第五章 POM项目实战(上海-悠悠)

5.1 页面对象模型Page Object Models

POM(Page Object Models) 页面对象模型已经成了写 web 自动化的一个标准模型。

5.1.1 页面对象模型Page Object Models

页面对象代表 Web 应用程序的一部分。电子商务 Web 应用程序可能有一个主页、一个列表页面和一个结帐页面。它们中的每一个都可以由页面对象模型表示。
页面对象通过创建适合您的应用程序的更高级别的 API 来简化创作，并通过在一个地方捕获元素选择器和创建可重用代码来避免重复来简化维护。

官方示例

```
# models/search.py
class SearchPage:
    def __init__(self, page):
        self.page = page
        self.search_term_input = page.locator(
            '[aria-label="Enter your search term"]'
        )

    def navigate(self):
        self.page.goto("https://bing.com")

    def search(self, text):
        self.search_term_input.fill(text)
        self.search_term_input.press("Enter")
```

然后可以在测试中使用页面对象。

```
# test_search.py
from models.search import SearchPage

# in the test
page = browser.new_page()
search_page = SearchPage(page)
search_page.navigate()
search_page.search("search query")
```

官方文档总是那么简洁精悍，接下来看下具体的实例

5.1.2 使用实例

以测试注册页面为例

注册页面根据输入框的内容，我们可以写多个有效等价，无效等价的测试用例，那么这些用例的操作最终都是在页面上的几个固定元素上点点点。

整个项目结构

```
├─cases
│  │  test_register.py
│  │  __init__.py
├─models
│  │  register_page.py
│  │  __init__.py
│  │
├─conftest.py
├─pytest.ini
```

于是可以把注册页的元素定位和操作，封装成一个RegisterPage 类

```python
from playwright.sync_api import Page
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

class RegisterPage:

    def __init__(self, page: Page):
        self.page = page
        self.locator_username = page.get_by_label("用 户 名:")
        self.locator_password = page.get_by_label("密      码:")
        self.locator_register_btn = page.locator('text=立即注册')
        self.locator_login_link = page.locator('text=已有账号? 点这登录')
        # 用户名输入框提示语
        self.locator_username_tip1 = page.locator(
            '[data-fv-validator="notEmpty"][data-fv-for="username"]'
        )
        self.locator_username_tip2 = page.locator(
            '[data-fv-validator="stringLength"][data-fv-for="username"]'
        )
        self.locator_username_tip3 = page.locator(
            '[data-fv-validator="regexp"][data-fv-for="username"]'
        )
        # 密码输入框提示语
        self.locator_password_tip1 = page.locator(
            '[data-fv-validator="notEmpty"][data-fv-for="password"]'
        )
        self.locator_password_tip2 = page.locator(
            '[data-fv-validator="stringLength"][data-fv-for="password"]'
        )
        self.locator_password_tip3 = page.locator(
            '[data-fv-validator="regexp"][data-fv-for="password"]'
        )
        # 账号或密码不正确!
        self.locator_register_error = page.locator('text=用户名已存在或不合法! ')

    def navigate(self):
        self.page.goto("http://127.0.0.1:8000/register.html")

    def fill_username(self, username):
        self.locator_username.fill(username)

    def fill_password(self, password):
        self.locator_password.fill(password)

    def click_register_button(self):
        self.locator_register_btn.click()
```

```
    def click_login_link(self):
        self.locator_login_link.click()
```

conftest.py 前置和后置操作代码

```
from playwright.sync_api import sync_playwright
import pytest


@pytest.fixture(scope="session")
def context_chrome():
    p = sync_playwright().start()
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()
    yield context
    # 实现用例后置
    context.close()
    browser.close()
    p.stop()
```

用例部分

```python
from models.register_page import RegisterPage
from playwright.sync_api import expect
import pytest
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

class TestRegister:

    @pytest.fixture(autouse=True)
    def start_for_each(self, context_chrome):
        print("for each--start: 打开新页面访问注册页")
        self.page = context_chrome.new_page()
        self.register = RegisterPage(self.page)
        self.register.navigate()
        yield
        print("for each--close: 关闭注册页")
        self.page.close()

    def test_register_1(self):
        """用户名为空，点注册"""
        self.register.fill_username('')
        self.register.fill_password('123456')
        self.register.click_register_button()
        # 断言
        expect(self.register.locator_username_tip1)\
            .to_be_visible()
        expect(self.register.locator_username_tip1)\
            .to_contain_text("不能为空")

    def test_register_2(self):
        """用户名大于30字符"""
        self.register.fill_username('hello world hello world hello world')
        # 断言
        expect(self.register.locator_username_tip2)\
            .to_be_visible()
        expect(self.register.locator_username_tip2)\
            .to_contain_text("用户名称1-30位字符")
        # 断言 注册按钮不可点击
        expect(self.register.locator_register_btn)\
            .not_to_be_enabled()

    def test_register_3(self):
        """用户名有特殊字符"""
        self.register.fill_username('hello!@#')
        # 断言
        expect(self.register.locator_username_tip3)\
```

```
            .to_be_visible()
        expect(self.register.locator_username_tip3)\
            .to_contain_text("用户名称不能有特殊字符,请用中英文数字")
        # 断言 注册按钮不可点击
        expect(self.register.locator_register_btn)\
            .not_to_be_enabled()

    def test_register_4(self):
        """密码框不能为空"""
        self.register.fill_username('hello')
        self.register.fill_password('')
        self.register.click_register_button()
        # 断言
        expect(self.register.locator_password_tip1)\
            .to_be_visible()
        expect(self.register.locator_password_tip1)\
            .to_contain_text("不能为空")

    @pytest.mark.parametrize('test_input', ['abc12', 'abc1234567890abc1'])
    def test_register_5(self, test_input):
        """密码框6-16位"""
        self.register.fill_password(test_input)
        # 断言
        expect(self.register.locator_password_tip2)\
            .to_be_visible()
        expect(self.register.locator_password_tip2)\
            .to_contain_text("密码6-16位字符")

    def test_register_6(self):
        """密码框不能有特殊字符"""
        self.register.fill_password('abc123!')
        # 断言
        expect(self.register.locator_password_tip3)\
            .to_be_visible()
        expect(self.register.locator_password_tip3)\
            .to_contain_text("不能有特殊字符,请用中英文数字下划线")

    def test_login_link(self):
        """已有账号? 点这登录"""
        expect(self.register.locator_login_link)\
            .to_have_attribute("href", "login.html")
        self.register.click_login_link()
        expect(self.register.page).to_have_title('网站登录')

    def test_register_error(self):
        """测试注册正常流程， 已被注册过的账号"""
```

```
        self.register.fill_username('yoyo')
        self.register.fill_password('aa123456')
        self.register.click_register_button()
        # 断言提示语可见
        expect(self.register.locator_register_error).to_be_visible()

    def test_register_success(self):
        """测试注册正常流程， 注册成功"""
        # 生成随机账号
        import uuid
        self.register.fill_username(str(uuid.uuid4())[:8])
        self.register.fill_password('aa123456')
        self.register.click_register_button()
        # 断言提示语可见
        expect(self.register.page).to_have_title('首页')
```

整体下来感觉非常流畅，执行效率很高，很多断言方法都有了封装，很不错的体验。
pytest 自己不太会写的，playwright 官方还很贴心的提供了一个插件`pytest-playwright`专门用于写自动化测试用例。

5.2 pytest-playwright插件编写测试用例

pytest-playwright插件完美的继承了pytest 用例框架和playwright基础使用的封装，基本能满足工作中的常规需求了，不需要我们再做额外的插件开发。

5.2.1 pytest-playwright 环境准备

Playwright 建议使用官方的 pytest-playwright 插件来编写端到端测试。它提供上下文隔离，开箱即用地在多个浏览器配置上运行。或者，您可以使用该库使用您喜欢的测试运行程序手动编写测试基础设施。Pytest 插件使用 Playwright 的同步版本，还有一个可通过库访问的异步版本。

开始安装 Playwright 并运行示例测试以查看它的实际效果。

```
pip install pytest-playwright
```

安装所需的浏览器：

```
playwright install
```

仅需这一步即可安装所需的浏览器，并且不需要安装驱动包了,解决了selenium启动浏览器，总是要找对应驱动包的痛点。

5.2.2 快速开始

test_my_application.py 使用以下代码在当前工作目录或子目录中创建一个文件：

```
import re
from playwright.sync_api import Page, expect
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/

def test_homepage(page: Page):
    page.goto("https://playwright.dev/")

    # Expect a title "to contain" a substring.
    expect(page).to_have_title(re.compile("Playwright"))

    # create a locator
    get_started = page.get_by_role("link", name="Get started")

    # Expect an attribute "to be strictly equal" to the value.
    expect(get_started).to_have_attribute("href", "/docs/intro")

    # Click the get started link.
    get_started.click()

    # Expects the URL to contain intro.
    expect(page).to_have_url(re.compile(".*intro"))
```

默认情况下，测试将在 chromium 上运行。这可以通过 CLI 选项进行配置。
测试以无头模式运行，这意味着在运行测试时不会打开浏览器 UI。测试结果和测试日志将显示在终端中。

```
>pytest test_my_application.py
========== test session starts ==============
platform win32 -- Python 3.8.5, pytest-7.2.1, pluggy-1.0.0
rootdir: D:\demo\play_web\cases
plugins: base-url-2.0.0, playwright-0.3.0
collected 1 item

test_my_application.py .          [100%]

============ 1 passed in 2.40s ===
```

如果想看到打开的浏览器运行，可以加上`--headed`参数

```
>pytest test_my_application.py --headed
```

5.2.3 编写用例

Playwright 提供了`expect`将等待直到满足预期条件的功能。

```
import re
from playwright.sync_api import expect

expect(page).to_have_title(re.compile("Playwright"))
```

定位器是 Playwright 自动等待和重试能力的核心部分。定位器代表了一种随时在页面上查找元素的方法，并用于对诸如.click .fill 等之类的元素执行操作。

```
from playwright.sync_api import expect

get_started = page.get_by_role("link", name="Get started")

expect(get_started).to_have_attribute("href", "/docs/installation")
get_started.click()
```

Playwright Pytest 插件基于测试装置的概念，例如传递到您的测试中的内置页面装置。
由于浏览器上下文，页面在测试之间被隔离，这相当于一个全新的浏览器配置文件，每个测试都会获得一个全新的环境，即使在单个浏览器中运行多个测试也是如此。

```
from playwright.sync_api import Page

def test_basic_test(page: Page):
  # ...
```

您可以使用各种 fixture 在测试之前或之后执行代码，并在它们之间共享对象。
例如，具有自动使用 function 作用域的 fixture 类似于 beforeEach/afterEach。
具有module自动使用功能的作用域fixture的行为类似于 `beforeAll/afterAll`，它在所有测试之前和之后运行。

```
import pytest
from playwright.sync_api import Page


@pytest.fixture(scope="function", autouse=True)
def before_each_after_each(page: Page):
    print("beforeEach")
    # Go to the starting url before each test.
    page.goto("https://playwright.dev/")
    yield
    print("afterEach")

def test_main_navigation(page: Page):
    # Assertions use the expect API.
    expect(page).to_have_url("https://playwright.dev/")
```

5.2.4 运行测试用例

您可以运行单个测试、一组测试或所有测试。测试可以在一个浏览器或多个浏览器上运行。
默认情况下，测试以无头方式运行，这意味着在运行测试时不会打开浏览器窗口，并且会在终端中看到结果。如果您愿意，可以使用--headed标志以引导模式运行测试。

默认在 Chromium 上运行测试

```
pytest
```

运行单个测试文件

```
pytest test_login.py
```

运行一组测试文件

```
pytest tests/todo-page/ tests/landing-page/
```

使用函数名运行测试

```
pytest -k "test_add_a_todo_item"
```

在引导模式下运行测试

```
pytest --headed test_login.py
```

在特定浏览器上运行测试

```
pytest test_login.py --browser webkit
```

在多个浏览器上运行测试

```
pytest test_login.py --browser webkit --browser firefox
```

如果您的测试在具有大量 CPU 的机器上运行，您可以通过使用pytest-xdist一次运行多个测试来加快测试套件的整体执行时间：

```
# install dependency
pip install pytest-xdist
# use the --numprocesses flag
pytest --numprocesses auto
```

根据测试的硬件和性质，您可以设置numprocesses为从2机器上的 CPU 数量到 CPU 数量之间的任何值。如果设置得太高，您可能会注意到意外行为。

5.2.5 Pytest 插件参考

Playwright 提供了一个Pytest插件来编写端到端测试。要运行测试，请使用Pytest CLI。

```
pytest --browser webkit --headed
```

如果你想自动添加 CLI 参数而不指定它们，你可以使用pytest.ini文件：

CLI 相关参数
`--headed`：以有头模式运行测试（默认：无头）。
`--browser`：在不同的浏览器：chromium,firefox,webkit。它可以指定多次（默认值：chromium）。
`--browser-channel` 要使用的浏览器通道。
`--slowmo` 使用慢动作运行测试。
`--device` 要模拟的设备。
`--output` 测试生成的工件目录（默认值：test-results）。
`--tracing` 是否为每个测试记录轨迹。on、off或retain-on-failure（默认值：off）。
`--video` 是否为每次测试录制视频。on、off或retain-on-failure（默认值：off）。
`--screenshot` 是否在每次测试后自动捕获屏幕截图。on、off或only-on-failure（默认值：off）。

5.2.6 内置fixture

Function scope:
这些固定装置在测试功能中请求时创建，并在测试结束时销毁。

- context: 用于测试的新浏览器上下文。
- page: 用于测试的新浏览器页面。

Session scope:
这些固定装置在测试函数中请求时创建，并在所有测试结束时销毁。

- playwright: playwright实例。
- browser_type: 当前浏览器的BrowserType实例。
- browser：由 Playwright 启动的浏览器实例。
- browser_name: 浏览器名称作为字符串。
- browser_channel: 浏览器通道作为字符串。
- is_chromium, is_webkit, is_firefox: 相应浏览器类型的布尔值。

自定义fixture选项
对于browser和context，使用以下fixture来定义自定义启动选项。

- browser_type_launch_args：覆盖browser_type.launch()的启动参数。它应该返回一个字典。
- browser_context_args：覆盖browser.new_context()的选项。它应该返回一个字

典。

5.2.7 编写用例示例

写个简单的用例

```
# test_my_application.py
from playwright.sync_api import Page

def test_visit_admin_dashboard(page: Page):
    page.goto("/admin")
    # ...
```

使用带有--slowmo参数的 slow mo 运行测试。

```
pytest --slowmo 100
```

skip 某个浏览器

```
# test_my_application.py
import pytest

@pytest.mark.skip_browser("firefox")
def test_visit_example(page):
    page.goto("https://example.com")
    # ...
```

在指定浏览器上运行

```
# conftest.py
import pytest

@pytest.mark.only_browser("chromium")
def test_visit_example(page):
    page.goto("https://example.com")
    # ...
```

指定浏览器

```
pytest --browser-channel chrome
```

```
# test_my_application.py
def test_example(page):
    page.goto("https://example.com")
```

5.2.8 配置base-url

使用 base-url 参数启动 Pytest。该 pytest-base-url 插件用于允许您从配设置基本 url。

```
# test_my_application.py
def test_visit_example(page):
    page.goto("/admin")
    # -> Will result in http://localhost:8080/admin
```

运行

```
pytest --base-url http://localhost:8080
```

5.2.9 其它

忽略https 错误

```
# conftest.py
import pytest

@pytest.fixture(scope="session")
def browser_context_args(browser_context_args):
    return {
        **browser_context_args,
        "ignore_https_errors": True
    }
```

自定义浏览器窗口大小

```python
# conftest.py
import pytest

@pytest.fixture(scope="session")
def browser_context_args(browser_context_args):
    return {
        **browser_context_args,
        "viewport": {
            "width": 1920,
            "height": 1080,
        }
    }
```

设置手机设备

```python
# conftest.py
import pytest

@pytest.fixture(scope="session")
def browser_context_args(browser_context_args, playwright):
    iphone_11 = playwright.devices['iPhone 11 Pro']
    return {
        **browser_context_args,
        **iphone_11,
    }
```

或者通过 CLI指定参数`--device="iPhone 11 Pro"`

持久的 context

```python
# conftest.py
import pytest
from playwright.sync_api import BrowserType
from typing import Dict

@pytest.fixture(scope="session")
def context(
    browser_type: BrowserType,
    browser_type_launch_args: Dict,
    browser_context_args: Dict
):
    context = browser_type.launch_persistent_context("./foobar", **{
        **browser_type_launch_args,
        **browser_context_args,
        "locale": "de-DE",
    })
    yield context
    context.close()
```

使用它时，测试中的所有页面都是从持久上下文创建的

5.3 new_context上下文之base_url 参数

在做自动化测试的时候，我们经常是基于某个测试环境地址去测试某个项目，所以应该把它单独拿出来做为一个全局的配置。
其它地方用相对地址就行。在pytest用例里面可以用到 pytest-base-url 插件来实现。
playwright 不得不说设计的非常人性化，堪称 web 自动化界的“海底捞”服务，就差上厕所帮你扶着了~

5.3.1 使用场景

如下测试场景，在多个地方都会有访问的地址，并且环境地址都是一样`https://www.cnblogs.com`，也就是我们说的base_url地址。

```
from playwright.sync_api import sync_playwright


with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context()

    page = context.new_page()
    # 打开首页
    page.goto("https://www.cnblogs.com/")

    # 点点点后打开其他页
    page.goto("https://www.cnblogs.com/yoyoketang")

    context.close()
    browser.close()
```

当很多地方都用到 base_url 的时候，为了方便切换环境，应该单独拿出来,做全局配置

5.3.2 base_url 的使用

base_url 参数是在 new_context() 新建上下文的时候使用

优化后的代码如下：

```
from playwright.sync_api import sync_playwright


with sync_playwright() as p:
    browser = p.chromium.launch(headless=False)
    context = browser.new_context(base_url='https://www.cnblogs.com')

    page = context.new_page()
    # 打开首页
    page.goto("/")

    # 点点点后打开其他页
    page.goto("/yoyoketang")

    context.close()
    browser.close()
```

这样只需相对地址即可访问了。

5.3.3 pytest-playwright 使用

在 pytest-playwright 插件中已经自带了 pytest-base-url 插件，于是仅需在 pytest.ini 中配置

```
[pytest]
base_url=https://www.cnblogs.com
```

或者使用命令行参数

```
pytest --base-url https://www.cnblogs.com
```

在测试用例中写相对地址即可

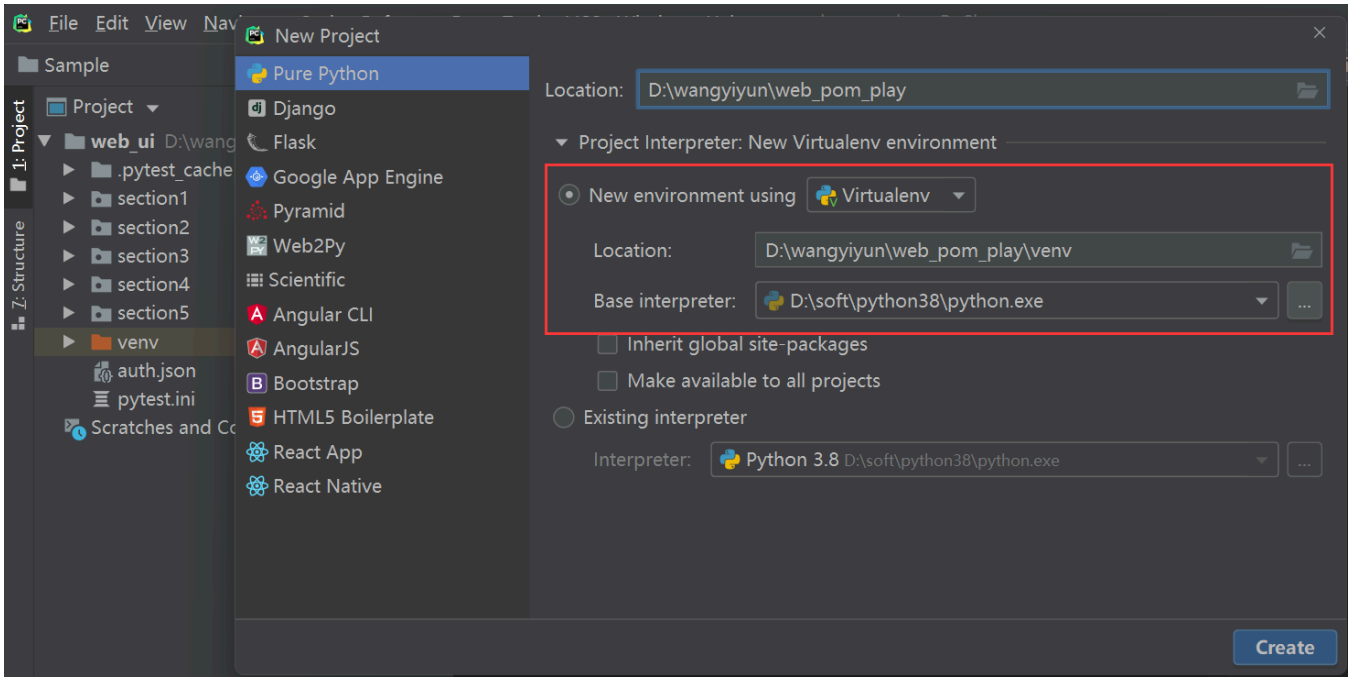
```
from playwright.sync_api import Page


def test_blog(page: Page):
    """首页"""
    page.goto("/")


def test_yoyo_blog(page: Page):
    """上海悠悠博客地址"""
    page.goto("/yoyoketang")
```

5.4 Playwright项目实战-web自动化框架搭建

通过前面课程的学习，已经具备了写 web 自动化项目的基本能力，从本章节开始，教大家进入项目实战内容。
先搭建环境，安装需要的包，搭建项目的框架结构，搭建好之后，按前面学到的 POM 项目设计模型写自动化测试用例。

5.4.1 环境准备

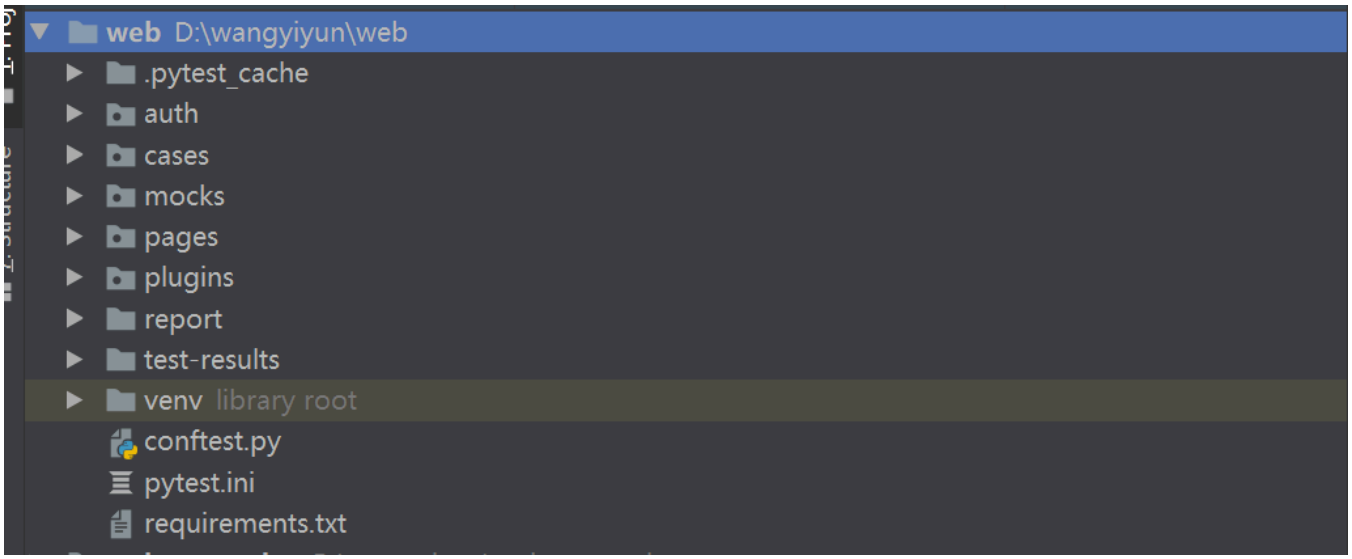
1.创建虚拟环境



5.4.2 完整项目结构

一个完整的项目结构

- auth 目录保存登录 cookies 信息
- cases 目录放测试用例
- mocks 目录放需要 mock 的数据
- pages 放封装的 Page 页面对象
- plugins 放第三方插件，或者自己开发的一些插件包
- report allure 报告目录， 自动生成
- test-resultes 保存 trace，video，png 截图内容，追踪用例记录，自动生成
- venu 虚拟环境，自动生成
- conftest.py pytest 框架的本地插件
- pytest.ini pytest 的配置文件
- requirements.txt 项目依赖包，自动生成



5.4.3 所需依赖包

完成一个文本项目的自动化，至少需要以下基本环境

```
pip install playwright
playwright install
pip install pytest-playwright
pip install pytest-base-url
pip install allure-pytest
```

安装完成后，通过 pip freeze 命令自动生成 requirements.txt 文件

```
pip freeze >requirements.txt
```

到这里项目搭建已经完成！

5.5 Playwright项目实战-登录页面用例

根据 POM 项目设计模式，先 pages 封装登录页面，再去 cases 写用例.
自动化测试的本质是把功能用例点点点的工作，用代码去完成

在写自动化用例之前，应该先有功能测试用例，对照功能用例，一个个翻译成自动化测试脚本。

5.5.1 Login页面封装

在 pages 目录创建 login_page.py, 封装登录页面抓取元素的属性，以及页面上操作方法

```python
from playwright.sync_api import Page


class LoginPage:

    def __init__(self, page: Page):
        self.page = page
        self.locator_username = page.get_by_label("用 户 名:")
        self.locator_password = page.get_by_label("密　　码:")
        self.locator_login_btn = page.locator('text=立即登录')
        self.locator_register_link = page.locator('text=没有账号？点这注册')
        # 用户名输入框提示语
        self.locator_username_tip1 = page\
            .locator('[data-fv-validator="notEmpty"][data-fv-for="username"]')
        self.locator_username_tip2 = page\
            .locator('[data-fv-validator="stringLength"][data-fv-for="username"]')
        self.locator_username_tip3 = page\
            .locator('[data-fv-validator="regexp"][data-fv-for="username"]')
        # 密码输入框提示语
        self.locator_password_tip1 = page\
            .locator('[data-fv-validator="notEmpty"][data-fv-for="password"]')
        self.locator_password_tip2 = page\
            .locator('[data-fv-validator="stringLength"][data-fv-for="password"]')
        self.locator_password_tip3 = page\
            .locator('[data-fv-validator="regexp"][data-fv-for="password"]')
        # 账号或密码不正确!
        self.locator_login_error = page\
            .locator('text=账号或密码不正确！ ')

    def navigate(self):
        self.page.goto("/login.html")

    def fill_username(self, username):
        self.locator_username.fill(username)

    def fill_password(self, password):
        self.locator_password.fill(password)

    def click_login_button(self):
        self.locator_login_btn.click()

    def click_register_link(self):
        self.locator_register_link.click()

    def login(self, username, password) -> None:
        """完整登录操作"""
```

```
        self.locator_username.fill(username)
        self.locator_password.fill(password)
        self.locator_login_btn.click()
```

5.5.2 用例设计

用例的编写根据平常的功能测试用例去写

```python
from pages.login_page import LoginPage
from playwright.sync_api import expect
import pytest
import allure


class TestLogin:
    """登录功能"""

    @pytest.fixture(autouse=True)
    def start_for_each(self, page):
        print("for each--start: 打开新页面访问登录页")
        self.login = LoginPage(page)
        self.login.navigate()
        yield
        print("for each--end: 后置操作")

    def test_login_1(self):
        """用户名为空，点注册"""
        self.login.fill_username('')
        self.login.fill_password('123456')
        self.login.click_login_button()
        # 断言
        expect(self.login.locator_username_tip1)\
            .to_be_visible()
        expect(self.login.locator_username_tip1)\
            .to_contain_text("不能为空")

    def test_login_2(self):
        """用户名大于30字符"""
        self.login.fill_username('hello world hello world hello world')
        # 断言
        expect(self.login.locator_username_tip2)\
            .to_be_visible()
        expect(self.login.locator_username_tip2)\
            .to_contain_text("用户名称1-30位字符")
        # 断言 登录按钮不可点击
        expect(self.login.locator_login_btn).not_to_be_enabled()

    def test_login_3(self):
        """用户名有特殊字符"""
        self.login.fill_username('hello!@#')
        # 断言
        expect(self.login.locator_username_tip3).to_be_visible()
        expect(self.login.locator_username_tip3)\
            .to_contain_text("用户名称不能有特殊字符")
```

```
        # 断言 注册按钮不可点击
        expect(self.login.locator_login_btn).not_to_be_enabled()

    @pytest.mark.parametrize("username,password,title",[
        ['yoyo', '12345678', '输入错误的密码'],
        ['yoyox1x2x3', '12345678', '输入错误的账号'],
    ])
    def test_login_error(self, username, password, title):
        """登录失败场景"""
        self.login.fill_username(username)
        self.login.fill_password(password)
        self.login.click_login_button()
        # 断言提示语可见
        expect(self.login.locator_login_error).to_be_visible()

    def test_login_success(self):
        """登录正常流程， 登录成功"""
        self.login.fill_username("yoyo")
        self.login.fill_password('aa123456')
        self.login.click_login_button()
        # 断言页面跳转到首页
        expect(self.login.page).to_have_title('首页')
        expect(self.login.page).to_have_url('/index.html')

    def test_login_success_2(self):
        """登录正常流程， 登录成功"""
        self.login.fill_username("yoyo")
        self.login.fill_password('aa123456')
        # 显示断言重定向
        with self.login.page.expect_navigation(url='**/index.html'):
            self.login.click_login_button()
        # 断言页面跳转到首页
        expect(self.login.page).to_have_title('首页')
        expect(self.login.page).to_have_url('/index.html')

    def test_login_ajax(self):
        """登录正常流程， 获取异步ajax 请求"""
        self.login.fill_username("yoyo")
        self.login.fill_password('aa123456')
        # 捕获ajax请求
        with self.login.page.expect_request('**/api/login') as req:
            self.login.click_login_button()
        print(req.value)  # 获取请求对象
        # 断言请求内容
        assert req.value.method == 'POST'
        assert req.value.header_value('content-type') == 'application/json'
```

```
        assert req.value.post_data_json == {
            'username': 'yoyo', 'password': 'aa123456'
        }

    def test_login_ajax_response(self):
        """登录正常流程， 获取异步ajax 请求返回结果"""
        self.login.fill_username("yoyo")
        self.login.fill_password('aa123456')
        # 捕获ajax请求
        with self.login.page.expect_response('**/api/login') as res:
            self.login.click_login_button()
        print(res.value)  # 获取请求对象
        print(res.value.url)
        print(res.value.status)
        print(res.value.ok)
        assert res.value.ok
        assert res.value.status == 200
```

5.6 Playwright项目实战-登录成功如何断言？

当页面登录成功的时候，会重定向到首页，这时候我们如何断言登录成功了？
登录成功我们有多种判断方式：
1.页面跳转到首页，直接断言首页的 title 和 url 地址是否正确
2.显示断言 expect_navigation， 判断导航到指定 url 地址
3.也可以断言 Ajax 请求，判断前端发过去的 Ajax 请求参数是否正确
4.还可以断言 Ajax 响应数据，判断接口响应是否正确

5.6.1 登录成功用例

1.页面跳转到首页，直接断言首页的 title 和 url 地址是否正确

```
def test_login_success(self):
    """登录正常流程， 登录成功"""
    self.login.fill_username("yoyo")
    self.login.fill_password('aa123456')
    self.login.click_login_button()
    # 断言页面跳转到首页
    expect(self.login.page).to_have_title('首页')
    expect(self.login.page).to_have_url('/index.html')
```

2.显示断言你 expect_navigation， 判断导航到指定 url 地址

```
def test_login_success_2(self):
    """登录正常流程， 登录成功"""
    self.login.fill_username("yoyo")
    self.login.fill_password('aa123456')
    # 显示断言重定向
    with self.login.page.expect_navigation(url='**/index.html'):
        self.login.click_login_button()
    # 断言页面跳转到首页
    expect(self.login.page).to_have_title('首页')
    expect(self.login.page).to_have_url('/index.html')
```

5.7 Playwright项目实战-参数化场景

如果用例的操作步骤都是一样，只是测试输入和输出结果不一样，这种情况就可以适用于参数化的场景

5.7.1 Pytest 参数化

pytest框架参数化的实现格式

```
import pytest


@pytest.mark.parametrize("test_input,expected",
    [
        ["3+5", 8],
        ["2+4", 6],
        ["6 * 9", 42]
    ]
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

5.7.2 用例参数化

登录失败的场景，输入用户名和密码不一样，得到的结果提示语也不一样

```
@pytest.mark.parametrize("username,password,title",[
        ['yoyo', '12345678', '输入错误的密码'],
        ['yoyox1x2x3', '12345678', '输入错误的账号'],
    ])
    def test_login_error(self, username, password, title):
        """登录失败场景"""
        self.login.fill_username('yoyo')
        self.login.fill_password('12345678')
        self.login.click_login_button()
        # 断言提示语可见
        expect(self.login.locator_login_error).to_be_visible()
```

5.8 Playwright项目实战-断言Ajax 异步请求

Playwright 不仅仅可以断言登录后重定向的页面，也可以断言点登录按钮，发过去的ajax 异步请求，以及接口返回过来的结果。

5.8.1 断言 ajax 异步请求

断言点登录按钮，发过去的请求参数与头部内容是否正确

```
def test_login_ajax(self):
    """登录正常流程， 获取异步ajax 请求"""
    self.login.fill_username("yoyo")
    self.login.fill_password('aa123456')
    # 捕获ajax请求
    with self.login.page.expect_request('**/api/login') as req:
        self.login.click_login_button()
    print(req.value)  # 获取请求对象
    # 断言请求内容
    assert req.value.method == 'POST'
    assert req.value.header_value('content-type') == 'application/json'
    assert req.value.post_data_json == {
        'username': 'yoyo', 'password': 'aa123456'
    }
```

5.8.2 断言ajax 请求结果

断言点登录按钮，ajax 请求返回的结果是否正确

```python
def test_login_ajax_response(self):
    """登录正常流程， 获取异步ajax 请求"""
    self.login.fill_username("yoyo")
    self.login.fill_password('aa123456')
    # 捕获ajax请求
    with self.login.page.expect_response('**/api/login') as res:
        self.login.click_login_button()
    print(res.value)  # 获取返回对象
    print(res.value.url)
    print(res.value.status)
    print(res.value.ok)
    assert res.value.ok
    assert res.value.status == 200
```

5.9 Playwright项目实战-a标签链接断言

a 标签的链接可以先断言href 属性， 也就是点开后会跳转的地址， 在页面上存在2种情况。
1.带有 `target="_blank"` 属性的链接时， 会打开一个新的标签页。
2.不带 `target="_blank"` 属性， 直接在当前页面刷新。

5.9.1 使用场景

先看a标签有没有`target="_blank"` 属性

我们课程里面的案例， 适用于第二种场景， 直接在当前页面刷新。

5.9.2 a标签断言

```
def test_login_register_link(self):
    """没有账号？点这注册  链接测试"""
    expect(self.login.locator_register_link)\
        .to_have_attribute("href", "register.html")
    self.login.click_register_link()
    expect(self.login.page).to_have_title('注册')
    expect(self.login.page).to_have_url('register.html')
```

5.10 Playwright项目实战-解决反复登录问题

在 web 网站自动化中，很多用例场景都是需要先登录才能访问的（注册和登录页面除外）如果把登录写到前置操作，就会导致重复登录问题。
前面说到可以用 cookies 的方式去登录，那么我们只需要在所有用例之前登录一次，保存 cookies 即可

5.10.1 全局先登录

全局仅运行一次，可以用 pytest 的 fixtures，设置 session 级别即可解决，
设置 `autouse=True` 它会在所以用例之前自动执行。

在 cases 目录创建 conftest.py 文件，作为整个项目用例的前置操作。

```
import pytest
from pages.login_page import LoginPage


@pytest.fixture(scope="session", autouse=True)
def login_save_auth(browser, base_url, pytestconfig):
    """登录，保存cookies"""
    context = browser.new_context(base_url=base_url)
    page = context.new_page()
    LoginPage(page).navigate()
    LoginPage(page).login("yoyo", "aa123456")
    # 等待登录成功页面重定向
    page.wait_for_url(url='**/index.html')
    # 保存storage state 到指定的文件
    storage_path = pytestconfig.rootpath.joinpath("auth/state.json")
    context.storage_state(path=storage_path)
    context.close()
```

5.10.2 context上下文加载cookies

在 context上下文添加参数， 也就是前面 pytest-playwright 插件提到的, 重写 browser_context_args 覆盖原来的 browser_context_args 。

```
@pytest.fixture(scope="session")
def browser_context_args(browser_context_args, playwright, pytestconfig):
    """
    添加context 上下文参数，默认每个页面加载cookies
    :param browser_context_args:
    :param playwright:
    :return:
    """

    return {
        "storage_state":pytestconfig.rootpath.joinpath("auth/state.json"),
        **browser_context_args,
    }
```

这样全局用例之前， 就会自动加载 cookies 了。

5.11 Playwright项目实战-随机生成注册账号

注册页面和登录页面用例差不多，只有一个场景不一样。
同一个账号不能反复注册，对于这种**测试数据不可逆**的场景， 我们可以用以下方式解决：
1.用随机账号注册， 每次生成不一样的账号
2.去连数据库删除已注册的账号
3.直接 mock 返回注册成功， 不关注接口的功能， 只关注前端注册成功后的业务逻辑。

5.11.1 生成随机账号

使用uuid生成随机账号

```
import uuid

print(str(uuid.uuid4())[:8])
```

5.11.2 注册成功场景

```
def test_register_success(self):
    """测试注册正常流程， 注册成功"""
    # 生成随机账号
    import uuid
    self.register.fill_username(str(uuid.uuid4())[:8])
    self.register.fill_password('aa123456')
    self.register.click_register_button()
    # 断言页面跳转到首页
    expect(self.register.page).to_have_title('首页')
    expect(self.register.page).to_have_url('/index.html')
```

5.12 Playwright项目实战-添加项目

新增项目界面只有一个 form 表单提交的功能， 当新增成功， 重定向到项目列表页面

5.12.1 封装页面对象

先封装页面对象

```python
from playwright.sync_api import Page


class AddProjectPage:

    def __init__(self, page: Page):
        self.page = page
        self.locator_project_name = page.get_by_label("项目名称:")
        self.locator_publish_app = page.get_by_label("所属应用:")
        self.locator_project_desc = page.get_by_label('项目描述:')
        self.locator_save_button = page.get_by_text('点击提交')

    def navigate(self):
        self.page.goto("/add_project.html")

    def fill_project_name(self, name):
        self.locator_project_name.fill(name)

    def fill_publish_app(self, text):
        self.locator_publish_app.fill(text)

    def fill_project_desc(self, text):
        self.locator_project_desc.fill(text)

    def click_save_button(self):
        self.locator_save_button.click()
```

5.12.2 写用例

完成用例编写

```python
from pages.add_project_page import AddProjectPage
from playwright.sync_api import expect
import pytest
import uuid


class TestAddProject:
    """添加项目"""

    @pytest.fixture(autouse=True)
    def start_for_each(self, page):
        print("for each--start: 打开添加项目页")
        self.add_project = AddProjectPage(page)
        self.add_project.navigate()
        yield
        print("for each--end: 后置操作")

    @pytest.mark.parametrize("name, app, desc", [
        ["abc!@", "", ""],
        ["aaaaabbbbbccccdddddeeeeeffffff1", "", ""],
        ["abc", "aa!@", ""]
    ])
    def test_add_project_disabled(self, name, app, desc):
        """异常场景-项目名称，无效等价：特殊字符/大于30个字符"""
        self.add_project.fill_project_name(name)
        self.add_project.fill_publish_app(app)
        self.add_project.fill_project_desc(desc)
        # 断言提交按钮状态 不可点击
        expect(self.add_project.locator_save_button).to_be_disabled()

    def test_add_project_null(self):
        """异常场景-项目名称不能为空"""
        self.add_project.fill_project_name("")
        self.add_project.fill_publish_app("")
        self.add_project.fill_project_desc("")
        self.add_project.click_save_button()
        # 断言提交按钮状态 不可点击
        expect(self.add_project.locator_save_button).to_be_disabled()

    def test_add_project_success(self, page):
        """提交成功，跳转到项目列表"""
        # 生成随机账号
        self.add_project.fill_project_name(
            str(uuid.uuid4()).replace('-', '')[:25]
        )
        self.add_project.fill_publish_app("xx")
```

```
        self.add_project.fill_project_desc("xxx")
        # 断言跳转到项目列表页
        with page.expect_navigation(url="**/list_project.html"):
            # 保存成功后，重定向到列表页
            self.add_project.click_save_button()
```

5.12.3 提交成功判断新增项目在列表

table表格定位与内容获取，参考 `2.28 table表格定位与数据获取`

获取第3列的数据放到list列表，断言新增的名称在列表里


```
# 断言新增模块名称  "登录"  在列表里
loc_table_module = self.add_project.page.locator(
    '//*[@id="table"]/tbody/tr/td[3]'
)
# 获取文本
all_module_names = [i.inner_text() for i in loc_table_module.all()]
print(f"获取页面 table 表格-模块名称列全部内容: {all_module_names}")
assert "登录" in all_module_names
```

完整用例代码

```python
    def test_add_project_success2(self, page: Page):
        """提交成功，判断新项目名称在列表"""
        # 生成随机账号
        new_project_name = str(uuid.uuid4()).replace('-', '')[:25]
        self.add_project.fill_project_name(new_project_name)
        self.add_project.fill_publish_app("xx")
        self.add_project.fill_project_desc("xxx")
        # 保存成功后
        self.add_project.click_save_button()
        # 点击保存后等页面重定向到table表格页
        self.add_project.page.wait_for_load_state('networkidle')
        # 断言新增项目在列表页
        print(f"新增项目名称: {new_project_name}")
        # 获取页面 table 表格-项目名称列全部内容
        loc_projects = self.add_project.page.locator(
            '//table[@id="table"]//td[3]/a')
        # 获取文本
        project_names = [i.inner_text() for i in loc_projects.all()]
        print(f"获取页面 table 表格-项目名称列全部内容: {project_names}")
        assert new_project_name in project_names
```

5.13 Playwright项目实战-Mock 新增项目400和500异常场景

如果你们项目是前后端分离的，你只关注前端功能（后端的功能给接口自动化人员去做）
那么我们不用管接口的返回内容，也不用跟数据库打交道。直接 mock 自己需要的数据即可。
有些调用第三方接口，也可以用 mock 拦截掉。

5.13.1 mock 数据

mocks/mock_api.py文件

```
import json

"""
/**** 模拟新增项目 返回 400  ***/
"""
mock_project_400 = {
    "url": "**/api/project",
    "handler": lambda route: route.fulfill(
        status=400,
        body=json.dumps({
            "errors":
                {
                    "project_name": "yo yo 已存在"
                },
            "message": "Input payload validation failed"
        })
    )
}

"""
/**** 模拟新增项目 返回 500  ***/
"""
mock_project_500 = {
    "url": "**/api/project",
    "handler": lambda route: route.fulfill(
        status=500,
        body="服务端错误"
    )
}
```

5.13.2 mock 用例场景

```
def test_add_project_400(self, page):
    """项目名称重复，弹出模态框"""
    self.add_project.fill_project_name("yo yo")
    self.add_project.fill_publish_app("xx")
    self.add_project.fill_project_desc("xxx")
    # mock 接口返回400
    page.route(**mock_api.mock_project_400)
    self.add_project.click_save_button()
    # 校验结果 弹出框文本包含
    expect(page.locator('.bootbox-body')).to_contain_text('已存在')

def test_add_project_500(self, page):
    """服务器异常 500 状态码"""
    self.add_project.fill_project_name("test")
    self.add_project.fill_publish_app("xx")
    self.add_project.fill_project_desc("xxx")
    # mock 接口返回500
    page.route(**mock_api.mock_project_500)
    self.add_project.click_save_button()
    # 校验结果 弹出框文本包含
    expect(page.locator('.bootbox-body')).to_contain_text('操作异常')
```

5.14 Playwright项目实战- 生成 Allure报告标题

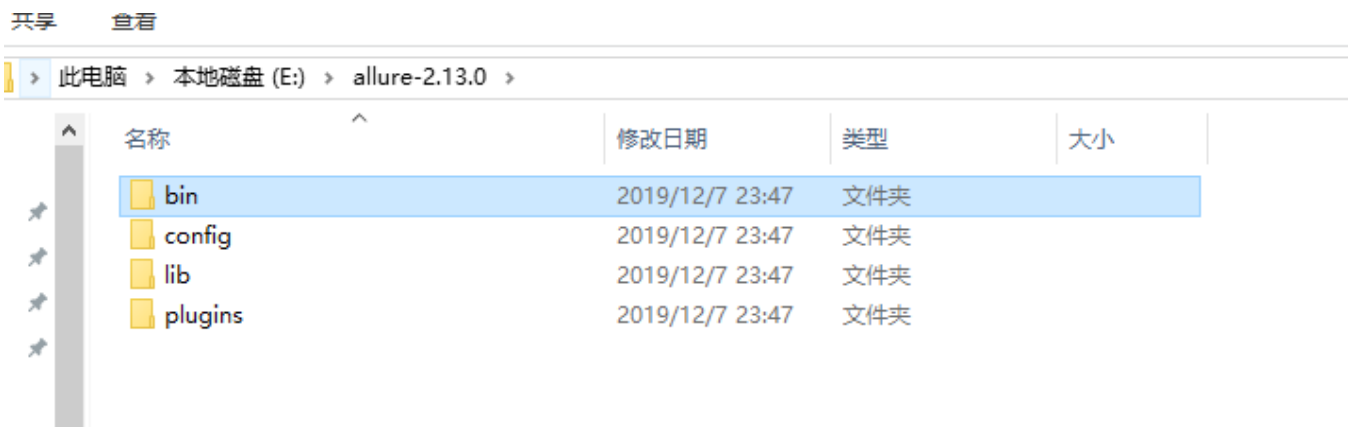
pytest 框架有给 allure-pytest 插件可以生成 allure 测试报告

5.14.1 Allure 环境准备

Allure 的环境分为2个部分，

- 一个是前面 pip 安装的 allure-pytest， 是用于在代码里面添加用例标题步骤描述等内容。
- 还有一个是用于生成 html 报告的本地命令行工具，需要去 github上下载最新版 https://github.com/allure-framework/allure2/releases

下载后解压在本地



由于 allure 是基于 Java 开发的，所以你本地还要安装 JDK 环境，相关环境自己百度去解决了。

allure 命令行工具解压后，需把 bin目录的路径添加到环境变量，然后重启 pycharm， 去检查命令有没生效！

```
(venv) D:\wangyiyun\web_pom_pla>allure
Usage: allure [options] [command] [command options]
  Options:
    --help
      Print commandline help.
    -q, --quiet
      Switch on the quiet mode.
      Default: false
    -v, --verbose
      Switch on the verbose mode.
      Default: false
    --version
      Print commandline version.
      Default: false
```

5.14.2 allure 的基本使用

使用方法	参数值	参数说明
`@allure.epic()`	epic描述	敏捷里面的概念，定义史诗，往下是feature
`@allure.feature()`	模块名称	功能点的描述，往下是story
`@allure.story()`	用户故事	用户故事，往下是title
`@allure.title(用例的标题)`	用例的标题	重命名html报告名称

使用方法	参数值	参数说明
`@allure.testcase()`	测试用例的链接地址	对应功能测试用例系统里面的case
`@allure.issue()`	缺陷	对应缺陷管理系统里面的链接
`@allure.description()`	用例描述	测试用例的描述
`@allure.step()`	操作步骤	测试用例的步骤
`@allure.severity()`	用例等级	blocker，critical，normal，minor，trivial
`@allure.link()`	链接	定义一个链接，在测试报告展现
`@allure.attachment()`	附件	报告添加附件

5.14.3 自动生成 feature 和 title

为了避免每个用例都去加`@allure.feature(功能点)` 和 `@allure.title(用例的标题)`

如下用例部分，我们可以根据
类的注释内容，自动生成 allure.feature(功能点)
测试用例的注释，自动生成 allure.title(用例的标题)，测试用例的注释默认是 description （描述）

```
class TestRegister:
    """注册功能"""


    def test_register_1(self):
        """用户名为空，点注册"""


    def test_register_2(self):
        """用户名大于30字符"""
```

在全局conftest.py 钩子部分实现

```python
from pytest import Item
import allure


def pytest_runtest_call(item: Item):  # noqa
    # 动态添加测试类的 allure.feature()
    if item.parent._obj.__doc__:  # noqa
        allure.dynamic.feature(item.parent._obj.__doc__)  # noqa
    # 动态添加测试用例的title 标题 allure.title()
    if item.function.__doc__: # noqa
        allure.dynamic.title(item.function.__doc__)  # noqa
```

5.14.4 添加用例步骤

用例步骤也不用在用例里面一个个去写，在封装 page 对象的时候，方法里面添加步骤

```python
class RegisterPage:

    def __init__(self, page: Page):
        self.page = page
        ......

    def navigate(self):
        with allure.step("导航到注册页"):
            self.page.goto("/register.html")

    def fill_username(self, username):
        with allure.step(f"输入用户名:{username}"):
            self.locator_username.fill(username)

    def fill_password(self, password):
        with allure.step(f"输入密码:{password}"):
            self.locator_password.fill(password)

    def click_register_button(self):
        with allure.step(f"点注册按钮"):
            self.locator_register_btn.click()

    def click_login_link(self):
        with allure.step(f"点登录链接"):
            self.locator_login_link.click()
```

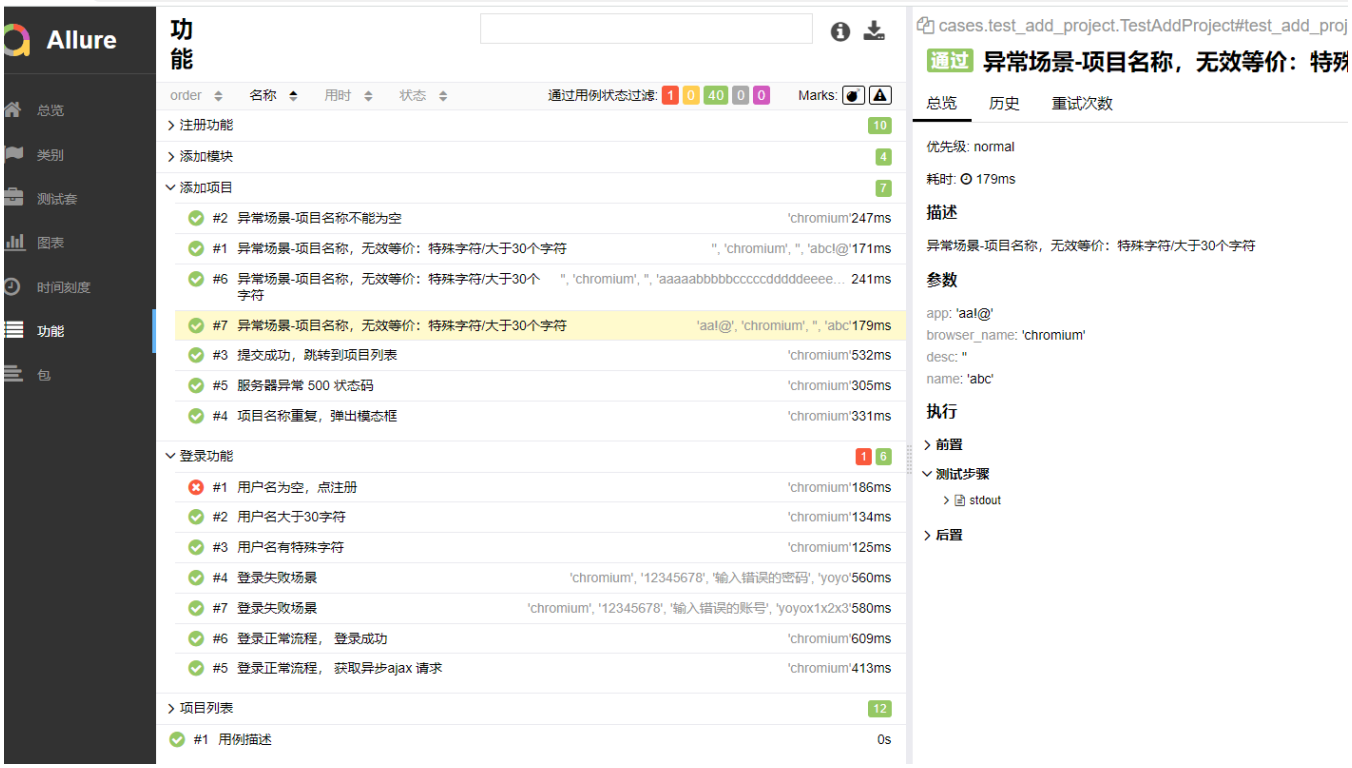
5.14.5 报告展示

使用 pytest 命令行运行用例

```
>pytest --alluredir ./report
==================== test session starts ============================
platform win32 -- Python 3.8.5, pytest-7.2.1, pluggy-1.0.0
baseurl: http://47.108.155.10
rootdir: D:\demo\play_web_p, configfile: pytest.ini
plugins: allure-pytest-2.13.0, base-url-2.0.0
collected 41 items
```

查看报告

```
allure serve ./report
```



最终测试报告会自动生成 feature 和 title, 以及测试步骤

5.15 Playwright项目实战- Allure 报告带上视频和截图

我们一般希望在用例失败的时候，在 allure 报告加上截图，录制视频，方便我们知道失败的原因。

pytest-playwright 插件有3 个参数，可以在用例失败的时候调用。
`--tracing` 是否为每个测试记录轨迹。on、off或retain-on-failure（默认值：off）。
`--video` 是否为每次测试录制视频。on、off或retain-on-failure（默认值：off）。
`--screenshot` 是否在每次测试后自动捕获屏幕截图。on、off或only-on-failure（默认值：off）。

5.15.1 添加命令行参数

我们可以在pytest.ini 中添加运行的默认带上的参数

```
[pytest]

addopts = --headed
          --tracing=retain-on-failure
          --screenshot=only-on-failure
          --video=retain-on-failure
```

这一步仅仅是把截图和视频保存到test-resultes 目录

5.15.2 allure 报告带上截图

添加用例失败截图, pytest-playwright 插件 先导入

```
import pytest
from playwright.sync_api import (
    Browser,
    BrowserContext,
    BrowserType,
    Error,
    Page,
    Playwright,
    sync_playwright,
)
from slugify import slugify
import tempfile
import allure
```

在 pytest-playwright 插件中找到context

```
if capture_screenshot:
    for index, page in enumerate(pages):
        human_readable_status = "failed" if failed else "finished"
        screenshot_path = _build_artifact_test_folder(
            pytestconfig, request, f"test-{human_readable_status}-{index+1}.png"
        )
        try:
            page.screenshot(timeout=5000, path=screenshot_path)
        except Error:
            pass
```

改成

```
if capture_screenshot:
    for index, page in enumerate(pages):
        human_readable_status = "failed" if failed else "finished"
        screenshot_path = _build_artifact_test_folder(
            pytestconfig, request, f"test-{human_readable_status}-{index+1}.png"
        )
        try:
            page.screenshot(timeout=5000, path=screenshot_path)
            # --------- 把截图放入allure报告 以下代码新增 --------------

            allure.attach.file(
                screenshot_path,
                name=f"{request.node.name}-{human_readable_status}-{index+1}",
                attachment_type=allure.attachment_type.PNG
            )
            # --------- 把截图放入allure报告  --------------        except Error:
            pass
```

添加视频功能

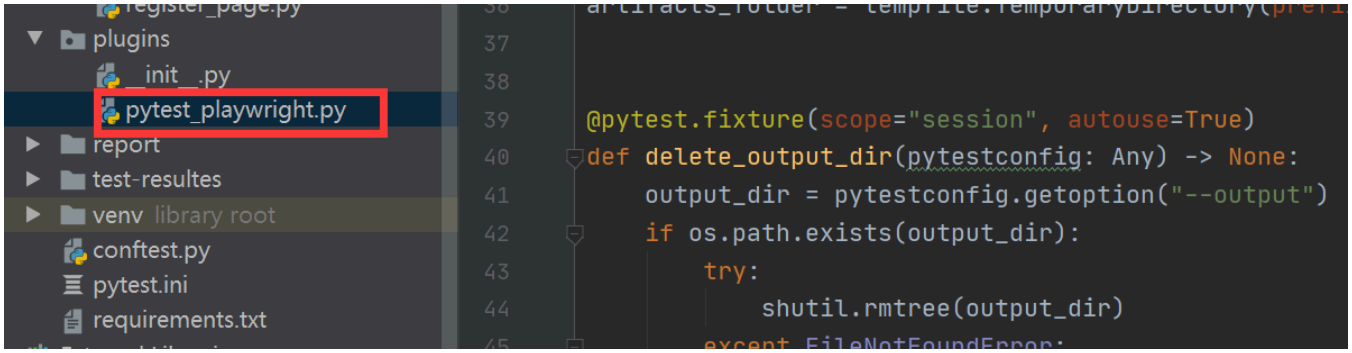
```
if preserve_video:
    for page in pages:
        video = page.video
        if not video:
            continue
        try:
            video_path = video.path()
            file_name = os.path.basename(video_path)
            video.save_as(
                path=_build_artifact_test_folder(
                    pytestconfig, request, file_name
                )
            )
        except Error:
            # Silent catch empty videos.
            pass
```

修改后

```
if preserve_video:
    for page in pages:
        video = page.video
        if not video:
            continue
        try:
            video_path = video.path()
            file_name = os.path.basename(video_path)
            file_path = _build_artifact_test_folder(
                pytestconfig, request, file_name
            )
            video.save_as(
                path=file_path
            )
            # 放入视频
            allure.attach.file(
                file_path,
                name=f"{request.node.name}-{human_readable_status}-{index + 1}",
                attachment_type=allure.attachment_type.WEBM
            )

        except Error:
            # Silent catch empty videos.
            pass
```

5.15.3 导入本地插件

由于改源码会导致，换个电脑环境就没有了，我们可以把插件复制到本地



在conftest.py 注册本地插件即可

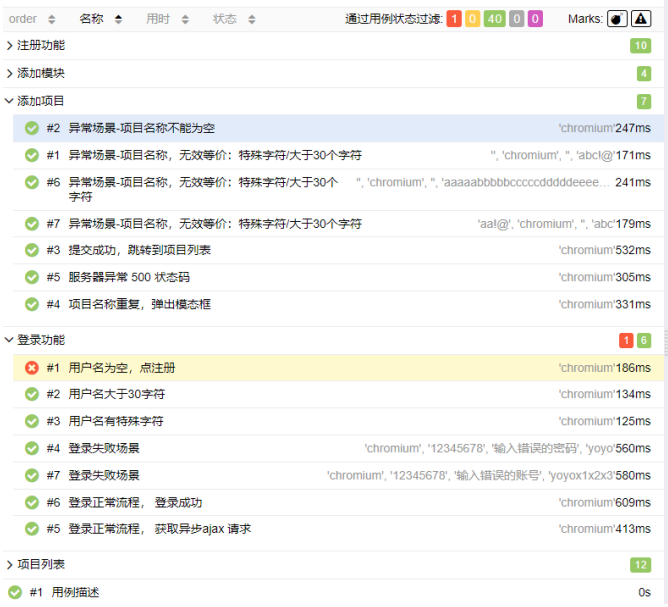
```python
from pytest import Item
import allure

# 本地插件注册
pytest_plugins = ['plugins.pytest_playwright']  # noqa


def pytest_runtest_call(item: Item):  # noqa
    # 动态添加测试类的 allure.feature()
    if item.parent._obj.__doc__:  # noqa
        allure.dynamic.feature(item.parent._obj.__doc__) # noqa
    # 动态添加测试用例的title 标题 allure.title()
    if item.function.__doc__: # noqa
        allure.dynamic.title(item.function.__doc__) # noqa
```

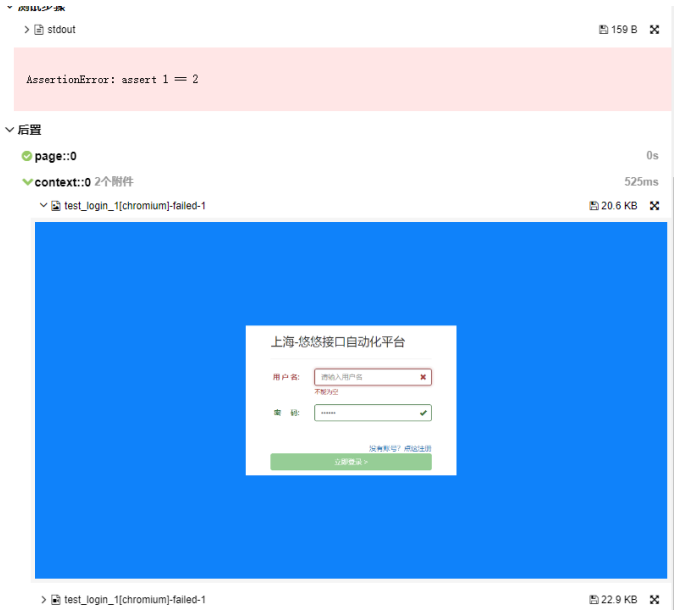
添加本地插件后需要在 pytest.ini 中禁用 pip 安装的 pytest-playwright 插件

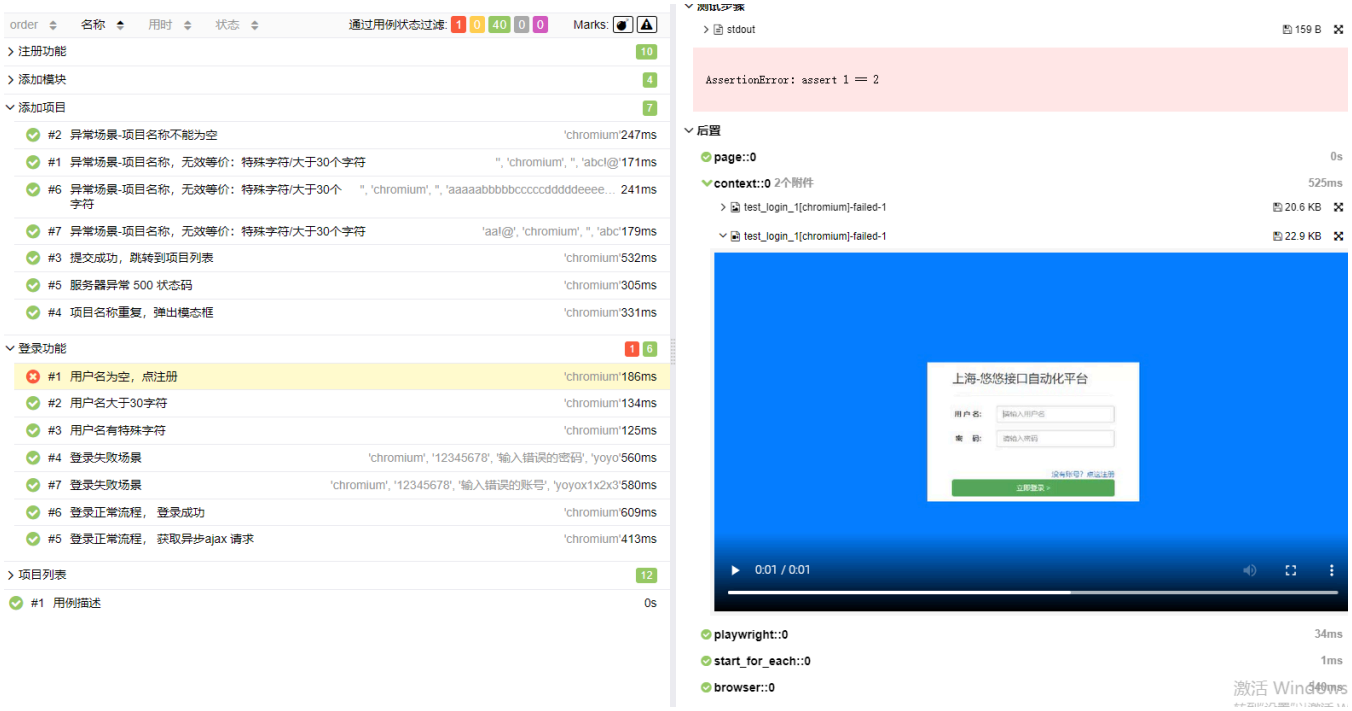
```
[pytest]
addopts = -p no:playwright
```

5.15.4 查看 Allure 报告

用例失败后自动截图







5.16 Playwright项目实战 - 多账号切换操作

pytest-playwright 插件可以让我们快速编写pytest格式的测试用例，它提供了一个内置的page 对象，可以直接打开页面操作。
但是有时候我们需要2个账号是操作业务流程，比如A账号创建了一个任务，需要用到B账号去操作审批动作等。
如果需要2个账号同时登录，可以使用 context 上下文，它可以做到环境隔离。

5.16.1 context 上下文环境隔离

使用 Playwright 编写的测试在称为浏览器上下文的隔离的全新环境中执行。这种隔离模型提高了可重复性并防止级联测试失败。

什么是测试隔离

测试隔离是指每个测试与另一个测试完全隔离。每个测试都独立于任何其他测试运行。这意味着每个测试都有自己的本地存储、会话存储、cookie 等。Playwright 使用 BrowserContext 实现了这一点，这相当于隐身式配置文件。它们的创建速度快、成本低，并且完全隔离，即使在单个浏览器中运行也是如此。Playwright 为每个测试创建一个上下文，并在该上下文中提供一个默认页面。

为什么测试隔离很重要

- 没有失败结转。如果一个测试失败，它不会影响另一个测试。
- 易于调试错误或不稳定，因为您可以根据需要多次运行单个测试。
- 并行运行、分片等时不必考虑顺序。

测试隔离有两种不同的策略：从头开始或在两者之间进行清理。在测试之间清理的问题是很容易忘记清理，有些东西是不可能清理的，比如“访问过的链接”。来自一个测试的状态可能会泄漏到下一个测试中，这可能会导致您的测试失败并使调试变得更加困难，因为问题来自另一个测试。从头开始意味着一切都是新的，因此如果测试失败，您只需查看该测试即可进行调试。

Playwright 如何实现测试

Playwright 使用浏览器上下文来实现测试隔离。每个测试都有自己的浏览器上下文。每次运行测试都会创建一个新的浏览器上下文。使用 Playwright 作为测试运行程序时，默认情况下会创建浏览器上下文。否则，您可以手动创建浏览器上下文。

```
browser = playwright.chromium.launch()
context = browser.new_context()
page = context.new_page()
```

浏览器上下文还可用于模拟涉及移动设备、权限、区域设置和配色方案的多页面场景

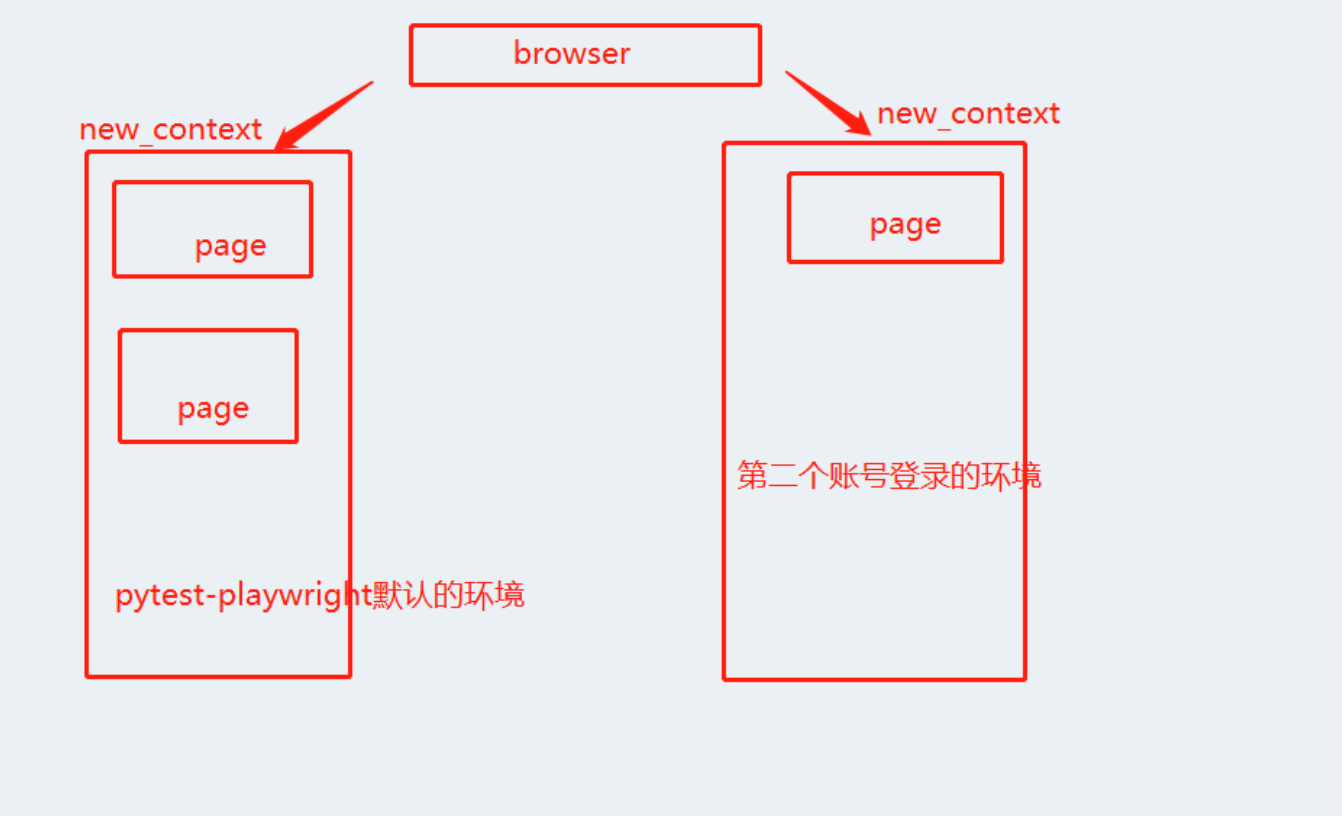
Playwright 可以在一个场景中创建多个浏览器上下文。当您想测试多用户功能（如聊天）时，这很有用。

```python
from playwright.sync_api import sync_playwright
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/


def run(playwright):
    # create a chromium browser instance
    chromium = playwright.chromium
    browser = chromium.launch()

    # create two isolated browser contexts
    user_context = browser.new_context()
    admin_context = browser.new_context()

    # create pages and interact with contexts independently

with sync_playwright() as playwright:
    run(playwright)
```

关于context上下文的详细介绍参考这篇https://www.cnblogs.com/yoyoketang/p/17142642.html

5.16.2 多账号登录解决方案

pytest-playwright 插件默认有一个context 和page 的fixture



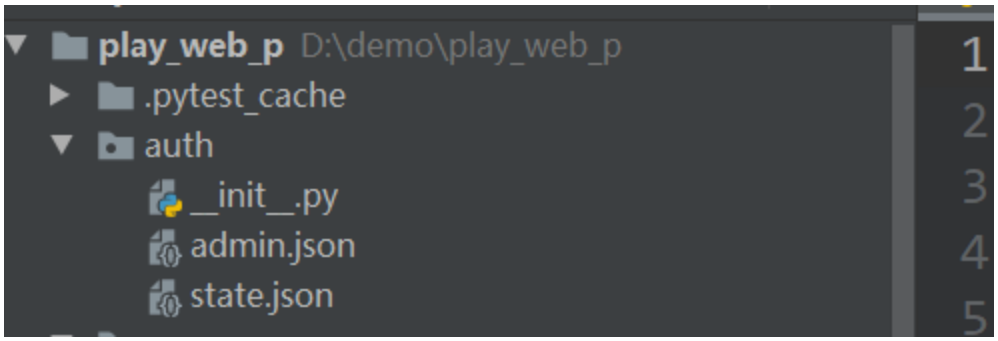
可以用pytest-playwright 插件自带的 page 对象，先登录用户 A

用户 B 的登录，重新创建另外一个上下文环境

conftest.py

```python
import pytest
from pages.login_page import LoginPage
"""
全局默认账号使用 "yoyo", "******"  在cases 目录的conftest.py 文件下
涉及多个账号切换操作的时候
我们可以创建新的上下文，用其它账号登录
"""
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/


@pytest.fixture(scope="session")
def save_admin_cookies(browser, base_url, pytestconfig):
    """
     admin 用户登录后保存admin.json  cookies信息
    :return:
    """
    context = browser.new_context(base_url=base_url, no_viewport=True)
    page = context.new_page()
    LoginPage(page).navigate()
    LoginPage(page).login("admin", "***********")
    # 等待登录成功页面重定向
    page.wait_for_url(url='**/index.html')
    # 保存storage state 到指定的文件
    storage_path = pytestconfig.rootpath.joinpath("auth/admin.json")
    context.storage_state(path=storage_path)
    context.close()


@pytest.fixture(scope="module")
def admin_context(browser, base_url, pytestconfig):
    """
    创建admin上下文，加载admin.json数据
    :return:
    """
    context = browser.new_context(
        base_url=base_url,
        no_viewport=True,
        storage_state=pytestconfig.rootpath.joinpath("auth/admin.json"),
    )
    yield context
    context.close()
```

运行完成后，会生成第二个账号对应的 admin.json 数据



在用例中，我们传 admin_context 参数就可以得到 B 账号的上下文 context 对象了，基于 context 对象创建page页面对象

```
"""
整个项目中的上下文对象
page  用例中直接传page,默认使用登录后的context上下文创建的page对象
admin_context  针对admin用户登录后的上下文环境
"""
from playwright.sync_api import BrowserContext, Page
import pytest
import uuid
from pages.add_project_page import AddProjectPage
from pages.project_list_page import ProjectListPage
# 上海悠悠 wx:283340479
# blog:https://www.cnblogs.com/yoyoketang/


class TestMoreAccounts:
    """多账号切换操作示例"""

    @pytest.fixture(autouse=True)
    def start_for_each(self, page: Page, admin_context: BrowserContext):
        print("for each--start: 打开添加项目页")
        # 用户1
        self.user1_project = AddProjectPage(page)
        self.user1_project.navigate()
        # 用户2
        page2 = admin_context.new_page()
        self.user2_project = ProjectListPage(page2)
        self.user2_project.navigate()
        yield
        print("for each--end: 后置操作")
        page.close()
        page2.close()

    def test_delete_project(self):
        """
        测试流程:
        step--A账号登录，创建项目xxx
        step--B账号登录，删除项目xxx
        :return:
        """
        # 账号 1 添加项目
        test_project_name = str(uuid.uuid4()).replace('-', '')[:25]
        self.user1_project.fill_project_name(test_project_name)
        self.user1_project.fill_publish_app("xx")
        self.user1_project.fill_project_desc("xxx")
        # 断言跳转到项目列表页
        with self.user1_project.page.expect_navigation(url="**/list_project.html"):
```

```
        # 保存成功后，重定向到列表页
        self.user1_project.click_save_button()

    # 账号 2 操作删除
    self.user2_project.search_project(test_project_name)
    with self.user2_project.page.expect_request("**/api/project**"):
        self.user2_project.click_search_button()
    self.user2_project.page.wait_for_timeout(3000)
    self.user2_project.locator_table_delete.click()
    # 确定删除
    with self.user2_project.page.expect_response("**/api/project**") as resp:
        self.user2_project.locator_boot_box_accept.click()
    # 断言删除成功
    resp_obj = resp.value
    assert resp_obj.status == 200
```

以上是多账号同时登录，操作页面的解决思路，完整的项目代码示例在网易云课程上下载 https://study.163.com/course/introduction.htm?courseId=1213382811&*trace_c_p_k2*=6dc935a2397941faac5676bb9c57d1e9

网易云视频课程

网易云视频完整课程地址https://study.163.com/course/courseMain.htm?courseId=1213382811&share=2&shareId=480000002230338


Playwright + Python
Web 自动化测试
上海-悠悠