Pytest 自动化测试框架简介

为什么要采用自动化测试

自动化测试在后端开发中常常被人忽视: 一个功能, 能跑就行——至于代码是否正确，等到前端对接的同学汇报问题后，又懊悔自己当初怎么没注意到这个点。

如果你对一个接口的验证还在 Postman 一条一条手动输入验证, 或者在浏览器里不断发起各种请求: 那么自动化测试就很有必要了。

随着项目不断壮大, 自动化(测试)带来的好处会越来越多, 包括但不限于:

- 对代码的**正确性**抱有信心
- 更**快**的开发流程(减少了手动测试的重复劳动)
- **不怕**代码改动带来的 bug(在回归测试中辨别)
- 另一种形式的**文档**(随功能加入不断更新)

有一种**测试驱动开发**的开发范式, 在定义好接口后先完成对测试代码的编写, 再开始开发.

自动化测试我们通常分成以下三类:

- **单元测试**：关注的是软件的最小可测试部分，通常是单个函数、方法或类。
- **集成测试**：在单元测试的基础上，检查多个单元或组件如何一起工作。当不同的软件单元组合在一起时，可能会产生单元测试无法发现的接口问题。
- **功能测试**（也称为系统测试或端到端测试）：关注的是软件的整体功能，即软件作为一个整体是否能够满足用户的需求和期望。

本文档主要介绍的是 Python 的 Pytest 自动化测试框架.

Unittest or Pytest

Python 的单元测试框架最常用的两者是 `unittest`(内置) 和 `pytest`. Jetbrain 曾在 2023 年做过一次开发者调查, Pytest 在榜上还有一席之地.

比较一下 `unittest` 和 `pytest` 在使用上的差异:

```python
# pytest
def test_something():
    x = 3
    assert x == 4
```

你只需要 `pytest` 一下就可以自动运行这些测试用例(你甚至不需要注册它们, `pytest` 会自动 `discover` 它们)!

`unittest` 在设计上借鉴了 Java 的 `java.junit`, 它看起来是这样的:

```python
import unittest
class TestSomething(unittest.TestCase):
    def test_something(self):
        x = 3
        self.assertEqual(x, 4)

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

huh... `pytest` 除了比 `unittest` 看起来没那么啰嗦以外, 它还能:

- 提供更清晰的错误信息: 测试失败得到的信息更可读, 同时可以通过更细致的选项控制
- 参数化: 如在不同数据集上执行对单个单元的测试

· 富插件架构: 高度自定义化

快速开始

安装

创建虚拟环境并激活虚拟环境

```
python -m venv .venv
.venv\Scripts\activate # powershell
```

用 `pip` 安装 `pytest`

```
pip install pytest
```

CLI 使用

`pytest` 的命令行工具会自动检测当前目录下是否有 `test_*.py` 文件, 并运行所有 `test_` 开头的函数/类:

```
pytest
```

你也可以指定测试文件:

```
pytest tests/test_something.py
```

运行 `tests` 目录下(递归寻找全部可行的)全部的测试用例

```
pytest tests/
```

断言重写

```
# failure_demo.py
def test_eq_text():
    assert 'spam' == 'eggs'

def test_eq_similar_text():
    assert 'foo 1 bar' == 'foo 2 bar'

def test_eq_long_text():
    a = '1'*100 + 'a' + '2'*100
    b = '1'*100 + 'b' + '2'*100
    assert a == b

def test_eq_list():
    assert [0, 1, 2] == [0, 1, 3]

def test_eq_dict():
    assert {'a': 0, 'b': 1} == {'a': 0, 'b': 2}

def test_eq_set():
    assert {0, 10, 11, 12} == {0, 20, 21}

def test_eq_longer_list():
    assert [1, 2] == [1, 2, 3]

def test_not_in_text_single():
    text = 'single foo line'
    assert 'foo' not in text
```

后端部 BACK END DEPARTMENT

pytest 对 assert 语句进行了重写. 正常来说, Python 的 assert 在不满足断言条件时会抛出 AssertionError:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/besthope/pylab/pytest-tutorial/basic/failure_demo.py", line 12, in
test_eq_long_text
    assert a == b
AssertionError
```

而具体的报错原因, 它只定位到了 a == b, 而具体程序运行时的上下文信息全无, 对于我们调试毫无作用!

而 pytest 通过某种复杂的机制, 除了爆出 AssertionError 外, 还会把它包装成 Failed 异常, 并显示出更详细的错误信息. 你可以用 pytest 运行测试文件:

```
pytest failure_demo.py
```

pytest 的报错消息甚至可以具体到第几个键值对的值对应不上:

```
    def test_eq_dict():
>       assert {'a': 0, 'b': 1} == {'a': 0, 'b': 2}
E       AssertionError: assert {'a': 0, 'b': 1} == {'a': 0, 'b': 2}
E
E         Omitting 1 identical items, use -vv to show
E         Differing items:
E         {'b': 1} != {'b': 2}
E         Use -v to get more diff
```

这里的 assert 可以看成是 unittest 中的 assertEqual, 但 assert 语句比 assertXX 更加灵活:

```
def test_assert_introspection():
    x = y = 0              # with unittest.py
    assert x               # self.assertTrue(x)
    assert x == 1          # self.assertEqual(x, 1)
    assert x != 2          # self.assertNotEqual(x, 2)
    assert not x           # self.assertFalse(x)
    assert x < 3 or y > 5  # ?
```

常用测试选项

- -v: verbose, 显示更详细的输出消息
- -vv: very verbose, 显示更详细的输出消息
- --tb: traceback
  - 如果 --tb=no 表明不显示 traceback 信息
  - 如果 --tb=line 表明只显示 traceback 信息的最后一行
- --setup-show: 显示执行顺序
- -s: 允许打印语句

接口使用

通过在程序中添加 pytest 提供的接口, 我们可以控制更细致的测试.

手动断言

你可以手动调用 pytest.fail 来使得该测试用例失败, 从而实现某种意义上的手动断言:

```
import pytest
from dataclasses import dataclass, field
```

```
@dataclass
class Student:
    name: str
    id: int = field(default=None, compare=False)

def assert_identical(s1: Student, s2: Student):
    __tracebackhide__ = True
    assert s1 == s2
    if s1.id != s2.id:
        pytest.fail(f"ID don't match: {s1.id} != {s2.id}")

class TestEquality:
    def test_identical():
        s1 = Student("Alice", 1)
        s2 = Student("Alice", 1)
        assert_identical(s1, s2)

    def test_identical_fail():
        s1 = Student("Alice", 1)
        s2 = Student("Alice", 2)
        assert_identical(s1, s2)
```

用 `pytest.raises` 触发已知的 Exception 而不是 `AssertionError`:

```
import pytest

def divide(x, y):
    return x / y

def test_raises():
    with pytest.raises(ZeroDivisionError):
        divide(3, 0)
```

依赖注入: Fixtures

Fixture 是 pytest 最核心的机制之一, 它可以帮助我们创建测试用例所需的资源, 并在测试用例执行前或执行后自动释放资源. 它能用一种简洁的方式做到 `unittest` 中的 `setup` 与 `teardown`.

所谓的“依赖注入”是指，将组件的依赖关系（如服务、工具类、数据源等）的创建和维护工作交给外部容器或框架来管理，而不是在组件内部直接创建:

```
# dependecy injection
class Sender:
    def __init__(self, server):
        self.server = server

server = UDPServer()
sender = Sender(server)
```

```
# no dependency injection
class Sender:
    def __init__(self):
        self.server = UDPServer()

sender = Sender()
```

这种做法非常适合对象编程的思想.

在 Web 框架里, 对一个创建对象的请求, 我们通常会在测试用例中创建数据库连接, 并在测试用例执行后关闭连接. 利用 `fixture`, 我们可以这样完成:

```
@pytest.fixture()
def db():
    conn = sqlite3.connect(":memory:") # setup
    yield conn
```

```
    conn.close() # teardown


def test_db(db):
    # GIVEN a database connection
    cur = db.cursor()
    # WHEN we execute a query
    cur.execute(query)
    # THEN we should see the expected result
    expected = ...
    assert cur.fetchone() == expected
```

```
$ pytest --setup-show
```

不同的测试用例可以使用相同的 `fixture`, 只要保证这个 fixture 可以正确识别到.

fixture 默认名称即函数名, 当然你也可以手动命名 fixture:

```
@pytest.fixture(name="ultimate_answer")
def ultimate_answer_fixture():
    return 42
```

通常来说, `fixture` 会存放在 `conftest.py` 文件中, `pytest` 会自动导入这个文件里的 fixtures, 这样在同一个模块下的测试用例都可以共享这个 fixture.

当然你可以手动指定一个 fixture 的**作用域**(scope), 默认为函数级(`function`), 例如你不想一个 fixture 在一次测试**会话**(session)中被重复执行(例如创建对象), 可以指定

```
@pytest.fixture(scope="session")
def some_students():
    return [
        ...
    ]
```

在每个测试类的每个方法之前执行一次

```
@pytest.fixture(scope="class")
def some_students():
    return [
        ...
    ]
```

此外, fixture 可以自动导入到当前的作用域下. 可以开启 `autouse` 选项

```
@pytest.fixture(autouse=True, scope="session")
def footer_session_scope():
    """Report the time at the end of a session."""
    yield
    now = time.time()
    print("--")
    print(
    "finished : {}".format(
    time.strftime("%d %b %X", time.localtime(now))
    )
    )
    print("-----------------")


@pytest.fixture(autouse=True)
def footer_function_scope():
```

```
    """Report test durations after each function."""
    start = time.time()
    yield
    stop = time.time()
    delta = stop - start
    print("\ntest duration : {:0.3} seconds".format(delta))
```

```
pytest -s -v
```

```
test_autouse.py::test_empty PASSED
test duration : 0.00241 seconds
--
finished : 27 Oct 17:06:26
-----------------
```

模拟

pytest 框架内置一些非常常用的 fixture, 其中之一是 monkeypatch.

mock (模拟)是自动化测试中很重要的一个概念, 它用于在测试过程中临时修改或替换代码的一部分(patch), 模拟依赖项来测试行为.

利用 mock 可以避免在测试中产生副作用，例如发送真正的邮件或进行网络请求，同时也可以提高测试的运行速度，因为避免了与外部系统的交互。

例如, 对密码验证逻辑的验证, 我们需要依赖外部 IO 输入 getpass. 但是在测试中我们希望“模拟”一组用户输入, 用一个固定的字符串函数代替 getpass, 那么采用 monkeypatch 来更改这部分的代码:

```
import pytest
import getpass

def validate_password():
    return getpass.getpass() == "super-secure" #...not

def test_validate_password_good(monkeypatch: pytest.MonkeyPatch):
    monkeypatch.setattr(getpass, "getpass", lambda: "super-secure")
    assert validate_password()

def test_validate_password_bad(monkeypatch: pytest.MonkeyPatch):
    monkeypatch.setattr(getpass, "getpass", lambda: "wrong-password")
    assert not validate_password()
```

软件工程的一大重要概念是“为测试而设计”(Design for Testability)，也就是要让设计出的代码易于测试。如果你的代码可以通过 mock 大幅简化测试流程的话，那么程序的解耦性自然就得已规范了。

创建临时文件目录

内置 fixture: tmp_path 和 tmp_path_factory

```
def test_tmp_path(tmp_path):
    file = tmp_path / "file.txt" # return a pathlib.Path object
    file.write_text("Hello")
    assert file.read_text() == "Hello"

def test_tmp_path_factory(tmp_path_factory):
    path = tmp_path_factory.mktemp("sub")
    file = path / "file.txt"
```

```
    file.write_text("Hello")
    assert file.read_text() == "Hello"
```

标记

mark 是一个用于标记测试用例的装饰器，它允许你根据标记来选择性地运行测试，或者对测试用例进行分类。

一些内置的 marker 有:

- skip: 跳过某些测试用例
- skipif: 跳过某些条件的测试用例
- xfail: 标记测试用例为期望失败
- parametrize: 参数化测试用例

跳过/预期失败

为什么要用到 skip 和 xfail 这类的标记? 看起来是在做无用功, 但是对于一个有着明确版本管理的项目, 这些标记可以帮助我们预先确定功能实现:

```
@pytest.mark.skip(reason="Student doesn't support < comparison yet")
def test_less_than():
    s1 = Student("Alice")
    s2 = Student("Bob")
    assert s1 < s2 # implement < operator later
```

更细致的, 通过判断版本号来控制什么时候需要运行测试用例, 什么时候跳过:

```
@pytest.mark.skipif(
parse(cards.__version__).major < 2,
reason="Student < comparison not supported in 1.x",
)
def test_less_than():
    s1 = Student("Alice")
    s2 = Student("Bob")
    assert s1 < s2
```

xfail 标记为预期失败的. 如果你有一个接口发现了 BUG, 但你不知道该如何修复, 你可以写一条 xfail 的单元测试来帮助他人复现这个问题.

当之后版本更新后, 你可能会发现这条单元测试变成了 XPASSED 状态: 你预计它失败, 但它通过了. 就表明旧版本的 BUG 被修复了.

这些标记对于测试驱动开发(TDD)的范式很有用!

自定义标记

帮助我们对测试用例分类:

```
import pytest
import time

# marking/test_marking.py

@pytest.mark.slow
@pytest.mark.webtest
def test_slow_api():
    time.sleep(1)

@pytest.mark.webtest
```

```
def test_api():
    pass

def test_fast():
    pass
```

测试带有 webtest 标签的用例(-m 选项):

```
pytest -m "webtest"
```

```
========================= test session starts ==========
collected 15 items / 13 deselected / 2
selected

test_marking.py ..
```

你还可以组合使用(用 and/or/not 关键字)

```
pytest -m "slow and webtest" -v
```

```
========================= test session starts ==========
test_marking.py::test_slow_api PASSED
[100%]

=================== 1 passed, 14 deselected ===================
```

参数化

我们编写测试函数一般遵循着:

- Given/Arrange: 准备测试环境
- When/Act: 触发测试事件
- Then/Assert: 验证测试结果

这样的三步流程. 但, 很多情况下, 我们只会更换测试用例, 而后续流程相同, 就带来了很多重复的代码:

```
# card/carddb implementation
import pytest
import dataclasses

@dataclasses.dataclass
class Card:
    content: str
    state: str

class CardDatabase:
    def __init__(self):
        self.cards = []

    def add_card(self, card):
        self.cards.append(card)
        return len(self.cards) - 1

    def finish(self, idx):
        self.cards[idx].state = "done"

    def get_card(self, idx):
        return self.cards[idx]
```

```python
# test_finish.py
def test_finish_in_prog(db: CardDatabase):
    index = db.add_card(Card("second version", state="in prog"))
    db.finish(index)
    card = db.get_card(index)
    assert card.state == "done"


def test_finish_from_done(db: CardDatabase):
    index = db.add_card(Card("write a book", state="done"))
    db.finish(index)
    card = db.get_card(index)
    assert card.state == "done"


def test_finish_from_todo(db):
    index = db.add_card(Card("create a course", state="todo"))
    db.finish(index)
    card = db.get_card(index)
    assert card.state == "done"
```

一种解决方案是采用 for loop 来缩略代码

```python
def test_finish(db):
    for c in [
        Card("write a book", state="done"),
        Card("second edition", state="in prog"),
        Card("create a course", state="todo"),
    ]:
        index = db.add_card(c)
        db.finish(index)
        card = db.get_card(index)
        assert card.state == "done"
```

但在测试汇报中测试用例数的信息丢失了:

```
========================== test session starts ==========================
collected 1 item
test_finish_combined.py . [100%]
=========================== 1 passed in 0.01s ===========================
```

为了解决这个问题, 可以使用 `@pytest.mark.parametrize` 这个标记

```python
@pytest.mark.parametrize(
    "start_summary, start_state",
    [
        ("write a book", "done"),
        ("second edition", "in prog"),
        ("create a course", "todo"),
    ],
)
def test_finish(db, start_summary, start_state):
    initial_card = Card(summary=start_summary, state=start_state)
    index = db.add_card(initial_card)
    db.finish(index)
    card = db.get_card(index)
    assert card.state == "done"
```

- `db` 作为 fixture 注入

- 此外还有 mark 注入的额外两个参数
- 就可以编写“通用”测试

```
$ pytest -v test_func_param.py::test_finish
======================== test session starts ========================
collected 3 items
test_func_param.py::test_finish[write a book-done] PASSED [ 33%]
test_func_param.py::test_finish[second edition-in prog] PASSED [ 66%]
test_func_param.py::test_finish[create a course-todo] PASSED [100%]
========================= 3 passed in 0.05s =========================
```

你还可以对 fixture 参数化:

```
@pytest.fixture(params=["done", "in prog", "todo"])
def start_state(request):
    return request.param


def test_finish(db, start_state):
    c = Card("write a book", state=start_state)
    index = db.add_card(c)
    db.finish(index)
    card = db.get_card(index)

    assert card.state == "done"
```

`params` 会导致对 fixture 的多次调用, 每次的参数会存放到 `request.param` 里(`request` 也是一个内置 fixture).

更复杂的参数化还可以使用 `pytest_generate_tests`. 具体大家可以查看文档.

Flask 的测试

利用 `fixture` 创建测试用数据库、客户端:

```
@pytest.fixture(scope="session")
def app():
    load_dotenv(".testenv")
    app = create_app(testing=True)
    return app

@pytest.fixture
def db(app : Flask):
    from app.extensions import db as _db
    _db.app = app

    with app.app_context():
        _db.create_all()

    yield _db

    _db.session.close()
    _db.drop_all()

@pytest.fixture
def client(app : Flask):
    return app.test_client()

@pytest.fixture
```

```
def runner(app: Flask):
    return app.test_cli_runner()
```

对接口测试

利用 `client` 发送请求并验证响应:

```
def test_request_example(client):
    response = client.get("/posts")
    assert b"<h2>Hello, World!</h2>" in response.data
```

推荐利用 `url_for` 获取接口的 URL, 避免代码重构带来的错误.

对 client 发起请求的操作类似于 `requests` 库的做法, 例如传入表单数据只需要设置 `data` 参数, 查询字符串 `query_string={"key": "value", ...}`, json 请求 `json={"key": "value", ...}` 等.

```
from pathlib import Path

# get the resources folder in the tests folder
resources = Path(__file__).parent / "resources"

def test_edit_user(client):
    response = client.post("/user/2/edit", data={
        "name": "Flask",
        "theme": "dark",
        "picture": (resources / "picture.png").open("rb"),
    })
    assert response.status_code == 200
```

获取 session 以及修改 session:

```
from flask import session

def test_access_session(client):
    with client:
        client.post("/auth/login", data={"username": "flask"})
        # session is still accessible
        assert session["user_id"] == 1

    # session is no longer accessible
    # ...

def test_modify_session(client):
    with client.session_transaction() as session:
        # set a user id without going through the login route
        session["user_id"] = 1

    # session is saved now

    response = client.get("/users/me")
    assert response.json["username"] == "flask"
```

需要上下文的测试

```
def test_validate_user_edit(app):
    with app.test_request_context(
        "/user/2/edit", method="POST", data={"name": ""}
    ):
        # call a function that accesses `request`
        messages = validate_edit_user()
```

```
    assert messages["name"][0] == "Name cannot be empty."
```

参考最佳实践

参考一些大型的 flask 项目.

参考资料

pytest 文档: 官方文档

Python Testing with pytest: Simple, Rapid, Effective, and Scalable: 很好的一本书

后端部
BACK END DEPARTMENT