**以OpenCart切入UI测试框架**

目录

[1 总览 1](#_Toc204523735)

[1.1 项目结构 1](#_Toc204523736)

[1.2 项目架构 2](#_Toc204523737)

[2 OpenCart配置 3](#_Toc204523738)

[2.1 OpenCart介绍 3](#_Toc204523739)

[2.2 OpenCart网站限制 3](#_Toc204523740)

[2.3 Docker安装配置OpenCart 4](#_Toc204523741)

[2.3.1 配置文件 4](#_Toc204523742)

[2.3.2 安装网页 6](#_Toc204523743)

[3 调度器 7](#_Toc204523744)

[4 Yaml数据设计 7](#_Toc204523745)

[5 Playwright初始化 9](#_Toc204523746)

[5.1 Init\_page页面初始化 9](#_Toc204523747)

[5.1.1 连接函数 9](#_Toc204523748)

[5.1.2 实现 9](#_Toc204523749)

[6 PO设计 10](#_Toc204523750)

[6.1 POM介绍 10](#_Toc204523751)

[6.2 行为级别 11](#_Toc204523752)

[6.3 页面继承 11](#_Toc204523753)

[6.3.1 例子 11](#_Toc204523754)

[6.4 PO行为管理 13](#_Toc204523755)

[6.4.1 执行函数 13](#_Toc204523756)

[6.4.2 断言函数 13](#_Toc204523757)

[6.4.3 工具函数 13](#_Toc204523758)

[7 总结 14](#_Toc204523759)

# 总览

本项目整体做法与先前接口测试（https://blog.csdn.net/qq\_74086587/article/details/149326339?spm=1001.2014.3001.5502）基本一致，主要是针对UI测试特化设计了Yaml数据以及PO页面的设计。与接口测试不同的是，UI测试的用例执行强依赖于PO的页面操作，同时断言也嵌套于PO中，也会少了很多工具类，主要都是依靠PO做整体的执行和测试操作。

## 项目结构

- base

  - dispatchbase.py  调度器基类,扶着整体逻辑的拼接和PO行为的调度

  - playwrightobj.py  playwright封装类，主要用于获取page

  - precon.yaml 前置条件接口数据，未实现

  - preconditions.py  前置条件执行类，用于分发处理前置条件，部分实现

  - teardown.yaml  后置条件接口数据，未实现

  - teardown.py 后置条件执行类，用于分发处理后置条件，部分实现

- conf

  - conf.yaml 配置文件，配置文件夹、运行参数等信息

  - configloader.py 配置加载类，用于加载配置文件数据

- data  测试数据存储文件夹

- logs  日志内容存储文件夹

- opencart\_docker  基于docker的opencart环境

- pages  页面对象文件夹，封装页面元素

  - base\_page.py  根基类，所有页面对象都继承此类，封装所有页面都有的action和一些功能

  - ac\_base\_page.py 账号基类

  - ac\_.... 账号相关页面，包括个人信息、注册、登录等

  - gds\_base\_page.py  商品页面基类

  - gds\_.... 商品相关页面，包括商品列表、详情、购物车等

- reports  运行报告存储文件夹

- screenshots  截图存储文件夹

- tests/test\_entry.py  入口测试文件，调用测试用例

- utils

  - caseloader.py 用例加载类，配合钩子从指定文件夹加载用例数据

  - recordlogs.py 日志记录类，用于记录日志

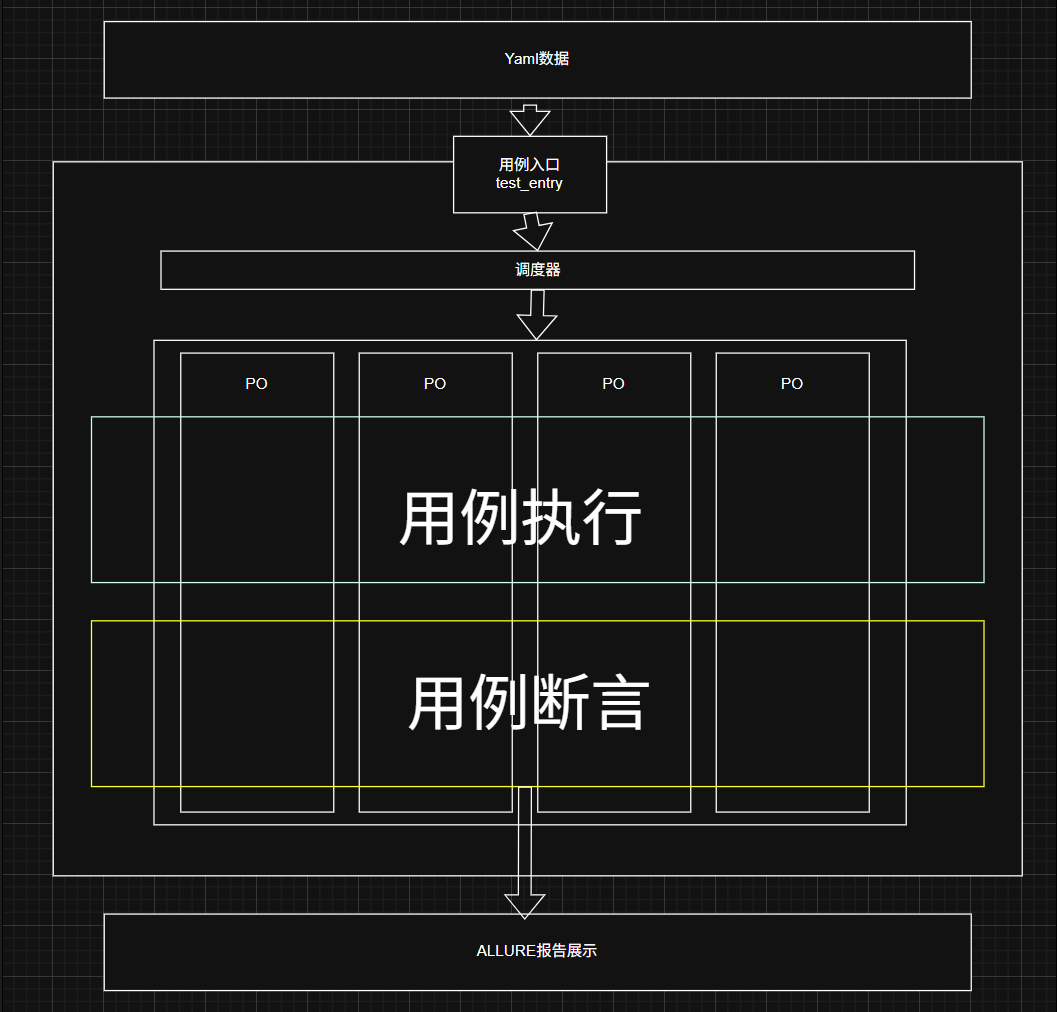
- conftest.py 全局测试配置，主要配置各种钩子的执行

- pytest.ini pytest配置文件

- run.py  运行入口文件，调用pytest

## 项目架构

架构基本如下，其内部用例执行是强依赖于PO的，和接口测试不同的是，是不能一套框架只改数据就能复用的，最重要的部分PO的设计是定制化，而且是很测试的内容强相关且占大部分内容的。

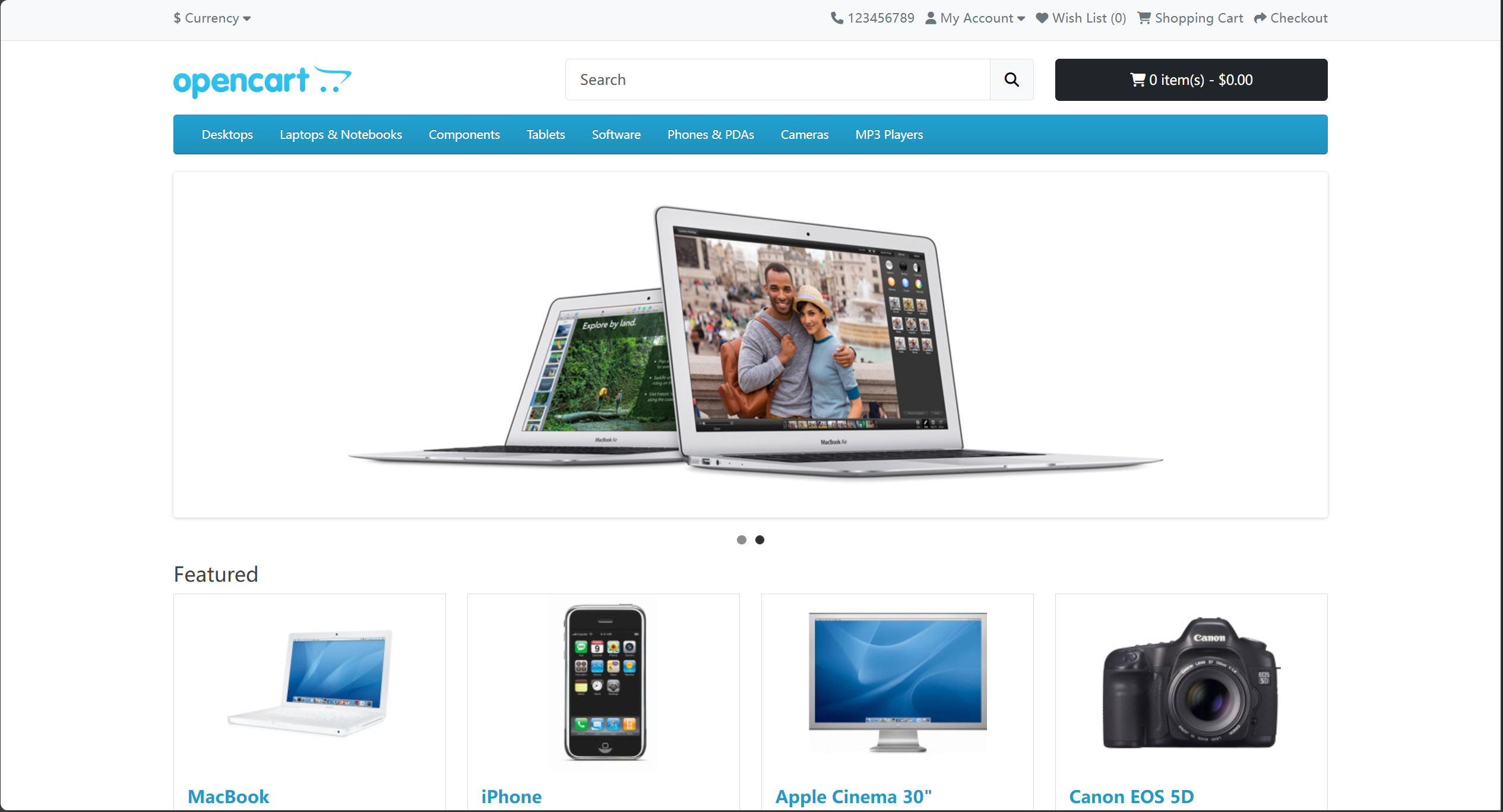


# OpenCart配置

## OpenCart介绍

OpenCart 是一款基于 PHP 的开源电商系统，采用 MVC 架构，支持多语言、多币种和多商店管理，内置商品、订单、客户、促销、支付和物流等完整模块，适合中小型电商平台快速搭建。其前端使用 Twig 模板和 Bootstrap，后端使用 MySQL 数据库，插件生态丰富，可通过 Marketplace 快速扩展功能，部署简单，便于二次开发，常用于学习、原型开发或轻量级商业站点。

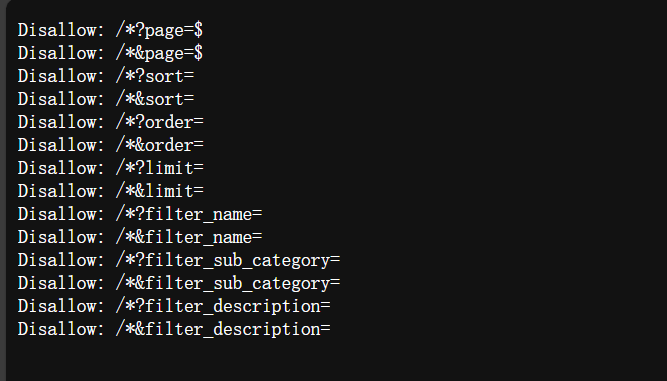
可见：<https://demo.opencart.com/>



## OpenCart网站限制

可以通过<https://demo.opencart.com/>查看演示网站，本来应该是可以直接通过在线网站来操作的，但是我尝试的过程中发现破不了反爬虫。通过网站robots.txt发现其似乎静止机器人爬虫操作：

原因应该主要是存在真实的用户注册、登录等存储，避免遭受外部攻击所以全部禁止了。



所以后面决定通过本地部署Opencart来进行UI访问以进行接口测试。

## Docker安装配置OpenCart

Opencart是开源，所以可以自行查询或者问生成式AI如何实现本地部署，我是使用VsCode在Docker进行本地化部署，基本步骤是：

1. 安装WSL
2. 安装Docker Desktop
3. VsCode安装Docker插件
4. 创建文件夹（例如opencart\_docker）绑定wsl
5. 拉取或下载发行版（含upload文件夹的）的github源码的upload文件夹到项目根目录
6. 配置Dockerfile、php.ini、docker-compose.yml后为文件夹等授权
7. 打开本地页面进行安装

注：上面内容只是大致流程，细节操作我忘记了，反正是弄了挺久才配置好，主要是拉取一些内容或者授权等比较花时间。

### 配置文件

#### Dockerfile

这里是定义一些内容安装和命令操作

FROM php:8.1-apache

# 安装依赖

RUN apt-get update && apt-get install -y \

    libpng-dev libjpeg-dev libfreetype6-dev libzip-dev zip unzip \

    libxml2-dev libonig-dev libcurl4-openssl-dev zlib1g-dev \

    && docker-php-ext-configure gd --with-freetype --with-jpeg \

    && docker-php-ext-install mysqli gd zip

# 拷贝 php.ini

COPY php.ini /usr/local/etc/php/conf.d/custom.ini

# 拷贝 OpenCart 上传目录到 Apache 默认网站目录

COPY upload/ /var/www/html/

# 给 Apache 用户权限

RUN chown -R www-data:www-data /var/www/html/

# 启用 Apache rewrite 模块

RUN a2enmod rewrite

# 启动 Apache 服务

CMD ["apache2-foreground"]

#### php.ini

这里最好就是直接设置这个base\_dir为空，这样肯定可以，如果指定反而会出错还是什么样，可以自己先试，不行再改成这样。

open\_basedir=

display\_errors = On

error\_reporting = E\_ALL & ~E\_WARNING

#### docker-compose.yml

需要注意的是安装时要求设置的数据库这里设置的数据库

services:

  db:

    image: mysql:5.7

    container\_name: opencart-db

    restart: unless-stopped

    environment:

      MYSQL\_ROOT\_PASSWORD: rootpass123

      MYSQL\_DATABASE: opencart

      MYSQL\_USER: opencartuser

      MYSQL\_PASSWORD: opencartpass

    volumes:

      - db\_data:/var/lib/mysql

  web:

    build: .

    container\_name: opencart-web

    ports:

      - "8080:80"

    volumes:

      - ./upload:/var/www/html

      - ./php.ini:/usr/local/etc/php/conf.d/custom.ini

    depends\_on:

      - db

    restart: unless-stopped

volumes:

  db\_data:

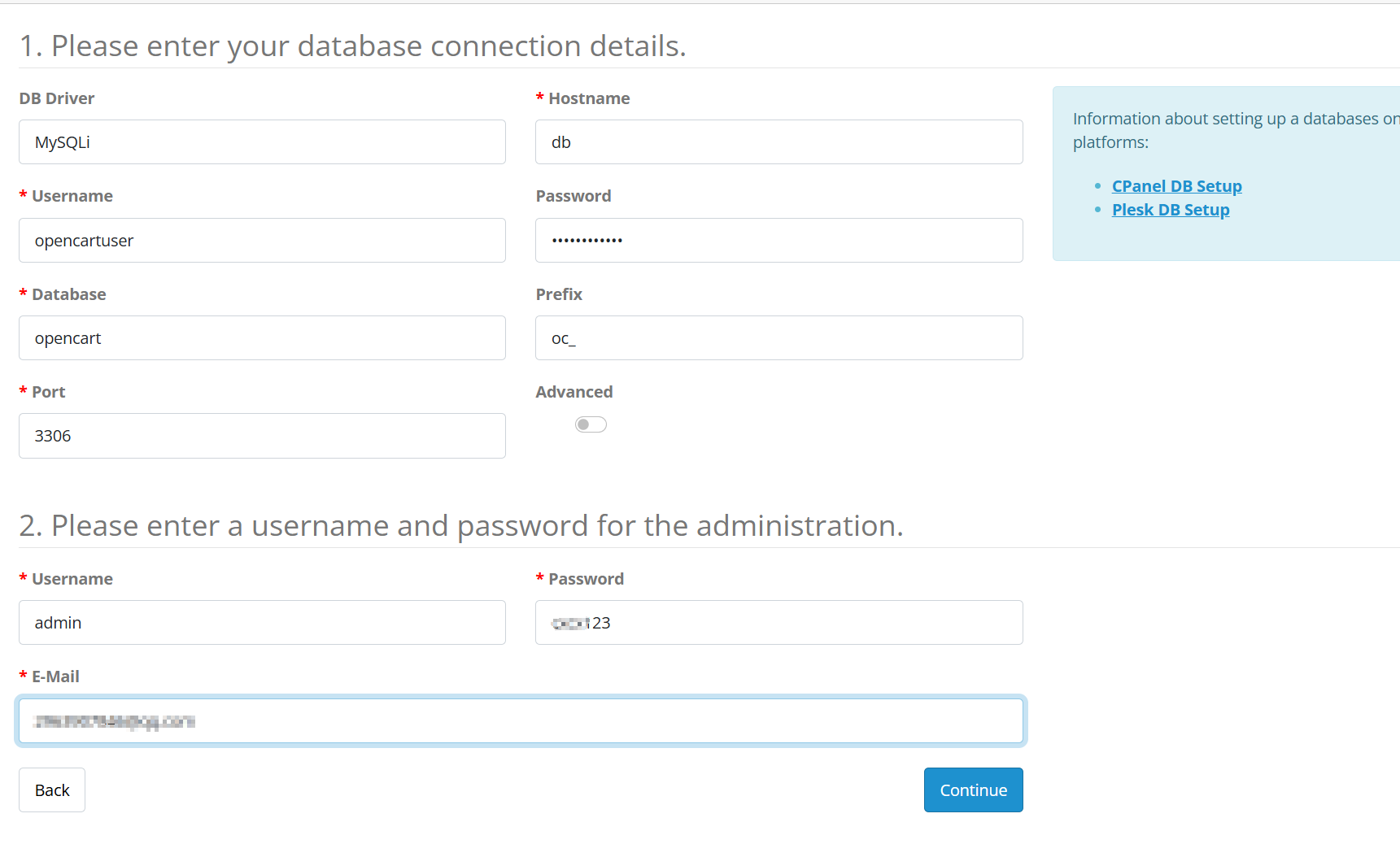
### 安装网页

上面源码和配置文件都弄好后通过下面指令启动服务

# docker-compose down

docker-compose up -d –build

随后打开网站<http://localhost:8080/>即可看到安装页面，随后是比对一些配置，这里我忘记具体的指令了，大概是授权和目录什么的，可以自行搜索询问一下，然后就是进入数据库什么的一些配置，这里数据库配置需要与前面一致，例如：



一步步安装完成后可以先打开8080/admin管理员后台进行一些安装临时文件夹的伤处和数据文件夹的移动，同时用户管理、订单管理页面也就是这个了，然后就可以像demo.opencart那样使用了。

# 调度器

这个UI测试框架使用的是高抽象+统一用例入口，我们通过测试入口函数记录allure消息后就通过调度器进行真正的用例执行，调度器就是真正的用例执行与验证模块，主要操作就是读取数据，取出一步步的PO和action执行，整体执行完成后再一个断言，整体结构如下

class DispatchBase:

def \_\_init\_\_(self):

    def get\_select(self, page\_key):

    def excute\_test(self,steps,vlds):

    def step\_excute(self,step,valid = False):

    def dispatch\_assert(self,vlds):

其中：

1. get\_select 主要用于模块、PO类的转化
2. def excute\_test 负责整合串联流程
3. def step\_excute 负责单步骤执行，包含调用转化出PO和函数执行
4. def dispatch\_assert 负责业务后独立断言的执行，内部还是依靠step\_excute

注意：

1. 调度器主要处理Playwright相关异常和一些没考虑到的，其它业务逻辑设计上的包括一些数据加载、跳转等连同断言一起记录并抛出断言异常，不需要调度器处理。
2. step\_excute是实际联系PO类和函数执行的方法，用例执行和断言执行都依赖于它，通过vliad字段表明是否属于断言

# Yaml数据设计

与接口测试类似，yaml设计需要指明用例的执行操作、验证断言，但是不同的是，UI测试不需要什么太多的上下文管理，而且大部分都为流程话测试，不会像接口测试一样有很多的单个接口的独立测试。

我们需要考虑的是如何方便获取用例执行的操作和前后置操作的管理，下面是我的设计：

- meta:

    feature: 用户模块                      #用于allure的feature标签

    story: 用户注册                        #用于allure的story标签

    case\_name: 用户首页正常注册后登录       #用例名称

    preconditions:                        #前置条件

      - url\_home

    teardown:                             #后置条件

      - logout

  #测试步骤

  steps:

    - step\_name: 点击账号-注册

      page\_name: home                     #PO页面表示，用于在调度器转化为page对象，需要注意命名转化逻辑

      action: top\_navigate                #PO行为，也就是PO对象的某个函数方法

      data:                               #函数可能需要的数据

        nv\_name: Register

    - step\_name: 注册账号

      page\_name: ac\_register

      action: register

      data:

        first\_name: ccc

        last\_name: xxx

        email: 12345@gg.com

        password: 123456789

  #验证步骤，本质上与测试步骤一致，主要是隔离开的独立验证，整体格式与steps基本一致，只是vld\_name

  validation:

    - vld\_name: 验证登录成功

      page\_name: now

      action: val\_url

      data:

        expected: customer\_token

- meta：

  ......

其中：

1. meta字段为元消息，主要用于allure级别划分和前后置操作的处理，这里的前后置条件大部分都是通过接口实现，这样更稳定更快速，所以目前是空着的。
2. steps和validation字段格式基本一致，对应调度器的执行分析都是使用同一个step\_excute，但是对应的函数可以单个函数对应执行原本steps中多个复合步骤以实现连贯的一次性断言
3. steps中，单个step的page\_name需要遵守统一规范，action则是直接对应函数名即可，data表示函数参数，可选。

# Playwright初始化

Playwright的初始化主要是进行浏览器的连接和页面的获取，此外，这里我们整体选择初始化一个全局的playwrightobject对象用作全局变量，表示共享同一个页面进行操作，如需继续多线程执行，可以通过每个线程独立的playwright对象实例化来实现。

整体设计结构如下：

class PlaywrightObject:

def \_\_init\_\_(self):

def init\_page(self):

    def get\_page(self):

def reset\_page(self):

    def close(self):

其中init\_page是核心，主要涉及浏览器连接，下面详细说明

## Init\_page页面初始化

### 连接函数

连接函数使用**launch\_persistent\_context**而不是简单的一层层获取，其使用真实浏览器绑定用户目录，整体使用是和用户真实使用基本一致的，同时也能通过真实目录利用浏览器本身自动进行cookie的管理，此外通过用户目录的指定还能实现隔离管理。

### 实现

需要注意的是，这里的browserinfo就是包含浏览器可执行文件（例如C:/Program Files (x86)/Microsoft/Edge/Application/msedge.exe）、用户目录的一个字典，这里用户目录是可以自动创建，所以放在一个好管理的地方也是ok的，此外此处的self.playwright是再类初始化中通过sync\_playwright().start()获取的，所以后面会需要通过close()主动关闭

xct\_path = self.browerinfo.get("xct\_path")

user\_data\_dir = self.browerinfo.get("user\_data\_dir")

# 持久化连接，直接得到context

context = self.playwright.chromium.launch\_persistent\_context(

    headless = self.headless,

    executable\_path = xct\_path,

    user\_data\_dir = user\_data\_dir,

    ignore\_default\_args=["--enable-automation"],

    args=[

        "--disable-password-manager-reauthentication",

        "--disable-save-password-bubble",

        "--disable-features=PasswordManager",

        "--disable-password-generation",

        "--disable-autofill-keyboard-accessory-view",

        "--disable-popup-blocking",

        "--disable-notifications",

        "--disable-translate",

        "--disable-infobars",

        "--no-default-browser-check",

        "--disable-autofill-keyboard-accessory-view",

        "--disable-save-password-bubble",

        "--disable-features=PasswordManager",  # 彻底关闭密码管理器

        "--start-maximized"

    ],

    locale="en-US",

    extra\_http\_headers={"Accept-Language": "en-US"}

)

self.page = context.new\_page()

# PO设计

PO设计就是UI测试的核心了，用例的执行、断言都是依靠PO类的设计。

需要着重注意的是，PO并不是页面跳转，而是页面操作，实际的页面跳转需要主动跳转，这里的PO的作用是提供操作，例如PO就是衣柜，分男女，page就是人，PO只负责提供衣服（功能），如果page不对就会出错。

也就是说，PO只是封装针对具体页面的操作，我们需要自行确认当前的页面就是与PO对应的页面和连贯上下文。

## POM介绍

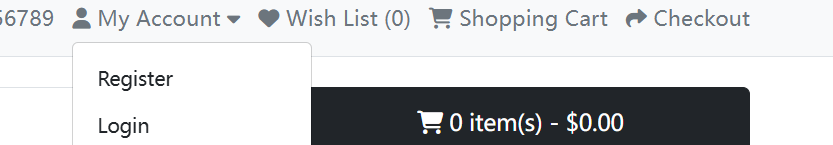
**POM（Page Object Model）** 是一种设计模式，常用于 **UI自动化测试** 中，用于提升测试代码的可维护性、复用性与可读性。**核心思想**是将页面上的每个“页面”或“模块”**封装成一个类**（Page Object），其中封装该页面的元素定位（locators）与操作行为（methods）。

简单来说就是**封装**，在使用是不直接使用原生接口，从而避免直接暴露locator、click（）等元操作。同时一个PO对应一个页面，对于针对性的测试也相对更好维护。

## 行为级别

行为级别指的就是页面的操作，低级行为指定就是靠近原生的点击、输入这些操作，由于多元素都可能用到这些操作，所及使用时的功能是不明确的，而这要求外部数据的使用也更加臃肿，可能会要求把元素的定位传入。

而所谓高级行为级别就是指把具体的行为进行封装，例如一个组策操作可能包括输入姓、输入名、输入邮箱、输入密码、点击勾选用户协议、点击注册这样一套流程，而我们高级别的行为封装就会把整个行为叫“注册”然后封装成一个函数。但是高级别有一个问题，有一些操作本质上其实是类似，例如导航到登录、导航到注册、导航到商品收藏，此时我们不会太细到每一种导航都写一个而是把相似业务统一起来成一个“顶部导航”函数，这种我称为中级封装，也是我实际的选择。



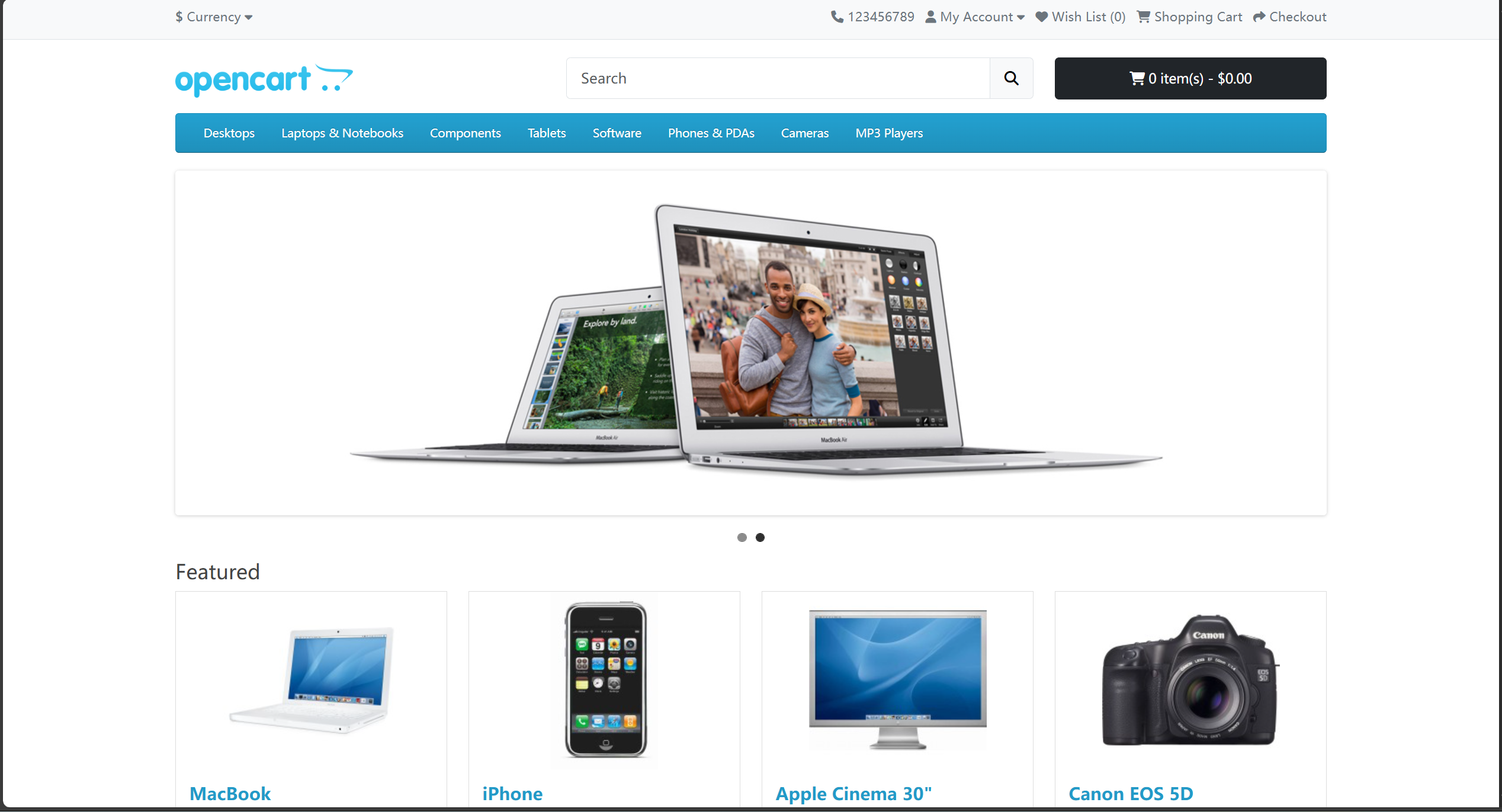
## 页面继承

页面继承是一个简化代码的常用操作，意思就是不同页面间有相同功能我们可以抽离出来用一个父类整体表示，通过一级级的继承抽离来简化代码，优化功能。一个简单的例子，我们有base\_page为所有页面的基类，有ac\_base\_page为继承base\_page的个人账户相关内容的基类，再下面才是登陆注册等具体PO。

下面是具体的例子说明

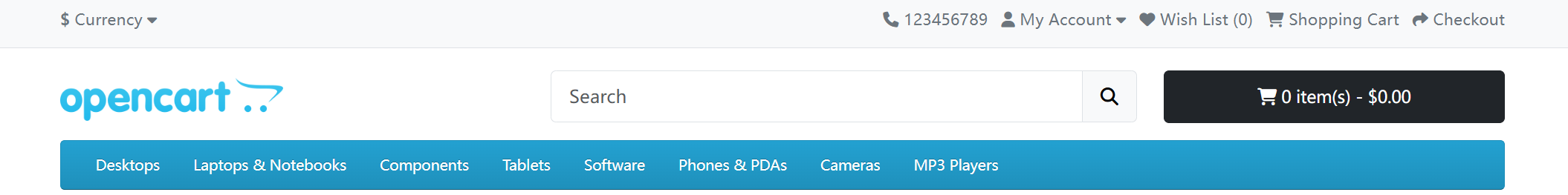
### 例子

下面的opencart网站我们观察



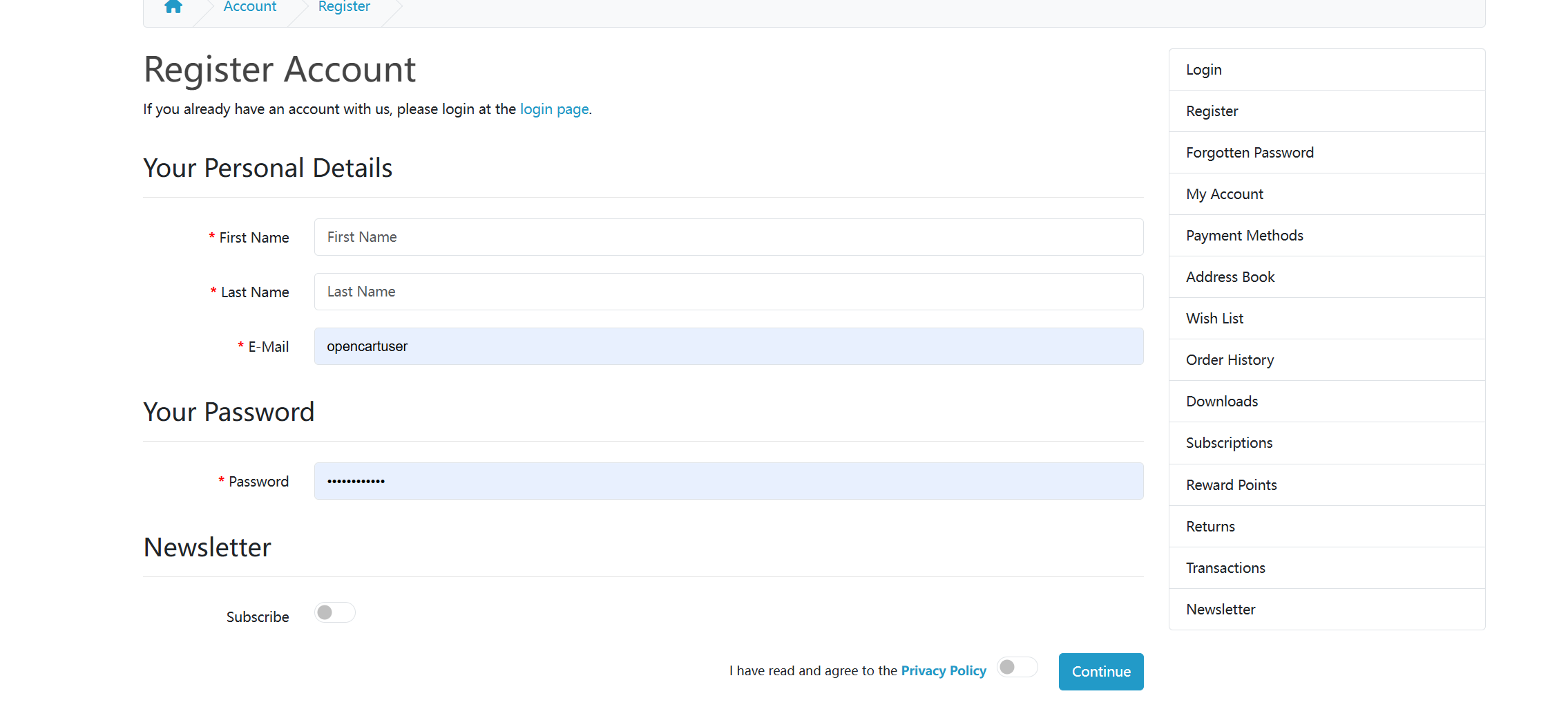
发现顶部的顶部内容、搜索、左上角货币切换、黑色的缩略购物车是所有页面都有的，意味着他们的功能是通用的，所以我们定义一个所有PO的基类base\_page，里面包含：

1. def top\_navigate(self, data):顶部登陆注册、购物车等导航
2. def goods\_navigate(self, data):商品导航，也就是蓝色这一行
3. def search(self, data):商品搜索
4. def view\_cart(self,data = None):黑色缩略购物车的操作
5. def currency\_change(self, data):货币切换

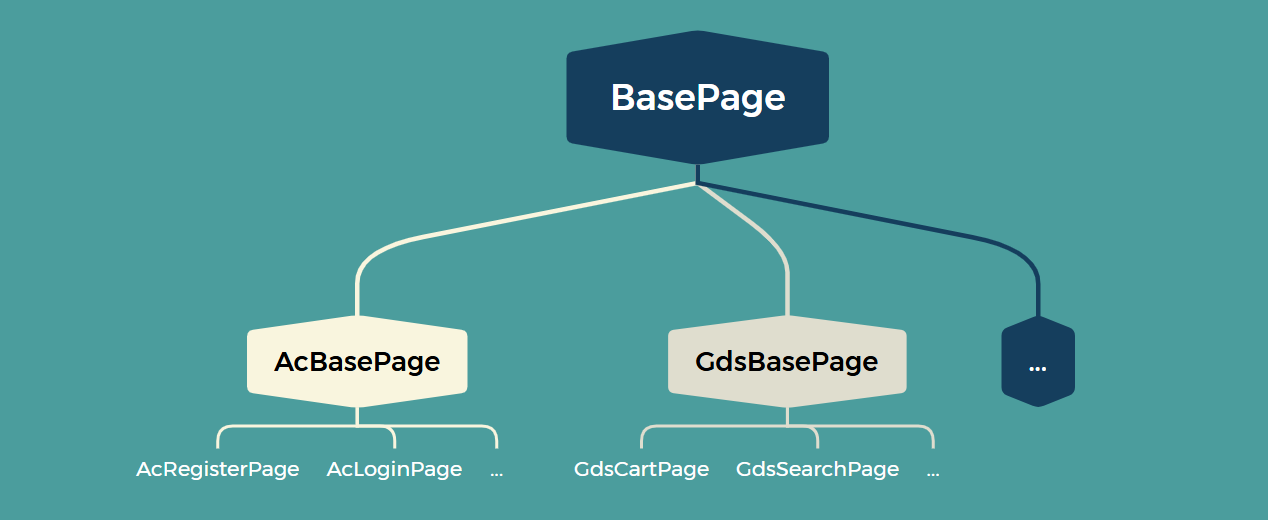


此外，在后续的使用中，我们还会把一些页面的辅助功能，例如页面截图、导航时间处理、url断言等通用内容加入，总之只要是PO相关的、通用的操作都可以往base\_page加。

然后我们有发现注册、登录、个人信息等页面都有这个右边这一条导航，所以我们又定义一个account相关内容的父类，这样之后例如注册页面就继承这个AcBasePage来实现



基本结构就是这样，此外也会出现一些多继承的PO例如收藏页面



## PO行为管理

PO的函数简单分为下面几种

1. 用例执行函数
2. 用例断言函数
3. 工具函数

### 执行函数

执行函数就是用例执行的核心，最主要的执行，这里的业务执行一方面要具体、一方面不能过于细，此外要考虑好该业务操作是否会影响其它操作的执行。

而执行函数将是UI测试最耗时间的地方，又枯燥又耗时，这是因为需要查找好定位表达、一方面要顾虑精简、一方面要顾虑稳定性、一方面又要顾虑速度；此外由于页面操作功能繁多，所以要写的执行函数也会很多。

### 断言函数

此处的断言函数是更广泛的断言，并不全是validation字段的断言，而是会包含一些用例执行时的单步骤断言。

**什么意思呢？**

一般来说，vld字段的断言是业务流程执行完成后的检查，例如把某个商品加入购物车，此时业务断言可能需要额外到达购物车检查是否有这个商品、包括商品配置、数量等是否正确，这是因为我们执行“加入购物车”的过程中是不确定是否加入成功、我们是不会把打开购物车检查作为执行流程的，但是——有一些操作在业务执行时是会相互影响的，例如导航的跳转，因为只有跳转成功了才能进行下一步操作，其中这会涉及PO的切换，所以我们必须在中间进行断言。

而中间断言和业务独立断言可能是会有重复的，而且独立断言是可以涉及更加复杂的多步骤的，甚至可以使用执行函数，例如检查购物车，我们可能只需要定义一个叫做“val\_cart”的函数，他的操作可能是：切换到GdsCartPage

1. 从任意页面点击顶部导航跳转购物车（基类功能，所以不要求目前PO）
2. 检查商品存在

这里的顶部导航其实就是一个独立的step函数，但是在独立断言中我们可以尽可能精简的写所以可以调用到这个功能。对应过于复杂的验证，我们可以直接就像用例执行那样一步步写，所以是非常灵活的。

### 工具函数

工具函数就是一些page级别或者函数需要的辅助功能，常见的例如截图操作和导航等待函数，这个导航函数其实既是断言也是工具，主要是因为playwright等待导航跳转需要注册事件，同时导航也会影响到后续操作的进行，实现：

    def navigation\_check(self, lambda\_func, name="nav\_action"):

        """

        导航跳转封装

        """

        try:

            # 表示预期出现页面导航跳转

            with self.page.expect\_navigation():

                lambda\_func()

        except TimeoutError:

            logs.error(f"❌ {name} 执行页面跳转\*\*超时\*\*")

            allure.attach(f"{name} 执行页面跳转\*\*超时\*\*", name="❌ 执行页面跳转\*\*超时\*\*", attachment\_type=allure.attachment\_type.TEXT)

            self.get\_screenshot(name, "navigation\_timeout", "❌ 执行页面跳转\*\*超时\*\*【截图】")

            raise AssertionError(f"{name} 执行页面跳转失败")

        logs.error(f"✅ {name} 执行页面跳转\*\*成功\*\*")

        allure.attach(f"{name} 执行页面跳转\*\*成功\*\*", name="✅ 执行页面跳转\*\*成功\*\*", attachment\_type=allure.attachment\_type.TEXT)

由于是UI测试，所以一般断言失败都会附带截图到allure报告中

# 总结

这个项目首先是建立在之前接口测试的知识上的，所以配置加载、单用例入口、日志记录这些其实都是基本一致的，所以没有做过多的说明。最主要的还是调度器、yaml和PO的配合。