addoption('--host_addr', action='store', default=None, help='传入host地址')
    parser.addoption('--startall', action='store', default=None, help='标示执行全部用例')

# 2. 可通过方法返回，然后通过fixture透传，注意：这里使用pytestconfig获取参数
@pytest.fixture()
def host_address1(pytestconfig):
    # 这里的host_add自定义命名，将整个结果返回
    logging.info("[pytestconfig为]: {}".format(pytestconfig.))
    return pytestconfig.getoption('--host_addr')


# 3. 可通过方法返回，然后通过fixture透传，注意：这里使用request获取参数
```

```
@pytest.fixture()
def host_address2(request):
    # 这里的host_add自定义命名，将整个结果返回
    return request.config.getoption('--host_addr')
```

注意：`pytestconfig` 其实是 `request.config` 的快捷方式，所以也可以自定义固件实现命令行参数读取。

```
import pytest
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s %(message)s')


def test_cmd_1(host_address1):
    """通过夹具拿到外部变量"""
    logging.info(host_address1)
    if host_address1:
        host = host_address1
    else:
        host = '1.1.1.1'
    logging.info("使用的host是：{}".format(host))


def test_cmd_2(host_address2):
    """通过夹具拿到外部变量"""
    logging.info(host_address2)
    if host_address2:
        host = host_address2
    else:
        host = '1.1.1.1'
    logging.info("使用的host是：{}".format(host))


def test_cmd_3(pytestconfig):
    """通过pytest变量获取"""
    logging.info(pytestconfig.getoption('--host_addr'))
```

使用下面两种命令即可指定参数，注意：这里的S是用显示详情的

1. `pytest -s test_cmd_case.py --host_addr 127.0.0.1`
2. `pytest -s test_cmd_case.py --host_addr=127.0.0.1`

注意：不要使用一下特殊变量，比如：`host`就会和一些插件/pytest本身的一些命名重复，而导致出错

练习：从外部获取参数，通过`--env=pro`可以指定测试生产环境

```
@pytest.fixture(scope="function")
def tb(pytestconfig):
    logging.info("[环境参数为]: {}".format(pytestconfig.getoption("--env")))
    if pytestconfig.getoption("--env") == "pro":
        return TsmsBase(env=2)
    else:
        return TsmsBase()
```

```
    def test_sign_api_create_07(self, tb):
        """演示环境切换"""
        data = tb.sign_data()
        tb.req_post("sign", data)
        assert isinstance(tb.json.get("sign_id"), int)
```

内置固件

内置固件可以理解为pytest的一个内置变量，这个变量可以用来完成一些特殊功能，比如：`tempdir`，`tmpdir_factory`，`pytestconfig`，`request`等，所以我们在使用变量命名的时候，要避开这些

临时文件与目录

- `tmpdir` 只有 function 作用域，只能在函数内使用。

```
def test_tmpdir_1(tmpdir):
    """单独用例使用，用完默认删除"""
    """只能在函数中使用"""
    a_dir = tmpdir.mkdir('mytmpdir')
    a_file = a_dir.join('tmpfile.txt')

    a_file.write('hello, pytest!')

    assert a_file.read() == 'hello, pytest!'
```

- tmpdir_factory 可以在所有作用域使用，包括 function, class, module, session

```
@pytest.fixture(scope='module')
def my_tmpdir_factory(tmpdir_factory):
    """定义一个夹具，则可以指定作用域"""
    a_dir = tmpdir_factory.mktemp('mytmpdir')
    a_file = a_dir.join('tmpfile.txt')
```

```
    a_file.write('hello, pytest!')

    return a_file


def test_tempdir(my_tmpdir_factory):
    """借用夹具"""
    # 获取路径
    logging.info(my_tmpdir_factory.read())
    # 获取内容
    logging.info(my_tmpdir_factory.read())
```

命令行固件

使用 `pytestconfig`，可以很方便的读取命令行参数和配置文件。详细用法，见：命令模式章节

```
```