pytest allure 测试框架

岳泽俊

前言

pytest 是 python 的一种测试框架，相比于 robotframework，测试脚本的编写无需了解 robotframework 特殊的语言格式殊的类型转换（此处我必须吐槽 robotframework 默认参数类型为 str 类型这个设置，实在太反人类了）等；相比于 unittest 代码更加简洁、插件更加丰富，尤其是 allure 插件，可以轻松的生成非常漂亮的测试报告。

所以我想与诸位分享下 pytest 测试框架技术。当然学习一个技术的最好方式就是读源码和官方文档，但是考虑到官方文档非常的详细，很多接口对于实际项目操作而言是冗余的。于是我站在项目实用性的角度上抽出了我认为十分使用的接口及使用说明给大家。

这篇文章除去前言共有三个章节：

- pytest 该章节用户可以了解到以下内容：
  - 如何安装 pytest
  - pytest 的编写规则
  - pytest 库的常用接口
  - 命令行规则
  - 相关配置文件
- allure 该章节用户可以了解到以下内容：
  - 如何安装 allure
  - allure 库的常用接口
  - 命令行规则
- 测试脚本编写该章节用户可以了解到以下内容：
  - 项目上如何使用 pytest+allure 编写测试脚本

文档中如有疏漏或不对之处，请批评指正。

希望各位都能从中有所收获。

目录

pytest

环境安装

```
pip install pytest
```

约束规则

- 测试文件名要符合 test_*.py 或*_test.py 格式（例如 test_modulename.py）
- 测试类要以 Test 开头，且不能带有 init 方法。在单个测试类中，可以包含一个或多个 test_开头的函数
- 测试函数要以 test_开头

常用接口

- **assert(条件，补充)**

  用途：为测试结果做断言、为断言不通过的结果添加说明信息

  示例：

```
import pytest

def test_function_001(self):
    result = 2 + 3
    assert result == 5, "3+3 应该等于 6"  # 为断言不通过的结果添加说明信息
```

- **pytest.raises(异常类)**

  用途：判断期望结果是否是指定异常类

  示例：

```
import pytest

def test_function_001(self):                # 测试结果为失败
    with pytest.raises(Exception):
        pass

def test_function_002(self):                # 测试结果为成功
    with pytest.raises(Exception, match=r"运行错误*?") as err_info: #正则匹配错误
        raise RuntimeError("运行错误")
    assert err_info.type == RuntimeError                # 判断错误的类型是否正确
    assert err_info.value.args[0] == "运行错误 11"       # 判断错误的提示信息是否正确
```

- **@pytest.mark.标签名**

  用途：用于直接标记类、方法或函数

  示例：

```
import pytest
```

```
@pytest.mark.happytest                  # 可以给类打标签
@pytest.mark.smoke                      # 也可以多个标签
class Test_Module:                      # 标记类
    @pytest.mark.fulltest
    def test_function_001(self):    # 标记方法
        pass

@pytest.mark.smoke        # 单个标签
def test_function_101():        # 标记函数
    pass
```

- **@pytest.mark.skip(原因)**

  用途：直接跳过被修饰的类、方法或函数

  示例：

```
import pytest

@pytest.mark.skip("该需求在 XX 项目上已遗弃")
def test_function(self):
    pass
```

- **@pytest.mark.skipif(条件,原因)**

  用途：当满足某个条件时，跳过被修饰的类、方法或函数

  示例：

```
import pytest

@pytest.mark.skipif(project == 'Project_Name', reason='项目 Project_Name 上该需求已被遗弃')
def test_function(self):
    pass
```

- **pytest.skip(原因)**

  用途：在测试执行期间（也可以在 SetUp/TearDown 期间）强制跳过后续的步骤

  示例：

```
import pytest
import sys

def test_function():
    if not sys.platform.startswith("win"):
        pytest.skip("skipping windows-only tests")
```

- **pytest.importorskip(模块名)**

  用途：当引入某个模块失败时，跳过后续部分的执行

  示例：

```
import pytest

pytest.importorskip(sys)                    # 当导入 sys 模块失败时，跳过后续执行步骤

def test_function():
    if not sys.platform.startswith("win"):
        pytest.skip("skipping windows-only tests")

import pytest

pytest.importorskip(sys, minversion="0.3")    # 当导入 sys 模块失败或 sys 的版本低于 0.3 时，跳过后续执行步骤

def test_function():
    if not sys.platform.startswith("win"):
        pytest.skip("skipping windows-only tests")
```

- **@pytest.mark.parametrize(参数名，参数值列表)**

  用途：实现测试用例参数化。适用于同一份 case，不同值的输入测试

  示例：

```
import pytest

# 传入单个参数
@pytest.mark.parametrize("input", ["输入值 1", "输入值 2", "输入值 3", "输入值 4", "输入值 5"])
def test_case_001(input):
    ...


    assert '输入值' == input

# 传入多个参数
@pytest.mark.parametrize("user,pwd",
                         [("yuezejun_1_0", "123456"), ("yuezejun_1_1", "123456"),
```

```
                    pytest.param("yuezejun_1", "123456", marks=pytest.mark.xfail),
                    pytest.param("yuezejun_2", "123456", marks=pytest.mark.skip)])
def test_case_002(user, pwd):

    assert user == "Stark"
```

- **@pytest.mark.timeout(时间)**

  前提：依赖 pytest-timeout

  用途：判断用例的执行时间是否超时，如果超时，直接中断

  注意事项：该装饰器与 allure 并不契合。如果该用例超时，allure 测试结果中不会显示 fail

  示例：

```
import pytest

@pytest.mark.timeout(2)
def test_testcase_name(self):
    time.sleep(3)
    assert 1 == 1
```

- **@pytest.mark.flaky(重试次数，间隔时间)**

  前提：依赖 pytest-rerunfailures

  用途：如果测试失败则重试指定次数。只要有一次通过，则认为测试成功

  示例：

```
import pytest

a = 0

@pytest.mark.flaky(reruns=5, reruns_delay=1) # 重试 5 次，间隔等待 1 秒
def test_testcase_name(self):            # 该 case 的结果为成功
    global a
    a = a +1
    if a >= 2:
        assert True
    else:
        assert False
```

- **@pytest.fixture()**

  用途：可用于设置测试用例的前置、后置操作

  示例：

```
import pytest

@pytest.fixture()
def login():
    print('登录相关操作')
    yield
    print('系统退出登录')

class Test_Login():
    def test_testcase_name_001(self, login):       #先执行登录，再执行 case，最后退出登录
        print('这个 case 需要登录')

    def test_testcase_name_002(self):              # 只运行 case
        print('这个 case 不需要登录')
    def test_testcase_name_003(self, login):       #先执行登录，再执行 case，最后退出登录
        print('这个 case 也需要登录')

# 运行类的 fixture
import pytest

@pytest.fixture()
def login():
    print('登录相关操作')
    yield
    print('系统退出登录')

@pytest.mark.usefixtures("login") # 先执行登录，再执行所有 class 下的 case，最后退出登录
class Test_Login():
    def test_testcase_name_001(self):
        print('运行 test_testcase_name_001')
    def test_testcase_name_002(self):
        print('运行 test_testcase_name_002')
    def test_testcase_name_003(self):
        print('运行 test_testcase_name_003')
# 相互调用的例子
```

```
import pytest

@pytest.fixture()
    print('登录相关操作')
    yield
    print('系统退出登录')

@pytest.fixture()
def log(login):
    print('开启日志功能')
    yield
    print('关闭日志功能')

class Test_Login():
    def test_testcase_name_001(self, log):          # 1.登录 2.打开日志 3.执行用例 4.关闭日志 5.退出登录
        print('运行 test_testcase_name_001')
    def test_testcase_name_002(self):               # 只运行 case
        print('运行 test_testcase_name_002')
    def test_testcase_name_003(self, log):          # 1.登录 2.打开日志 3.执行用例 4.关闭日志 5.退出登录
        print('运行 test_testcase_name_003')
# scope = 'function' -> 所有文件的测试用例执行前都会执行一次
# scope = 'class' -> 测试文件中测试类执行前都会执行一次
# scope = 'module' -> 每一个模块*.py 文件执行前都会执行一次
# scope = 'session' -> 所有测试文件执行前执行一次
# 如果 scope='session',不要像 function，class，module 一样，与 case 放在一起，需放在 conftest.py 下
# autouse = True 所有符合需求的 scope 自动被运行
import pytest

@pytest.fixture(scope = 'function'， autouse = True)
def login():
    print('登录相关操作')
    yield
    print('系统退出登录')

class Test_Login():
```

```
    def test_testcase_name_001(self):                #先执行登录，再执行 case，最后退出登录
        print('这个 case 需要登录')
    def test_testcase_name_002(self):                #先执行登录，再执行 case，最后退出登录
        print('这个 case 也需要登录')
```

运行规则

- 直接 pytest 只要符合用例的命名全都执行
- 使用 pytest -v 可以打印出详细的运行日志
- 使用 pytest -s 可以打印出你在代码中的 print 的输出
- 使用 pytest -k " 类名 " 除了这个类下面的用例其他都执行
- 使用 pytest -k " 函数名 " 除了这个函数名下面的用例其他都执行
- 使用 pytest -x 只要用例执行失败脚本立即停止后面的用例不再运行
- 使用 pytest --maxfail = [数量] 当用例执行失败的个数到达你设定的数量脚本停止不再运行
- 使用 pytest -m 运行所有@pytest.mark.[标记名] 标记的用例其他的用例不运行

补充

conftest.py

概述：conftest.py 是 pytest 特有的本地测试配置文件，可把 hook 或 fixture 写在此文件中

注意事项：

- conftest.py 配置脚本名称是固定的，不能改名称
- 不需要 import 导入，测试文件运行前会自动执行当前目录下的 conftest.py 文件

常用 hook：

- pytest_configure
  - 在解析命令行选项后或导入其他 conftest 文件时调用
- pytest_sessionstart
  - 创建 Session 对象后、执行收集测试用例前调用
- pytest_collection_finish
  - 收集测试用例结束时调用
- pytest_runtestloop
  - 收集测试用例结束后、执行主运行测试循环
- pytest_runtest_makereport
  - 会在 setup、call、teardown 三个阶段调用
- pytest_sessionfinish
  - 整个测试运行完成后、将退出状态返回到系统之前

- pytest_terminal_summary
  - 测试运行结束后，向终端添加测试摘要
- pytest_unconfigure
  - 在退出测试过程前调用

示例：

```
import pytest

# 参考文档：https://docs.pytest.org/en/stable/_modules/_pytest/hookspec.html

def pytest_configure(config):
    print("插件初始化代码")

def pytest_sessionstart(session):
    print("测试会话开始")

def pytest_collection_finish(session):
    print("测试收集结束")

def pytest_runtestloop(session):
    print("测试循环")

@pytest.hookimpl(hookwrapper=True, tryfirst=True)
def pytest_runtest_makereport(item, call):
    # 对于给定的测试用例(item)和调用步骤(call)，返回一个测试报告对象
(_pytest.runner.TestReport)
    out = yield
    report = out.get_result()
    if call.when == "setup":
        # 测试前置阶段
        pass
    elif call.when == "call":
        # 测试阶段
        # 打印测试用例的名称和结果
        print("id {} result {}".format(report.nodeid,report.outcome))
        # 如果测试结果是失败，打印原因
        if report.outcome == "failed":
            print("reason {}".format(report.longrepr))
    elif call.when == "teardown":
```

```
    # 测试结束阶段
    pass

def pytest_sessionfinish(session, exitstatus):
    print("测试会话结束，退出状态:", exitstatus)

def pytest_terminal_summary(terminalreporter, exitstatus, config):
    passed = len(terminalreporter.stats.get("passed", []))
    failed = len(terminalreporter.stats.get("failed", []))
    skipped = len(terminalreporter.stats.get("skipped", []))
    print("测试结果: 通过={}, 失败={}, 跳过={}".format(passed, failed, skipped))

def pytest_unconfigure(config):
    print("插件清理")
```

pytest.ini

概述：pytest 的配置文件

注意事项：

- pytest.ini 配置文件名称是固定的，不能改名称
- 不需要 import 导入，测试文件运行前会自动执行当前目录下的 pytest.ini 文件

常用参数：

- markers
  - 作用：测试用例中添加了 @pytest.mark.标签名 装饰器，如果不在 pytest.ini 中添加 markers 选项的话，就会报 warnings
- xfail_strict
  - 作用：设置 xfail_strict = True 可以让被@pytest.mark.xfail 修饰的，结果显示为 XPASS 的测试用例被报告为失败
- testpaths
  - 作用：定义 pytest 运行的测试目录
- norecursedirs
  - 作用：将指定目录或文件排除为测试源，多个路径用空格隔开
- addopts
  - 作用：设置命令行选项
- python_files
  - 作用：更改 pytest 关于文件名的约束规则
- python_classes
  - 作用：更改 pytest 关于测试类的约束规则
- python_functions
  - 作用：更改 pytest 关于测试函数的约束规则

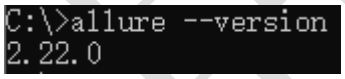
示例：

```
[pytest]
markers =
    module: this is module
    function: this is function

xfail_strict = True

testpaths = ./testcase_dir

norecursedirs = .git build

addopts = -s --alluredir=./allure-results --clean-alluredir

python_files =     test_*  *_test  test*
python_classes =   test*   test*
python_functions = test_*  test*
```

allure

环境安装

1. 从 github 上下载 allure（https://github.com/allure-framework/allure2/releases）后解压，将解压后的 bin 路径配置到环境变量中
2. 在 cmd 窗口中运行 allure --version，如果显示版本号则表示 allure 已经安装、环境已经配置



3. 安装 pytest 插件

```
pip install pytest
pip install allure-pytest
```

常用接口

- **@allure.epic(信息)**

用途：结构化 allure 的测试报告，是测试报告中的最顶级根目录

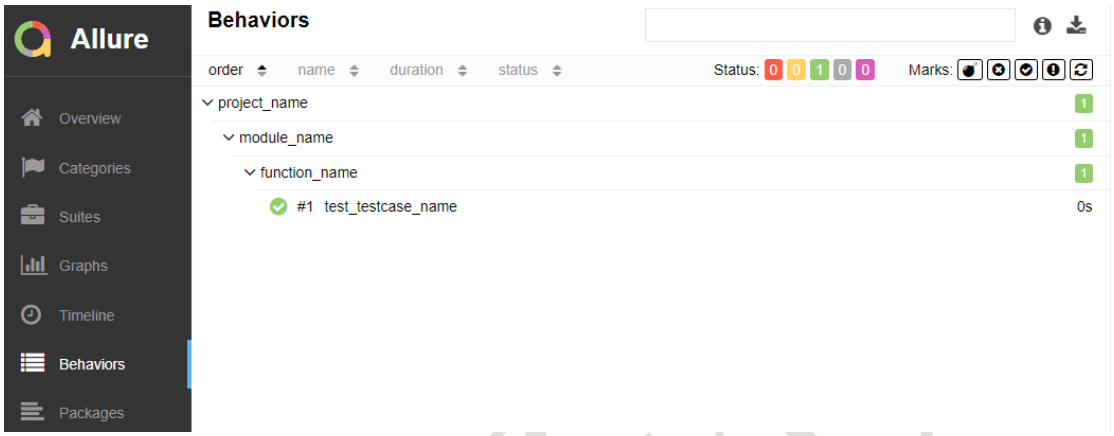
epic>feature>story>title,常用于表示项目名称

示例：

```
import pytest
import allure

@allure.epic("project_name")
```

```
@allure.feature("module_name")
class Test_module_name:
    @allure.story("function_name_")
    @allure.title('test_testcase_name')
    def test_testcase_name(self):
        pass
```

测试报告页面效果如下：



- **@allure.feature(模块名)**

  用途：结构化 allure 的测试报告，是测试报告中的次级根目录 epic>feature>story>title,常用于表示模块名称

  示例：同@allure.epic()

- **@allure.story(功能名)**

  用途：结构化 allure 的测试报告，是测试报告中的次次级根目录 epic>feature>story>title,常用于表示功能名称

  示例：同@allure.epic()

- **@allure.title(测试用例 ID)**

  用途：结构化 allure 的测试报告，是测试报告中的最底层文件 epic>feature>story>title,常用于表示测试用例的 id

  示例：同@allure.epic()

- **@allure.description(测试用例简述)**

  用途：在测试结果报告页面增加该用例的简述

  示例：

```
import pytest
import allure

@allure.epic("project_name")
@allure.feature("module_name")
class Test_module_name:
    @allure.story("function_name")
```

```
    @allure.title('test_testcase_name')
    @allure.description("description 是对测试用例的简述")
    @allure.testcase("https://www.gitlab.com", name="用例管理系统")
    @allure.issue("https://jira.yunxi.tv/login.jsp", name="缺陷系统")
    @allure.link("https://www.google.com", name="相关链接")
    @allure.severity(allure.severity_level.CRITICAL)
    def test_testcase_name(self):
        result = False
        with allure.step("步骤 1：初始化环境"):
            pass
        with allure.step("步骤 2：执行操作步骤"):
            pass
        with allure.step("步骤 3：判断测试结果"):
            # 如果执行失败，则增加日志附件
            if not result:
                allure.attach("用于测试", name="测试日志_1",
attachment_type=allure.attachment_type.TEXT)
                allure.attach.file(r"E:\log.txt", name="测试日志_2",
attachment_type=allure.attachment_type.TEXT)
            assert result
```

测试报告页面效果如下：

- **@allure.testcase(链接，名称)**

  用途：在测试结果报告页面增加该用例相关的链接，常用于关联测试用例存放地址

  示例：同@allure.description()

- **@allure.issue(链接，名称)**

  用途：在测试结果报告页面增加该用例相关的链接，常用于关联缺陷系统中的缺陷编号

  示例：同@allure.description()

- **@allure.link(链接，名称)**

  用途：在测试结果报告页面增加该用例相关的链接，常用于关联与该用例相关的其他信息

  示例：同@allure.description()

- **@allure.severity(用例等级)**

  用途：在测试结果报告页面增加该用例相关的等级，用于表明该用例的重要性

  示例：同@allure.description()

- **allure.attach(消息，显示名称，消息所属类型)**

  用途：在测试结果报告页面增加该用例相关的附件内容

  示例：同@allure.description()

- **allure.attach.file(文件路径，显示名称，文件类型)**

  用途：在测试结果报告页面增加该用例相关的附件文件

  示例：同@allure.description()

- **allure.step(操作步骤)**

  用途：在测试结果报告页面增加该用例相关的操作步骤

  示例：同@allure.description()

运行规则

- 使用 pytest --alluredir 指定 allure 运行日志的保存路径
- 使用 pytest --clean-alluredir 清空保存的 allure 历史运行日志
- 使用 allure generate -c -o allure-reports 通过当前目录下的 allure 运行日志生成报告 html

测试脚本编写

测试用例一般会有以下 9 个基础要素：

1. 被测模块：用于表示被测件的名称
2. 用例编号：用于自动化测试平台、质量管理平台的引用 。辅助测试人员快速定位至具体的单条测试用例
3. 需求编号：将测试用例与需求关联起来。当产品需求发生变更时，可以快速锁定涉及用例，进行更新
4. 测试简述：用于概述测试任务，辅助阅读者理解该条测试的目标
5. 用例优先级：用于表述测试用例的重要级别。是测试计划安排、提交问题严重度的参考标准之一
6. 测试环境：是对进行测试时软、硬件环境的描述
7. 测试步骤：依据需求文档中被测件功能相关输入参数，所编写的相关测试操作
8. 预期结果：需求文档中被测件的预期输出结果
9. 测试类型：例如"smoke"、"happy path"、"full test"，用于满足不同项目节点对测试报告覆盖度的需求

测试脚本代码层面的需求

要素名称	代码需求
被测模块	使用"@pytest.mark"及”@allure.feature“记录被测件模块
用例编号	测试脚本中函数名使用“test_用例编号”格式标记用例编号
需求编号	● 使用"@pytest.mark"标记用例相关联的需求编号 ● 使用"@allure.story"标记用例相关的需求名称

测试简述	使用"@allure.description"记录该测试用例的简述
用例优先级	使用”@allure.severity"标记用例的优先级 ■ blocker：阻塞缺陷（功能未实现，无法下一步） ■ critical：严重缺陷（功能点缺失 ■ normal： 一般缺陷（边界情况，格式错误） ■ minor：次要缺陷（界面错误与 ui 需求不符） ■ trivial： 轻微缺陷（必须项无提示，或者提示不规范）
测试环境	使用”@pytest.fixture"实现测试环境的准备
测试步骤	使用"allure.step"来记录测试步骤
预期结果	● 期望结果如果不是异常（Exception 类），则应使用“assert”做断言，并且必须为断言不通过的结果添加说明信息 ● 期望结果如果是异常（Exception 类），则应使用”pytest.assert“做断言
测试类型	使用"@pytest.mark"标记用例的类型

示例

以下代码表示有两个被测件，模块名分别为 module_A 与 module_B

module_A（有两个功能 function_001 与 function_002）

function_001（有两条相关测试用例编号为 testcase_name_001 与 testcase_name_002）

- 需求编号 id 为 functionid_1_1

testcase_name_001

- 测试优先级为 blocker
- 测试类型为 smoke_test 测试

testcase_name_002

- 测试优先级为 minor
- 测试类型为 full_test 测试

function_002（有一条相关测试用例编号为 testcase_name_003）

- 需求编号 id 为 functionid_2_1

testcase_name_003

- 测试优先级为 blocker
- 测试类型为 smoke_test 测试

module_B（有一个功能 function_003）

function_003（有一条相关测试用例编号为 testcase_name_004）

- 需求编号 id 为 functionid_3_1

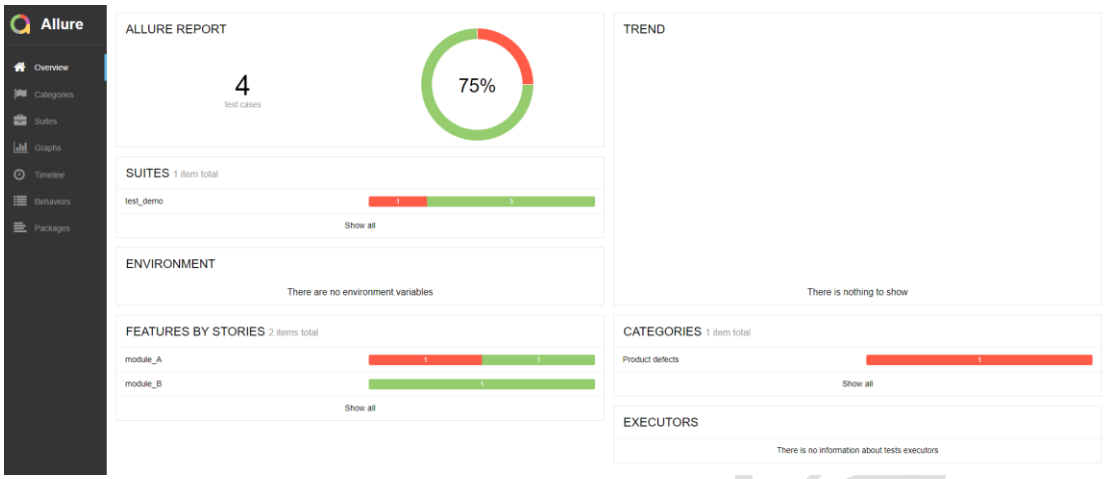
testcase_name_004

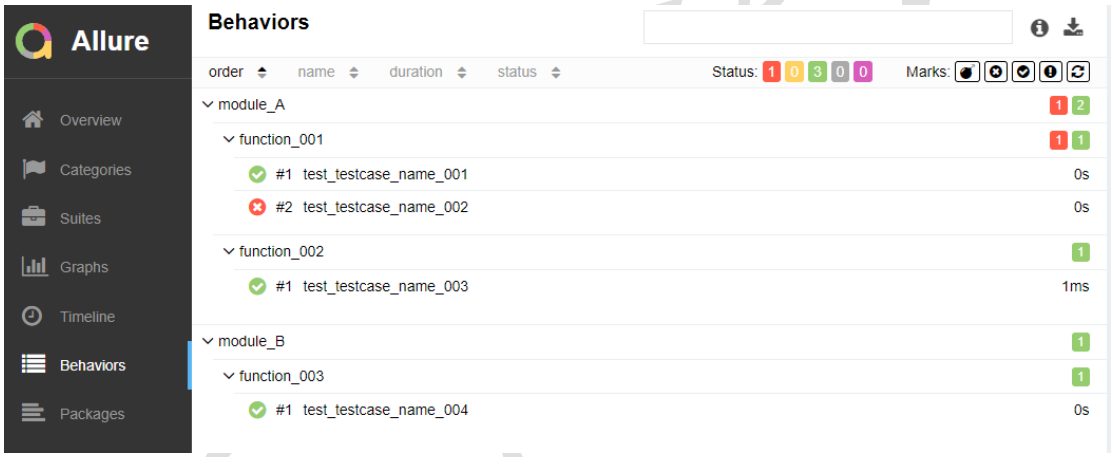
- 测试优先级为 trivial
- 测试类型为 full_test 测试

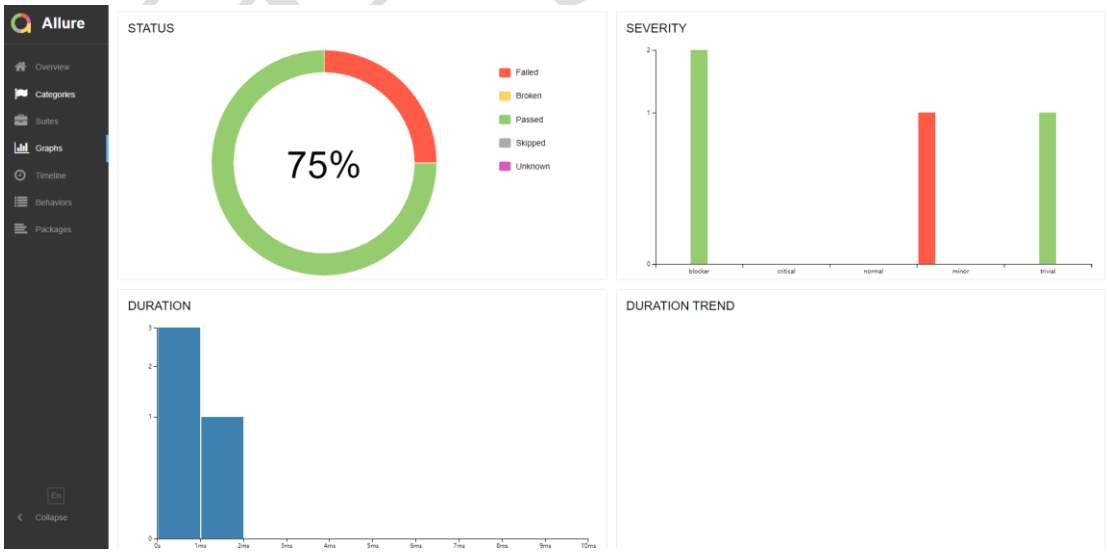
```
import allure
import pytest

@pytest.mark.module_A
@allure.feature("module_A")
class Test_module_A:
    @pytest.mark.function_001
    @allure.story("function_001")
    @pytest.mark.functionid_1_1
    @allure.severity(allure.severity_level.BLOCKER)
    @pytest.mark.smoke_test
    @allure.title('test_testcase_name_001')
    @allure.description("测试用例 testcase_name_001 的概述")
    def test_testcase_name_001(self):
        with allure.step("步骤 1：初始化环境"):
            pass
        with allure.step("步骤 2：执行操作步骤"):
            pass
        with allure.step("步骤 3：判断测试结果"):
            assert True

    @pytest.mark.function_001
    @allure.story("function_001")
    @pytest.mark.functionid_1_1
    @allure.severity(allure.severity_level.MINOR)
    @pytest.mark.full_test
    @allure.title('test_testcase_name_002')
    @allure.description("测试用例 testcase_name_002 的概述")
    def test_testcase_name_002(self):
        with allure.step("步骤 1：初始化环境"):
            pass
        with allure.step("步骤 2：执行操作步骤"):
            pass
        with allure.step("步骤 3：判断测试结果"):
            assert False
```

```
    @pytest.mark.function_002
    @allure.story("function_002")
    @pytest.mark.functionid_2_1
    @allure.severity(allure.severity_level.BLOCKER)
    @pytest.mark.smoke_test
    @allure.title('test_testcase_name_003')
    @allure.description("测试用例 testcase_name_003 的概述")
    def test_testcase_name_003(self):
        with allure.step("步骤 1：初始化环境"):
            pass
        with allure.step("步骤 2：执行操作步骤"):
            pass
        with allure.step("步骤 3：判断测试结果"):
            assert True

@pytest.mark.module_B
@allure.feature("module_B")
class Test_module_B:
    @pytest.mark.function_003
    @allure.story("function_003")
    @pytest.mark.functionid_3_1
    @allure.severity(allure.severity_level.TRIVIAL)
    @pytest.mark.full_test
    @allure.title('test_testcase_name_004')
    @allure.description("测试用例 testcase_name_004 的概述")
    def test_testcase_name_004(self):
        with allure.step("步骤 1：初始化环境"):
            pass
        with allure.step("步骤 2：执行操作步骤"):
            pass
        with allure.step("步骤 3：判断测试结果"):
            assert True
```

测试报告示意图


Allure
Overview
Categories
Suites
Graphs
Timeline
Behaviors
Packages
ALLURE REPORT
4
test cases
75%
TREND
There is nothing to show
SUITES 1 item total
test_demo
Show all
ENVIRONMENT
There are no environment variables
FEATURES BY STORIES 2 items total
module_A
module_B
Show all
CATEGORIES 1 item total
Product defects
Show all
EXECUTORS
There is no information about tests executors


Allure
Overview
Categories
Suites
Graphs
Timeline
Behaviors
Packages
Behaviors
order
name
duration
status
Status: 1 0 3 0 0
Marks:
module_A 1 2
function_001 1 1
#1 test_testcase_name_001 0s
#2 test_testcase_name_002 0s
function_002 1
#1 test_testcase_name_003 1ms
module_B 1
function_003 1
#1 test_testcase_name_004 0s


Allure
Overview
Categories
Suites
Graphs
Timeline
Behaviors
Packages
Collapse
STATUS
75%
Failed
Broken
Passed
Skipped
Unknown
SEVERITY
blocker
critical
normal
minor
trivial
DURATION
DURATION TREND