1. 项目架构说明

       该项目架构基于楼主公司的一款B/S软件设计，大家也可以根据自己的被测软件来构建适合自己的架构层级，当然也可以参考楼主的。做自动化测试项目，当搞懂了思想和方法，其实都是万变不离其宗，就跟写代码一样，语言万千种，唯一不变的就是语言中殊途同归的思想，因此，玩会了套路自然就能凌驾于套路之上，运用并加入自己的东西。在PyCharm中新建如下的项目层级：

       有过开发经历的小伙伴都知道，有个好的交互式开发工具对于我们创建和管理清晰的项目架构很方便，PyCharm就是一款交互良好的python开发工具。楼主上面的项目层级中部分目录和目录下的文件没展开，下面显示一个完整的目录结构，并说明每个目录是用来干嘛？放什么东西？

        当然这个项目层级设计，不是楼主一时间就固定下来的，也是在不断的摸索和采坑中，不断调整出的一个适合自己的框架目录层级。项目框架设计好了后，接下来就是慢慢补充内容一步步实现上面每个目录需要的东西。来吧，开始造轮子~

三、配置文件设计

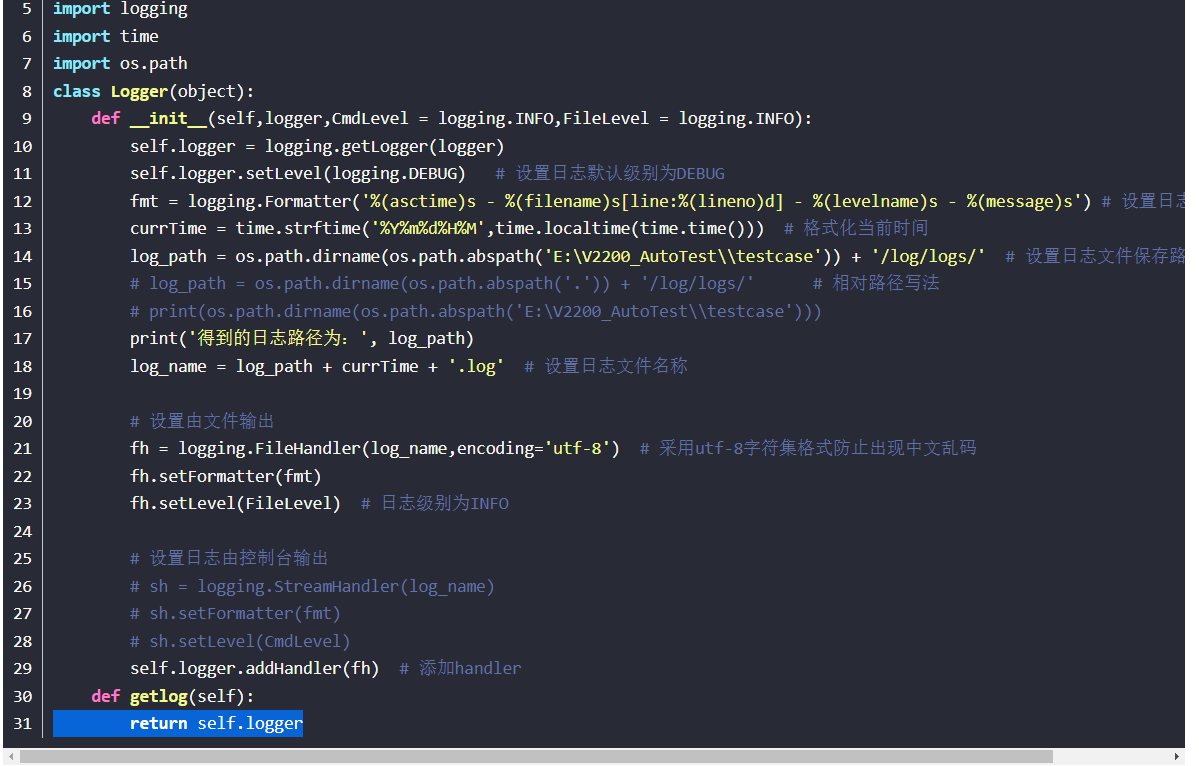
       首先，对于上面的config.ini的配置文件进行配置。说到配置文件，不管是开发人员还是测试人员都不会陌生，还有xml、txt等格式的配置文件，配置文件就是用来配置一些参数和固定的变量值，一般是程序固定不变的东西我们就放这里面，用于程序直接调用，如果修改配置文件中变量的值，程序调用该变量就会产生不同的输出行为。如在做自动化测试时，我们可以将测试的不同浏览器写入到该文件中，当我们需要调用firefox浏览器时，将参数设置成firefox即可，测试脚本将会在火狐浏览器进行。如下图，在配置文件中设置了浏览器参数、测试url、邮件服务器、邮件发送和接收者等参数。

四、日志类模块的实现

       说到日志，大家都明白日志的作用，最明显的作用就是在程序的关键位置或者步骤节点下设置日志输出，当程序运行时，会输出日志，日志是我们查看程序运行情况和查找错误的重要手段。因此，对于自动化测试也是如此，我们需要知道自动化执行的情况以及执行错误的情况发生了什么，那就需要给你的自动化测试项目封装一个日志类的功能模块，用于输出日志。python语言封装了一个叫logging的标准库模块，能够设置日志等级以及怎么输出、输出到哪里。对于logging模块，大家可以自己针对性去学习该模块的使用。

       我们在上面的项目层级的models目录下创建log.py文件，在这个模块文件下定义一个叫做Logger的日志类，完成日志类的封装，编辑如下代码

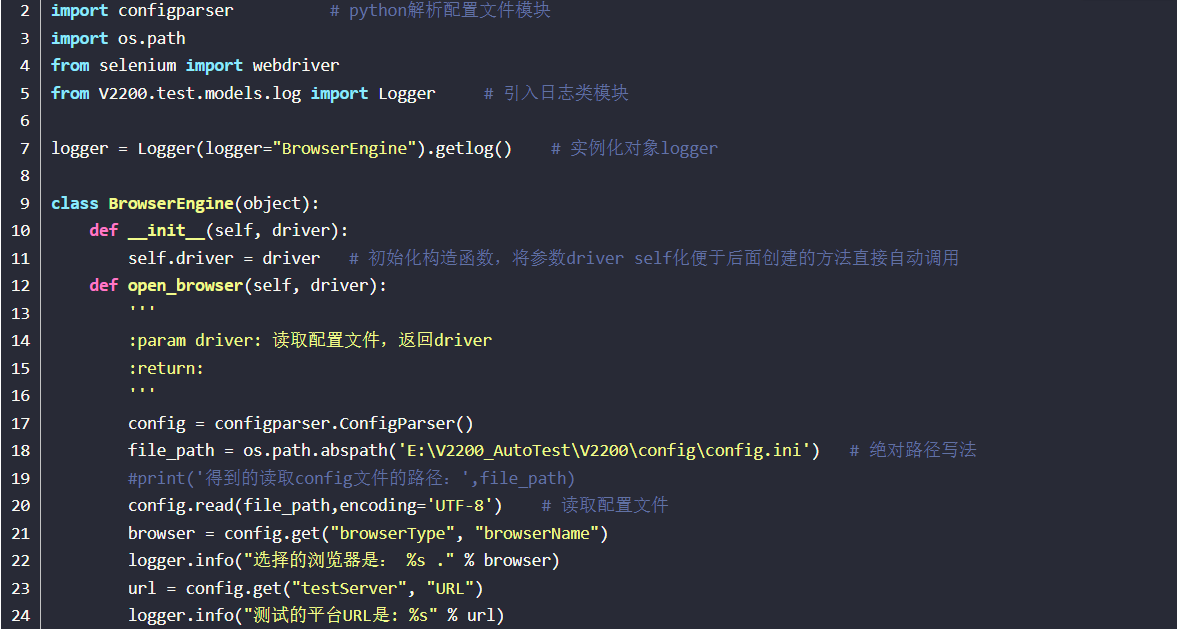
import logging  
import time  
import os.path  
  
  
class Logger(object):  
 def \_\_init\_\_(self, logger, CmdLevel=logging.INFO, FileLevel=logging.INFO):  
 self.logger = logging.getLogger(logger)  
 self.logger.setLevel(logging.DEBUG) # 设置日志默认级别为DEBUG  
 fmt = logging.Formatter('%(asctime)s - %(filename)s[line:%(lineno)d] - %(levelname)s - %(message)s') # 设置日志输出格式  
 currTime = time.strftime('%Y%m%d%H%M', time.localtime(time.time())) # 格式化当前时间  
 log\_path = os.path.dirname(os.path.abspath('E:\V2200\_AutoTest\\testcase')) + '/log/logs/' # 设置日志文件保存路径  
 # log\_path = os.path.dirname(os.path.abspath('.')) + '/log/logs/' # 相对路径写法  
 # print(os.path.dirname(os.path.abspath('E:\V2200\_AutoTest\\testcase')))  
 print('得到的日志路径为：', log\_path)  
 log\_name = log\_path + currTime + '.log' # 设置日志文件名称  
  
 # 设置由文件输出  
 fh = logging.FileHandler(log\_name, encoding='utf-8') # 采用utf-8字符集格式防止出现中文乱码  
 fh.setFormatter(fmt)  
 fh.setLevel(FileLevel) # 日志级别为INFO  
  
 # 设置日志由控制台输出  
 # sh = logging.StreamHandler(log\_name)  
 # sh.setFormatter(fmt)  
 # sh.setLevel(CmdLevel)  
 self.logger.addHandler(fh) # 添加handler  
  
 def getlog(self):  
 return self.logger



 这样我们就自定义封装了一个简单的日志类模块，设置了日志输出级别、输出格式以及输出日志的位置，后面其他模块需要输出日志时，就可以调用引入该日志类。

 五、浏览器模块的实现

     日志类实现简单封装后，继续造轮子~。此部分用于封装浏览器模块，主要实现打开和关闭不同浏览器的方法，这里就用到了POM的思想，咱们封装了浏览器的类型和打开关闭方法，那么后面每条测试脚本就可以直接调用打开和关闭浏览器方法，脚本只需要专注具体的测试业务逻辑的实现即可。在models目录下新建broser\_engine.py文件，自定义一个叫做BrowserEngine类，实现浏览器模块的封装，代码如下：

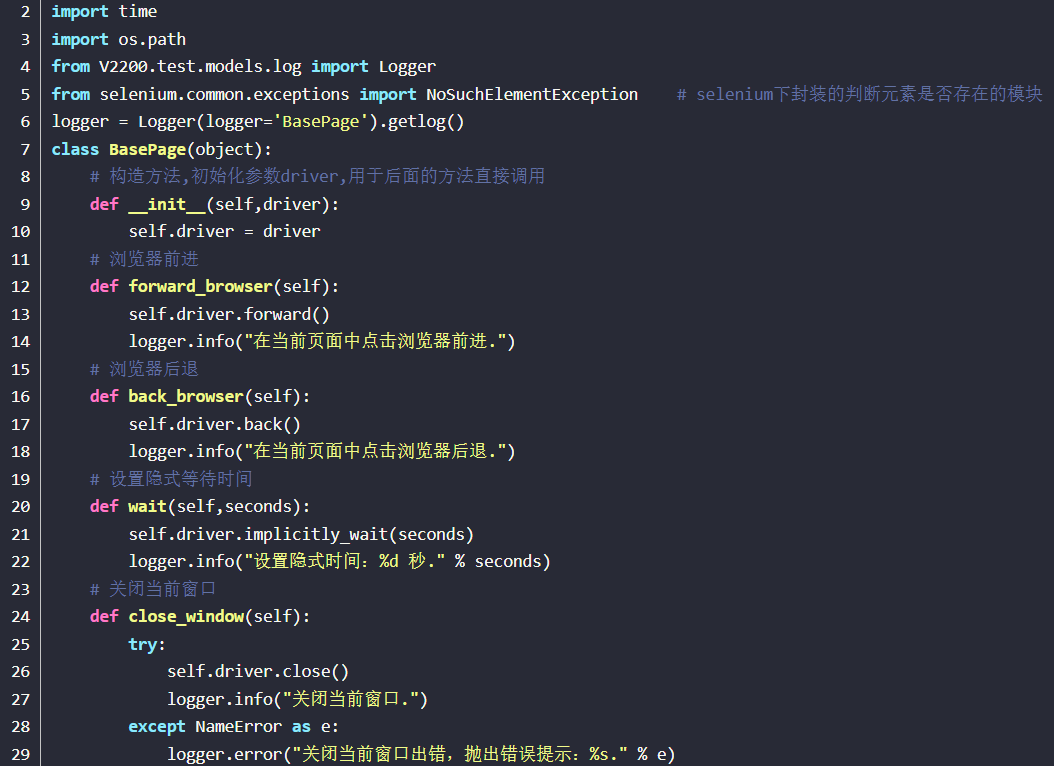


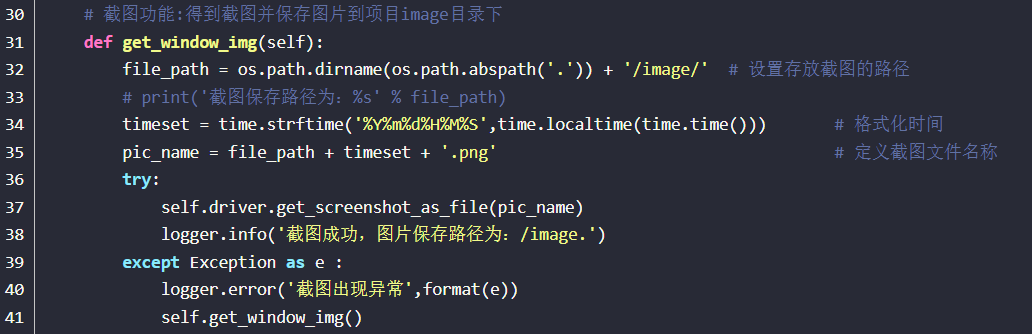


import configparser # python解析配置文件模块  
import os.path  
from selenium import webdriver  
from V2200.test.models.log import Logger # 引入日志类模块  
  
logger = Logger(logger="BrowserEngine").getlog() # 实例化对象logger  
  
  
class BrowserEngine(object):  
 def \_\_init\_\_(self, driver):  
 self.driver = driver # 初始化构造函数，将参数driver self化便于后面创建的方法直接自动调用  
  
 def open\_browser(self, driver):  
 *'''* ***:param*** *driver: 读取配置文件，返回driver* ***:return****:  
 '''* config = configparser.ConfigParser()  
 file\_path = os.path.abspath('E:\V2200\_AutoTest\V2200\config\config.ini') # 绝对路径写法  
 # print('得到的读取config文件的路径：',file\_path)  
 config.read(file\_path, encoding='UTF-8') # 读取配置文件  
 browser = config.get("browserType", "browserName")  
 logger.info("选择的浏览器是： %s ." % browser)  
 url = config.get("testServer", "URL")  
 logger.info("测试的平台URL是: %s" % url)  
  
 if browser == "Firefox":  
 driver = webdriver.Firefox()  
 logger.info("Starting firefox browser.")  
 elif browser == "Chrome":  
 driver = webdriver.Chrome()  
 logger.info("Starting Chrome browser.")  
 elif browser == "Ie":  
 driver = webdriver.Ie()  
 logger.info("Starting IE browser.")  
  
 driver.get(url) # 得到测试的url  
 logger.info("浏览器的版本为：%s" % driver.capabilities['version']) # 获取浏览器版本  
 driver.maximize\_window()  
 logger.info("最大化浏览器窗口.")  
 driver.implicitly\_wait(10)  
 return driver  
  
 def quit\_browser(self):  
 self.driver.quit()

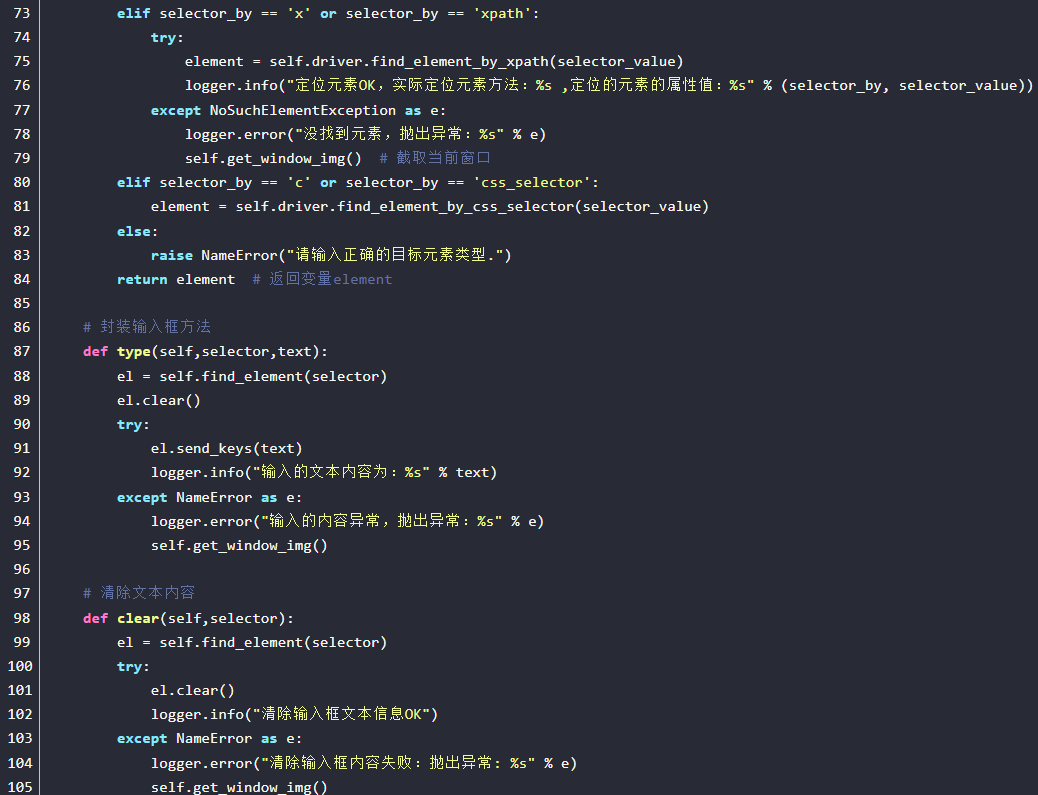
六、页面基类的实现

        此部分用于**封装页面基类**，主要用于封装一些**常用的公共方法**，如截图方法、元素定位方法、元素通用操作方法、警示框处理方法等等，**只要软件页面一些常用的操作都可以写在该页面基类中**，这个页面基础类就类似于一个公共函数库一样，封装这些方法，后面有需要的地方直接调用即可。如下代码，已经封装了8大元素定位方法、截图方法、鼠标点击方法、警示框处理方法等，后续根据自己的需要自行补充丰富一些常用的功能函数或者方法。这里可以着重看下8大元素定位方法的封装.











import time  
import os.path  
from V2200.test.models.log import Logger  
from selenium.common.exceptions import NoSuchElementException # selenium下封装的判断元素是否存在的模块  
  
logger = Logger(logger='BasePage').getlog()  
  
  
class BasePage(object):  
 # 构造方法,初始化参数driver,用于后面的方法直接调用  
 def \_\_init\_\_(self, driver):  
 self.driver = driver  
  
 # 浏览器前进  
 def forward\_browser(self):  
 self.driver.forward()  
 logger.info("在当前页面中点击浏览器前进.")  
  
 # 浏览器后退  
 def back\_browser(self):  
 self.driver.back()  
 logger.info("在当前页面中点击浏览器后退.")  
  
 # 设置隐式等待时间  
 def wait(self, seconds):  
 self.driver.implicitly\_wait(seconds)  
 logger.info("设置隐式时间：%d 秒." % seconds)  
  
 # 关闭当前窗口  
 def close\_window(self):  
 try:  
 self.driver.close()  
 logger.info("关闭当前窗口.")  
 except NameError as e:  
 logger.error("关闭当前窗口出错，抛出错误提示：%s." % e)  
  
 # 截图功能:得到截图并保存图片到项目image目录下  
 def get\_window\_img(self):  
 file\_path = os.path.dirname(os.path.abspath('.')) + '/image/' # 设置存放截图的路径  
 # print('截图保存路径为：%s' % file\_path)  
 timeset = time.strftime('%Y%m%d%H%M%S', time.localtime(time.time())) # 格式化时间  
 pic\_name = file\_path + timeset + '.png' # 定义截图文件名称  
 try:  
 self.driver.get\_screenshot\_as\_file(pic\_name)  
 logger.info('截图成功，图片保存路径为：/image.')  
 except Exception as e:  
 logger.error('截图出现异常', format(e))  
 self.get\_window\_img()  
  
 # 8大页面元素（对象）定位方法的封装  
 def find\_element(self, selector):  
 *'''  
 使用‘=>’作为字符串分割符，后续实际测试用例根据输入的元素selector\_by和selector\_value 进行选择元素的定位类型* ***:param*** *selector:* ***:return****: element  
 '''* element = ''  
 if '=>' not in selector:  
 return self.driver.find\_element\_by\_id(selector)  
 selector\_by = selector.split('=>')[0] # 按=>分割符进行切割字符串，返回一个列表，得到列表的第一个元素，即元素的定位方法  
 selector\_value = selector.split('=>')[1] # 得到列表的第二个元素，即元素定位的值  
  
 if selector\_by == 'i' or selector\_by == 'id':  
 try:  
 element = self.driver.find\_element\_by\_id(selector\_value)  
 logger.info("定位元素OK，实际定位元素方法：%s ,定位的元素的属性值：%s" % (selector\_by, selector\_value))  
 except NoSuchElementException as e:  
 logger.error("没找到元素，抛出异常：%s" % e)  
 self.get\_window\_img() # 截取当前窗口  
 elif selector\_by == 'n' or selector\_by == 'name':  
 element = self.driver.find\_element\_by\_name(selector\_value)  
 elif selector\_by == 'c' or selector\_by == 'class\_name':  
 element = self.driver.find\_element\_by\_class\_name(selector\_value)  
 elif selector\_by == 'l' or selector\_by == 'link\_text':  
 element = self.driver.find\_element\_by\_link\_text(selector\_value)  
 elif selector\_by == 'p' or selector\_by == 'partial\_link\_text':  
 element = self.driver.find\_element\_by\_partial\_link\_text(selector\_value)  
 elif selector\_by == 't' or selector\_by == 'tag\_name':  
 element = self.driver.find\_element\_by\_tag\_name(selector\_value)  
 elif selector\_by == 'x' or selector\_by == 'xpath':  
 try:  
 element = self.driver.find\_element\_by\_xpath(selector\_value)  
 logger.info("定位元素OK，实际定位元素方法：%s ,定位的元素的属性值：%s" % (selector\_by, selector\_value))  
 except NoSuchElementException as e:  
 logger.error("没找到元素，抛出异常：%s" % e)  
 self.get\_window\_img() # 截取当前窗口  
 elif selector\_by == 'c' or selector\_by == 'css\_selector':  
 element = self.driver.find\_element\_by\_css\_selector(selector\_value)  
 else:  
 raise NameError("请输入正确的目标元素类型.")  
 return element # 返回变量element  
  
 # 封装输入框方法  
 def type(self, selector, text):  
 el = self.find\_element(selector)  
 el.clear()  
 try:  
 el.send\_keys(text)  
 logger.info("输入的文本内容为：%s" % text)  
 except NameError as e:  
 logger.error("输入的内容异常，抛出异常：%s" % e)  
 self.get\_window\_img()  
  
 # 清除文本内容  
 def clear(self, selector):  
 el = self.find\_element(selector)  
 try:  
 el.clear()  
 logger.info("清除输入框文本信息OK")  
 except NameError as e:  
 logger.error("清除输入框内容失败：抛出异常: %s" % e)  
 self.get\_window\_img()  
  
 # 封装点击元素的动作  
 def click(self, selector):  
 el = self.find\_element(selector)  
 try:  
 el.click()  
 logger.info("点击元素动作完成")  
 except NameError as e:  
 logger.error("点击事件失败，抛出异常：%s" % e)  
  
 # 获取打开的url地址标题  
 def get\_page\_title(self):  
 logger.info("当前打开的url地址标题为：%s" % self.driver.title)  
 return self.driver.title  
  
 # 获取警示框，并得到提示框信息和关闭提示框  
 def get\_alert(self):  
 el = self.driver.switch\_to.alert # 获取窗口弹窗的方法  
 try:  
 assert '用户名或者密码错误' in el.text # el.text方法获取提示框内容  
 logger.info("弹窗提示正确")  
 el.accept() # 点击弹窗确认按钮  
 except Exception as e:  
 print('弹窗提示错误', format(e))  
  
 @staticmethod # 静态方法：不强制要求传递参数，类可以不用实例化就能调用该方法  
 def sleep(seconds):  
 time.sleep(seconds)  
 logger.info("等待时间是：%s 秒" % seconds)

七**、登陆页面元素的封装**

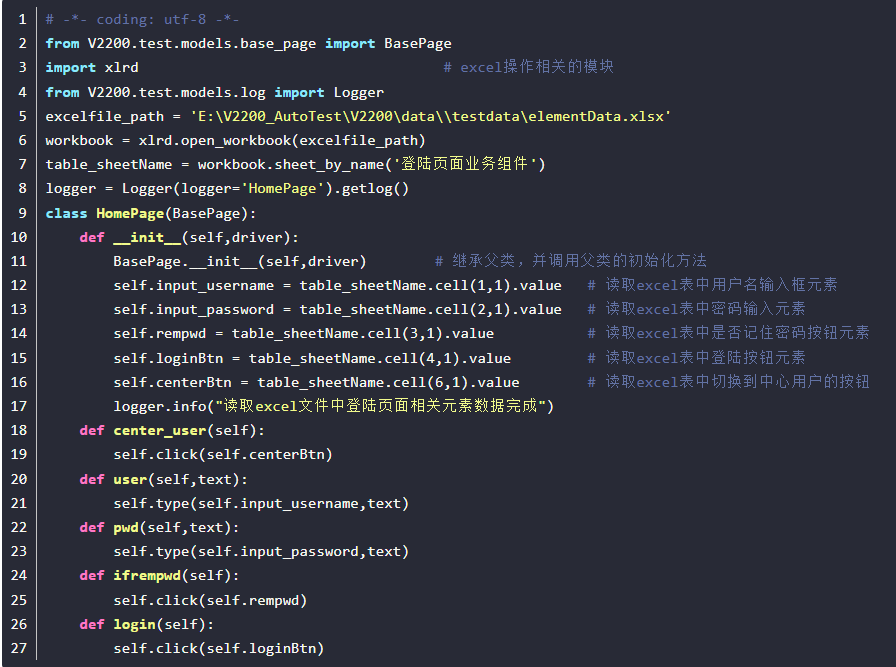
        在上面我们实现了页面基类的封装，下图为楼主公司的一个软件登陆页面，在page\_obj目录下新建home\_page.py实现这个登陆页面元素定位和元素操作方法的封装.

这里引入了第三方的**xlrd**模块，用于读取excel文件中的数据，当然还有**xlwt**模块用于向excel写数据，说白了这两个模块就是实现操作excel，上面代码只用到了xlrd模块，在编写上面登陆页面的封装前，咱们先将登陆页面定位的元素和元素属性写到对应的excel表中，这样做的好处就是**实现测试数据和测试脚本的分离**，**如果页面元素发生变化，那么我们就只需要修改excel中的元素属性而不需要修改代码**，在data/testdata目录下新建名称为elementData.xlsx的文件，excel编辑内容如下：

**后续各个页面的元素都是定位以及元素的数据都是可以写在不同的sheet中**，至于xlrd模块具体向excel中读数据的方法以及使用，这里也是不做介绍，自己百度学习练习下就知道了。

软件其他页面的封装也是类似，按照上面的思想来就OK了。

原文链接：<https://blog.csdn.net/fei347795790/article/details/88980656>

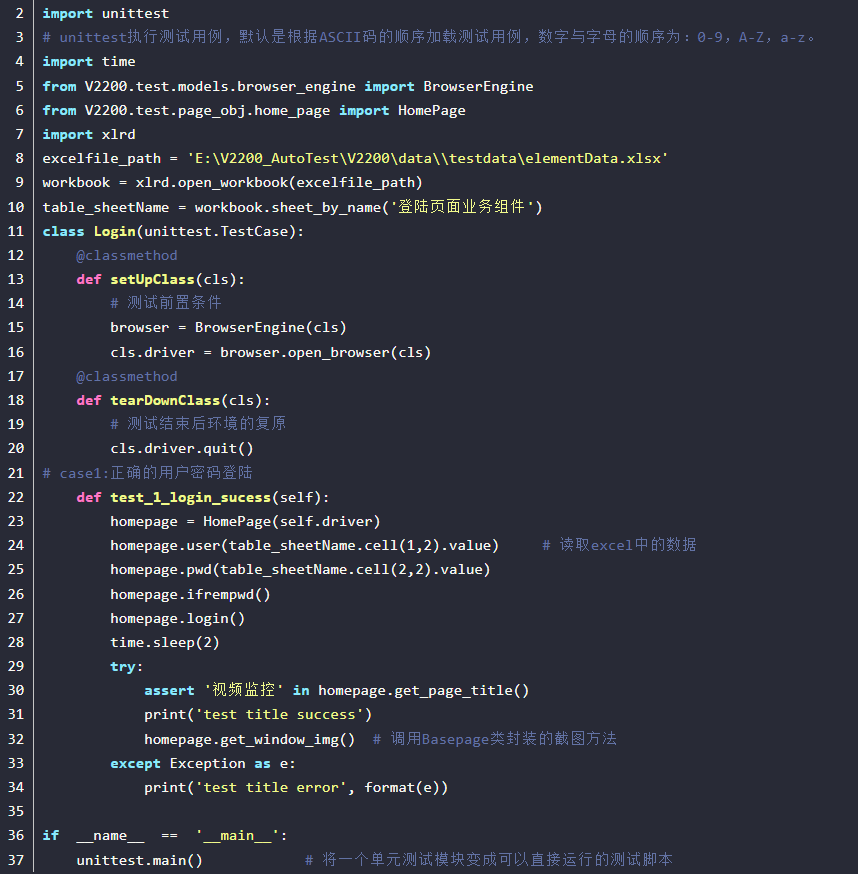


from V2200.test.models.base\_page import BasePage  
import xlrd # excel操作相关的模块  
from V2200.test.models.log import Logger  
excelfile\_path = 'E:\V2200\_AutoTest\V2200\data\\testdata\elementData.xlsx'  
workbook = xlrd.open\_workbook(excelfile\_path)  
table\_sheetName = workbook.sheet\_by\_name('登陆页面业务组件')  
logger = Logger(logger='HomePage').getlog()  
class HomePage(BasePage):  
 def \_\_init\_\_(self,driver):  
 BasePage.\_\_init\_\_(self,driver) # 继承父类，并调用父类的初始化方法  
 self.input\_username = table\_sheetName.cell(1,1).value # 读取excel表中用户名输入框元素  
 self.input\_password = table\_sheetName.cell(2,1).value # 读取excel表中密码输入元素  
 self.rempwd = table\_sheetName.cell(3,1).value # 读取excel表中是否记住密码按钮元素  
 self.loginBtn = table\_sheetName.cell(4,1).value # 读取excel表中登陆按钮元素  
 self.centerBtn = table\_sheetName.cell(6,1).value # 读取excel表中切换到中心用户的按钮  
 logger.info("读取excel文件中登陆页面相关元素数据完成")  
 def center\_user(self):  
 self.click(self.centerBtn)  
 def user(self,text):  
 self.type(self.input\_username,text)  
 def pwd(self,text):  
 self.type(self.input\_password,text)  
 def ifrempwd(self):  
 self.click(self.rempwd)  
 def login(self):  
 self.click(self.loginBtn)

八、登陆页面测试脚本的编写

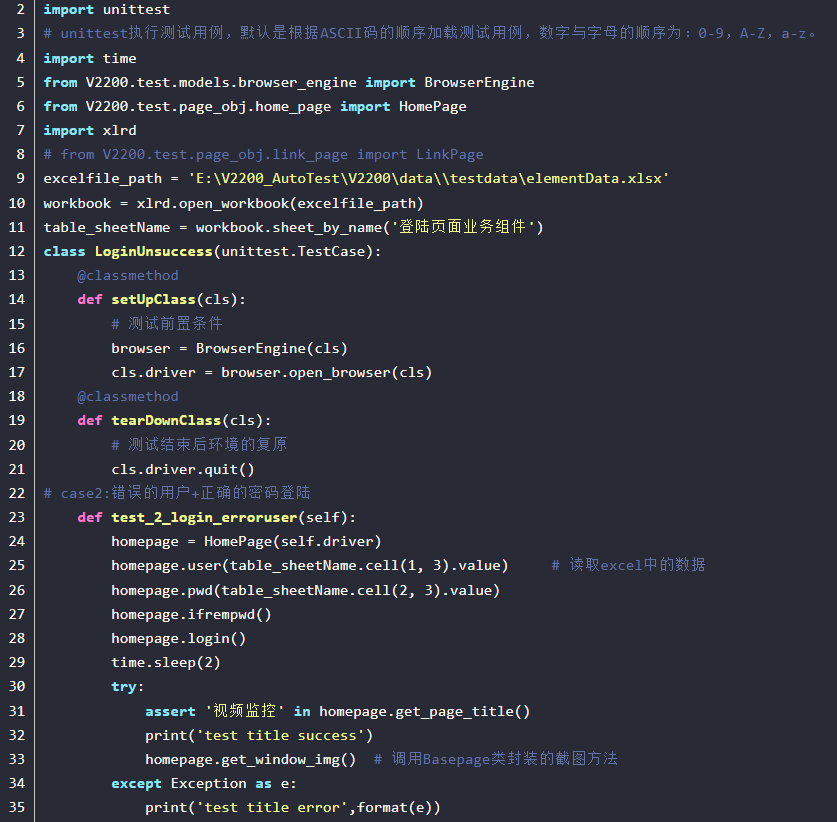
    通过上面封装的基类和登陆页面类，在unittest框架下开始编写具体的测试脚本。测试脚本在testcase下，如登陆功能的脚本我们写在：testcase/login\_page\_case目录下，其他页面的脚本写在对应的目录下。在testcase/login\_page\_case目录下创建：test\_login\_success.py和test\_login\_unsuccess.py分别表示登陆成功的脚本和登陆不成功的脚本。如下代码：

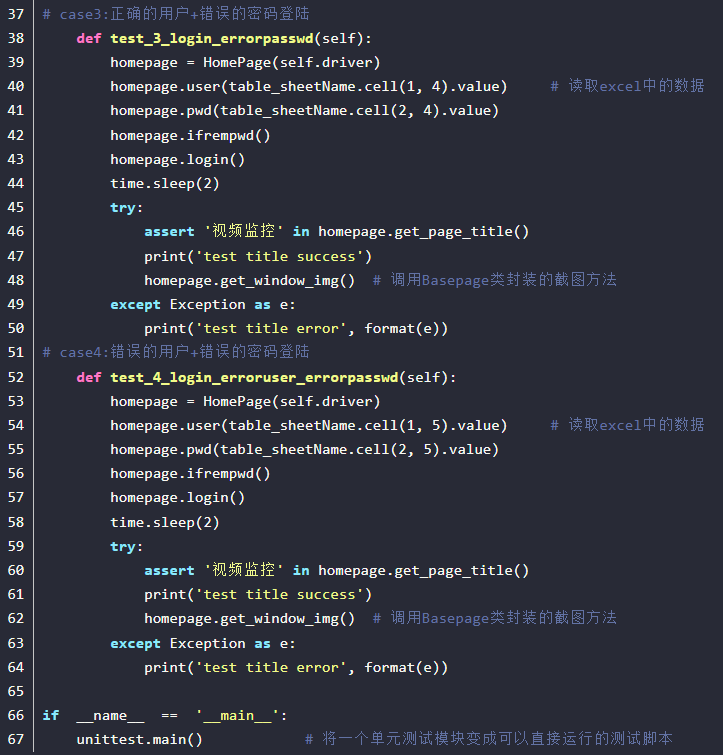
登陆成功代码



import unittest  
# unittest执行测试用例，默认是根据ASCII码的顺序加载测试用例，数字与字母的顺序为：0-9，A-Z，a-z。  
import time  
from V2200.test.models.browser\_engine import BrowserEngine  
from V2200.test.page\_obj.home\_page import HomePage  
import xlrd  
  
excelfile\_path = 'E:\V2200\_AutoTest\V2200\data\\testdata\elementData.xlsx'  
workbook = xlrd.open\_workbook(excelfile\_path)  
table\_sheetName = workbook.sheet\_by\_name('登陆页面业务组件')  
  
  
class Login(unittest.TestCase):  
 @classmethod  
 def setUpClass(cls):  
 # 测试前置条件  
 browser = BrowserEngine(cls)  
 cls.driver = browser.open\_browser(cls)  
  
 @classmethod  
 def tearDownClass(cls):  
 # 测试结束后环境的复原  
 cls.driver.quit()  
  
 # case1:正确的用户密码登陆  
 def test\_1\_login\_sucess(self):  
 homepage = HomePage(self.driver)  
 homepage.user(table\_sheetName.cell(1, 2).value) # 读取excel中的数据  
 homepage.pwd(table\_sheetName.cell(2, 2).value)  
 homepage.ifrempwd()  
 homepage.login()  
 time.sleep(2)  
 try:  
 assert '视频监控' in homepage.get\_page\_title()  
 print('test title success')  
 homepage.get\_window\_img() # 调用Basepage类封装的截图方法  
 except Exception as e:  
 print('test title error', format(e))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main() # 将一个单元测试模块变成可以直接运行的测试脚本

登陆不成功代码：



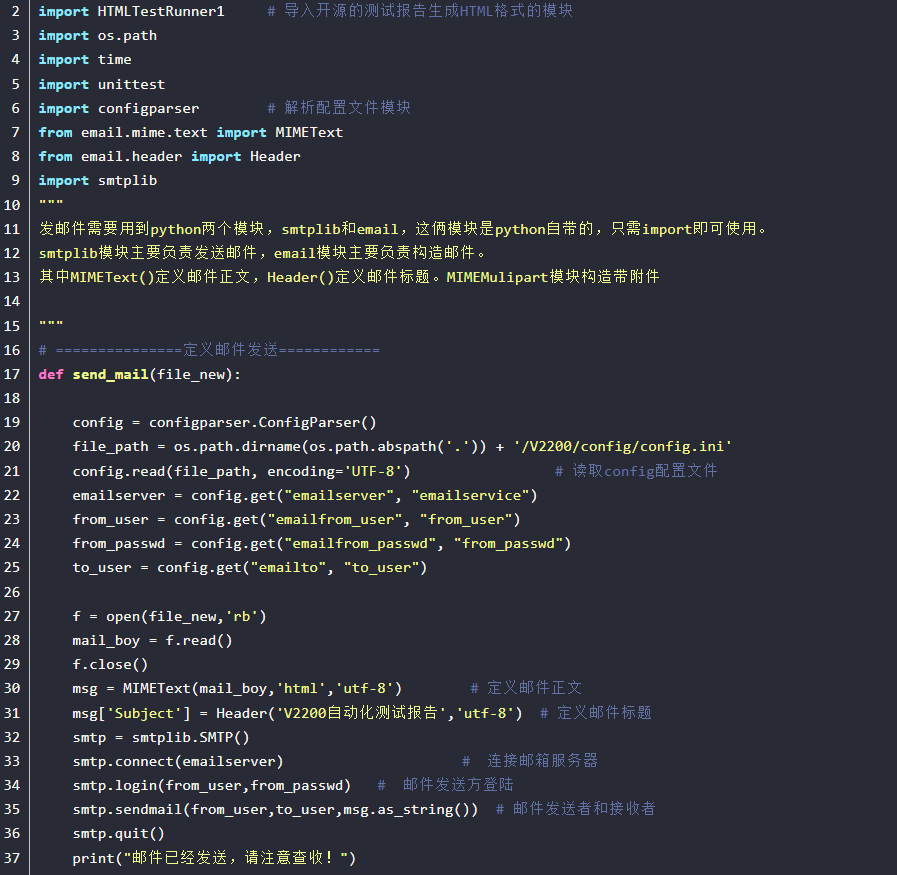


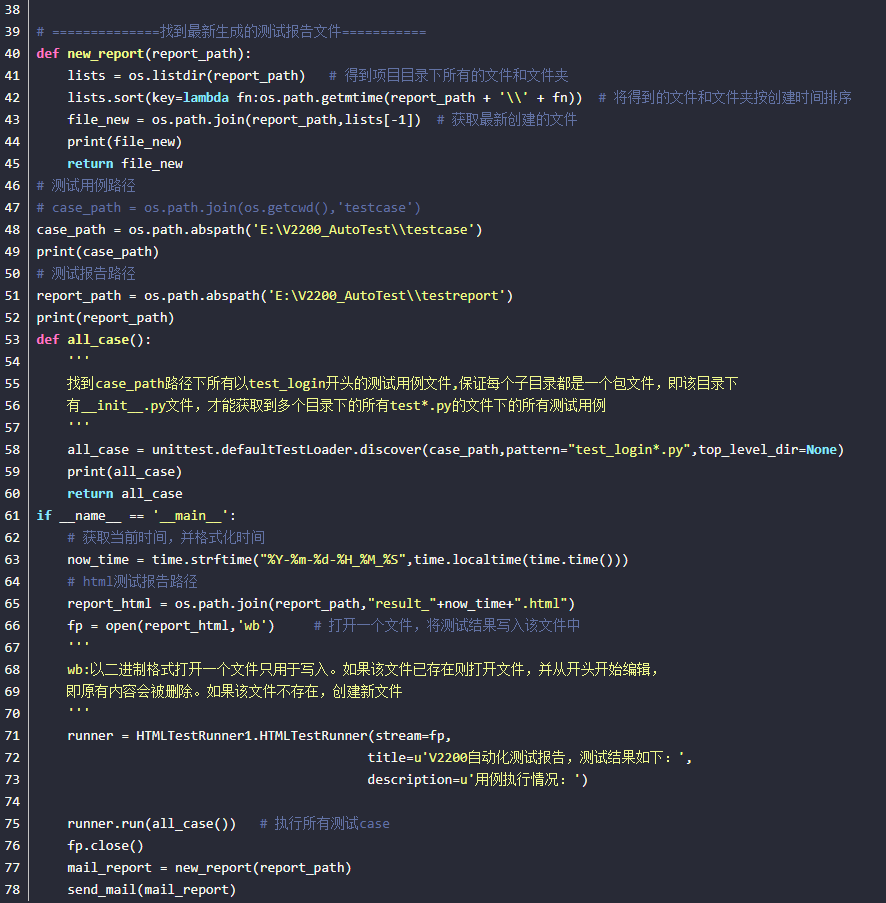
import unittest  
# unittest执行测试用例，默认是根据ASCII码的顺序加载测试用例，数字与字母的顺序为：0-9，A-Z，a-z。  
import time  
from V2200.test.models.browser\_engine import BrowserEngine  
from V2200.test.page\_obj.home\_page import HomePage  
import xlrd  
  
# from V2200.test.page\_obj.link\_page import LinkPage  
excelfile\_path = 'E:\V2200\_AutoTest\V2200\data\\testdata\elementData.xlsx'  
workbook = xlrd.open\_workbook(excelfile\_path)  
table\_sheetName = workbook.sheet\_by\_name('登陆页面业务组件')  
  
  
class LoginUnsuccess(unittest.TestCase):  
 @classmethod  
 def setUpClass(cls):  
 # 测试前置条件  
 browser = BrowserEngine(cls)  
 cls.driver = browser.open\_browser(cls)  
  
 @classmethod  
 def tearDownClass(cls):  
 # 测试结束后环境的复原  
 cls.driver.quit()  
  
 # case2:错误的用户+正确的密码登陆  
 def test\_2\_login\_erroruser(self):  
 homepage = HomePage(self.driver)  
 homepage.user(table\_sheetName.cell(1, 3).value) # 读取excel中的数据  
 homepage.pwd(table\_sheetName.cell(2, 3).value)  
 homepage.ifrempwd()  
 homepage.login()  
 time.sleep(2)  
 try:  
 assert '视频监控' in homepage.get\_page\_title()  
 print('test title success')  
 homepage.get\_window\_img() # 调用Basepage类封装的截图方法  
 except Exception as e:  
 print('test title error', format(e))  
  
 # case3:正确的用户+错误的密码登陆  
 def test\_3\_login\_errorpasswd(self):  
 homepage = HomePage(self.driver)  
 homepage.user(table\_sheetName.cell(1, 4).value) # 读取excel中的数据  
 homepage.pwd(table\_sheetName.cell(2, 4).value)  
 homepage.ifrempwd()  
 homepage.login()  
 time.sleep(2)  
 try:  
 assert '视频监控' in homepage.get\_page\_title()  
 print('test title success')  
 homepage.get\_window\_img() # 调用Basepage类封装的截图方法  
 except Exception as e:  
 print('test title error', format(e))  
  
 # case4:错误的用户+错误的密码登陆  
 def test\_4\_login\_erroruser\_errorpasswd(self):  
 homepage = HomePage(self.driver)  
 homepage.user(table\_sheetName.cell(1, 5).value) # 读取excel中的数据  
 homepage.pwd(table\_sheetName.cell(2, 5).value)  
 homepage.ifrempwd()  
 homepage.login()  
 time.sleep(2)  
 try:  
 assert '视频监控' in homepage.get\_page\_title()  
 print('test title success')  
 homepage.get\_window\_img() # 调用Basepage类封装的截图方法  
 except Exception as e:  
 print('test title error', format(e))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main() # 将一个单元测试模块变成可以直接运行的测试脚本

九**、测试执行控制模块**

      完成上面的测试脚本编写后，对于自动化测试还需要有一个**测试执行控制**的部分，用来**控制执行哪些用例集**，生成HTML可视化的**测试报告**，并实现测试报告**邮件发送**。

在runtest目录下新建**run\_all\_case.py**，编辑如下代码：





import HTMLTestRunner1 # 导入开源的测试报告生成HTML格式的模块  
import os.path  
import time  
import unittest  
import configparser # 解析配置文件模块  
from email.mime.text import MIMEText  
from email.header import Header  
import smtplib  
  
"""  
发邮件需要用到python两个模块，smtplib和email，这俩模块是python自带的，只需import即可使用。  
smtplib模块主要负责发送邮件，email模块主要负责构造邮件。  
其中MIMEText()定义邮件正文，Header()定义邮件标题。MIMEMulipart模块构造带附件  
  
"""  
  
  
# ===============定义邮件发送============  
def send\_mail(file\_new):  
 config = configparser.ConfigParser()  
 file\_path = os.path.dirname(os.path.abspath('.')) + '/V2200/config/config.ini'  
 config.read(file\_path, encoding='UTF-8') # 读取config配置文件  
 emailserver = config.get("emailserver", "emailservice")  
 from\_user = config.get("emailfrom\_user", "from\_user")  
 from\_passwd = config.get("emailfrom\_passwd", "from\_passwd")  
 to\_user = config.get("emailto", "to\_user")  
  
 f = open(file\_new, 'rb')  
 mail\_boy = f.read()  
 f.close()  
 msg = MIMEText(mail\_boy, 'html', 'utf-8') # 定义邮件正文  
 msg['Subject'] = Header('V2200自动化测试报告', 'utf-8') # 定义邮件标题  
 smtp = smtplib.SMTP()  
 smtp.connect(emailserver) # 连接邮箱服务器  
 smtp.login(from\_user, from\_passwd) # 邮件发送方登陆  
 smtp.sendmail(from\_user, to\_user, msg.as\_string()) # 邮件发送者和接收者  
 smtp.quit()  
 print("邮件已经发送，请注意查收！")  
  
  
# ==============找到最新生成的测试报告文件===========  
def new\_report(report\_path):  
 lists = os.listdir(report\_path) # 得到项目目录下所有的文件和文件夹  
 lists.sort(key=lambda fn: os.path.getmtime(report\_path + '\\' + fn)) # 将得到的文件和文件夹按创建时间排序  
 file\_new = os.path.join(report\_path, lists[-1]) # 获取最新创建的文件  
 print(file\_new)  
 return file\_new  
  
  
# 测试用例路径  
# case\_path = os.path.join(os.getcwd(),'testcase')  
case\_path = os.path.abspath('E:\V2200\_AutoTest\\testcase')  
print(case\_path)  
# 测试报告路径  
report\_path = os.path.abspath('E:\V2200\_AutoTest\\testreport')  
print(report\_path)  
  
  
def all\_case():  
 *'''  
 找到case\_path路径下所有以test\_login开头的测试用例文件,保证每个子目录都是一个包文件，即该目录下  
 有\_\_init\_\_.py文件，才能获取到多个目录下的所有test\*.py的文件下的所有测试用例  
 '''* all\_case = unittest.defaultTestLoader.discover(case\_path, pattern="test\_login\*.py", top\_level\_dir=None)  
 print(all\_case)  
 return all\_case  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # 获取当前时间，并格式化时间  
 now\_time = time.strftime("%Y-%m-%d-%H\_%M\_%S", time.localtime(time.time()))  
 # html测试报告路径  
 report\_html = os.path.join(report\_path, "result\_" + now\_time + ".html")  
 fp = open(report\_html, 'wb') # 打开一个文件，将测试结果写入该文件中  
 '''  
 wb:以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，  
 即原有内容会被删除。如果该文件不存在，创建新文件  
 '''  
 runner = HTMLTestRunner1.HTMLTestRunner(stream=fp,  
 title=u'V2200自动化测试报告，测试结果如下：',  
 description=u'用例执行情况：')  
  
 runner.run(all\_case()) # 执行所有测试case  
 fp.close()  
 mail\_report = new\_report(report\_path)  
 send\_mail(mail\_report)

该代码，编写了怎么获取需要执行的测试用例脚本和引入生成可视化测试报告的模块和发送邮件模块等，对于这几个模块自己多学习下就能掌握。

十、测试执行效果

    通过上面这些类的封装以及测试脚本的编写，算是完成了我们自动化测试框架的基本具备的东西。忙活了这么久，是时候来看看咱们的效果了。PyCharm中运行run\_all\_case.py，运行完成后的效果如下：

　咱们再看看log/logs路径下生成的日志，就如下图这样：

 同时执行完成后，在testreport目录下会生成HTML格式的可视化测试报告文件，用浏览器打开效果如下：

　这报告是不是很酷炫啊，O(∩\_∩)O哈哈~

     还有测试报告发送邮件给到指定的邮箱哦，如果你的自动化测试执行完了，可以把该自动化测试报告自动邮件发给你的leader，领导看到了是不是对你另眼相看？楼主上面的代码设置发送的是楼主公司内网使用的邮箱：foxmail，效果如下：

十一、整个自动化测试框架的总结和反思

        其实到第十节的介绍，楼主算是成功的做出了一个轻量级的测试框架，but，回过头来继续思考，还是有诸多需要优化和待下一步解决的问题：

1.页面基类还需要补充更多的公共函数或者方法；

2.可视化HTML**测试报告内容还不够丰富**，没有完善的测试执行失败的用例的详细描述和测试截图附件显示；

3.整个框架的部分逻辑还需要优化和改进；

4.待解决的问题：**没实现测试脚本的持续集成和定时执行，现在想到的是配合jenkins**持续集成来达到自动构建测试执行任务；

5.想独立开发一个web测试平台，现在想到的是学习Django的web框架来开发一个自动化测试平台；