Table of Contents

Web自动化测试课程

PO模式

目标

1. 深入理解方法封装的思想
2. 能够使用方法封装的思想对代码进行优化
3. 深入理解PO模式的思想
4. 熟练掌握PO模式的分层思想

PO模式学习思路

采用版本迭代的方式来学习，便于对不同版本的优缺点进行对比和理解。

- V1：不使用任何设计模式和单元测试框架
- V2：使用UnitTest管理用例
- V3：使用方法封装的思想，对代码进行优化
- V4：采用PO模式的分层思想对代码进行拆分
- V5：对PO分层之后的代码继续优化
- V6：PO模式深入封装，把共同操作提取封装到父类中，子类直接调用父类的方法

无模式

目标

1. 熟悉web自动化测试代码编写的基本流程
2. 掌握如何使用UnitTest管理测试脚本

---

1. 案例说明

对TPshop项目的登录模块进行自动化测试。

> 提示：登录模块包含了很多测试用例，比如：账号不存在、密码错误、验证码错误、登录成功等等。
> 为了节省时间我们只选取几个有代表性的用例来演示。

2. V1版本

不使用任何设计模式和单元测试框架。
每个文件里编写一个用例，完全的面向过程的编程方式。

登录功能-账号不存在

```
from selenium import webdriver

# 创建浏览器驱动对象，并完成初始化操作
driver = webdriver.Firefox()
driver.maximize_window()
driver.implicitly_wait(10)
driver.get("http://localhost")

"""
登录功能-账号不存在
"""

# 点击首页的‘登录’链接，进入登录页面
driver.find_element_by_link_text("登录").click()

# 输入用户名
driver.find_element_by_id("username").send_keys("13099999999")

# 输入密码
```

```
driver.find_element_by_id("password").send_keys("123456")

# 输入验证码
driver.find_element_by_id("verify_code").send_keys("8888")

# 点击‘登录’按钮
driver.find_element_by_name("sbtbutton").click()

# 获取提示信息
msg = driver.find_element_by_class_name("layui-layer-content").text
print("msg=", msg)

# 关闭驱动对象
driver.quit()
```

登录功能-密码错误

```
from selenium import webdriver

# 创建浏览器驱动对象，并完成初始化操作
driver = webdriver.Firefox()
driver.maximize_window()
driver.implicitly_wait(10)
driver.get("http://localhost")

"""
登录功能-密码错误
"""

# 点击首页的‘登录’链接，进入登录页面
driver.find_element_by_link_text("登录").click()

# 输入用户名
driver.find_element_by_id("username").send_keys("13012345678")

# 输入密码
driver.find_element_by_id("password").send_keys("error")

# 输入验证码
driver.find_element_by_id("verify_code").send_keys("8888")

# 点击‘登录’按钮
driver.find_element_by_name("sbtbutton").click()

# 获取提示信息
msg = driver.find_element_by_class_name("layui-layer-content").text
print("msg=", msg)
```

```
# 关闭驱动对象
driver.quit()
```

3. V2版本

使用UnitTest管理用例，并断言用例的执行结果

```
import unittest
from selenium import webdriver

class TestLogin(unittest.TestCase):
    """
    对登录模块的功能进行测试
    """

    @classmethod
    def setUpClass(cls):
        cls.driver = webdriver.Firefox()
        cls.driver.maximize_window()
        cls.driver.implicitly_wait(10)
        cls.driver.get("http://localhost")

    @classmethod
    def tearDownClass(cls):
        cls.driver.quit()

    def setUp(self):
        # 打开首页
        self.driver.get("http://localhost")

        # 点击首页的‘登录’链接，进入登录页面
        self.driver.find_element_by_link_text("登录").click()

    # 账号不存在
    def test_login_username_is_error(self):
        # 输入用户名
        self.driver.find_element_by_id("username").send_keys("13099999999")

        # 输入密码
        self.driver.find_element_by_id("password").send_keys("123456")

        # 输入验证码
        self.driver.find_element_by_id("verify_code").send_keys("8888")

        # 点击‘登录’
        self.driver.find_element_by_name("sbtbutton").click()
```

```
        # 断言提示信息
        msg = self.driver.find_element_by_class_name("layui-layer-content").text
        print("msg=", msg)
        self.assertIn("账号不存在", msg)

    # 密码错误
    def test_login_password_is_error(self):
        # 输入用户名
        self.driver.find_element_by_id("username").send_keys("13012345678")

        # 输入密码
        self.driver.find_element_by_id("password").send_keys("error")

        # 输入验证码
        self.driver.find_element_by_id("verify_code").send_keys("8888")

        # 点击‘登录’
        self.driver.find_element_by_name("sbtbutton").click()

        # 断言提示信息
        msg = self.driver.find_element_by_class_name("layui-layer-content").text
        print("msg=", msg)
        self.assertIn("密码错误", msg)
```

方法封装

目标

1. 深入理解方法封装的思想
2. 能够使用方法封装的思想对代码进行优化

---

1. 方法封装

方法封装是将一些有共性的或多次被使用的代码提取到一个方法中，这样可以避免代码冗余，且容易维护， 只需要改一个方法所有需要调用此方法的方法结果都会被修改。

目的：用最少的代码实现最多的功能

---

2. V3版本

使用方法封装的思想，对代码进行优化。

2.1 定义获取驱动对象的工具类

对登录流程的代码进行优化，定义获取驱动对象的工具类

```
# utils.py

class DriverUtil:
    """
    浏览器驱动工具类
    """

    _driver = None

    @classmethod
    def get_driver(cls):
        """
        获取浏览器驱动对象，并完成初始化设置
        :return: 浏览器驱动对象
        """
        if cls._driver is None:
            cls._driver = webdriver.Firefox()
            cls._driver.maximize_window()
```

```
            cls._driver.implicitly_wait(10)
            cls._driver.get("http://localhost")
        return cls._driver

    @classmethod
    def quit_driver(cls):
        """
        关闭浏览器驱动
        """
        if cls._driver:
            cls._driver.quit()
            cls._driver = None
```

2.2 封装“获取弹出框的提示消息”

对登录流程的代码进行优化，封装'获取弹出框的提示消息'的方法

```
# utils.py

def get_tips_msg():
    """
    获取弹出框的提示消息
    :return: 消息文本内容
    """
    msg = DriverUtil.get_driver().find_element_by_class_name("layui-layer-content").tex
t
    return msg
```

PO模式介绍

目标

1. 深入理解PO模式的思想
2. 熟练掌握PO模式的分层思想

---

1. 存在的问题

在做UI自动化时定位元素特别依赖页面，一旦页面发生变更就不得不跟着去修改定位元素的代码。

举例：假设要对一个元素进行点击操作，而且会经常对该元素进行操作，那么你就可能会编写多处如下代码

```
driver.find_element_by_id("login-btn").click()
```

存在的问题：

- 如果开发人员修改了这个元素的id，这时候你就不得不修改所有对应的代码
- 存在大量冗余代码

> 思考：如何来解决这个问题呢？

2. PO模式

PO是**Page Object**的缩写，PO模式是自动化测试项目开发实践的最佳设计模式之一。
核心思想是通过对界面元素的封装减少冗余代码，同时在后期维护中，若元素定位发生变化， 只需要调整页面元素封装的代码，提高测试用例的可维护性、可读性。

PO模式可以把一个页面分为三层，对象库层、操作层、业务层。

- 对象库层：封装定位元素的方法。
- 操作层：封装对元素的操作。
- 业务层：将一个或多个操作组合起来完成一个业务功能。比如登录：需要输入帐号、密码、点击登录三个操作。

2.1 引入PO模式的好处

引入PO模式前

- 存在大量冗余代码

- 业务流程不清晰
- 后期维护成本大

引入PO模式后

- 减少冗余代码
- 业务代码和测试代码被分开，降低耦合性
- 维护成本低

PO模式实践

目标

1. 能够采用PO模式的分层思想对页面进行封装

---

1. V4版本

采用PO模式的分层思想对代码进行拆分

```
from po.utils import DriverUtil


class LoginPage:
    """
    对象库层
    """

    def __init__(self):
        self.driver = DriverUtil.get_driver()

        # 用户名输入框
        self.username = None
        # 密码
        self.password = None
        # 验证码输入框
        self.verify_code = None
        # 登录按钮
        self.login_btn = None
        # 忘记密码
        self.forget_pwd = None

    def find_username(self):
        return self.driver.find_element_by_id("username")

    def find_password(self):
        return self.driver.find_element_by_id("password")

    def find_verify_code(self):
        return self.driver.find_element_by_id("verify_code")

    def find_login_btn(self):
```

```
        return self.driver.find_element_by_name("sbtbutton")

    def find_forget_pwd(self):
        return self.driver.find_element_by_partial_link_text("忘记密码")


class LoginHandle:
    """
    操作层
    """

    def __init__(self):
        self.login_page = LoginPage()

    def input_username(self, username):
        self.login_page.find_username().send_keys(username)

    def input_password(self, pwd):
        self.login_page.find_password().send_keys(pwd)

    def input_verify_code(self, code):
        self.login_page.find_verify_code().send_keys(code)

    def click_login_btn(self):
        self.login_page.find_login_btn().click()

    def click_forget_pwd(self):
        self.login_page.find_forget_pwd().click()


class LoginProxy:
    """
    业务层
    """

    def __init__(self):
        self.login_handle = LoginHandle()

    # 登录
    def login(self, username, password, verify_code):
        # 输入用户名
        self.login_handle.input_username(username)
        # 输入密码
        self.login_handle.input_password(password)
        # 输入验证码
        self.login_handle.input_verify_code(verify_code)
        # 点击登录按钮
        self.login_handle.click_login_btn()
```

```
    # 跳转到忘记密码页面
    def to_forget_pwd_page(self):
        # 点击忘记密码
        self.login_handle.click_forget_pwd()
```

```
import unittest

from po import utils
from po.utils import DriverUtil
from po.v4.page.login_page import LoginProxy


class TestLogin(unittest.TestCase):
    """
    对登录模块的功能进行测试
    """

    @classmethod
    def setUpClass(cls):
        cls.driver = DriverUtil.get_driver()
        cls.login_proxy = LoginProxy()

    @classmethod
    def tearDownClass(cls):
        DriverUtil.quit_driver()

    def setUp(self):
        # 打开首页
        self.driver.get("http://localhost")

        # 点击首页的‘登录’链接，进入登录页面
        self.driver.find_element_by_link_text("登录").click()

    # 账号不存在
    def test_login_username_is_error(self):
        self.login_proxy.login("13099999999", "123456", "8888")

        # 断言提示信息
        msg = utils.get_tips_msg()
        print("msg=", msg)
        self.assertIn("账号不存在", msg)

    # 密码错误
    def test_login_password_is_error(self):
        self.login_proxy.login("13012345678", "123456", "8888")

        # 断言提示信息
```

```
        msg = utils.get_tips_msg()
        print("msg=", msg)
        self.assertIn("密码错误", msg)
```

2. V5版本

对PO分层之后的代码继续优化

1. 优化对象库层的代码，抽取元素的定位方式，把定位信息定义在对象的属性中，便于集中管理
2. 优化操作层的代码，针对输入操作应该先清空输入框中的内容再输入新的内容

```
from selenium.webdriver.common.by import By

from po.utils import DriverUtil


class LoginPage:
    """
    对象库层
    """

    def __init__(self):
        self.driver = DriverUtil.get_driver()

        # 用户名
        self.username = (By.ID, "username")
        # 密码
        self.password = (By.ID, "password")
        # 验证码输入框
        self.verify_code = (By.ID, "verify_code")
        # 登录按钮
        self.login_btn = (By.NAME, "sbtbutton")
        # 忘记密码
        self.forget_pwd = (By.PARTIAL_LINK_TEXT, "忘记密码")

    def find_username(self):
        return self.driver.find_element(self.username[0], self.username[1])

    def find_password(self):
        return self.driver.find_element(self.password[0], self.password[1])

    def find_verify_code(self):
        return self.driver.find_element(self.verify_code[0], self.verify_code[1])

    def find_login_btn(self):
        return self.driver.find_element(self.login_btn[0], self.login_btn[1])
```

```
    def find_forget_pwd(self):
        return self.driver.find_element(self.forget_pwd[0], self.forget_pwd[1])


class LoginHandle:
    """
    操作层
    """

    def __init__(self):
        self.login_page = LoginPage()

    def input_username(self, username):
        self.login_page.find_username().clear()
        self.login_page.find_username().send_keys(username)

    def input_password(self, pwd):
        self.login_page.find_password().clear()
        self.login_page.find_password().send_keys(pwd)

    def input_verify_code(self, code):
        self.login_page.find_verify_code().clear()
        self.login_page.find_verify_code().send_keys(code)

    def click_login_btn(self):
        self.login_page.find_login_btn().click()

    def click_forget_pwd(self):
        self.login_page.find_forget_pwd().click()


class LoginProxy:
    """
    业务层
    """

    def __init__(self):
        self.login_handle = LoginHandle()

    # 登录
    def login(self, username, password, verify_code):
        # 输入用户名
        self.login_handle.input_username(username)
        # 输入密码
        self.login_handle.input_password(password)
        # 输入验证码
        self.login_handle.input_verify_code(verify_code)
        # 点击登录按钮
        self.login_handle.click_login_btn()
```

```
# 跳转到忘记密码页面
def to_forget_pwd_page(self):
    # 点击忘记密码
    self.login_handle.click_forget_pwd()
```

PO模式深入封装

目标

1. 能够采用继承的思想对PO模式进行深入的封装

---

1. V6版本

把共同操作提取封装到父类中，子类直接调用父类的方法，避免代码冗余

```
# base_page.py

from po.utils import DriverUtil

class BasePage:
    """
    基类-对象库层
    """

    def __init__(self):
        self.driver = DriverUtil.get_driver()

    def find_element(self, location):
        return self.driver.find_element(location[0], location[1])


class BaseHandle:
    """
    基类-操作层
    """

    def input_text(self, element, text):
        """
        在输入框里输入文本内容，先清空再输入
        :param element: 要操作的元素
        :param text: 要输入的文本内容
        """
        element.clear()
        element.send_keys(text)
```

```
from selenium.webdriver.common.by import By
```

```python
from po.v6.common.base_page import BasePage, BaseHandle


class LoginPage(BasePage):
    """
    对象库层
    """

    def __init__(self):
        super().__init__()

        # 用户名输入框
        self.username = (By.ID, "username")
        # 密码
        self.password = (By.ID, "password")
        # 验证码
        self.verify_code = (By.ID, "verify_code")
        # 登录按钮
        self.login_btn = (By.NAME, "sbtbutton")
        # 忘记密码
        self.forget_pwd = (By.PARTIAL_LINK_TEXT, "忘记密码")

    def find_username(self):
        return self.find_element(self.username)

    def find_password(self):
        return self.find_element(self.password)

    def find_verify_code(self):
        return self.find_element(self.verify_code)

    def find_login_btn(self):
        return self.find_element(self.login_btn)

    def find_forget_pwd(self):
        return self.find_element(self.forget_pwd)


class LoginHandle(BaseHandle):
    """
    操作层
    """

    def __init__(self):
        self.login_page = LoginPage()

    def input_username(self, username):
        self.input_text(self.login_page.find_username(), username)
```

```
    def input_password(self, pwd):
        self.input_text(self.login_page.find_password(), pwd)

    def input_verify_code(self, code):
        self.input_text(self.login_page.find_verify_code(), code)

    def click_login_btn(self):
        self.login_page.find_login_btn().click()

    def click_forget_pwd(self):
        self.login_page.find_forget_pwd().click()


class LoginProxy:
    """
    业务层
    """

    def __init__(self):
        self.login_handle = LoginHandle()

    # 登录
    def login(self, username, password, verify_code):
        # 输入用户名
        self.login_handle.input_username(username)
        # 输入密码
        self.login_handle.input_password(password)
        # 输入验证码
        self.login_handle.input_verify_code(verify_code)
        # 点击登录按钮
        self.login_handle.click_login_btn()

    # 跳转到忘记密码页面
    def to_forget_pwd_page(self):
        # 点击忘记密码
        self.login_handle.click_forget_pwd()
```

日志收集

目标

1. 理解日志的相关概念
2. 掌握日志的基本用法
3. 掌握日志的高级用法

日志相关概念

目标

1．了解日志的概念
2．理解日志的作用
3．掌握常见的日志级别

---

1. 日志

概念：日志就是用于记录系统运行时的信息，对一个事件的记录；也称为Log。

1.1 日志的作用

- 调试程序
- 了解系统程序运行的情况，是否正常
- 系统程序运行故障分析与问题定位
- 用来做用户行为分析和数据统计

1.2 日志级别

> 日志级别：是指日志信息的优先级、重要性或者严重程度

常见的日志级别

日志级别	描述
DEBUG	调试级别，打印非常详细的日志信息，通常用于对代码的调试
INFO	信息级别，打印一般的日志信息，突出强调程序的运行过程
WARNING	警告级别，打印警告日志信息，表明会出现潜在错误的情形，一般不影响软件的正常使用
ERROR	错误级别，打印错误异常信息，该级别的错误可能会导致系统的一些功能无法正常使用
CRITICAL	严重错误级别，一个严重的错误，这表明系统可能无法继续运行

说明

- 上面列表中的日志级别是从上到下依次升高的，即：DEBUG < INFO < WARNING < ERROR

< CRITICAL，而日志的信息量是依次减少的；

- 当为程序指定一个日志级别后，程序会记录所有日志级别大于或等于指定日志级别的日志信息，而不是仅仅记录指定级别的日志信息；
- 一般建议只使用DEBUG、INFO、WARNING、ERROR这四个级别

日志的基本用法

目标

1. 掌握如何设置日志级别
2. 掌握如何设置日志格式
3. 掌握如何将日志信息输出到文件中

---

1. logging模块

Python中有一个标准库模块logging可以直接记录日志

1.1 基本用法

```
import logging

logging.debug("这是一条调试信息")
logging.info("这是一条普通信息")
logging.warning("这是一条警告信息")
logging.error("这是一条错误信息")
logging.critical("这是一条严重错误信息")
```

1.1 设置日志级别

logging中默认的日志级别为WARNING，程序中大于等于该级别的日志才能输出，小于该级别的日志不会被打印出来。

设置日志级别

```
logging.basicConfig(level=logging.DEBUG)
```

如何选择日志级别

- 在开发环境和测试环境中，为了尽可能详细的查看程序的运行状态来保证上线后的稳定性，可以使用DEBUG或INFO级别的日志获取详细的日志信息，这是非常耗费机器性能的。
- 在生产环境中，通常只需要记录程序的异常信息、错误信息等，这样既可以减小服务器的I/O压力，也可以提高获取错误日志信息的效率和方便问题的排查。

1.2 设置日志格式

默认的日志的格式为：
日志级别:Logger名称:日志内容

自定义日志格式：

```
logging.basicConfig(format="%(levelname)s:%(name)s:%(message)s")
```

format参数中可能用到的格式化信息：

占位符	描述
%(name)s	Logger的名字
%(levelno)s	数字形式的日志级别
%(levelname)s	文本形式的日志级别
%(pathname)s	调用日志输出函数的模块的完整路径名，可能没有
%(filename)s	调用日志输出函数的模块的文件名
%(module)s	调用日志输出函数的模块名
%(funcName)s	调用日志输出函数的函数名
%(lineno)d	调用日志输出函数的语句所在的代码行
%(created)f	当前时间，用UNIX标准的表示时间的浮 点数表示
%(relativeCreated)d	输出日志信息时的，自Logger创建以来的毫秒数
%(asctime)s	字符串形式的当前时间。默认格式是 “2003-07-08 16:49:45,896”
%(thread)d	线程ID。可能没有
%(threadName)s	线程名。可能没有
%(process)d	进程ID。可能没有
%(message)s	用户输出的消息

示例代码：

```
import logging

fmt = '%(asctime)s %(levelname)s [%(name)s] [%(filename)s(%(funcName)s:%(lineno)d)] - %(message)s'
logging.basicConfig(level=logging.INFO, format=fmt)

logging.debug("调试")
logging.info("信息")
logging.warning("警告")
logging.error("错误")
```

1.3 将日志信息输出到文件中

默认情况下Python的logging模块将日志打印到了标准输出中（控制台）

将日志信息输出到文件中：

```
logging.basicConfig(filename="a.log")
```

示例代码：

```
import logging

fmt = '%(asctime)s %(levelname)s [%(name)s] [%(filename)s(%(funcName)s:%(lineno)d)] - %
(message)s'
logging.basicConfig(filename="a.log", level=logging.INFO, format=fmt)

logging.debug("调试")
logging.info("信息")
logging.warning("警告")
logging.error("错误")
```

日志的高级用法

目标

1. 了解logging日志模块四大组件
2. 掌握如何讲日志输出到多个Handler中

---

1. logging日志模块四大组件

组件名称	类名	功能描述
日志器	Logger	提供了程序使用日志的入口
处理器	Handler	将logger创建的日志记录发送到合适的目的输出
格式器	Formatter	决定日志记录的最终输出格式
过滤器	Filter	提供了更细粒度的控制工具来决定输出哪条日志记录，丢弃哪条日志记录

logging模块就是通过这些组件来完成日志处理的

1.1 组件之间的关系

- 日志器（logger）需要通过处理器（handler）将日志信息输出到目标位置，如：文件、sys.stdout、网络等；
- 不同的处理器（handler）可以将日志输出到不同的位置；
- 日志器（logger）可以设置多个处理器（handler）将同一条日志记录输出到不同的位置；
- 每个处理器（handler）都可以设置自己的格式器（formatter）实现同一条日志以不同的格式输出到不同的地方。
- 每个处理器（handler）都可以设置自己的过滤器（filter）实现日志过滤，从而只保留感兴趣的日志；

简单点说就是：日志器（logger）是入口，真正干活儿的是处理器（handler），处理器（handler）还可以通过过滤器（filter）和格式器（formatter）对要输出的日志内容做过滤和格式化等处理操作。

1.2 Logger类

Logger对象的任务：

- 向程序暴露记录日志的方法

- 基于日志级别或Filter对象来决定要对哪些日志进行后续处理
- 将日志消息传送给所有感兴趣的日志handlers

如何创建**Logger**对象

```
logger = logging.getLogger()
logger = logging.getLogger("myLogger")
```

logging.getLogger()方法有一个可选参数name，该参数表示将要返回的日志器的名称标识，如果不提供该参数，则返回root日志器对象。 若以相同的name参数值多次调用getLogger()方法，将会返回指向同一个logger对象的引用。

Logger常用的方法

方法	描述
logger.debug() logger.info() logger.warning() logger.error() logger.critical()	打印日志
logger.setLevel()	设置日志器将会处理的日志消息的最低严重级别
logger.addHandler()	为该logger对象添加一个handler对象
logger.addFilter()	为该logger对象添加一个filter对象

1.3 Handler类

Handler对象的作用是将消息分发到handler指定的位置，比如：控制台、文件、网络、邮件等。Logger对象可以通过addHandler()方法为自己添加多个handler对象。

如何创建**Handler**对象

在程序中不应该直接实例化和使用Handler实例，因为Handler是一个基类，它只定义了Handler应该有的接口。 应该使用Handler实现类来创建对象，logging中内置的常用的Handler包括：

Handler	描述
logging.StreamHandler	将日志消息发送到输出到Stream，如std.out, std.err或任何file-like对象。
logging.FileHandler	将日志消息发送到磁盘文件，默认情况下文件大小会无限增长
logging.handlers.RotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按大小切割
logging.hanlders.TimedRotatingFileHandler	将日志消息发送到磁盘文件，并支持日志文件按时间切割

logging.handlers.HTTPHandler	将日志消息以GET或POST的方式发送给一个HTTP服务器
logging.handlers.SMTPHandler	将日志消息发送给一个指定的email地址

Handler常用的方法

方法	描述
handler.setLevel()	设置handler将会处理的日志消息的最低严重级别
handler.setFormatter()	为handler设置一个格式器对象
handler.addFilter()	为handler添加一个过滤器对象

1.4 Formatter类

Formatter对象用于配置日志信息的格式。

如何创建Formatter对象

```
formatter = logging.Formatter(fmt=None, datefmt=None, style='%')
    fmt：指定消息格式化字符串，如果不指定该参数则默认使用message的原始值
    datefmt：指定日期格式字符串，如果不指定该参数则默认使用"%Y-%m-%d %H:%M:%S"
    style：Python 3.2新增的参数，可取值为 '%', '{'和 '$'，如果不指定该参数则默认使用'%'
```

---

2. 将日志信息同时输出到控制台和文件中

实现步骤分析

```
1.创建日志器对象
2.创建控制台处理器对象
3.创建文件处理器对象
4.创建格式化器对象
5.把格式化器添加到处理器中
6.把处理器添加到日志器中
```

定义日志格式

```
fmt = '%(asctime)s %(levelname)s [%(name)s] [%(filename)s(%(funcName)s:%(lineno)d)] - %
(message)s'
formatter = logging.Formatter(fmt)
```

把日志输出到控制台

```
logger = logging.getLogger()
sh = logging.StreamHandler()
sh.setFormatter(formatter)
logger.addHandler(sh)
```

把日志输出到文件中

```
fh = logging.FileHandler("./b.log")
fh.setFormatter(formatter)
logger.addHandler(fh)
```

---

3. 每日生成一个日志文件

定义Handler对象

```
fh = logging.handlers.TimedRotatingFileHandler(filename, when='h', interval=1, backupCo
unt=0)
    将日志信息记录到文件中，以特定的时间间隔切换日志文件。
    filename: 日志文件名
    when: 时间单位，可选参数
        S - Seconds
        M - Minutes
        H - Hours
        D - Days
        midnight - roll over at midnight
        W{0-6} - roll over on a certain day; 0 - Monday
    interval: 时间间隔
    backupCount: 日志文件备份数量。如果backupCount大于0，那么当生成新的日志文件时，
                 将只保留backupCount个文件，删除最老的文件。
```

示例代码：

```
import logging.handlers

logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

# 日志格式
fmt = "%(asctime)s %(levelname)s [%(filename)s(%(funcName)s:%(lineno)d)] - %(message)s"
formatter = logging.Formatter(fmt)

# 输出到文件，每日一个文件
fh = logging.handlers.TimedRotatingFileHandler("./a.log", when='MIDNIGHT', interval=1,
backupCount=3)
```

```
fh.setFormatter(formatter)
fh.setLevel(logging.INFO)
logger.addHandler(fh)
```

项目实战

目标

1. 熟悉自动化测试的流程
2. 能够对一个web项目实现自动化测试

自动化测试流程

目标

1. 熟悉自动化测试的流程

1. 自动化测试的流程

1. 需求分析
2. 挑选适合做自动化测试的功能
3. 设计测试用例
4. 搭建自动化测试环境 [可选]
5. 设计自动化测试项目的架构 [可选]
6. 编写代码
7. 执行测试用例
8. 生成测试报告并分析结果

项目介绍

1. 项目介绍

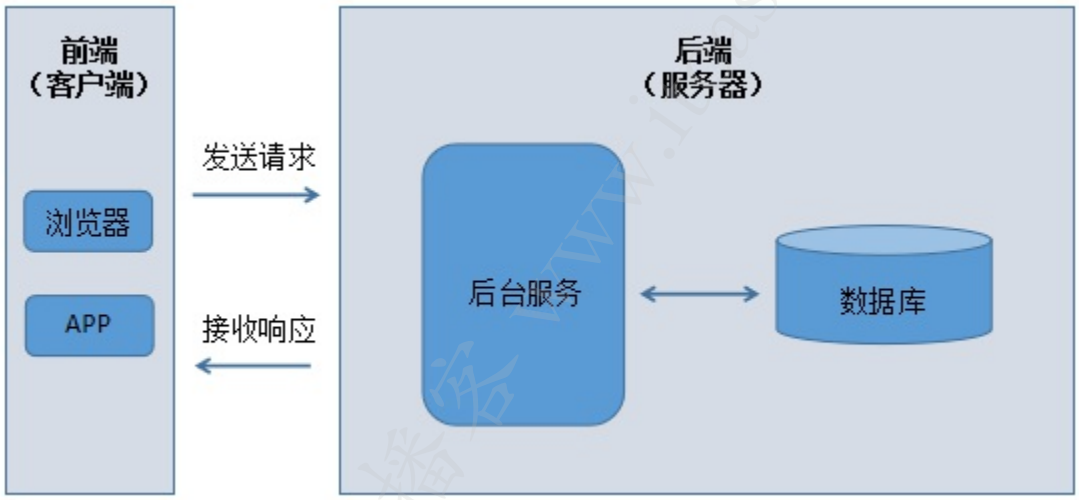
项目名称

TPshop开源商城系统

项目描述

TPshop是一个电子商务B2C电商平台系统，功能强大，安全便捷。适合企业及个人快速构建个性化网上商城。
包含PC+IOS客户端+Adroid客户端+微商城，系统PC+后台是基于ThinkPHP MVC构架开发的跨平台开源软件，设计得非常灵活，具有模块化架构体系和丰富的功能，易于与第三方应用系统无缝集成，在设计上，包含相当全面，以模块化架构体系，让应用组合变得相当灵活，功能也相当丰富。

项目架构



用例设计

目标

1．掌握如何编写自动化测试用例文档

1. 编写自动化测试用例的原则

1. 自动化测试用例一般只实现核心业务流程或者重复执行率较高的功能。
2. 自动化测试用例的选择一般以“正向”逻辑的验证为主。
3. 不是所有手工用例都可以使用自动化测试来执行。
4. 尽量减少多个用例脚本之间的依赖。
5. 自动化测试用例执行完毕之后，一般需要回归原点。

2. 编写测试用例

ID	模块	优先级	测试标题	预置条件	步骤描述	测试数据	预期结果	测试结果
001	登录	P0	正确的用户名和密码，成功登录	1.打开首页 2.点击登录链接	1. 输入用户名 2. 输入密码 3. 输入验证码 4. 点击登录按钮	1.用户名：13012345678 2.密码：123456 3.验证码：8888	1.登录成功，页面跳转至我的商城信息页	
002	购物车	P0	搜索商品并添加到购物车	1.用户成功登录 2.进入首页	1.在首页搜索框内输入搜索关键词并点击搜索按键 2.在商品列表中点击搜索到的商品 3.在商品详情页点击添加购物车	1.关键词：小米6	1.提示：添加成功 2.购物车页存在已添加的商品	
003	订单	P0	下订单	1.用户成功登录 2.进入购物车页面	1.进入购物车页面 2.点击‘全选’选中所有商品 3.点击‘去结算’按钮 4.进入填写核对订单页面 5.点击‘提交订单’按钮		1.跳转到订单支付页面 2.提示：订单提交成功，请您尽快付款！	
004	订单	P0	支付	1.用户成功登录 2.进入首页	1.点击‘我的订单’ 2.进入后台订单管理页面 3.点击‘待付款’ 4.点击‘立即支付’ 5.进入订单支付页面 6.选择‘货到付款’ 7.点击‘确认支付方式’		1.跳转到支付成功页面 2.提示：订单提交成功，我们将在第一时间给你发货！	

项目搭建

目标

1. 掌握如何进行自动化测试框架的搭建

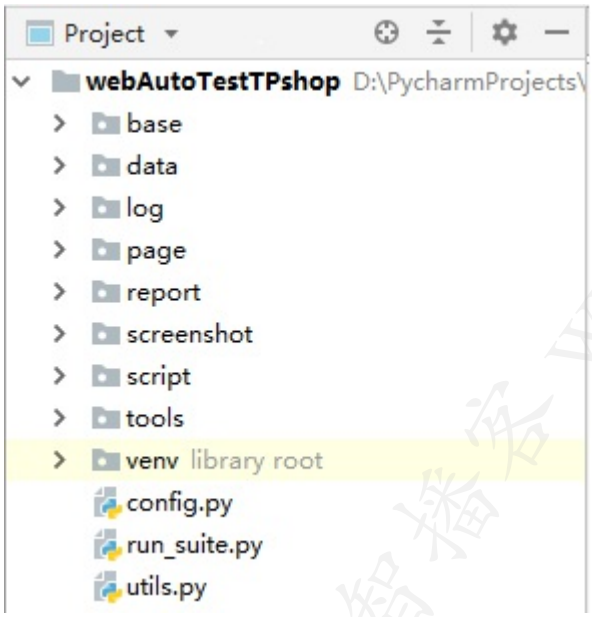
---

1. 初始化项目

1.1 新建项目

项目名称：webAutoTestTPshop

1.2 创建目录结构



1.3 安装依赖包

- 安装 `selenium` 包
- 安装 `parameterized` 包
- 添加 `HTMLTestRunner`

---

2. 初始化代码

- 驱动工具类
- 封装PO基类，定义 `BasePage` 和 `BaseHandle`

编写代码

目标

1. 掌握如何采用PO模式的分层思想对页面进行封装
2. 掌握如何使用UnitTest管理项目中的测试用例

---

1. 抽取PO

根据用例分析待测功能，提取页面对象

1. 定义页面对象文件

```
登录页：login_page.py
首页：index_page.py
后台页面(个人中心页)：home_page.py
商品搜索页：goods_search_page.py
商品详情页：goods_detail_page.py
购物车页：cart_page.py
下订单页：order_page.py
订单支付页：order_pay_page.py
我的订单页：my_order_page.py
```

2. 分别编写对象库层、操作层、业务层的代码

2. 编写测试脚本

1. 定义测试脚本文件

```
登录模块：test_login.py
购物车模块：test_cart.py
订单模块：test_order.py
```

2. 使用unittest管理测试脚本

3. 执行测试脚本

1. 使用unittest执行测试脚本
2. 调试代码

完善代码

目标

1. 掌握如何把数据驱动应用到项目中
2. 掌握如何使用UnitTest生成测试报告

---

1. 数据驱动

1.1 定义数据文件

1. 定义存放测试数据的目录，目录名称：data
2. 分模块定义数据文件

   登录模块：login.json
   购物车模块：cart.json
   订单模块：order.json

3. 根据业务编写用例数据

1.2 测试数据参数化

修改测试脚本，使用 `parameterized` 实现参数化

---

2. 生成测试报告

使用HTMLTestRunner生成测试报告

```
report_file = "./report/report{}.html".format(time.strftime("%Y%m%d-%H%M%S"))
with open(report_file, "wb") as f:
    runner = HTMLTestRunner(f, title="TPshop商城自动化测试报告", description="Win10.Firefox")
    runner.run(suite)
```