用deepseek+vscode自动完成测试脚本

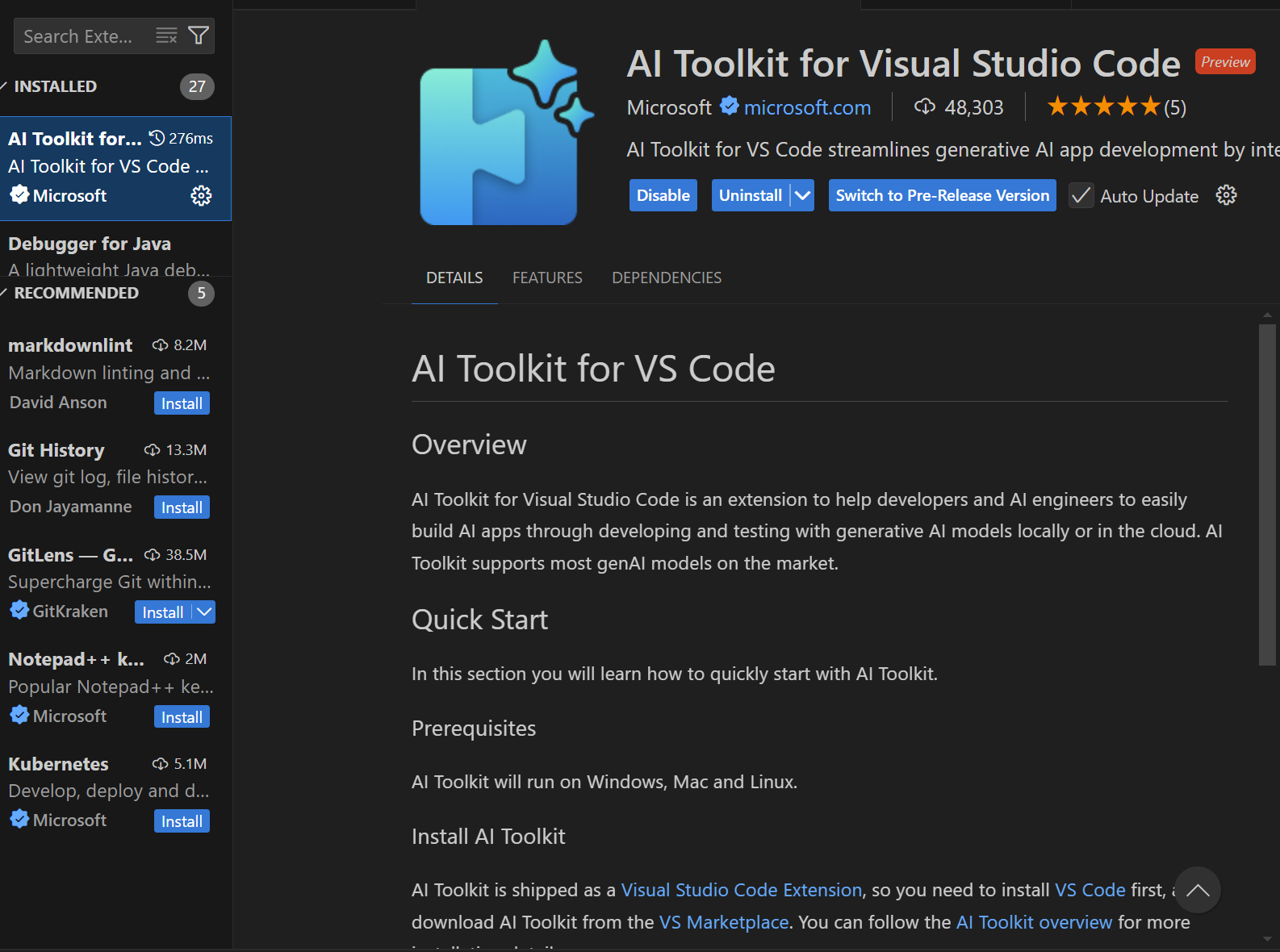
2025-01-31T21:29:12+08:00

近几日Deepseek大火，我这里也尝试了一下，确实很强。而目前vscode的AI toolkit插件也已经集成了deepseek R1，这里就介绍下在vscode中利用deepseek帮助我们完成自动化测试脚本的实践分享

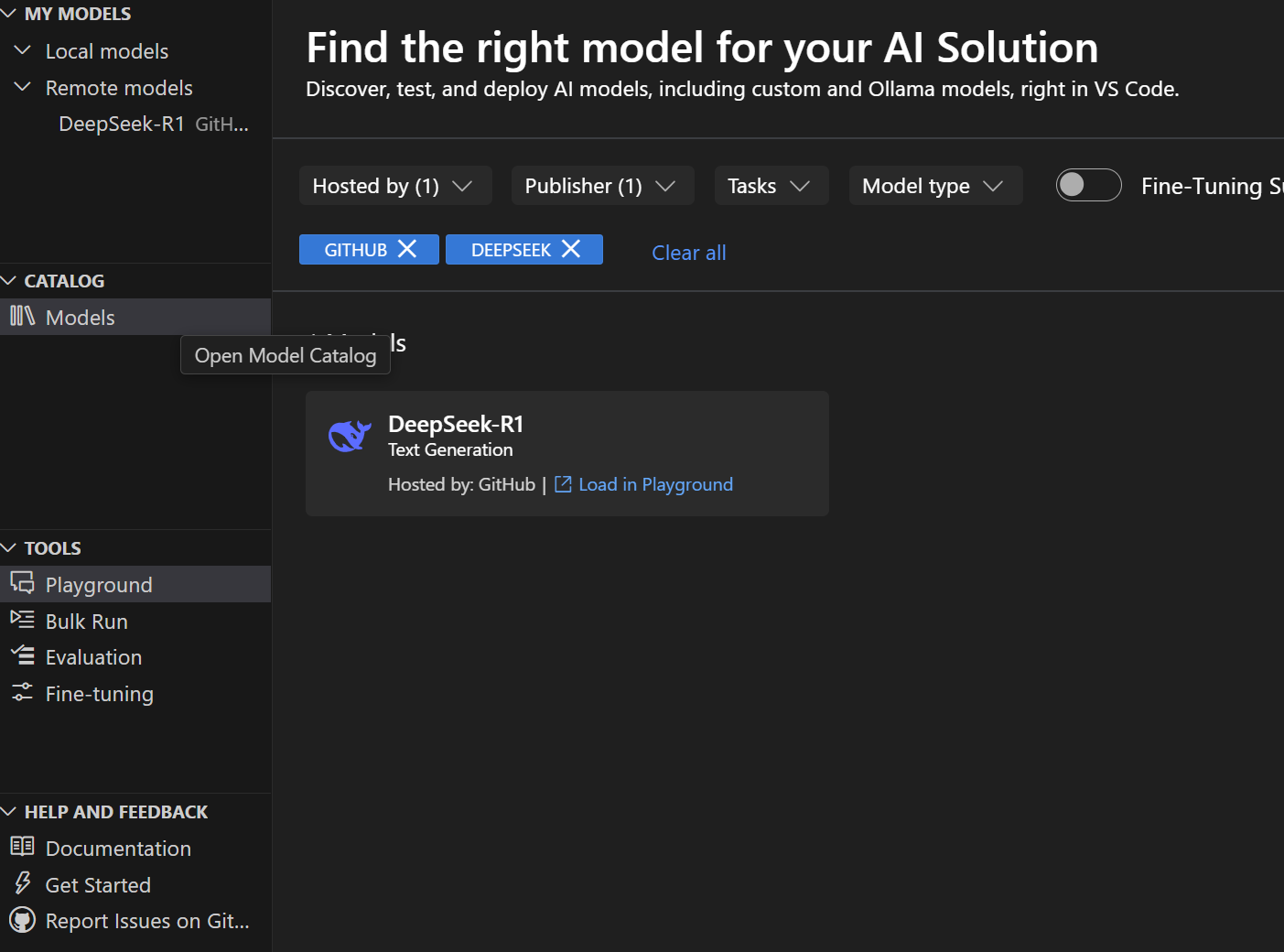
## 安装AI ToolKit并启用Deepseek

微软官方提供了一个针对AI辅助的插件，也就是 **AI Toolkit for Visual Studio Code**， 这个插件可以利用各种开源或商用的大模型，目前也已经集成了Deepseek R1模型，可以让我们在VS code中方便地利用Deepseek来帮助我们的代码开发

首先在插件中心中找到**AI Toolkit**并完成安装



安装完成插件以后，在插件页面，选择catalog中的models模型页面，并通过筛选条件 hosted by Github, 以及 publisher Deepseek 使用Deepseek-R1模型



## AI帮助生成脚本

有了deepseek模型后，我们就可以新建一个playgroud让deepseek帮助我们生成自动化脚本了，以下是我让Deepseek生成的针对saucedemo的测试脚本



### 思考过程

嗯，我现在需要帮忙设计一个基于pytest和Playwright的自动化测试项目，针对saucedemo.com的登录和checkout流程，使用POM模式，用户数据放在user.yml里，还要包含异常场景。好的，我得一步一步来。

首先，应该确定项目的结构。POM模式的话，通常会有page对象、测试用例、配置文件、数据文件等。可能需要这样的目录结构：

* tests/
  + pages/
    - login\_page.py
    - inventory\_page.py
    - cart\_page.py
    - checkout\_page.py
  + test\_login.py
  + test\_checkout.py
* conftest.py
* user.yml
* pytest.ini

然后，用户数据存在user.yml里，里面应该有不同的用户类型，比如标准用户，锁定用户，错误密码，没有用户等。然后测试用例要读取这些数据。

接下来是Page Object Model的实现。每个页面比如登录页、商品页，购物车页、结算页都有自己的类，封装元素和操作。比如LoginPage有用户名、密码输入框，登录按钮，错误提示的方法。

测试用例应该使用pytest的参数化来覆盖不同的用户场景。比如正常登录成功，无效密码，锁定用户等。同时，checkout流程需要正常流程的测试，可能还有填写信息不完整的情况。

然后考虑conftest.py里面设置Playwright的fixture，比如browser，page，以及各个page对象的fixture。比如，每次测试前都转到登录页，登录后到库存页，这样可以处理前置条件。

异常场景的话，例如登录时的各种错误信息，结账时的表单验证，比如必须填first name等。需要编写这些测试用例，并验证错误提示是否正确。

用户的数据如何加载呢？可能用pyYAML库来读取user.yml文件，里面分不同用户的数据。测试时用@pytest.mark.parametrize参数化输入不同的用户数据。

比如，用户数据分成valid\_users，invalid\_password，locked\_user等键。在测试登录的时候，参数化这些情况，触发不同的测试用例。

在测试checkout流程时，可能需要先登录，然后添加商品到购物车，进入购物车，进入结算页，填写信息，完成checkout。这时候，正常的测试用例需要用有效的数据，而异常情况比如不填写某些字段需要验证错误提示。

还需要处理元素的选择器，Playwright的定位方法，比如使用CSS选择器或文本定位。例如，登录页可能通过id定位用户名和密码输入框。

此外，测试完成后需要清理状态吗？比如，购物车中的商品或者是否需要登出，或者每个测试用例之间保持独立，可能使用 fixtures 来确保各自的状态。

也可能需要处理异步操作，不过Playwright是自动处理等待的，所以在页面导航或元素点击时不需要额外处理。

现在，具体的步骤：

1. 创建项目结构，安装依赖：pytest, playwright, pyyaml。
2. 编写user.yml，包含不同类型的用户数据。
3. 实现各个页面的Page类，包含元素定位和操作方法。
4. 在conftest.py中定义fixture，例如初始化page，创建各page对象的实例。
5. 编写测试用例，参数化用户数据，覆盖正常和异常流程。
6. 处理测试中的等待和断言，确保元素的存在或文本正确。

需要检查测试用例是否覆盖以下场景：

登录：

* 正确用户密码登录成功。
* 错误密码提示错误信息。
* 锁定用户无法登录。
* 空用户名或密码提示错误。

Checkout：

* 正常填写所有信息，完成订单。
* 必填字段缺失（如first name为空）提示错误。
* 取消订单流程。

现在，user.yml的结构应该是什么样的？

可能如下：

users: valid\_user: username: standard\_user password: secret\_sauce invalid\_password\_user: username: standard\_user password: wrongpass locked\_user: username: locked\_out\_user password: secret\_sauce missing\_username: username: “” password: secret\_sauce missing\_password: username: standard\_user password: “”

在测试登录的时候，参数化这些情况。比如，对于valid\_user，应该登录成功，跳转到/inventory.html。对于invalid\_password，显示Epic sadface: Username and password do not match。锁定的用户提示Epic sadface: Sorry, this user has been locked out。缺少用户名或密码提示相应的错误信息。

对于checkout流程，可行的步骤是：

登录后，添加一个商品到购物车，点击购物车图标，进入购物车页，点击checkout按钮，填写信息，继续，完成。

所以测试用例需要参数化填写的信息是否正确。例如：

正常情况下的信息：

first\_name: John last\_name: Doe zip: 12345

异常情况，比如不填first name，是否提示错误。

在编写这些测试时，需要使用页面对象的方法来完成每个步骤，比如添加商品，进入购物车，填写信息，继续按钮等。

可能的问题点：

元素的定位是否正确，特别是动态生成的内容或者异步加载的部分是否需要等待。不过Playwright有自动等待机制，所以选择元素时只要正确选择应该没问题。

需要断言的地方包括：登录后的页面URL是否正确，订单完成后的确认信息是否存在，错误提示是否显示正确的文本。

比如，登录成功后应该跳转到/inventory.html；checkout完成后应该显示“Thank you for your order”。

另外，可能需要清理购物车，但因为每个测试可能独立运行，或者每个测试都重新开始会话，所以在fixture中可以处理，如在每次测试前登录并重置状态？

或者使用类似于beforeEach的fixture，让每个测试用例开始时处于某个状态。比如，登录后的页面。

此外，测试购物流程需要添加商品到购物车，所以库存页面的类需要有添加商品的方法。

Page类示例：

InventoryPage可以有添加某个商品的方法，比如点击“Add to cart”按钮。比如，根据商品名称定位对应的按钮。不过这可能比较复杂，也许在页面上使用通用的方法。

或者，在InventoryPage中有方法add\_item\_to\_cart(item\_name)，用于点击对应商品的添加按钮。这时需要使用Playwright的定位方法，例如：

def add\_item\_to\_cart(self, item\_name): self.page.locator(f”text={item\_name}“).locator(”xpath=..”).locator(“button”).click()

不过这可能需要根据实际的DOM结构来确定。可能需要inspector查看元素结构。

例如，每个商品项的结构可能类似：

Sauce Labs Backpack

Add to cart

所以，找到对应的名称元素，然后找到其父级元素中的按钮。

或者，更简单的办法是每个商品的“Add to cart”按钮有不同的id，比如 add-to-cart-sauce-labs-backpack，或其他属性。需要看实际网站的结构。

登录后的页面可能有多个商品，每个商品对应的添加按钮可能需要通过其文本来定位对应的按钮。或者通过其他属性。

现在，需要实际查看saucedemo.com的结构。可能使用Playwright的代码生成功能来获取正确的选择器。

假设，例如，每个商品的“Add to cart”按钮的id是动态生成的或者类似，例如对于“Sauce Labs Backpack”，对应的按钮的id可能是add-to-cart-sauce-labs-backpack，那么可以直接通过这个id去定位。

这样的话，InventoryPage中可以有一个方法：

def add\_product\_to\_cart(self, product\_name): product\_id = product\_name.lower().replace(’ ‘,’-’) self.page.click(f”#add-to-cart-{product\_id}“)

当然，这需要查看实际的网站结构是否正确。

假设正确的商品example在这里是Sauce Labs Backpack，其id可能确实是这样的，这样方法可行。

另外，购物车图标上的数量是否正确，当商品添加后，购物车的图标显示数量为1。当进入购物车页，是否有对应的商品。

需要注意，每个操作之后的页面变化是否需要新的Page对象实例，比如在InventoryPage点击购物车后跳转到CartPage，所以该操作应该返回CartPage的实例，从而能够继续操作。

这样方法的返回值处理可能成为问题。比如：

在InventoryPage中的方法：

def go\_to\_cart(self): self.page.click(“.shopping\_cart\_link”) return CartPage(self.page)

类似这样，每个导航动作返回新的Page对象。

这会需要在各个页面的方法中处理返回相应的页面对象。

然后是测试流程的组装：

在测试checkout的时候，可能步骤很多，需要写一个大的测试用例，分步骤进行：

例如：

def test\_checkout\_process(login\_page, inventory\_page, checkout\_data): # 登录 login\_page.login(“standard\_user”, “secret\_sauce”) # 添加商品 inventory\_page.add\_product\_to\_cart(“Sauce Labs Backpack”) inventory\_page.go\_to\_cart() # 进入购物车页面，点击checkout cart\_page.click\_checkout() # 填写checkout信息 checkout\_page.fill\_shipping\_info(checkout\_data) checkout\_page.continue\_to\_overview() # 确认订单 checkout\_page.finish\_checkout() # 断言是否成功 assert checkout\_page.get\_order\_confirmation() == “Thank you for your order!”

当然，这可能涉及到许多页面的跳转，每个页面之间需要用对应的方法来连接。

异常场景的话，比如在填写信息的时候不输入first name，点击continue时，出现错误提示。

此时，测试用例可能参数化不同的数据，比如：

@pytest.mark.parametrize(“first\_name, last\_name, zip\_code, error\_message”, [ (““,”Doe”, “12345”, “Error: First Name is required”), (“John”, ““,”12345”, “Error: Last Name is required”), (“John”, “Doe”, ““,”Error: Postal Code is required”), ]) def test\_checkout\_missing\_fields(…, first\_name, last\_name, zip\_code, error\_message): # 填信息时，漏掉对应字段 checkout\_page.fill\_shipping\_info(first\_name, last\_name, zip\_code) checkout\_page.continue\_to\_overview() assert checkout\_page.get\_error\_message() == error\_message

可能还需要考虑，错误信息提示元素的定位是否正确，以及当出现错误时是否阻止进入下一步。

现在，问题可能是如何组织这些数据和页面方法。

另外，关于用户数据的读取，测试用例需要从YAML文件中加载。可能的做法是在conftest.py里面写个fixture读取YAML的数据，然后参数化测试用例。

但pytest的参数化可能需要特定的方式处理，可能将YAML数据转换成测试参数。或者使用外部数据源来参数化测试。

例如，在conftest.py中读取user.yml的所有用户数据，并为登录测试创建一个参数化的fixture。

或者，可以在测试用例中直接读取YAML文件中的数据，并将其参数化。

比如：

import yaml

def load\_users(): with open(‘user.yml’) as f: data = yaml.safe\_load(f) return data[‘users’]

然后在测试用例中用：

@pytest.mark.parametrize(“user”, [ (“valid\_user”), (“invalid\_password\_user”), (“locked\_user”), # … 其他情况 ], indirect=True) def test\_login(login\_page, user): login\_page.login(user[‘username’], user[‘password’]) # 断言等操作

不过，如何将数据转换为参数可能需要更细致的处理，比如将各个用户数据提取成各个测试实例的参数。

可能更好的方法是将不同的用户情况分别写成不同的测试数据项。例如，把不同用户类型数据的键列出来，然后根据这些键从数据中加载相应的参数。

或者，在所有参数化时，导入所有用户数据作为参数。

比如：

import yaml import pytest

with open(‘user.yml’) as f: user\_data = yaml.safe\_load(f)[‘users’]

login\_test\_data = [ (user\_data[‘valid\_user’], True, None), # 正常用户，登录成功，没有错误 (user\_data[‘invalid\_password\_user’], False, ‘Epic sadface: Username and password do not match’), (user\_data[‘locked\_user’], False, ‘Epic sadface: Sorry, this user has been locked out.’), # … 其他情况 ]

@pytest.mark.parametrize(“user, success, error\_msg”, login\_test\_data) def test\_login\_user(login\_page, user, success, error\_msg): login\_page.navigate() login\_page.login(user[‘username’], user[‘password’]) if success: assert login\_page.current\_url == ‘https://www.saucedemo.com/inventory.html’ else: assert login\_page.get\_error\_message() == error\_msg

这时候，需要在user.yml里的每个用户数据中有一个error键，或者在参数化的准备阶段处理错误信息的映射。

不过这样的参数方式可能更灵活，能够同时指定预期结果。

对于YAML的结构，可能需要更详细的数据项，每个用户数据包含预期的错误消息。

例如，user.yml部分内容：

valid\_user: username: standard\_user password: secret\_sauce expected: success: true error: null

invalid\_password\_user: username: standard\_user password: wrong expected: success: false error: “Epic sadface: Username and password do not match”

locked\_user: username: locked\_out\_user password: secret\_sauce expected: success: false error: “Epic sadface: Sorry, this user has been locked out.”

这样参数化时可以更具结构化。但在编写YAML时需要考虑每个用例的期望结果。

这样，测试用例的参数可能会是直接从YAML中读取的结构，然后通过解析不同的用户组生成测试用例参数。

不过，这可能有些复杂。另一个办法是，将每个测试案例定义为YAML中的一个项目，包含输入数据和预期结果。

总之，这部分的处理可能需要更多设计。

然后，关于Playwright的fixture设置，在conftest.py里可以使用pytest-playwright提供的fixture。例如，使用pytest-playwright插件中的page fixture。不过这可能需要安装pytest-playwright库，并调用playwright的安装步骤。

安装依赖包括：

pytest-playwright playwright pyyaml

执行playwright install安装浏览器。

然后，在conftest.py中，定义如何初始化各个page对象，并将它们作为fixture注入到测试用例中。

例如：

import pytest from playwright.sync\_api import Page from pages.login\_page import LoginPage from pages.inventory\_page import InventoryPage

##### 其他页面的导入

@pytest.fixture def login\_page(page: Page): return LoginPage(page)

@pytest.fixture def inventory\_page(page: Page): return InventoryPage(page)

等等。这样每个测试用例如果需要用到特定的页面对象，可以直接在参数中声明需要的fixture。

例如，一个测试用例函数可能接收login\_page和inventory\_page作为参数。

接下来，各个Page类的示例。

LoginPage类是这样的：

class LoginPage: def **init**(self, page): self.page = page self.username\_input = page.locator(“#user-name”) self.password\_input = page.locator(“#password”) self.login\_button = page.locator(“#login-button”) self.error\_message = page.locator(“.error-message-container”)

def navigate(self):  
 self.page.goto("https://www.saucedemo.com")  
  
def login(self, username, password):  
 self.username\_input.fill(username)  
 self.password\_input.fill(password)  
 self.login\_button.click()  
  
def get\_error\_message(self):  
 return self.error\_message.inner\_text()

但实际情况需要根据saucedemo.com的实际HTML结构调整选择器。比如，错误信息的元素可能是div.error-message-container带有data-test=“error”的属性，其文本可能在某个子元素内。

比如说，实际的网站中，错误信息的HTML可能是：

Epic sadface: Username and password do not match any user in this service

那么，error\_message的locator需要定位到h3元素的文本。或者，定位到整个容器，然后用inner\_text()获取文本。

或者，正确的选择器可能是：

error\_message = page.locator(‘[data-test=“error”]’)

但Playwright支持多种选择方式，比如用data-testid属性等，所以在编写时需要查证正确的元素定位方式。

所以可能更好的方式是使用data-test属性来选择元素，例如登录页的数据测试属性：

用户名输入：data-test=“username”

密码输入：data-test=“password”

登录按钮：data-test=“login-button”

错误信息元素：data-test=“error”，所以其元素可以定位为：

self.error\_message = page.locator(‘[data-test=“error”]’)

然后其文本内容可以用error\_message.inner\_text()来获取，包含错误信息的关键部分。

至于其他页面，比如库存页面，购物车页等：

InventoryPage可能有添加商品到购物车的方法，跳转到购物车的方法，或许购物车数量的显示。例如：

class InventoryPage: def **init**(self, page: Page): self.page = page

def add\_product\_to\_cart(self, product\_name): # 这里假设商品名称对应的添加按钮如何定位，假设data-test属性中有对应的add-to-cart按钮，例如 data-test=“add-to-cart-sauce-labs-backpack” product\_id = product\_name.lower().replace(’ ‘,’-‘) self.page.locator(f’[data-test=“add-to-cart-{product\_id}”]’).click()

def get\_cart\_count(self): return self.page.locator(‘.shopping\_cart\_badge’).inner\_text()

def go\_to\_cart(self): self.page.locator(‘.shopping\_cart\_link’).click() return CartPage(self.page)

CartPage类，可能有checkout按钮：

class CartPage: def **init**(self, page: Page): self.page = page

def checkout(self):  
 self.page.locator('[data-test="checkout"]').click()  
 return CheckoutPage(self.page)

CheckoutPage类：

class CheckoutPage: def **init**(self, page: Page): self.page = page self.first\_name\_input = page.locator(‘[data-test=“firstName”]’) self.last\_name\_input = page.locator(‘[data-test=“lastName”]’) self.postal\_code\_input = page.locator(‘[data-test=“postalCode”]’) self.continue\_button = page.locator(‘[data-test=“continue”]’) self.error\_message = page.locator(‘[data-test=“error”]’)

def fill\_shipping\_info(self, first\_name, last\_name, postal\_code): self.first\_name\_input.fill(first\_name) self.last\_name\_input.fill(last\_name) self.postal\_code\_input.fill(postal\_code)

def continue\_to\_overview(self): self.continue\_button.click()

def get\_error\_message(self): return self.error\_message.inner\_text().strip()

然后，在完成信息填写后的页面，可能有finish按钮，在CheckoutOverviewPage类中：

class CheckoutOverviewPage: def **init**(self, page: Page): self.page = page self.finish\_button = page.locator(‘[data-test=“finish”]’) self.confirmation\_message = page.locator(‘.complete-header’)

def finish\_checkout(self): self.finish\_button.click()

def get\_confirmation\_message(self): return self.confirmation\_message.inner\_text()

当然，这些需要根据实际页面的结构进行调整，比如确认信息的元素是否正确。

这样，整体测试流程就可以通过各个页面的方法来串联。

现在编写测试用例的步骤：

测试登录流程：

参数化不同的用户数据，判断登录是否成功，或者是否出现正确的错误提示。

例如：

在test\_login.py中：

import pytest from playwright.sync\_api import Page

@pytest.mark.parametrize(“user\_key, expected\_success, expected\_error”, [ (“valid\_user”, True, None), (“invalid\_password\_user”, False, “Epic sadface: Username and password do not match”), (“locked\_user”, False, “Epic sadface: Sorry, this user has been locked out.”), (“missing\_username”, False, “Epic sadface: Username is required”), (“missing\_password”, False, “Epic sadface: Password is required”), ]) def test\_login(login\_page, user\_key, expected\_success, expected\_error): # 从YAML加载用户数据，这里假设有一个fixture可以加载全部的测试用户数据 users = load\_users() user = users[user\_key] login\_page.navigate() login\_page.login(user[‘username’], user[‘password’])

if expected\_success: assert login\_page.page.url == “https://www.saucedemo.com/inventory.html” else: assert login\_page.get\_error\_message() == expected\_error

需要写出load\_users函数，或者使用fixture来传递给测试用例。

这里的load\_users可能是在conftest.py中的一个fixture：

import pytest import yaml

@pytest.fixture def users(): with open(‘user.yml’) as f: data = yaml.safe\_load(f) return data[‘users’]

然后测试用例可以接收users作为一个fixture：

def test\_login(login\_page, users, user\_key, expected\_success, …): user = users[user\_key] …

但这种参数化可能需要结合indirect参数的使用，或者在参数化时动态获取用户数据。

或者，将参数化的数据和用户密钥对应起来，通过字典获取正确的用户数据。

这可能需要对pytest的参数化更深入的处理，例如使用间接参数化，将用户键作为参数，然后在测试函数内部获取相应用户的数据。

或者，这种参数传递方式可能不太直接，可能需要重新设计参数化的结构。

可能最好的方式在参数化的列表中直接指定用户数据的键，然后在测试用例中使用该键来索引users fixture中的对应数据。

例如：

@pytest.mark.parametrize(“user\_key, expected\_success, expected\_error”, [ (“valid\_user”, True, None), (“invalid\_password\_user”, False, “Epic sadface: Username and password do not match”), (“locked\_user”, False, “Epic sadface: Sorry, this user has been locked out.”), (“missing\_username”, False, “Epic sadface: Username is required”), (“missing\_password”, False, “Epic sadface: Password is required”), ]) def test\_login(login\_page, users, user\_key, expected\_success, expected\_error): user = users[user\_key] login\_page.navigate() login\_page.login(user[‘username’], user[‘password’]) if expected\_success: assert login\_page.page.url == “https://www.saucedemo.com/inventory.html” else: assert login\_page.get\_error\_message() == expected\_error

需要确保user.yml中存在相应的键，且各用户的数据正确。

对于checkout流程的正常测试：

在test\_checkout.py中：

def test\_successful\_checkout(login\_page, inventory\_page, users): valid\_user = users[‘valid\_user’] login\_page.login(valid\_user[‘username’], valid\_user[‘password’]) inventory\_page.add\_product\_to\_cart(“Sauce Labs Backpack”) cart\_page = inventory\_page.go\_to\_cart() checkout\_page = cart\_page.checkout() checkout\_page.fill\_shipping\_info(“John”, “Doe”, “12345”) checkout\_page.continue\_to\_overview() overview\_page = CheckoutOverviewPage(checkout\_page.page) overview\_page.finish\_checkout() assert overview\_page.get\_confirmation\_message() == “Thank you for your order!”

这需要定义CheckoutOverviewPage类的相应元素和方法。

对于异常测试：

比如，填写信息时省略必填字段：

@pytest.mark.parametrize(“first\_name, last\_name, zip\_code, expected\_error”, [ (““,”Doe”, “12345”, “Error: First Name is required”), (“John”, ““,”12345”, “Error: Last Name is required”), (“John”, “Doe”, ““,”Error: Postal Code is required”), ]) def test\_checkout\_error\_messages(login\_page, inventory\_page, users, first\_name, last\_name, zip\_code, expected\_error): valid\_user = users[‘valid\_user’] login\_page.login(valid\_user[‘username’], valid\_user[‘password’]) inventory\_page.add\_product\_to\_cart(“Sauce Labs Backpack”) cart\_page = inventory\_page.go\_to\_cart() checkout\_page = cart\_page.checkout() checkout\_page.fill\_shipping\_info(first\_name, last\_name, zip\_code) checkout\_page.continue\_to\_overview() assert checkout\_page.get\_error\_message() == expected\_error

这里，需要确定实际的错误提示是否是类似这样的文本。比如，在reqreal的网站上，不填first name并继续时，错误提示是否真的是“Error: First Name is required”。需要查看实际页面的情况，可能错误信息是否包含在data-test=“error”元素中，或者是否是其他结构。

综上所述，整个项目需要：

* 正确配置POM的各个页面类。
* 正确读取用户数据并参数化测试用例。
* 处理不同测试场景的流程，尤其是Test checkout的流程可能需要连续的页面导航和状态维护。
* 对异常情况进行断言，如错误消息的正确性。

可能还需要对页面元素的同步进行确认，尽管Playwright自动等待，但例如，点击按钮后的跳转是否完成可能会需要确认页面的某些元素，这样更稳定。

对于一些异步动作，比如点击按钮后元素变化，可以不用等待，因为Playwright内部已经处理了。

总的来说，这样的设计应该能够覆盖所需的自动化测试需求。接下来，我需要根据这些思路编写具体的代码文件，并验证是否正确工作。

### 具体实现

为了基于pytest和Playwright完成针对saucedemo.com的自动化测试登录和checkout流程，并遵循POM模式，以下是详细的实现方案：

### 项目结构

project-root/  
├── tests/  
│ ├── pages/  
│ │ ├── login\_page.py  
│ │ ├── inventory\_page.py  
│ │ ├── cart\_page.py  
│ │ ├── checkout\_page.py  
│ │ └── checkout\_overview\_page.py  
│ ├── test\_login.py  
│ └── test\_checkout.py  
├── conftest.py  
├── user.yml  
└── pytest.ini

### 依赖安装

1. 安装所需库：

pip install pytest playwright pyyaml pytest-playwright  
playwright install

### 文件实现

#### user.yml

users:  
 valid\_user:  
 username: standard\_user  
 password: secret\_sauce  
 invalid\_password\_user:  
 username: standard\_user  
 password: wrong\_pass  
 locked\_user:  
 username: locked\_out\_user  
 password: secret\_sauce  
 missing\_username:  
 username: ""  
 password: secret\_sauce  
 missing\_password:  
 username: standard\_user  
 password: ""

#### pages/login\_page.py

class LoginPage:  
 def \_\_init\_\_(self, page):  
 self.page = page  
 self.username\_input = page.locator('[data-test="username"]')  
 self.password\_input = page.locator('[data-test="password"]')  
 self.login\_button = page.locator('[data-test="login-button"]')  
 self.error\_message = page.locator('[data-test="error"]')  
  
 def navigate(self):  
 """导航到登录页"""  
 self.page.goto("https://www.saucedemo.com")  
  
 def login(self, username, password):  
 """填充并提交登录表单"""  
 self.username\_input.fill(username)  
 self.password\_input.fill(password)  
 self.login\_button.click()  
   
 def get\_error\_message(self):  
 """获取错误提示文本"""  
 return self.error\_message.inner\_text()

#### pages/inventory\_page.py

class InventoryPage:  
 def \_\_init\_\_(self, page):  
 self.page = page  
  
 def add\_product\_to\_cart(self, product\_name):  
 """将指定商品添加到购物车"""  
 product\_id = product\_name.lower().replace(' ', '-')  
 self.page.locator(f'[data-test="add-to-cart-{product\_id}"]').click()  
  
 def go\_to\_cart(self):  
 """进入购物车页面"""  
 self.page.locator('.shopping\_cart\_link').click()  
 return CartPage(self.page)

#### pages/cart\_page.py

class CartPage:  
 def \_\_init\_\_(self, page):  
 self.page = page  
   
 def checkout(self):  
 """进入结算页面"""  
 self.page.locator('[data-test="checkout"]').click()  
 return CheckoutPage(self.page)

#### pages/checkout\_page.py

class CheckoutPage:  
 def \_\_init\_\_(self, page):  
 self.page = page  
 self.first\_name\_input = page.locator('[data-test="firstName"]')  
 self.last\_name\_input = page.locator('[data-test="lastName"]')  
 self.postal\_code\_input = page.locator('[data-test="postalCode"]')  
 self.continue\_button = page.locator('[data-test="continue"]')  
 self.error\_message = page.locator('[data-test="error"]')  
  
 def fill\_shipping\_info(self, first\_name, last\_name, postal\_code):  
 """填写配送信息"""  
 self.first\_name\_input.fill(first\_name)  
 self.last\_name\_input.fill(last\_name)  
 self.postal\_code\_input.fill(postal\_code)  
  
 def continue\_to\_overview(self):  
 """提交并进入订单概览"""  
 self.continue\_button.click()  
   
 def get\_error\_message(self):  
 """获取错误提示文本"""  
 return self.error\_message.inner\_text()

#### pages/checkout\_overview\_page.py

class CheckoutOverviewPage:  
 def \_\_init\_\_(self, page):  
 self.page = page  
 self.finish\_button = page.locator('[data-test="finish"]')  
 self.confirmation\_message = page.locator('.complete-header')  
  
 def finish\_checkout(self):  
 """完成订单"""  
 self.finish\_button.click()  
  
 def get\_confirmation\_message(self):  
 """获取订单完成确认信息"""  
 return self.confirmation\_message.inner\_text()

#### conftest.py

import pytest  
import yaml  
from playwright.sync\_api import Page  
from tests.pages.login\_page import LoginPage  
from tests.pages.inventory\_page import InventoryPage  
from tests.pages.cart\_page import CartPage  
from tests.pages.checkout\_page import CheckoutPage  
  
@pytest.fixture(scope="session")  
def users():  
 with open('user.yml', 'r') as f:  
 data = yaml.safe\_load(f)  
 return data['users']  
  
@pytest.fixture  
def login\_page(page: Page):  
 login\_page = LoginPage(page)  
 login\_page.navigate()  
 return login\_page  
  
@pytest.fixture  
def inventory\_page(login\_page: LoginPage, users):  
 login\_page.login(users['valid\_user']['username'], users['valid\_user']['password'])  
 return InventoryPage(login\_page.page)  
  
@pytest.fixture  
def cart\_page(inventory\_page: InventoryPage):  
 return inventory\_page.go\_to\_cart()  
  
@pytest.fixture  
def checkout\_page(cart\_page: CartPage):  
 return cart\_page.checkout()

#### test\_login.py

import pytest  
  
@pytest.mark.parametrize("user\_key, expected\_success, expected\_error", [  
 ("valid\_user", True, None),  
 ("invalid\_password\_user", False, "Epic sadface: Username and password do not match"),  
 ("locked\_user", False, "Epic sadface: Sorry, this user has been locked out."),  
 ("missing\_username", False, "Epic sadface: Username is required"),  
 ("missing\_password", False, "Epic sadface: Password is required"),  
])  
def test\_login(login\_page, users, user\_key, expected\_success, expected\_error):  
 user = users[user\_key]  
 login\_page.login(user['username'], user['password'])  
   
 if expected\_success:  
 assert login\_page.page.url == "https://www.saucedemo.com/inventory.html"  
 else:  
 assert login\_page.get\_error\_message() == expected\_error

#### test\_checkout.py

import pytest  
  
def test\_successful\_checkout(inventory\_page, checkout\_page):  
 inventory\_page.add\_product\_to\_cart("sauce-labs-backpack")  
 checkout\_page.fill\_shipping\_info("John", "Doe", "12345")  
 checkout\_page.continue\_to\_overview()  
   
 overview\_page = CheckoutOverviewPage(checkout\_page.page)  
 overview\_page.finish\_checkout()  
 assert overview\_page.get\_confirmation\_message() == "Thank you for your order!"  
  
@pytest.mark.parametrize("first\_name, last\_name, postal\_code, expected\_error", [  
 ("", "Doe", "12345", "Error: First Name is required"),  
 ("John", "", "12345", "Error: Last Name is required"),  
 ("John", "Doe", "", "Error: Postal Code is required"),  
])  
def test\_checkout\_errors(checkout\_page, first\_name, last\_name, postal\_code, expected\_error):  
 checkout\_page.fill\_shipping\_info(first\_name, last\_name, postal\_code)  
 checkout\_page.continue\_to\_overview()  
 assert checkout\_page.get\_error\_message() == expected\_error

### 知识点和解释

1. **POM设计模式**:
   * 每个页面（如登录页、商品页）被封装为一个类，页面元素和操作抽象为方法，从而提高代码复用性和可维护性。
2. **数据驱动测试**:
   * 用户数据与测试代码分离，存储在YAML文件中，提升了可维护性，方便添加新的测试场景。
3. **异常场景覆盖**:
   * 通过参数化测试验证不同错误输入（如锁定用户、错误密码、缺失必填字段）是否会触发正确的错误提示。
4. **Fixture管理**:
   * 利用pytest的fixture管理浏览器和页面对象，确保每个测试用例的独立性，减少重复代码。
5. **Playwright自动等待机制**:
   * Playwright内置的自动等待确保在操作元素前其已处于可交互状态，增强了测试的稳定性。

此方案覆盖了正常和异常的业务流程，确保核心功能点（登录、购物车、结算）的自动化验证，提高了软件质量。

### 结论

参照给出的结果，基本不用做什么微调就可以执行成果，确实极大提升了我们进行自动化实现的效率。最难能可贵的是，deepseek给出的思考过程，对于我们代码实现和用例设计的要点也有极大的参考价值！