第5章

扩展现有自动化测试框架

5.1 开发 pytest 插件

在 2.3.10 节和 2.3.12 节中使用了 pytest-xdist、pytest-html 和 allure-pytest 3 个插件，插件是 pytest 的强大特性之一，其为第三方开发者提供了扩展 pytest 的无限可能。

在开发插件之前，首先了解一下插件的加载顺序：

第 1 步，禁用命令行参数-p no:plugin_name 指定的插件。该方法可禁用包括内置插件在内的所有插件。

第 2 步，加载全部内置插件。内置插件是通过扫描 pytest 的内部包(_pytest 包)加载的。

第 3 步，加载使用命令行参数-p plugin_name 指定的插件。

第 4 步，加载使用 setuptools 入口点注册的插件。

第 5 步，加载使用 PYTEST_PLUGINS 环境变量指定的插件。

第 6 步，加载 conftest.py 文件中全局变量 pytest_plugins 指定的插件。

5.1.1 使用 pytest Hook

pytest 的 Hook 回调函数是开发外部插件的基础，因此首先需要了解它们。

pytest 的 Hook 回调函数分为以下 6 类。

(1) 引导回调函数：引导回调函数的作用是便于内部或外部插件尽早地注册。

(2) 初始化回调函数：供插件或 conftest.py 文件用于初始化操作。

(3) 收集回调函数：用于文件和目录的收集。

(4) 测试运行回调函数：用于测试运行，它们大都接受一个 Item 对象。

（5）报告回调函数：用于测试报告的回调函数。

（6）调试/交互回调函数：用于开发调试或异常交互的回调函数。

下面以参数化测试的回调函数 pytest_generate_tests()为例介绍 pytest 回调函数的使用，pytest_generate_tests()属于收集类的回调函数。为此新增 extend_test_framework 包，在 extend_test_framework 包中新增 use_pytest_hook 模块，并编写 pytest_generate_tests()函数和 test_add()测试函数的代码。

【例 5-1】 使用 pytest_generate_tests()回调函数。

```
from pytest import main

from chapter_02.learning_unittest.calculator import Calculator


def pytest_generate_tests(metafunc):
    metafunc.parametrize(('num_01', 'num_02', 'expected'), [
        (1, 2, 3),
        (1, 3, 4)
    ])


def test_add(num_01, num_02, expected):
    assert Calculator().add(num_01, num_02) == expected


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

以上代码使用了 pytest_generate_tests()回调函数，其参数 metafunc 是一个 Metafunc 对象，用于生成参数化的测试函数/方法。Metafunc 对象包含一个 parametrize()方法，其使用方式类似于 2.3.7 节中介绍的@pytest.mark.parametrize 装饰器。

除了直接放在模块中，也可以将 pytest_generate_tests()回调函数放在 conftest.py 文件中，这样，在 conftest.py 文件的作用域内所有测试函数/方法均被参数化了。在 pytest 中，这种在 conftest.py 文件中使用回调函数的方式被称为 local per-directory plugins(本地每个目录单独的插件)。

5.1.2 开发本地插件

在 pytest 中，插件的实质就是包含了一个或多个 Hook 回调函数的独立 Python 模块。理解这一点后，开发一个插件就非常简单了。

先做一些准备工作。在 extend_test_framework 包中新增一个 calculator_plugin 子包，在 calculator_plugin 子包中新增 calculator_plugin 模块，将 5.1.1 节中的 pytest_generate_tests()回调函数复制到 calculator_plugin 模块中。

calculator_plugin 模块就是一个插件，为了使用插件，新增了一个名为 use_calculator_plugin 的 Python 模块，并在其中编写使用 calculator_plugin 插件的代码。

【例 5-2】　使用本地插件。

```
from pytest import main

from chapter_02.learning_unittest.calculator import Calculator

pytest_plugins = 'chapter_05.extend_test_framework.calculator_plugin.calculator_plugin'


def test_add(num_01, num_02, expected):
    assert Calculator().add(num_01, num_02) == expected


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

从以上代码可以看出，使用插件非常简单，只需要在测试模块中使用 pytest_plugins 全局变量注册即可。

除了在测试模块中注册，也可以在 conftest.py 文件中注册，代码如下：

```
pytest_plugins = 'chapter_05.extend_test_framework.calculator_plugin.calculator_plugin'
```

另外，如果需要注册多个插件，可将插件以列表形式传递给 pytest_plugins 全局变量，代码如下：

```
pytest_plugins = ['path.to.plugin_01', 'path.to.plugin_02']
```

本节开发的插件是一个本地插件，即没有共享到公共仓库中的插件，如果其他人需要使用该插件，就只能复制它的源代码。因此为了让其他人可以使用自己开发的插件，需要将插件共享到公共仓库，这就是 5.1.3 节将要介绍的可安装插件。

说明

如果只在公司内部共享插件，可以使用公司内部的私有仓库，将插件打包并上传到私有仓库后，公司内部的其他人可通过私有仓库下载和安装插件，有关私有仓库详见 7.3 节。

5.1.3　开发可安装的插件

开发可安装的插件需要依赖 setuptools 进行打包配置，配置完成后再进行打包和上传到公共仓库，本节以上传到 PyPI 为例。

在 extend_test_framework 包中新增 setup 模块，该模块中存放的是打包所需的配置信息。

【例 5-3】　打包所需的配置信息。

```
from setuptools import setup

setup(
```

```
    name = 'pytest - demo - plugin',
    version = '1.0.0',
    description = 'pytest 示例插件',
    author = '卢家涛',
    author_email = '522430860@qq.com',
    url = 'https://github.com/lujiatao2',
    packages = ['chapter_05.extend_test_framework.calculator_plugin'],
    entry_points = {
        'pytest11': ['calculator_plugin = chapter_05.extend_test_framework.calculator_
plugin.calculator_plugin']},
    classifiers = ['Framework :: Pytest']
)
```

以上代码只调用了 setuptools 的 setup()函数，现对以上 setup()函数的参数解释如下。

（1）name：应用程序的名称，这里指 pytest 插件的名称。在 PyPI 中，该应用程序的 URL 会被指定为 https://pypi.org/project/pytest-demo-plugin/，并可使用 pip install pytest-demo-plugin 命令来安装它。pytest 插件一般使用 pytest-为前缀命名，但这只是规范，并非强制要求。

（2）version：应用程序的版本号。

（3）description：应用程序的简单描述。

（4）author：作者姓名。

（5）author_email：作者电子邮箱。

（6）url：项目主页。

（7）packages：需要打包的目录。

（8）entry_points：入口点。要被 pytest 识别为插件，需将 Key 设置为 pytest11。

（9）classifiers：分类器。PyPI 将读取分类器的列表数据对应用程序进行分类，比如应用程序支持 Python 3.7，则可以写成 Programming Language:: Python :: 3.7。以上代码中的分类器将该应用程序分类为 pytest 框架，即 Framework:: Pytest。

说明

以上代码只是使用了基本的打包配置，更多打包配置详见 6.10.1 节。

为了能够顺利打包，在打包之前最好进行打包前的命令校验，命令如下：

```
python chapter_05\extend_test_framework\setup.py check
```

如果回显结果有告警，那么需要解决告警。比如缺失 URL，回显如下：

```
warning: check: missing required meta - data:url
```

如果回显结果没有告警，那么可以进行正式打包操作了。

由于当前 Python 应用程序的主流打包格式是 wheel，因此在打包之前需要安装 wheel 依赖，执行命令即可安装 wheel，命令如下：

```
pip install wheel
```

接着执行命令将应用程序打包到工程的 dist 目录，命令如下：

```
python chapter_05\extend_test_framework\setup.pysdist bdist_wheel
```

执行完成后，查看工程的 dist 目录，可以看到此时已经生成了 pytest_demo_plugin-1.0.0-py3-none-any.whl 和 pytest-demo-plugin-1.0.0.tar.gz 两个打包文件，前者是 wheel 格式的安装包，后者是源代码的压缩包。

打包完成后就可以上传到 PyPI 了。但在上传之前还需要安装 twine，它用于上传应用程序到 PyPI，执行命令即可安装 twine，命令如下：

```
pip install twine
```

最后执行命令上传应用程序到 PyPI，命令如下：

```
twine upload dist/*
```

执行以上命令时会要求输入 PyPI 的用户名和密码（可在 PyPI 上免费注册账户），输入成功后会自动开始上传，上传完成后可访问 PyPI 查看该应用程序，如图 5-1 所示。

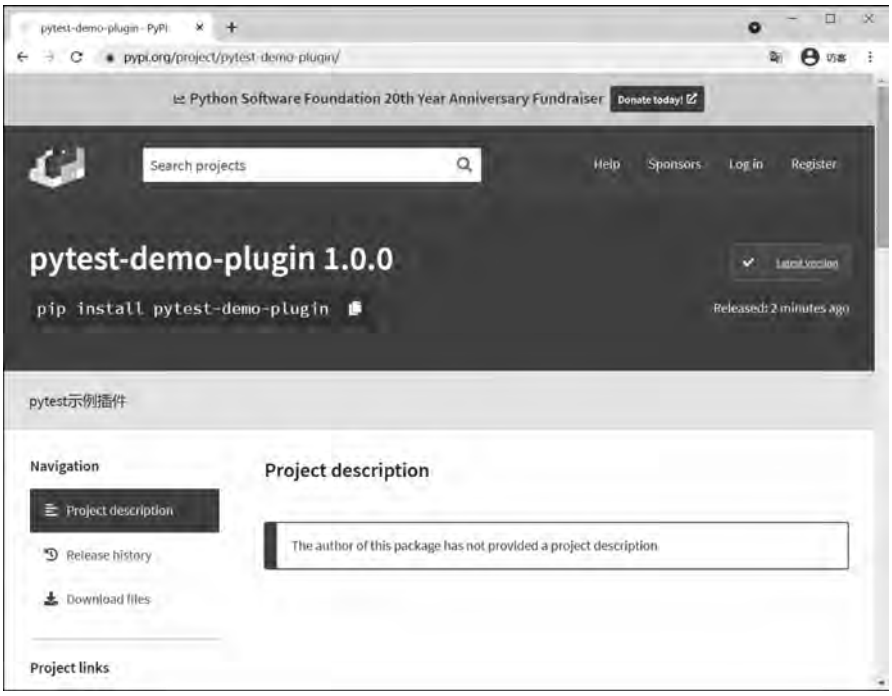

图 5-1　pytest-demo-plugin 的 PyPI 主页

说明

当修改应用程序后，必须修改版本号，否则 PyPI 将禁止上传的动作。即使将 PyPI 上的应用程序删除掉，也不能再上传相同版本号的应用程序。

由于当前工程存在 pytest-demo-plugin 插件的源代码，为了避免冲突，这里重新创建了一个工程 mastering-test-automation-for-test 用于 pytest-demo-plugin 插件的测试。该工程

同样也会作为第 6 章的测试工程。

由于这里使用了 Python 虚拟环境，因此需要在当前工程中重新安装 pytest，然后执行命令安装 pytest-demo-plugin 插件，命令如下：

```
pip install pytest - demo - plugin
```

安装完成后，在 mastering-test-automation-for-test 工程中新增 use_pytest_demo_plugin 包，并将 mastering-test-automation 工程中的 chapter_02\learning_unittest\calculator.py 模块复制到 use_pytest_demo_plugin 包中，最后新增 use_pytest_demo_plugin 模块，代码如下：

```
from pytest import main

from use_pytest_demo_plugin.calculator import Calculator


def test_add(num_01, num_02, expected):
    assert Calculator().add(num_01, num_02) == expected


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

执行命令运行测试代码，命令如下：

```
pytest -q use_pytest_demo_plugin\use_pytest_demo_plugin.py
```

执行结果为通过，因此说明插件已经被正确安装和使用了。

5.2 使用 Requests Hook

在实际项目中，当请求响应后通常需要先判断响应是否正常，如果正常，才会从响应中提取指定内容作为后续使用或作为实际结果进行断言。

第一种判断响应是否正常的方式是判断状态码，当状态码为 4××或 5××，表示响应异常。在 extend_test_framework 包中新增 use_requests_hook 模块，代码如下：

```
import requests

response = requests.get('https://httpbin.org/get')
if response.status_code < 400:
    print(response.json())
else:
    raise RuntimeError(f'请求失败：{response.reason}')
```

以上代码访问了 Response 对象的 status_code 属性来获取状态码，如果状态码小于 400，那么将以 JSON 形式打印响应体，否则抛出运行时异常。Response 对象的 reason 属性表示状态文本，如 404 Not Found 中的 Not Found。

在 Requests 中，Response 对象提供了一个 ok 属性来直接判断状态码是否小于 400，如

果小于400，就返回True，否则返回False。因此以上代码中的if语句可以被重构为使用ok属性，代码如下：

```
if response.ok:
    print(response.json())
else:
    raise RuntimeError(f'请求失败：{response.reason}')
```

说明

ok属性实际上是一个使用@property装饰器修饰的方法，其内部实现是依赖了raise_for_status()方法，而raise_for_status()方法的作用是检测状态码是否为4××或5××，如果是，就抛出HTTPError异常。

另一种判断响应是否正常的方式是判断响应体的code，这种判断方式依赖于具体项目。例如，约定当code不等于0时就代表请求异常，并且在message（具体名称取决于实际项目，也可能叫msg、errorMessage、errorMsg等）中描述异常的具体原因，代码如下：

```
response = requests.get('http://ims.lujiatao.com/api/goods-category')
body = response.json()
if body['code'] == 0:
    print(body)
else:
    raise RuntimeError(f'请求失败：{body["msg"]}')
```

由于还未登录IMS，因此直接获取物品分类会报错，执行以上代码后，执行结果如下：

```
Traceback (most recent call last):
  File "E:/Software_Testing/Software Development/Python/PycharmProjects/mastering-test-automation/chapter_05/extend_test_framework/use_requests_hook.py", line 13, in <module>
    raise RuntimeError(f'请求失败：{body["msg"]}')
RuntimeError: 请求失败：未登录!
```

现在的问题是如果自动化测试用例中需要调用成百上千个接口，每个接口都手动检测响应是否正常，那将存在大量冗余代码。幸亏Requests提供了Hook功能，可以在请求中注册回调函数来解决这个问题。

针对状态码的校验，可编写回调函数，代码如下：

```
def check_status_code(response, *args, **kwargs):
    if response.ok:
        return response
    else:
        raise RuntimeError(f'请求失败：{response.reason}')
```

回调函数将Response对象作为了第一个参数，然后在函数体中对它进行进一步的校验操作。

回调函数编写完成后，可在请求中传递给hooks参数以注册该回调函数，代码如下：

```
requests.get('http://ims.lujiatao.com/login', hooks={'response': check_status_code})
```

hooks 参数接受一个字典作为参数，目前字典 Key 仅支持 response，它表示对响应执行回调操作。

同理，也可以将判断响应体 code 的逻辑封装到回调函数中，代码如下：

```
def check_body_code(response, *args, **kwargs):
    body = response.json()
    if body['code'] == 0:
        return body
    else:
        raise RuntimeError(f'请求失败：{body["msg"]}')
```

要在请求中注册多个回调函数，只需要将它们以列表形式依次传入即可，代码如下：

```
requests.get('http://ims.lujiatao.com/api/goods-category', hooks={'response':[check_
status_code, check_body_code]})
```

另外，可将回调函数应用于会话，这样可对会话中的每个请求都执行相同的回调操作，代码如下：

```
with Session() as session:
    session.hooks['response'].append(check_status_code)
    session.hooks['response'].append(check_body_code)
    ...
```

append()方法用于注册单个回调函数，若需一次性注册多个，则可使用 extend()方法，代码如下：

```
session.hooks['response'].extend([check_status_code, check_body_code])
```

5.3 实现 Selenium 等待条件和事件监听器

5.3.1 实现 Selenium 等待条件

在 4.2.6 节中使用 WebDriverWait 类进行显式等待时，用到了 expected_conditions 模块内置的 invisibility_of_element_located 类作为等待条件，而由于 invisibility_of_element_located 类中显式定义了__call__()方法，因此其实例是可以被直接调用的。而 WebDriverWait 类的 until()方法，实际上是接受一个方法作为参数，既然如此完全可以不使用内置的 expected_conditions 模块，转而实现自定义的等待条件。

1. 使用 lambda 表达式

lambda 表达式实际上是一个匿名函数，因此可以将 4.2.6 节中的 invisibility_of_element_located 类直接替换为 lambda 表达式。

【例 5-4】 使用 lambda 表达式实现 Selenium 等待条件。

```
from selenium.webdriver import Chrome
from selenium.webdriver.support.wait import WebDriverWait
```

```
with Chrome() as driver:
    driver.get('http://wft.lujiatao.com/')
    driver.find_element_by_link_text('处理等待').click()
    web_driver_wait = WebDriverWait(driver, 60, poll_frequency=1)
    web_driver_wait.until(lambda d: not d.find_element_by_id('loading').is_displayed())
```

2. 使用独立的代码封装

Python 的 lambda 表达式被限制为只能包含一行代码，且其逻辑与数据是耦合在一起的。因此使用独立的代码封装是一种更好的选择，这里使用的是一个 Python 类作为独立的代码封装。为此在 extend_test_framework 包中新建一个 my_expected_conditions 模块，在该模块中新增一个 InvisibilityOfElement 类，该类意在提供与 invisibility_of_element_located 类一样的功能。

【例 5-5】 使用 Python 类实现 Selenium 等待条件。

```
class InvisibilityOfElement:

    def __init__(self, locator):
        self.locator = locator

    def __call__(self, driver):
        element = driver.find_element(*self.locator)
        return element if not element.is_displayed() else False
```

接着将 lambda 表达式替换为 InvisibilityOfElement 类即可，代码如下：

```
web_driver_wait.until(InvisibilityOfElement((By.ID, 'loading')))
```

5.3.2　实现 Selenium 事件监听器

事件监听器用于在特定事件发生时触发特定操作，常见的一个场景是当 Selenium 抛出异常时，对窗口进行截图留存，以便后续的回溯工作。

要实现一个事件监听器，需要继承 AbstractEventListener 类，并重写其中的方法，因为 AbstractEventListener 类中的方法都是空实现，如表 5-1 所示。

表 5-1　AbstractEventListener 类的方法

方法名称	方法含义
before_navigate_to	导航到目标页面之前触发的操作
after_navigate_to	导航到目标页面之后触发的操作
before_navigate_back	返回上一个页面之前触发的操作
after_navigate_back	返回上一个页面之后触发的操作
before_navigate_forward	前进到下一个页面之前触发的操作
after_navigate_forward	前进到下一个页面之后触发的操作
before_find	查找元素之前触发的操作
after_find	查找元素之后触发的操作
before_click	单击元素之前触发的操作

续表

方法名称	方法含义
after_click	单击元素之后触发的操作
before_change_value_of	改变元素值之前触发的操作
after_change_value_of	改变元素值之后触发的操作
before_execute_script	执行脚本之前触发的操作
after_execute_script	执行脚本之后触发的操作
before_close	关闭窗口之前触发的操作
after_close	关闭窗口之后触发的操作
before_quit	结束浏览器会话之前触发的操作
after_quit	结束浏览器会话之后触发的操作
on_exception	Selenium 抛出异常时触发的操作

下面以 on_exception()方法的重写为例介绍事件监听器的实现。为此新增 my_listener 模块，并新增一个 MyListener 类用于继承 AbstractEventListener 类。

【例 5-6】 实现 Selenium 事件监听器。

```
from datetime import datetime

from selenium.webdriver.support.abstract_event_listener import AbstractEventListener


class MyListener(AbstractEventListener):

    def on_exception(self, exception, driver):
        filename = datetime.now().strftime('%Y%m%d%H%M%S')
        driver.save_screenshot(fr'E:\Other\{filename}.png')
```

以上代码重写了 on_exception()方法，该方法用于在 Selenium 抛出异常时对屏幕进行截图保存，截图名称是当前时间，精确到秒。

事件监听器需要配合 EventFiringWebDriver 类才能使用，在 EventFiringWebDriver 的构造方法中将事件监听器对象作为参数传递即可。以访问 WFT 首页，并尝试查找一个不存在的元素为例。为此新增 implement_listener 模块，代码如下：

```
from selenium.webdriver import Chrome
from selenium.webdriver.support.event_firing_webdriver import EventFiringWebDriver

from chapter_05.extend_test_framework.my_listener import MyListener

with Chrome() as driver:
    new_driver = EventFiringWebDriver(driver, MyListener())
    new_driver.get('http://wft.lujiatao.com/')
    new_driver.find_element_by_id('no-id')
```

执行以上代码后，E:\Other 目录生成了截图。