1、安装 Python 和 Selenium 包

安装 Python：安装不同平台的 Python 可以在 http://python.org/download/

安装 Selenium：pip3 install -U selenium

PyCharm 设置：使用社区版，配置 Python 的解释器

2、Selenium WebDriver 基于 Python 的实例脚本(Demo)

1）从 Selenium 包导入 WebDriver 才能使用 Selenium WebDriver 的方法；

2）选用一个浏览器驱动实例，会提供一个接口去调用 Selenium 命令来跟浏览器交互；

3）设置 10s 隐式等待时间来定义 Selenium 执行步骤的超时时间；

4）调用 driver.get()方法访问该应用程序，方法调用后，WebDriver 会等待，一直到页面加载完成才继续执行脚本；

5) Selenium WebDriver 提供多种方法来定位和操作这些元素，例如设置值，单击按钮，在下拉组件中选择选项等；

这里使用 find_element_by_id 来定位搜索输入框；这个方法会返回第一个 id 属性值与输入参数匹配的元素；

（HTML 元素是用标签和属性定义的）

6）通过 send_keys()方法输入新的特定值，调用 submit()提交搜索请求；

7）加载搜索结果页面，我们读取结果列表的内容并打印输出；通过 find_elements_by_xpath 获取路径满足

class='c-abstract'的所有 div 标签，它将返回多于一个的元素列表；

8) 最后我们打印，获取到的标签的文本内容；在脚本的最后，我们可以使用 driver.quit()来关闭浏览器；

3、使用 unittest 编写单元测试以及写 Selenium WebDriver 测试

实现执行测试前置条件、测试后置条件，比对预期结果和实际结果，检查程序的状态，生成测试报告，创建数据驱动测试等功能；

1）Test Fixture（测试夹具）：

使用测试夹具，可以定义在单个或多个测试执行之前的准备工作和测试执行之后的清理工作；

2）Test Case（测试用例）：

unittest 中执行测试的最小单元，通过验证 unittest 提供的 assert 方法来验证一组特定的操作和输入以后得到的响应；

unittest 提供了一个名为 TestCase 的基础类，可以用来创建测试用例；

3）Test Suit（测试套件）：

一个测试套件是多个测试或测试用例的集合，是针对被测程序的对应的功能和模块创建的一组测试，一个测试套件内的测试用例将一起执行；

4）Test Runner（测试执行器）：

测试执行器负责测试执行调度并且生成测试结果给用户；

测试执行器可以使用图形界面、文本界面或者特定的返回值来展示测试执行结果；

5）Test Report（测试报告）：

测试报告展示所有执行用例的成功或者失败状态的汇总；包括失败的测试步骤的预期结果和实际结果，还有整体运行状况和运行时间的汇总；

6）一般测试过程中分为三个部分，即 3A's

① Arrange：初始化前置条件，初始化被测试的对象，相关配置和依赖；

② Act：执行功能操作；

③ Assert：用来校验实际结果与预期结果是否一致；

4、用 TestCase 类来实现一个测试

1）我们将通过集成 TestCase 类并且 在测试类中为每一个测试添加测试方法来创建单个测试或者一组测试；

测试用例使用 excel 维护，并且进行参数化，通过自定义 context 上下文管理的类，来操作 excel，对 excel 中的参数进行匹配和替换；

2）TestCase 中的常用的 assert 方法，最主要的任务是：

调用 assertEqual()来校验结果；

assertTrue()来验证条件；

assertRaises 来验证预期的异常；

通过使用第三方库 pymysql（Mysql）查询 SQL，和 TestCase 的返回值，进行匹配校验；

操作过程中重要的返回结果将通过调用 logger 来进行记录，以便快速定位问题；

3）除了添加测试，还可以添加测试夹具，setUp()方法和 tearDown()方法；

4）一个测试用例是从 setUp()方法开始执行，因此可以在每个测试开始前执行一些初始化的任务；此方法无参数，也无返回值；

5）接着编写 test 方法，这些测试方法命名为 test 开头，这种命名约定通知 test runner 哪个方法代表测试方法；

6）值得注意的是：test runner 能找到的每个测试方法，都会在执行测试方法之前先执行 setUp()方法，

这样有助于确保每个测试方法都能够依赖于相同的环境；

7）tearDown()方法会在测试执行完成之后调用，用来清理所有的初始值；

8）最后就是运行测试：为了能通过命令行测试，我们可以在测试中添加对 main 方法的调用；

9）优化：为了能让各个测试方法共用一个实例，我们可以创建类级别的 setUp()和 tearDown()方法：

1）通过 setUpClass()方法和 tearDownClass()方法及@classmethod 标识来实现；

2）这两个方法使在类级别初始化数据，替代了方法级别的初始化；

5、学习 unittest 提供的不同类型的 assert 方法

断言：

unittest 的 TestCase 类提供了很多实用的方法来校验预期结果和实际结果是否一致；以下为常用的集中断言方式：

assertEqual(a, b [, msg]);

assertNotEqual(a, b [, msg]);

assertTrue(x [, msg]); assertFalse(x [, msg]);

assertIsNot(a, b [, msg]);

assertRaises(exc, fun, *args, **kwds);

6、为一组测试创建 TestSuite

应用 unittest 的 TestSuites 特性，可以将不同的测试组成一个逻辑组，然后设置统一的测试套件，并通过一个命令来执行；

具体通过 TestSuites、TestLoader 和 TestRunner 类来实现的；
我们使用 TestSuites 类来定义和执行测试套件，将多可测试加到一个测试套件中；
还用 TestLoader 和 TextTestRunner 创建和运行测试套件；

7、使用 unittest 扩展来生成 HTML 格式的测试报告

8、如何进行元素定位

1）要搜索一个产品，需要先找到搜索框和搜索按钮，接着通过键盘输入要查询的关键字，最后用鼠标单击搜索按钮，提交搜索请求；

2）Selenium 提供了很多 find_element_by 方法定位页面元素，正常定位的话，相应的 WebElement 实例会被返回，

反之将抛出 NoSuchElementException 的异常；

3）8 种 find_element_by 方法：

——find_element_by_id()
——find_element_by_name()
——find_element_by_class_name()
——find_element_by_tag_name()
——find_element_by_xpath()
——find_element_by_css_selector()
——find_element_by_link_text()#标签之间的文本信息
——find_element_by_partial_link_text()

4）8 种 find_elements_by 方法按照一定的标准返回一组元素：

——find_elements_by_id()
——find_elements_by_name()
——find_elements_by_class_name()
——find_elements_by_tag_name()
——find_elements_by_xpath()
——find_elements_by_css_selector()
——find_elements_by_link_text()
——find_elements_by_partial_link_text()

5）值得一提的是 class 定位：class 属性是用来关联 CSS 中定义的属性的；

通过对元素 ID、name、class 属性来查找元素是最为普遍和快捷的方法；

也可以增加一个测试用例断言元素的可用性：

元素操作

（一）、xpath 复杂元素定位（终极总结，各种方法需要多练习定位）

1、//标签名[@属性名称=属性值]　* 匹配所有
2、逻辑运算 and or　//标签名[@属性名称=属性值 and 属性名称=属性值]
3、元素的文本内容　//标签名[text()="元素的文本内容"]　# 文本内容完全匹配
4、部分匹配：文本内容/属性值　contains(text()/@属性,部分值)
//标签名[contains(text(),"部分文本内容")]　# 太长了
//标签名[contains(@属性,"部分属性值")]　# id(不变动+变动) # class 有多个。
5、当你不能通过自己的属性唯一找到的时候，就要利用层级关系　。
5.1、层级定位　第一种方式
后一条件，是在前一个得到的结果之内去搜索。//条件 1//条件 2.....
//div[@id="u1"]//a[@name="tj_login"]
5.2 层级定位 - 轴定位　# 表达式　/轴定位名称::标签名[属性表达]
兄弟姐妹 - 直系的　有比你大的，有比你小的。
preceding-sibling: 哥哥姐姐
following-sibling：弟弟妹妹
//a[@name="tj_trvideo"]/following-sibling::a[@name="tj_login"]
//a[@name="tj_settingicon"]/preceding-sibling::a[@name="tj_login"]
爸爸：parent　祖先：ancestor
//a[@name="tj_trtieba"]/parent::div/a[@name="tj_login"]
//a[text()="百度首页"]/parent::div/following-sibling::div//a[@name="tj_login"]

（二）、等待-三种等待方式

1、强制等待
sleep(秒)
2、隐性等待
implicitly_wait(秒)
设置最长等待时间，在这个时间内加载完成，则执行下一步。
整个 driver 的会话周期内，设置一次即可，全局都可以使用。
3、显性等待
明确等到某个条件满足之后，再去执行下一步操作。
程序每隔 XX 秒看一眼，如果条件成立了，则执行下一步，否则继续等待，知道超过设置的最长时间，
抛出 TimeoutException.--

WebDriverWait 类，显性等待类;
WebDriverwait(dirver,等待市场，轮询周期).until/until_not()

expected_conditions 模块：提供一系列期望发生的条件。
presence_of_element_located:元素存在
visibility_of_element_located:元素可见

element_to_be_clickable:元素可点击
ps:这个类有很多判断方法。经常使用的就这几种!

例如:

1、当你的操作带来了页面的变化，请一定要等待
time.sleep(5)#傻等
2、隐形等待-智能等待：如果你 10 秒出现了，我就开始下一步操作。设置上限：30 秒 超时 TimeoutException
3、显性等待-智能等待：明确的条件（元素可见，窗口存在）。等待+条件（如果你 10 秒出现了，我就开始下一步操作。）
导包:

```
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
```

元素存在（html 里面，能找到）;
元素可见（存在并且可见-看得见大小-可见才可操作）;
元素可用（可见之后，才有可用的可能性。只读/不可点击-不可用）
等待条件表达
locater = (定位类型，定位表达式)

```
locater = (By.ID,'TANGRAM__PSP_10__footerULoginBtn')
```

EC.visibility_of_element_located(locater) #条件
等待元素可见

```
WebDriverWait(driver,30).until(EC.visibility_of_element_located(locater))
```

使用 sleep 短暂等待，辅助- 0.5 秒

```
time.sleep(0.5)
```

点击元素

```
driver.find_element_by_id('TANGRAM__PSP_10__footerULoginBtn').click()
```

（三）、打开新窗口部分核心代码

1、

```
#step1:获取窗口数
handles = driver.window_handles # >>> 只有一个 窗口
# step2:打开新窗口的操作
driver.find_element(*locator).click()    #本操作带来新窗口的出现   >>> 2 个窗口

############ 第二种 ###########
# step3:确认新窗口出现了，我再去操作他，等待新窗口出现
# EC.new_window_is_opened # 比窗口总数的大小  # 传一个窗口的总数
WebDriverWait(driver,10).until(EC.new_window_is_opened(handles))    #2>1 确认新窗口出现
# step4:再次获取 窗口的 handles
handles = driver.window_handles
# step5:切换 新的窗口
driver.switch_to.window(handles[-1])
```

2、

```
driver.find_element(*locator).click()    #本操作带来新窗口的出现
# 打开新的窗口
#    1、获取所有的窗口
handles = driver.window_handles
print(driver.window_handles)
#    当前窗口的 handle
print(driver.current_window_handle)
#    2、切换新的窗口
driver.switch_to.window(handles[-1])
print("切换之后的窗口为：",driver.current_window_handle)
```

项目实战+框架

(一)、自动化应用场景和用例设计

什么是 PO 模式?

1.页面对象模型(PO)是一种设计模式,用来管理维护一组页面元素的对象库.

2.在 PO 下,应用程序的每一个页面都有一个对应的 Page 类.

3.每一个 Page 类维护着该页面的元素集和操作这些元素的方法.

PO 模式的好处:

1.代码可读性强

2.可维护性高

3.复用性高

怎么设计 PO 模式:

设计的原则

1.抽象每一个页面

2.页面中元素不暴露,仅报错操作元素的方法

3.页面不应该有繁琐的继承关系

4.页面中不是所有元素都需要涉及到,核型业务元素做建模使用

5.把页面划分功能模块,在 Page 中实现这些功能方法

PO 的优势

1、PO 提供了一种业务流程与页面元素操作分离的模式，这使得测试代码变得更加清晰。

2、页面对象与用例分离，使得我们更好的复用对象。

3、可复用的页面方法代码会变得更加优化

4、更加有效的命名方式使得我们更加清晰的知道方法所操作的 UI 元素。例如我们要回到首页,

方法命名为：gotoHomePage()，通过方法名即可清晰的知道具体的功能实现。

PO 模式--项目

a、元素定位发生了变化...找到元素到底在那些文件当中，需要修改

b、相同功能-找到哪些地方用到了这些功能

HTML+DOM

元素定位 - 8 xpath- 多种

元素操作：4 大基本：send_keys,click,text,get_attribute

等待 - 3 种 sleep-助攻 显性等待-主攻 隐性等待-幕后/佯攻

3 种切换 - switch_to

window , iframe ， alert - 先找到，再切换。

下拉列表-Select 鼠标操作 - ActionChains 按键操作 - Keys

js 助攻 - 日历 / 修改属性 /滚动条

上传 - windows - autoit/pywin32 -windows --mac sikuli

(二)、项目实战

1、 项目 - web 自动化？ - 一份大概的执行计划。

为什么要做 web 自动化测试、接口自动化。?

自动化的目的:

前置、步骤、期望结果() - 1 点。写一份脚本 - 运行 N 遍 - 测试 N 次

1 - 2 个测试 - 10 个功能 - 2 天 - 上线频率：2 周 1 次

2 - 2 个测试 - 100 个功能 - 10 天 - 上线频率：2 周 1 次 - 所有功能都得测。

50 个功能 - 重复测试了 50 遍 - 同样的功能点点点。 -- 枯燥/重复/烦

-- 重复工作，没有提升。

-- 每次点的时候，不会覆盖所有的细节功能。

-- 上线，历史功能出问题。

尽量避免上线出问题 -- 客户反馈过的问题，适当加入自动化当中。

解放双手、解放时间 -- 提高一个测试反馈效率。

每次回归的功能都保持一致 -- 自动化的用例是什么，每次都运行的什么。

项目周期比较长 - 历史功能(稳定)

在开发后台接口的阶段，同步做接口自动化测试 -- 占比 70% - 100% (单接口 + 流程场景)

-- 团队的 质量意识/单元测试/测试能力高/沟通要及时

-- web 自动化 - 最接近用户操作。--占比 20% -- 主流程 + 易实现的异常用例

2、用工具和框架的区别?

项目实战 - web 自动化？ - 一份大概的执行计划。

1、了解被测对象 - 业务需求。

2、功能模块？1000 功能用例 - 5 大模块。 上线 bug 率最高的模块 - 核心模块

- 稳定的

3、测试用例？500 个功能用例 - 50-100 个左右 主流程业务 + 易实现的异常用例。

其它测试人员

实现 100 个功能用例 - 自动化

4、框架选型 - 自己写 - python+selenium+..... 2 周

帮助团队提升整体的测试水平，提升团队各个成员的测试技能，是每个测试人员的职责。（学会营造学习气氛，以自身影响他人学习，进步）

整体的测试水平。1 个人 - 用你最熟悉的技能来实现。 团队 - 工具(规范)

框架 - 规定哪些层级里放什么,规定编写规则-文件命名/定位的规则/注释要求等，准备示范的例子。

（好处）- 结构设计+规范。----管理、灵活扩展、维护工作量少

（好处）- 尽早定期评审团队成员脚本 - 发现问题，解决问题，定规则。

标准就是：较好的可读性、注释量要有以免回头看不懂写的啥。

3、实现脚本 - PO 模式

(1)、PO 模式 PageObject

测试用例 = 页面对象 + 测试数据

测试用例与测试对象分离

测试对象层：元素定位发生了变化？ 页面功能变化或者新增?

测试用例层：用例步骤、用例数据、测试数据 变化

目的：容易维护、容易扩展

(2)、写 web 用例原则：

1、**稳定性**最最最重要。可以**牺牲时间**来提高稳定性;

2、用例要保持独立性。 不依赖于其他的用例运行结果;

3、如果用例的流程很长，可以拆成几个用例，它就不独立。

4、尽量少的依赖环境数据（在任何情况下，都自给自足，自己创建条件）

(3)、分层设计思想：

PageLocators - > PageObjects

PageObjects - > TestCases

TestDatas - > TestCases

元素定位层

页面对象层

测试用例层

测试数据层

(4)、初衷和目的：解放双手，解放时间，提高反馈效果。

(5)、应用场景：冒烟-(正常场景，不一定要有断言，能不能用？)

回归-(全面覆盖，异常场景(准备工作很多，甚至需要人为干预))。

(三)、Pytest 框架的使用

1、Pytest 介绍

基于 unittest 之上的单元测试框架

(1)、自动发现测试模块和测试方法;

(2)、断言使用 assert+表达式即可;

(3)、可以设置会话(从运行所有用例开始-用例结束)级，模块(.py)级，类级(setupClass/teardownClass),

函数(测试用例)级的 fixtures，数据准备+清理工作

(4)、有丰富的插件，300+以上。==allure

(5)、测试用例不一定要放在测试类当中。

安装命令：pip install pytest

安装 html 报告插件：pip install pytest-html

pytest 插件地址：http://plugincompat.herokuapp.com/

pytest 收集测试用例的规则：

(1)、默认从当前目录中收集测试用例，即在哪个目录下运行 pytest 命令，则从哪个目录当中搜索；

(2)、搜索规则：

a、符合命名规则，test_*.py 或者 *_test.py 的文件；

b、以 test_开头的函数名；

c、以 Test 开头的测试类，（没有__init__函数）当中，以 test_开头的函数；

d、断言使用基本的 assert 即可；

Pytest 的特点：

1、简单灵活，容易上手，文档丰富；

2、支持参数化，可以细粒度地控制要测试的测试用例；

3、能够支持简单的单元测试和复杂的功能测试，

还可以用来做 selenium/appnium 等自动化测试、

接口自动化测试（pytest+requests）；

4、pytest 具有很多第三方插件，并且可以自定义扩展，比较好用的如:

pytest-selenium（集成 selenium）、

pytest-html（完美 html 测试报告生成）、

pytest-rerunfailures（失败 case 重复执行）、

pytest-xdist（多 CPU 分发）等；

5、测试用例的 skip 和 xfail 处理；

6、可以很好的和 CI 工具结合，例如 jenkins

2、pytest 之 mark 功能

mark 机制　4.6

先注册　pytest.ini　[pytest] markers=标签名:说明

去给用例打标签

@pytest.mark.已注册的标签名

测试类和模块：　类下面设置类属性值，模块下面设置全局变量。

pytestmarker=pytest.mark.已注册的标签名

多个标签：pytestmarker=[pytest.mark.已注册的标签名,pytest.mark.已注册的标签名]

3、pytest 之命令运行用例

使用命令行运行 pytest

4、pytest 之 fixture 功能

(1)、定义 fixture

1.1 创建了一个 conftest.py 文件

1.2 在 conftest 中，创建 fixture

1.3 定义函数，函数前面加上@pytest.fixture(scope=作用域)

函数内部：yield 隔开前置后后置的代码，之前是前置，之后是后置

yield 返回值（后面跟上返回值用于调用）

(2)、调用 fixture

在测试用例.测试类 前面加上(@pytest.mark.usefixtures("fixture 对应的函数名称"))；

fixture 对应的函数名称=它的返回值；

fixture 对应的函数名称作为测试用例的参数，将返回值传给测试用例；

fixure 在 conftest.py 当中，定义的时候，就已经决定了他的用例域，决定了它的命运；

fixture 可以有很多个；

无论在测试类、测试用例去主动调用 fixture,都不能够改变它的命运；

调用就是决定在哪儿去使用它。在哪个测试类？

pytest 的用例执行顺序：

基本原则：按照搜索规则，先匹配到的先执行。

1、文件名称：按名称名称顺序去搜索。先找到的，先去内部找用例。

2、在 py 文件内部：按照代码顺序去找用例。先找到的先执行。

(3)、fixture 暂不支持与 unittest 同用，断言都用 assert.

(4)、pytest 之 fixture 参数化-多运行，pytest 层级覆盖。测试用例与其同级或者在其子目录

(5)、fixture 的 scope 参数

scope 参数有四种，分别是'function','module','class','session'，默认为 function。

function：每个 test 都运行，默认是 function 的 scope

class：每个 class 的所有 test 只运行一次

module：每个 module 的所有 test 只运行一次

session：每个 session 只运行一次

(6)、setup 和 teardown 操作

setup，在测试函数或类之前执行，完成准备工作，例如数据库链接、测试数据、打开文件等

teardown，在测试函数或类之后执行，完成收尾工作，例如断开数据库链接、回收内存资源等

备注：也可以通过在 fixture 函数中通过 yield 实现 setup 和 teardown 功能

(7)、fixture 定义与调用

定义 == 定命运。session、modle、class、function

调用 == 你准备把它在哪儿用？

session:整个会话都有效。

module:模块内有效。

class:类内有效。

function:测试用例内有效。

conftest.py 文件。 === 定义多个 fixture.

5、pytest 之参数化---ddt

参数化　ddt　　参数名 = 用例的参数名称

在测试用例的前面加上：

@pytest.mark.parametrize("参数名",列表)

参数名：用来接收每一项数据，并作为测试用例的参数；

@pytest.mark.parametrize("参数 1，参数 2",[(数 1，数 2),(数 1，数 2)]);

排列组合。多个参数的值排列组合。在一个用例前面，使用多个 @pytest.mark.parametrize

示例：用例有 4 个：0,2/0,3/1,2/1,3

```
@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
    pass
```

6、pytest 之重运行

插件名称：rerunfailures

安装方法：pip install pytest-rerunfailures（失败 case 重复执行）

使用方式：

命令行参数形式：

命令：pytest -reruns 重试次数

比如：pytest --reruns 2 表示：运行失败的用例可以重新运行两次

命令：pytest --reruns 重试次数 --reruns-delay 次数之间设置的延时（单位：秒）

Pytest --reruns 2 --reruns-delay 5

表示失败的用例可以重新运行 2 次，第一次和第二次的时间间隔为 5 秒；

7、pytest 之 HTML 报告

测试报告　= junitxml,html,allure

1、先装插件

2、命令行的参数：

--html=相对路径/report.html　# 相对于 pytest 命令运行时，所在的根目录。

--alluredir=相对路径

3、安装 allure 命令行工具：下载，解压，配置环境变量

4、生成 allure 文件之后，用命令：allure serve alluredir

os.system("")

allure 与 jenkins 的集成、重运行机制、pytest 中的失败截图。

Pytest 可以生成多种样式的结果：　pytest plugin(下载插件用)

1、生成 JunitXML 格式的测试报告：命令：--junitxml = path

2、生成 result log 格式的测试报告：命令：--resultlog=report、log.txt

3、生成 Html 格式的测试报告：命令：--html=report\test_one_func.html(相对路径)

pytest -v -s -m demo --html=Outputs/reports/pytest_run_reports.html

8、pytest 之 allure 测试（allure 测试报告）

https://docs.qameta.io/allure/#_pytest

9、pytest 之 jenkins 持续集成

- jenkins 分别安装插件 Allure Jenkins Plugin 、HTML Publisher plugin
- 在系统设置中将 JDK / Maven 等其他一些基本配置(建议 jdk 版本 1.8)
- 在 jenkins 中添加 allure 执行工具

jenkins 添加 allure 执行工具

1.先下上面提到的 allure 执行的压缩文件(allure-commandline.zip)

2.在 jenkins 的系统配置——Allure Commandline —-去掉自动安装勾选框—-填写 name 及刚下载的文件夹的根目录路径

3.点击保存

- 新建 job, 在 job 中添加步骤 Allure report(Results: 第一行表示 xml 文件的路径; 第二行表示生成报告的路径)

目前做法

目前 job 的整个流程:

1.构建时，从 svn 上拉取最新的测试代码

2.执行测试脚本并且生成 allure 需要 xml 结果文件

3.通过 Allure report 生成测试报告

4.设置问题追踪

Web 自动化框架总结

1、分层设计思想、关键字驱动（在 basePage 中进行封装）

2、PO 模式:

- PageLocators 元素定位层
- PageObjects 测试对象
- TestCases 测试用例

3、分层:

- TestDatas 测试数据 -- 重用、不同的环境不同的数据

4、数据驱动: DDT 模块

- 异常场景，数据校验，刷新当前场景（异常）进行执行测试用例;
- 每个用例启动会话/所有用例启动会话
- 尽量不依赖测试环境
- 选择自己创造条件
- 尽量保证用例的独立性

- 提高用例的稳定性
- 连续运行 5 次（运行稳定后再放入 jenkins 当中）
- 尽早的加入持续集成
- 定位表达式的灵活度、等待方面的处理（决定用例的稳定性）
- 用例不宜复杂，拆分成多个用例（考虑成本，选择性部分用例就进行自动化用例，部分前置后置，可以用接口来代替）

5、basePage 关键字封装

1)、等待、查找元素、点击、输入、截图、获取文本、获取属性、窗口切换、iframe 切换、上传、select 等等

2)、元素的操作：等待、查找、操作

3)、实现了每一个页面的每一个操作，进行了异常捕获，失败截图，操作时间

6、PageObjects 当中，全部调用 basePage 提供的关键字，来封装业务函数

7、pytest 测试框架：更方便的筛选用例 - 冒烟 / 回归

更方便的组织用例：函数/类

特点：

1)、自动发现测试用例（test_*/*_test.py、test_的函数/Test 类并没有初始化函数中的 test_函数）

2)、assert 断言

3)、fixture(前置后置) 测试会话级别（session）、测试模块级别（module）、测试类（class）、测试用例（function）

4)、丰富的插件库，allure 库。

8、mark 功能：

1、标签名必须先注册：pytest.ini(根目录)

```
[pytest]
    markers=
        smoke:this is a smoke tag
        demo:demo
        login:login model
```

2、测试用例/测试类/测试模块

测试用例 ： @pytest.mark.smoke

测试类、测试模块：pytest.marker=[pytest.mark.smoke,pytest.mark.login]

9、pytest 命令行

1、pytest 在哪个目录下面运行，那就是在哪个目录当中去搜索用例

2、根据标签过滤用例：-m 标签名

3、-s -v 在控制台当中看到更详细的用例运行状态

10、pytest 的测试报告

1、xml --- 跟 jenkins 集成，就是解读的 xml 文件

2、html --- 插件：pytest-html 参数：--html=../../(相对 pytest 的命令执

行目录的相对路径 ）

11、conftest.py

1、共享前置后置

2、不同的包目录当中，可以自己的 conftest.py

3、定义前置后置

```
@pytest.fixtures(scope=)
def func1():
        # 前置
        yield  [返回值]
       # 后置

要在 func1 的基础上，有更多的操作
@pytest.fixtures(scope=)
def func2(func1):
        # 前置
        yield  [返回值]
       # 后置
```

4、测试类/测试函数 @pytest.mark.usefixtures("函数名称")

如果有返回值，那么把 函数名称 作为测试用例的参数即可。 函数名称 = 返回值。

12、数据驱动：pytest 参数化

```
1、@pytest.mark.parametrize("param1",[1,2,3,4])
    def test_11(param1):
        pass
2、@pytest.mark.parametrize("param1,param2",[(1,2),(3,4)])
    def test_11(param1,param2):
        pass

3、@pytest.mark.parametrize("param1",[1,2,3,4])
    @pytest.mark.parametrize("param2",[a,b])
    def test_11(param1,param2):  # 两个参数排列组合 ， 有 8 组测试用例。
        pass
```

13、失败重试机制：rerun 插件：pytest-rerun pytest 命令参数：(自行补充)pytest --rerun 5 --reruns-delay 1

14、pytest-allure 报告集成：

1、allure 命令环境的安装

2、pytest - 安装 allure 插件 命令参数：--alluredir=../..(相对 pytest 的命令执

行目录的相对路径 ）

3、allure -serve

4、jenkins 进一步集成：安装 jenkins 上的 allure 插件　　jenkins-allure-adopter

15、selenium 框架中需要创建的包名：

PageLocator

PageObjects

TestCases(conftest.py)

TestDatas

common：basepage、loggger、工程路径配置

outputs：日志、截图、报告

main.py：框架入口

API：接口

16、自动化运行测试、测试用例组织、测试报告自动生成、测试日志自动生成、失败截图的自动生成 、指定输出路径 、提供入口

编写规范：团队合作

文件命名规范、函数命名规范、通过的英名名称(login、action)

元素定位规范：非绝对定位　　定期互相检查

按模块分层次：事先搭好。

良好的注释：养成注释的习惯，对以后查看代码，优化代码有很大的帮助。
