

Python2 Selenium3 自动化测试

声明：

1、本书著作权归作者所有。未经作者书面或邮件许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

2、本书电子版以免费形式提供，任何个人、组织等不得利用本书获取任何形式的收入（包括但不限于现金、实物、虚拟货币等等）

联系作者：

作者：苦叶子

Email: lymking@foxmail.com

公众号：DeepTest 或 开源优测

公众号二维码



大家扫一扫关注公众号持续关注其他内容更新，谢谢

公众号其他内容（如何查看：关注公众号，从下方菜单开始）：



目录

一、源码分析.....	9
1.1 python Selenium3 架构说明	9
1.2 python Selenium3 源码分析概要	11
1.3 python Selenium3 源码 - 核心 package : remote 分析	12
1.4 python Selenium3 源码 - 核心 package : support.....	14
1.5 python Selenium3 源码 - webdriver 常用方法	16
1.6 python Selenium3 源码 - 鼠标键盘操作.....	20
二、基础入门.....	23
2.1 python Selenium3 开发环境搭建	23
2.1.1 环境搭建.....	23
2.1.2 安装包.....	23
2.1.3 python 安装过程	24
2.1.4 pyCharm 安装	27
2.1.5 第一个 python selenium 代码	29
2.2 Python Selenium Webdriver 驱动准备.....	30
2.2.1 前言.....	30
2.2.2 python 安装	30
2.2.3 升级最新的 pip.....	30
2.2.4 安装 webdriver	31
2.2.5 配置各种浏览器的驱动.....	31
2.3 webdriver 介绍&Selenium RC 的比较.....	32
2.3.1 什么是 webdriver ?	32

2.3.2 Selenium RC 和 webdriver 的区别.....	33
2.3.3 总结.....	36
2.4 Selenium 2.0 与 Selenium 3.0 介绍.....	37
2.4.1 什么是 Selenium	37
2.4.2 什么是 Selenium 2.0	37
2.4.3 什么是 Selenium 3.0	38
2.4.4 总结.....	38
2.5 创建你的第一个 webdriver python 代码	39
2.5.1 前言.....	39
2.5.2 webdriver python 代码.....	39
2.5.3 代码解释.....	42
2.5.4 运行代码.....	45
2.5.5 总结.....	46
2.6 Python 多线程 Selenium 跨浏览器测试.....	47
2.6.1 前言.....	47
2.6.2 什么是跨浏览器测试.....	47
2.6.3 为什么需要跨浏览器测试.....	48
2.6.4 如何执行跨浏览器测试.....	48
2.6.5 总结.....	51
2.7 在 Selenium Webdriver 中使用 XPath Contains、Sibling 函数定位.....	52
2.7.1 前言.....	52
2.7.2 Contains 函数.....	52

2.7.3 sibling 函数.....	53
2.7.4 xpath 常用函数.....	55
2.7.5 总结.....	56
2.8 Selenium Webdriver Desired Capabilities.....	56
2.8.1 前言.....	56
2.8.2 源码分析.....	57
2.8.3 DesiredCapabilities 示例.....	64
2.8.4 总结.....	65
2.9 基于 Excel 参数化你的 Selenium3 测试代码.....	65
2.9.1 前言.....	65
2.9.2 环境安装.....	65
2.9.3 xlrd 基本用法.....	65
2.9.4 代码示例.....	67
2.9.5 总结.....	71
2.10 Python Selenium 设计模式-POM.....	72
2.10.1 前言.....	72
2.10.2 为什么要用 POM.....	72
2.10.3 POM 是什么.....	73
2.10.4 POM 的优势.....	74
2.10.5 POM 实现示例.....	74
2.10.6 总结.....	81
三、高级示例.....	82

3.1 python Selenium3 示例 - 启动不同浏览器.....	82
3.1.1 启动 firefox 浏览器.....	82
3.1.2 启动 google 浏览器.....	83
3.1.3 启动 IE 浏览器	84
3.2 python Selenium3 示例 - Page Object Model.....	85
3.2.1 前言.....	85
3.2.2 Page 模式	86
3.2.3 结束语.....	89
3.3 python Selenium3 示例 - 利用 excel 实现参数化.....	89
3.3.1 前言.....	89
3.3.2 环境安装.....	89
3.3.3 一个简单的读写示例.....	90
3.3.4 结束语.....	93
3.4 python Selenium3 示例 - 日志管理	94
3.4.1 前言.....	94
3.4.2 简单日志.....	95
3.4.3 日志格式和级别控制.....	96
3.4.4 日志输入定向.....	98
3.4.5 日志配置.....	100
3.4.6 结束语.....	103
3.5 python Selenium3 示例 - 同步机制	103
3.5.1 前言.....	103

3.5.2 强制等待.....	104
3.5.3 隐性等待.....	105
3.5.4 显性等待.....	106
3.5.5 WebDriverWait 类.....	108
3.6 python Selenium3 示例 - SSL 处理.....	110
3.6.1 前言.....	110
3.6.2 面临的问题.....	111
3.6.3 结束语.....	113
3.7 python Selenium3 示例 - email 发送.....	114
3.7.1 前言.....	114
3.7.2 纯文本邮件.....	114
3.7.3 HTML 形式的邮件	115
3.7.4 带附件的邮件.....	116
3.7.5 群发邮件.....	118
3.7.6 综合示例.....	120
3.7.7 结束语.....	122
3.8 python Selenium3 示例 - 生成 HTMLTestRunner 测试报告	122
3.8.1 前言.....	122
3.8.2 什么是 HTMLTestRunner	122
3.8.3 HTMLTestRunner 安装	122
3.8.4 应用示例.....	122
3.8.5 报告效果.....	125

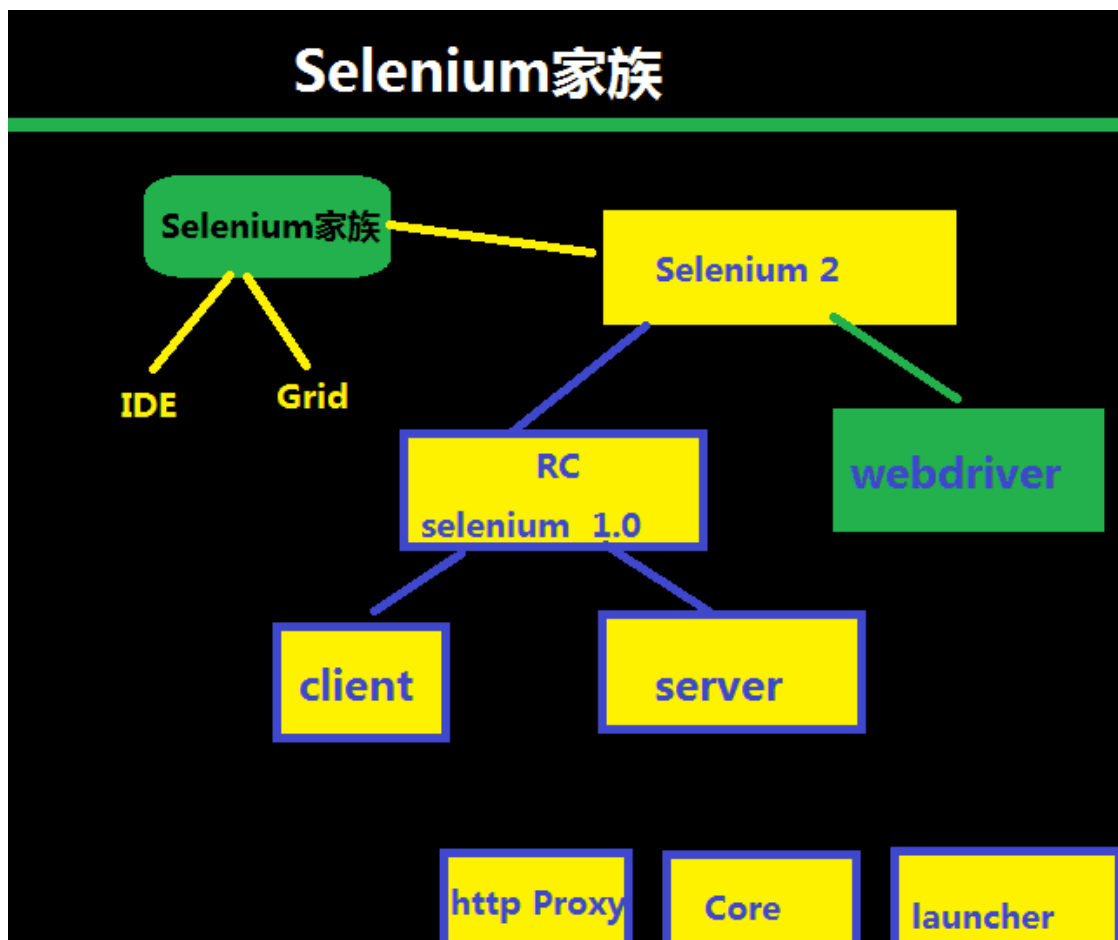
四、单元测试.....	126
4.1 基于 unittest 集成你的 Selenium3 测试.....	126
4.1.1 前言.....	126
4.1.2 测试用例.....	126
4.1.3 简单示例.....	126
4.1.4 关键代码说明.....	129
4.1.5 主入口说.....	129
4.1.6 代码组织说明.....	130
4.1.7 总结.....	130
4.2 python unittest 使用基本过程.....	130
4.2.1 前言.....	130
4.2.2 unittest 使用过程.....	131
4.4.3 unittest 命令	134
4.2.4 总结.....	135
4.3 python unittest 之关键 API 说明及示例	135
4.3.1 前言.....	135
4.3.2 TestCase 类 API.....	135
4.3.3 TestSuite 类 API.....	136
4.3.4 TestSuite 应用示例.....	138
4.3.5 TestLoader 类 API.....	140
4.3.6 TestResult 类	143
4.3.7 总结.....	144

4.4 python unittest 之断言及示例.....	145
4.4.1 前言.....	145
4.4.2 基本断言方法.....	145
4.4.3 比较断言.....	148
4.4.4 复杂断言.....	151
4.4.5 总结.....	153
4.5 python unittest 之异常测试.....	153
4.5.1 前言.....	153
4.5.2 assertRaises(exception, callable, <i>args</i> , * <i>kwds</i>).....	153
4.5.3 assertRaisesRegexp(exception, regexp, callable, <i>args</i> , * <i>kwds</i>).....	156
4.5.4 总结.....	159
4.6 python unittest 之加载及跳过测试方法和示例.....	159
4.6.1 前言.....	159
4.6.2 TestLoader 加载用例.....	159
4.6.3 unittest.skip 跳过测试方法.....	162
4.6.4 总结.....	166

一、源码分析

1.1 python Selenium3 架构说明

selenium 技术族谱



selenium 技术族谱

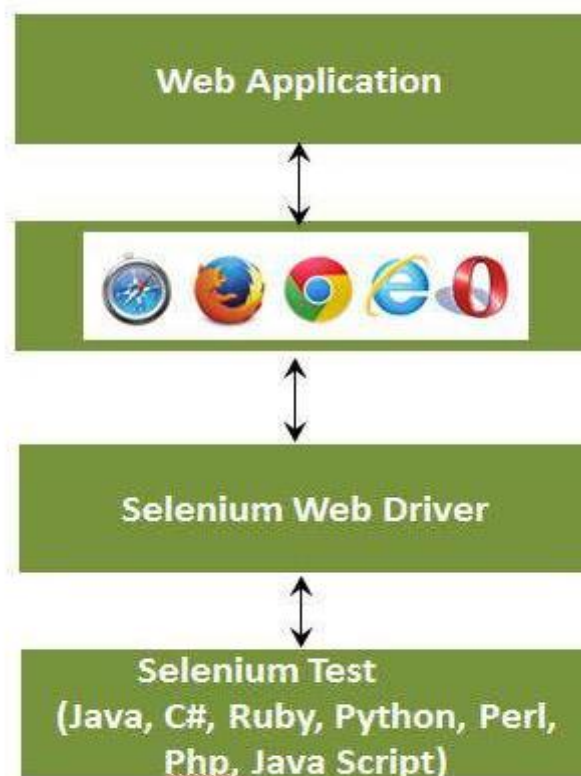
selenium 家族图说明：

- 1、IDE 主要用来学习和对 selenium 技术有个初步的了解用，不适合日常的自动化测试
- 2、grid 可以理解为 selenium grid，用于并行部署、测试、执行

3、selenium 2 包含了 1.0 和 2.0，一般情况我们说 Selenium3 是指 webdriver。目前 webdriver 已被纳入 w3c 标准，将成为浏览器端自动化测试的标准组件

webdriver 架构图

用一张最简洁的流程图形来标识 webdriver 的架构，如下



webdriver

webdriver 流程架构图说明

从图来看，webdriver 可以看做有四层，分别为：

1、业务脚本，支持 python、java、ruby、perl、php、js 等语言

2、selenium web driver 层，从前面几张的源码分析来看，支持 ie、google、firefox 等等各种常见的浏览器（默认支持 firefox，google、ie 需要下载对应的驱动）

3、浏览器层，几乎括揽了所有的浏览器

4、业务系统，即我们的测试对象

1.2 python Selenium3 源码分析概要

目录结构概要说明

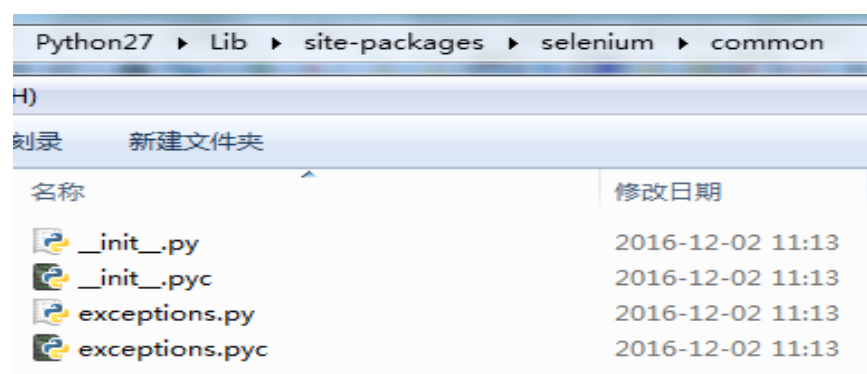
完整的路径是：C:\Python27\Lib\site-packages\selenium\

(注，笔者的 python 安装目录为 C:\python27)



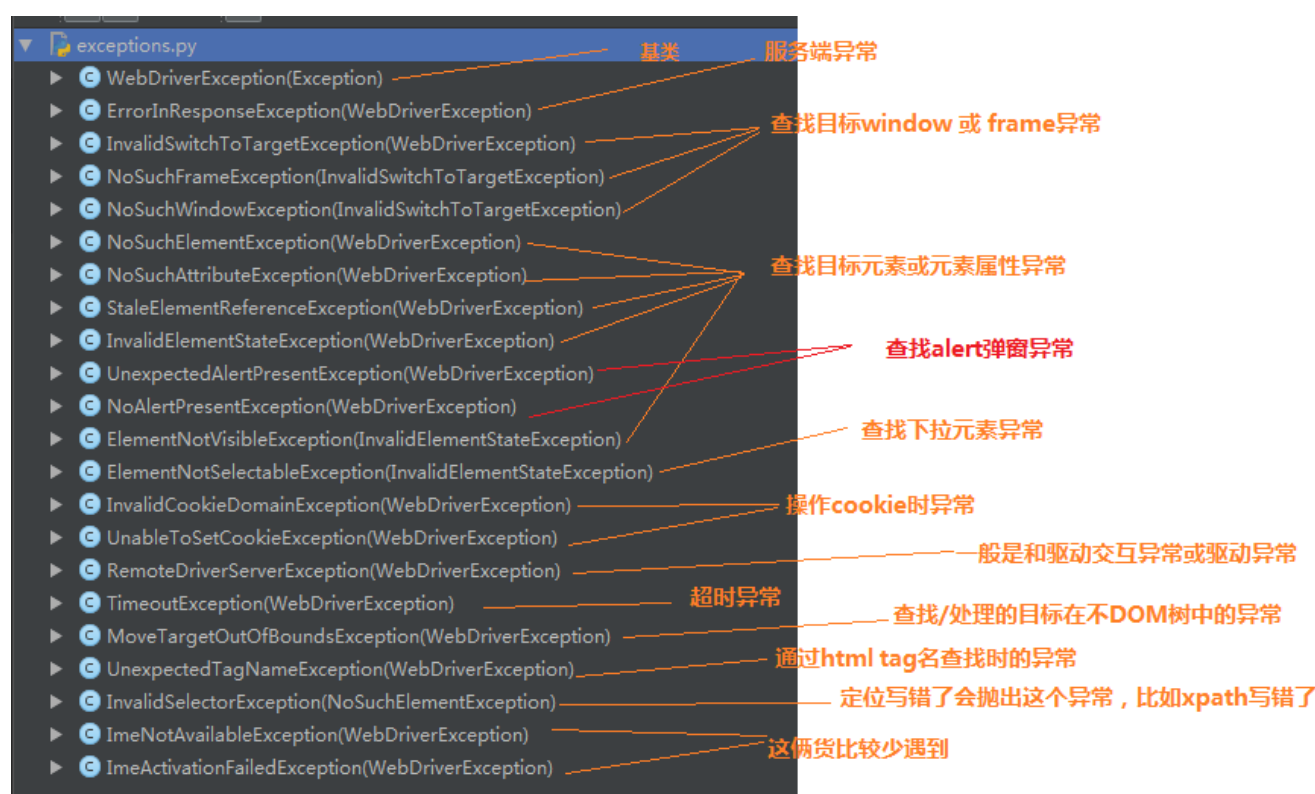
总体目录说明图

common/exceptions 模块分析



webdriver 异常定义

在 exceptions.py 中定义了 webdriver 各种异常处理类，如下图所示：



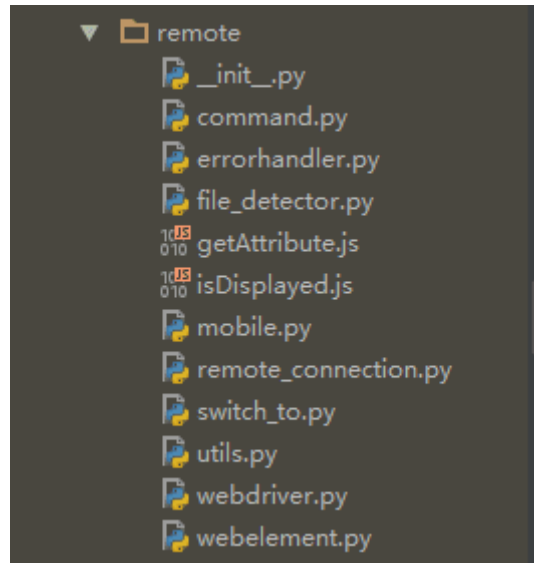
异常处理模块说明

本章就先暂时对总体目录和 common/exceptions 模块进行大概是分析说，后续逐步对其他模块进行一一说明。

1.3 python Selenium3 源码 - 核心 package：remote 分析

本章主要对 Selenium3 核心 package remote 进行说明，remote 主要包含了以下几个模块：

remote package 本地完整的路径为：C:\Python27\Lib\site-packages\selenium\webdriver\remote



remote 目录所有模块

remote 目录下所有模块说明

__init__.py 你懂的

command.py 定义了 webdriver 标准的命令常量，这些常量本身是没有意义的，但其标识了 webdriver 远程通信协议(webdriver's remote wire protocols)

errorhandler.py 定义和实现了 webdriver 错误编码和错误处理类

file_detector.py 定义和实现了文件侦测类

mobile.py 定义和实现了移动端的连接和 context 指令机制

remote_connection.py 扩展了 url_request.Request 实现，和 webdriver

remote server 通信交互就这个模块实现了，有兴趣的可以深入研究

switch_to.py 定义和实现了切换至 alert、window、frame、active 等系列动作

utils.py 定义和实现了一些辅助功能，例如格式化为 json、加载 json，压缩文件等等

webdriver.py 这是应用核心了，我们日常自动化测试调用的方法大都来源这个模块，**必须掌握**该模块提供的方法，该模块提供的方法有：浏览器操作类（例如 cookie 管理、刷 refresh），元素定位类（以 find_element_ 开头的各种定位函数）等等，这里就不一一列举，后续针对 webdriver 常用的方法，提供一个清单出来以供参考。

webelement.py web 元素操作类，定义了 web 元素操作的各种方法，必须掌握

重点强调

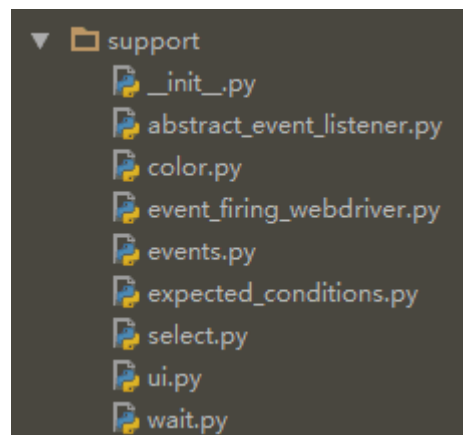
想要使用 webdriver 做好自动化测试，必须对 webdriver.py 和 webelement.py 这两个模块的源码进行深入研习和掌握，这样才能从知道在什么时候调用什么方法，每个方法的使用方式等等。

1.4 python Selenium3 源码 - 核心 package：support

目录说明

目录全路径: C:\Python27\Lib\site-packages\selenium\webdriver\support

注：笔者 python 安装在 C:\Python27
support package 下模块如下图所示：



support 目录

模块说明

`__init__.py` 你懂的

`abstract_event_listener.py` 事件监听器基类，定义了各种事件监听基础方法，该模块为做实现

`color.py` 定义和现实颜色转换支持类、方法和常量

`event_firing_webdriver.py` 定义和实现 `webdriver` 和 `WebElement` 事件触发类

`events.py` 统一对外导出 `AbstractEventListener` 和 `EventFiringWebDriver`

expected_conditions.py 针对页面 title、元素操作/文本/可见等、窗口打开等定义和实现了一系列的断言验证方法

select.py 定义和实现了对 select 标准下拉框元素的系列操作方法

ui.py 统一对外导出了 Select 和 WebDriverWait

wait.py 定义和实现了设置 webdriver 超时机制

重点强调

该 package 主要定义了一系列的辅助功能，日常自动化测试主要应用 select 和 wait 模块中的基本方法来设置 webdriver 的超时设置和对 select 标准的 html 元素下拉框处理（注，select 定义的方法不适用自定义方式实现的下拉框，所以在自动化测试过程中需要对具体的下拉框进行 html 的源码分析）

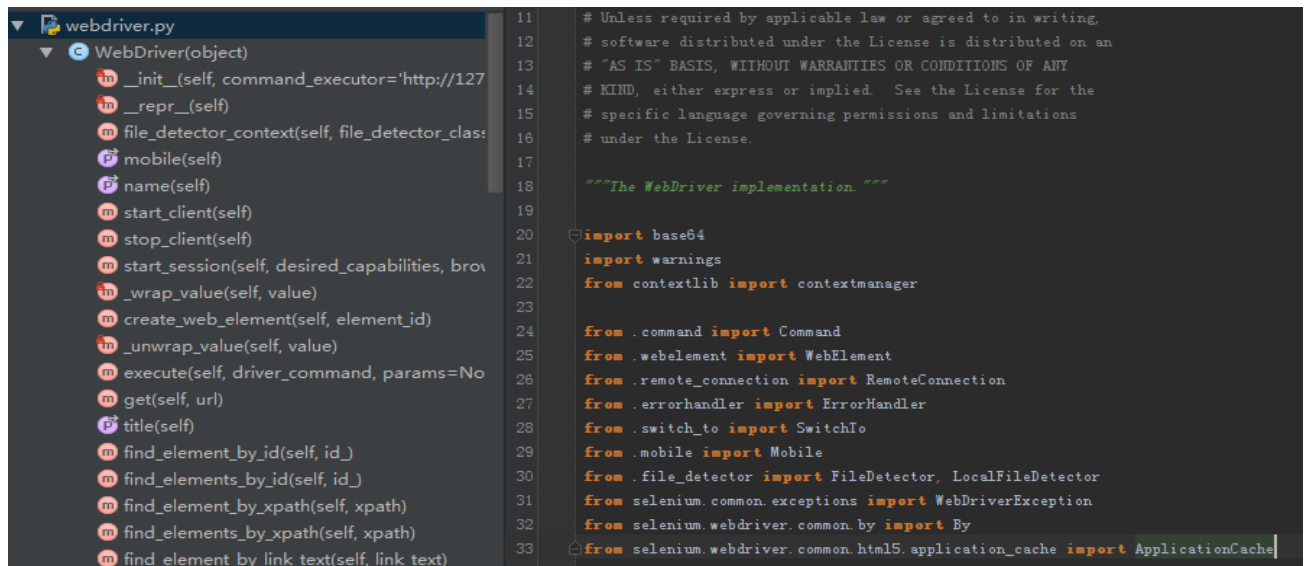
1.5 python Selenium3 源码 - webdriver 常用方法

完整路径:C:\Python27\Lib\site-

packages\selenium\webdriver\remote\webdriver.py

注：笔者 python 安装在 C:\Python27

webdriver.py 初步印象



webdriver.py

常用方法分类

一、全局操作类

start_session 使用指定的 desired capabilities 创建一个会话(session)

start_client新建一个 webdriver 会话 session 前调用，可以自定义初始化的动作

stop_client停止 webdriver 会话 session 后调用，可以自定义一些清理动作

create_web_element 创建一个 html 元素

get 在当前浏览器会话页打开指定的 url 网页

close 关闭当前浏览器窗口

quit 关闭 webdriver 会话，并把所有与该会话关联的浏览器窗口一起关闭

forward/back 浏览器历史浏览上一次/下一次操作

refresh 刷新浏览器

set_script_timeout设置脚本执行超时时间

set_page_load_timeout 设置页面加载超时时间

get_screenshot_as_file 截图并保存为文件

get_screenshot_as_png 截图并保存为 png 格式文件

get_screenshot_as_base64 截图成 base64 串

get_cookies/delete_all_cookies 获取/删除所有 cookie

get_cookie/delete_cookie 获取/删除指定的 cookie

二、元素定位类

find_element_by_id/find_elements_by_id 通过 id 查找一个或多个元素

find_element_by_xpath/find_elements_by_xpath 通过 xpath 查找一个或多个元素

find_element_by_link_text/find_elements_by_link_text 通过链接文本查找一个或多个元素（全匹配模式）

find_element_by_partial_link_text/find_elements_by_partial_link_text 通过部分链接文本查找一个或多个元素（部分匹配模式）

find_element_by_name/find_elements_by_name 通过元素名查找一个或多个元素

find_element_by_tag_name/find_elements_by_tag_name 通过 html 标记名查找一个或多个元素

find_element_by_class_name/find_elements_by_class_name 通过 class name 查找一个或多个元素

find_element_by_css_selector/find_elements_by_css_selector 通过

css 选择器查找一个或多个元素

三、js 执行类

execute_script 同步模式执行 js (等待 js 的执行完成, 才进入下一步)

execute_async_script 异步模式执行 js (不需要等待 js 的执行结果, 直接进行下一步)

四、窗口、元素操作类

current_window_handle 获取当前窗口的 handle

handlewindow_handles 获取当前 webdriver session 所有窗口的

maximize_window 最大化窗口

set_window_size 设置窗口大小

get_window_size 获取窗口大小

set_window_position 设置窗口位置

get_window_position 获取窗口位置

switch_to_window 切换至指定窗口

switch_to_default_content 切换至默认的 frame

switch_to_frame 切换至指定的 frame

switch_to_alert 切换至标准的 alert 窗口

switch_to_active_element 切换至当前激活的元素

五、基本信息读取类

desired_capabilities 获取当前会话的 desired_capabilities 信息

`current_url` 获取当前页面的 url

`page_source` 获取当前页面的源码

`title` 获取当前页面的标题

注：这里不会把所有的方法都列举出来，只会将常用的列举，有兴趣的朋友可以深入看看代码，更有利于掌握 webdriver。

1.6 python Selenium3 源码 - 鼠标键盘操作

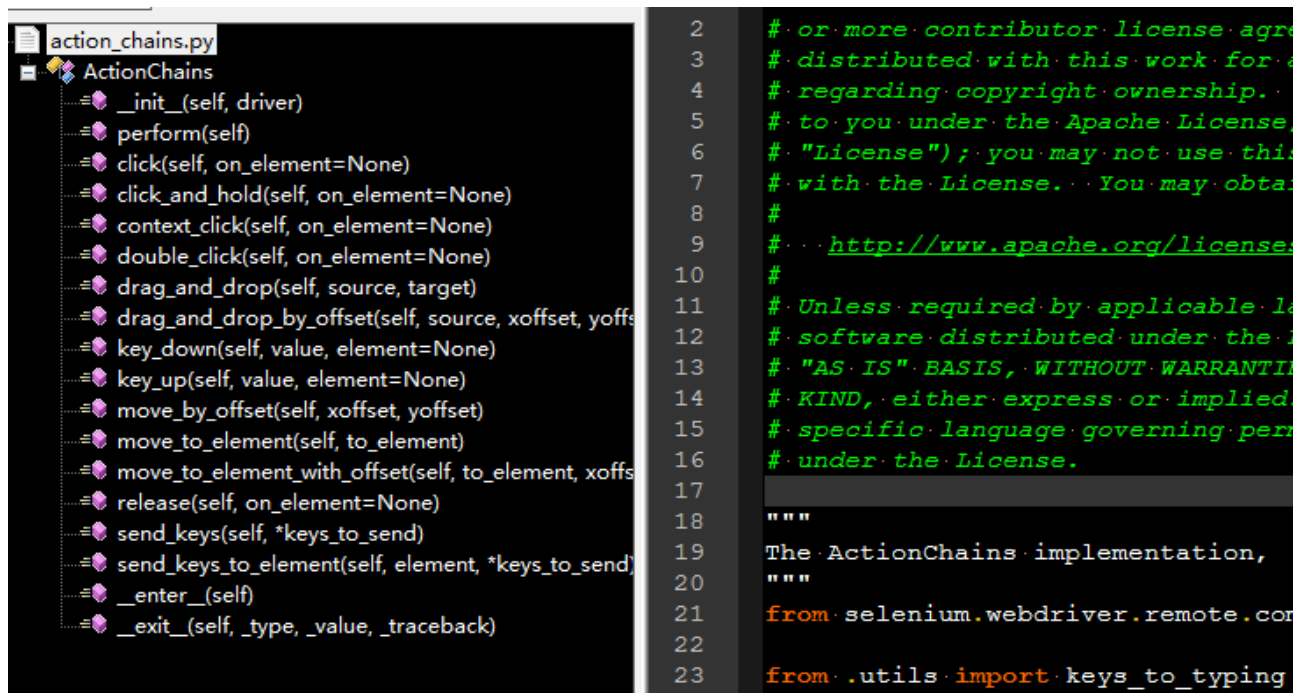
完整路径

C:\Python27\Lib\site-

packages\selenium\webdriver\common\action_chains.py

注：笔者 python 安装在 C:\Python27

初步印象



action_chains 【鼠标键盘动作】

方法说明

`__init__` 创建一个 actionChains , 需要传入一个实例化了的 webdriver 对象

`Click` 单击元素指定元素

`click_and_hold`在指定元素按下鼠标和 `release` 配套使用

`context_click` 右击

`double_click` 双击

`drag_and_drop` 拖曳动作

`drag_and_drop_by_offset` 从当前元素按下鼠标左键往指定的(x , y)坐标移动 ,

然后释放鼠标

`key_down` 按下指定的键盘按键

`key_up` 释放当前键盘按键

`move_by_offset` 将鼠标从当前位置移动至指定的 (x , y) 坐标

`move_to_element` 将鼠标移动至指定元素的中间位置

`move_to_element_with_offset` 将鼠标移动到指定的元素，其偏移 (x , y)

是相对该元素的 左上角的偏移

`release` 在当前元素释放鼠标，和 `click_and_hold` 配套使用

`send_keys` 在当前焦点的元素中模拟键盘输入

`send_keys_to_element` 给指定元素模拟键盘输入

`perform` 运行一组鼠标键盘动作

注意事项

本模块定义了所有的鼠标键盘动作控制，所有的键盘按键定义在 `keys.py` 中，有兴趣的朋友可以直接阅读源码进行了解。

二、基础入门

2.1 python Selenium3 开发环境搭建

2.1.1 环境搭建

基于 python 和 Selenium3 做自动化测试，你必须会搭建基本的开发环境，掌握 python 基本的语法和一个 IDE 来进行开发，这里通过详细的讲解，介绍怎么搭建 python 和 Selenium3 开发环境，并提供一个基本入门的代码，后续逐步提供系列实践文章。

2.1.2 安装包

- **python**

笔者使用 python2.7.13，请根据机器是 64 位还是 32 位来选择对应的 python 版本。

32 位下载：

<https://www.python.org/ftp/python/2.7.13/python-2.7.13.msi>

64 位下载：

<https://www.python.org/ftp/python/2.7.13/python-2.7.13.amd64.msi>

- **开发工具**

笔者使用 pyCharm 开源版本

<https://download.jetbrains.com/python/pycharm-community-2016.3.2.exe>

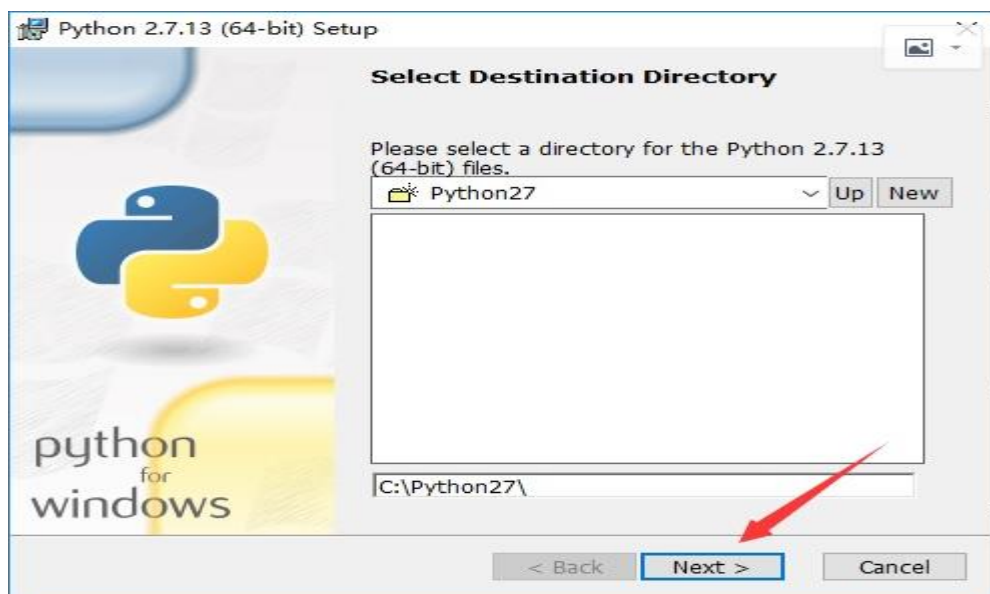
2.1.3 python 安装过程

双击已下载的 python 安装包，进入第一步



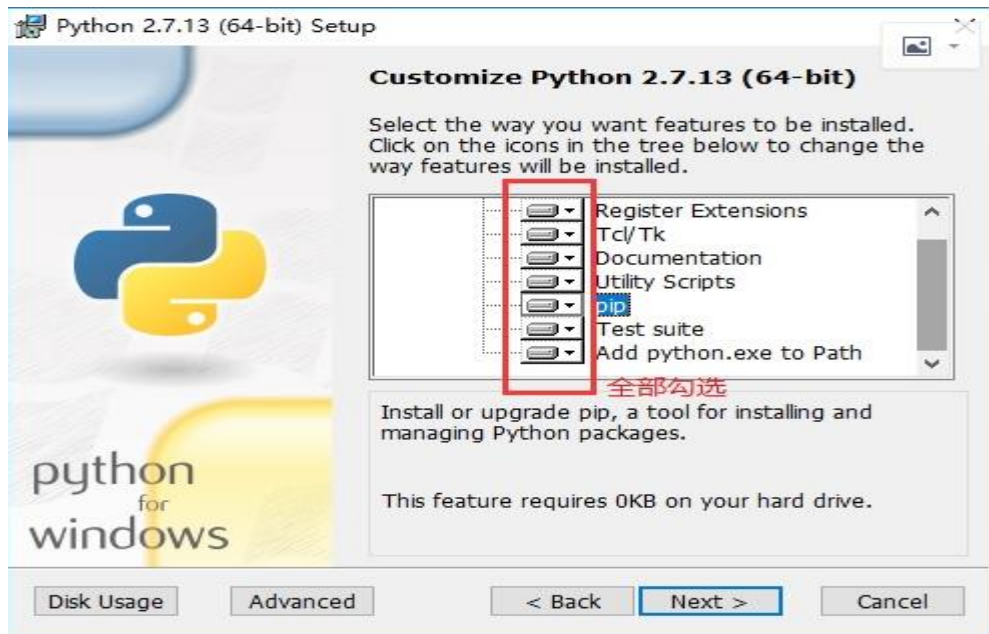
步骤一.png

设置安装目录，这里默认即可



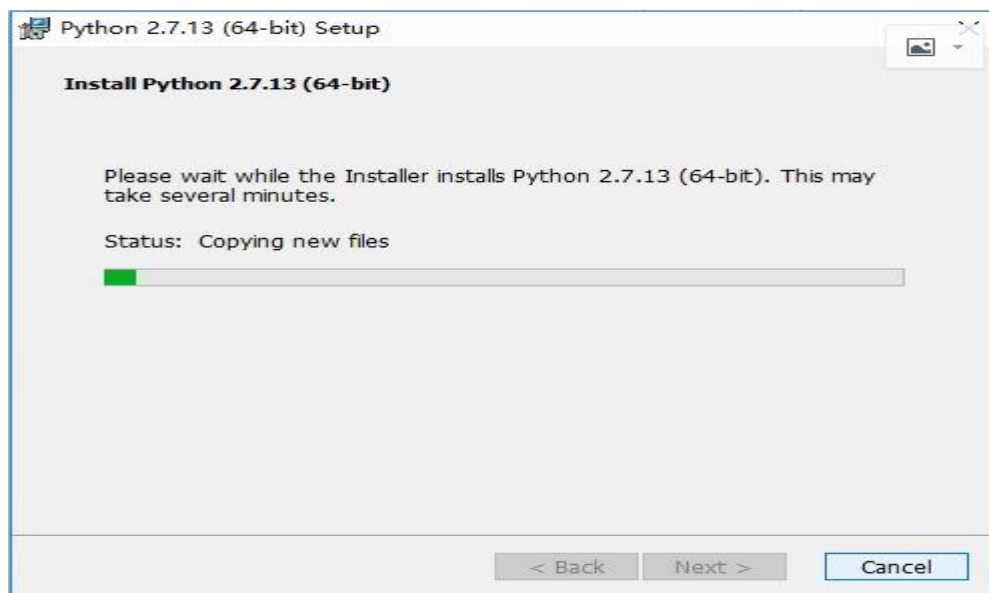
步骤二.png

设置要安装的可选包，全选是最佳的方式



步骤三.png

安装过程如下，坐等即可



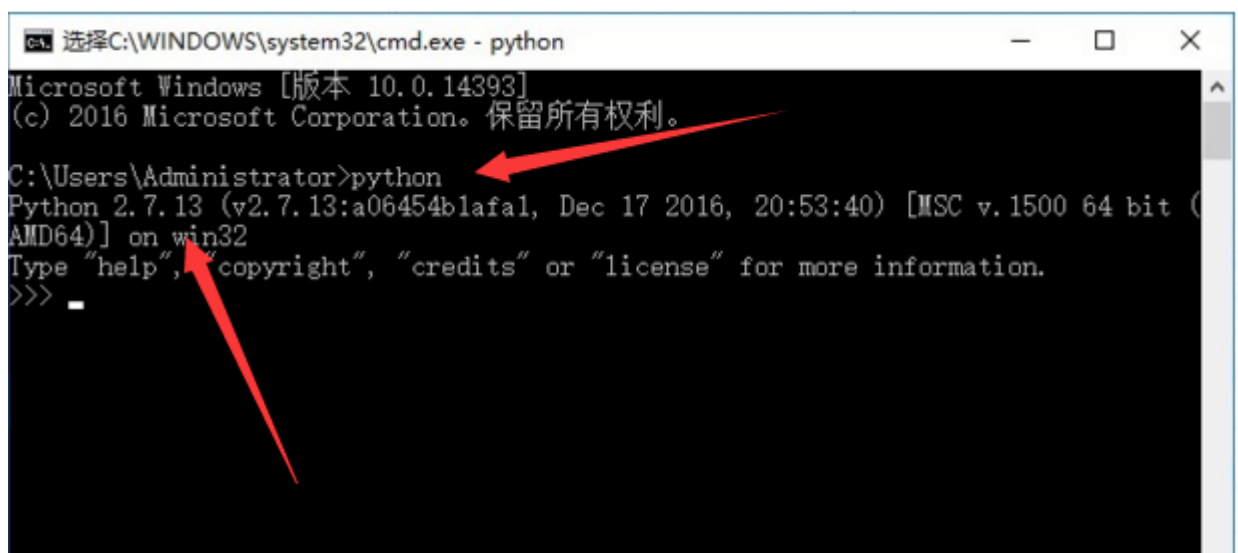
步骤四.png

安装完成，单击“Finish”完成安装



步骤五.png

打开命令行，输入 python,如图验证安装成功

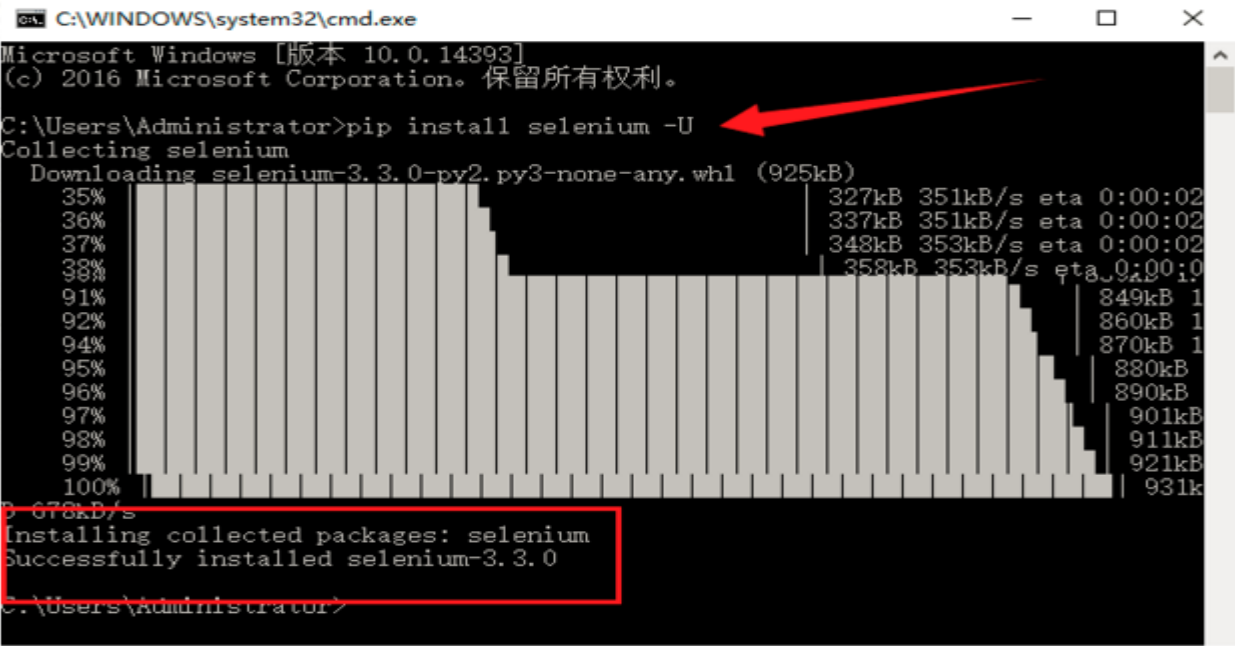


安装成功.png

安装 selenium，在命令行中输入一下命令

```
pip install selenium -U
```

安装成功后提示，如图所示



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation. 保留所有权利。

C:\Users\Administrator>pip install selenium -U
Collecting selenium
  Downloading selenium-3.3.0-py2.py3-none-any.whl (925kB)
    35% |#####| 327kB 351kB/s eta 0:00:02
    36% |#####| 337kB 351kB/s eta 0:00:02
    37% |#####| 348kB 353kB/s eta 0:00:02
    38% |#####| 358kB 353kB/s eta 0:00:02
    91% |#####| 849kB 1
    92% |#####| 860kB 1
    94% |#####| 870kB 1
    95% |#####| 880kB
    96% |#####| 890kB
    97% |#####| 901kB
    98% |#####| 911kB
    99% |#####| 921kB
   100% |#####| 931kB
678kB/s
Installing collected packages: selenium
Successfully installed selenium-3.3.0

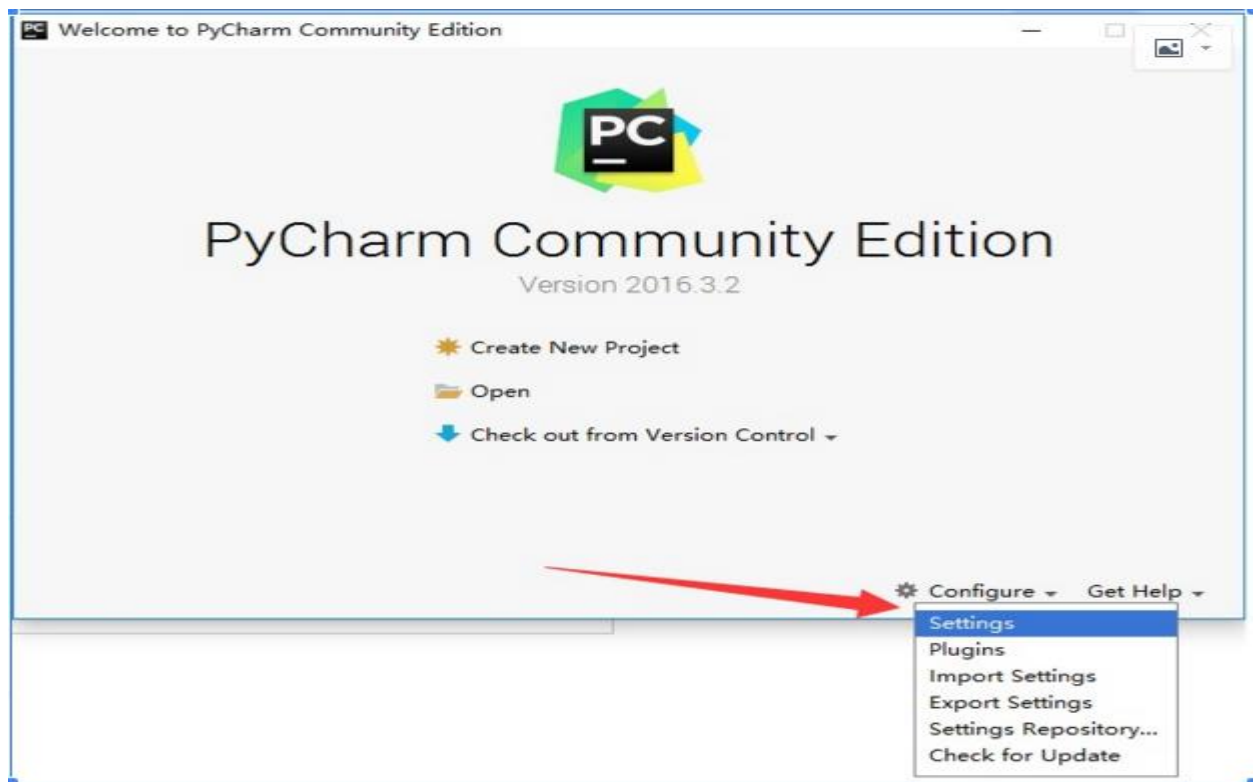
C:\Users\Administrator>
```

selenium 安装成功.png

2.1.4 pyCharm 安装

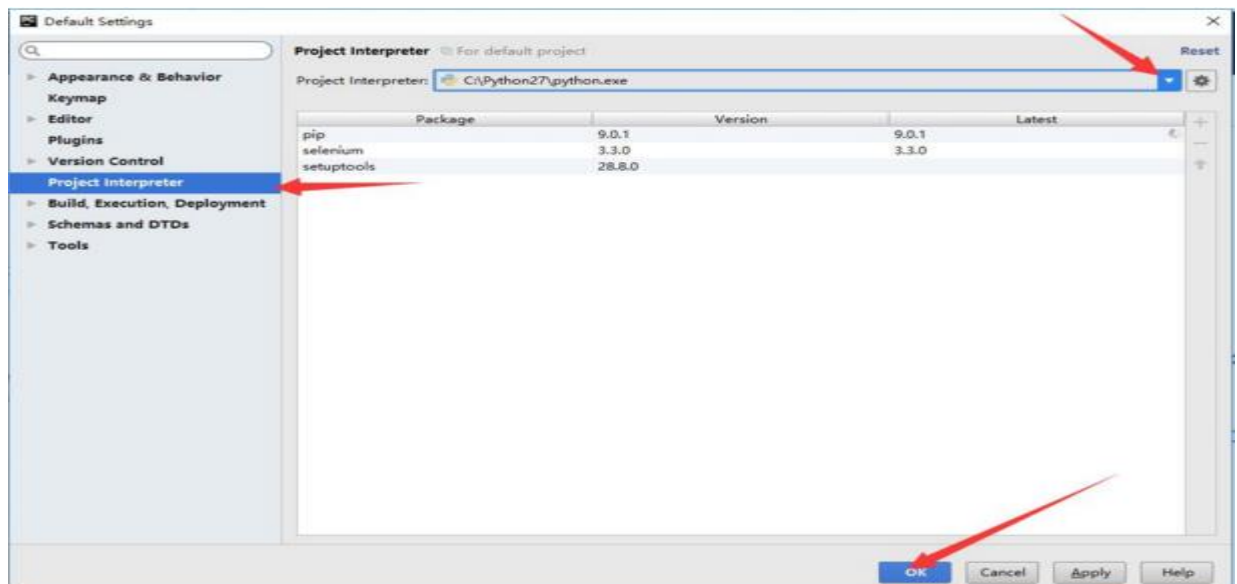
双击下载的安装包，按照默认步骤一步步安装即可。首次启动时，配置下

pycharm 的 python 解析器，如图所示



【启动界面.png】

配置 python 解析器，如图：



配置 python 解析器.png

2.1.5 第一个 python selenium 代码

```
# -*- coding:utf-8 -*-

from selenium import webdriver

from time import sleep

if __name__ == '__main__':

    # 初始化一个 webdriver 实例

    wd = webdriver.Firefox()

    # 访问百度

    wd.get("http://www.baidu.com")

    # 等待 5s

    sleep(5)

    # 关闭浏览器

    wd.close()
```

启动运行即可，如果出现错误，请确定 Firefox 版本是不是太新，需要降低版本

另附 google 和 ie 浏览器驱动下载地址**请选择最新版本进行下载**，下载后请放在 python 安装根目录下。

iedriver 下载地址：

<http://selenium-release.storage.googleapis.com/index.html>

chromedriver 下载地址：

<http://chromedriver.storage.googleapis.com/index.html>

2.2 Python Selenium Webdriver 驱动准备

2.2.1 前言

本次就 python webdriver 的安装和驱动不同浏览器的配置进行分享，以解决大家在入门过程中的一些基本的环境问题。

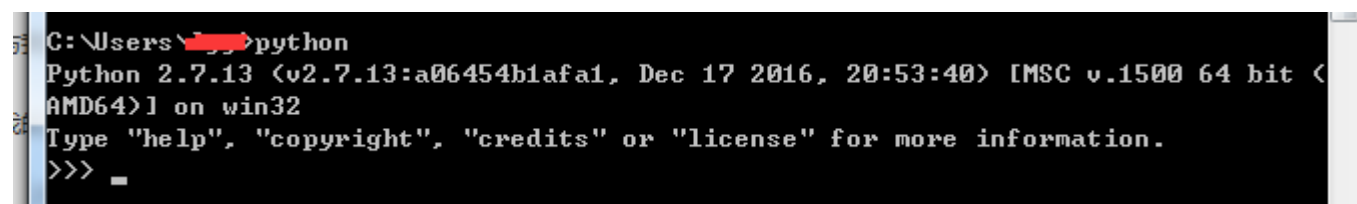
2.2.2 [python 安装](#)

目前 python 有 2.x 和 3.x 版本，笔者在这里推荐 2.x 版本。

从下述地址，根据自己操作系统的版本下载 32 位或 64 位的 python 2.x 最新版本：<https://www.python.org/downloads/>

双击下载的 python 安装包，默认或自定义安装路径，一步步的完成安装。

在命令行中，输入 python，回车，确保 python 已加入环境变量。如图：

A screenshot of a Windows command prompt window. The title bar shows 'C:\Users\...'. The command prompt shows the command 'python' being entered. The output is: 'Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:53:40) [MSC v.1500 64 bit <AMD64>] on win32'. Below this, it says 'Type "help", "copyright", "credits" or "license" for more information.' and then ' >>> ' with a cursor.

python-cmd.png

2.2.3 [升级最新的 pip](#)

在命令中输入以下命令，升级最新版的 pip

python -m pip install -U pip

为什么要升级 pip: 确保后续大家在使用 pip 安装 python 包时, 能获取最新最稳定的包。

2.2.4 [安装 webdriver](#)

在命令行中输入以下命令, 安装最新版的 webdriver

```
pip install selenium -U
```

注: webdriver 是 selenium 2 的一部分。

2.2.5 [配置各种浏览器的驱动](#)

firefox 浏览器

下载地址: <https://github.com/mozilla/geckodriver/releases> 下载后, 将解压的 geckodriver.exe 放至在 python 安装的根目录, 笔者放在 C:/Python27 下。

ie 浏览器

下载地址: <http://selenium-release.storage.googleapis.com/index.html>

请从中选择最新版, 注意是 32 位还是 64 位。 下载后, 将解压的 iedriver.exe 放至在 python 安装的根目录, 笔者放在 C:/Python27 下。

chrome 浏览器

下载地址:<http://chromedriver.storage.googleapis.com/index.html> 请从中选择最新版, 注意是 32 位还是 64 位。 下载后, 将解压的 chromedriver.exe 放至在 python 安装的根目录, 笔者放在 C:/Python27 下。

phantomjs

下载地址：<http://phantomjs.org/download.html> 请从中选择最新版，注意是 32 位还是 64 位。下载后，将解压的 phantomjs.exe 放至在 python 安装的根目录，笔者放在 C:/Python27 下。

2.3 webdriver 介绍&Selenium RC 的比较

2.3.1 什么是 webdriver ?

webdriver 是一个 web 自动化测试框架，不同于 selenium

IDE 只能运行在 firefox 上,webdriver 能够在不同的浏览器上执行你的 web 测试用例。其支持的浏览器有：Firefox、Chrome、IE、Edge、Safari、Opera、phantomjs 等等。

webdriver 支持使用不同的编程语言来写测试脚本，这是 selenium IDE 所无法做到的。对于测试人员来说至少具备：

掌握编程语言的判断分支语法

掌握基本的循环语法

webdriver 支持的编程语言有：

java

.net

php

python

perl

ruby

在本系列教程中，我们将使用 python 来写 webdriver 测试脚本。如果你对 python 的基本语法不够熟悉的话，请先参照《python 简明教程》学习（建议两个小时内完成学习）。

2.3.2 Selenium RC 和 webdriver 的区别

在 webdriver 发布前 selenium1.0 版本叫做 selenium remote control，简称 Selenium RC。webdriver 和 Selenium RC 之间有以下共同的特色：

它们都支持使用一种编程语言来设计你的测试脚本

它们均支持驱动多种浏览器来进行自动化测试

那它们之间有什么区别呢？接下来让我们一起看一看它们的区别：

1、架构

相比 Selenium RC 的架构，webdriver 的架构更加简洁。

webdriver 通过 OS 层级来控制浏览器

webdriver 运行你使用你喜欢的编程语言的 IDE 来进行脚本开发



webdriver 架构图

Selenium RC 的架构更为复杂。

在运行测试脚本前必须先启动 Selenium RC Server

Selenium RC Server 扮演着浏览器和测试脚本之间通信的桥梁角色

当开始运行测试脚本时，Selenium RC Server 会 Selenium Core (js 代码) 注入到浏览器中以达成控制浏览器

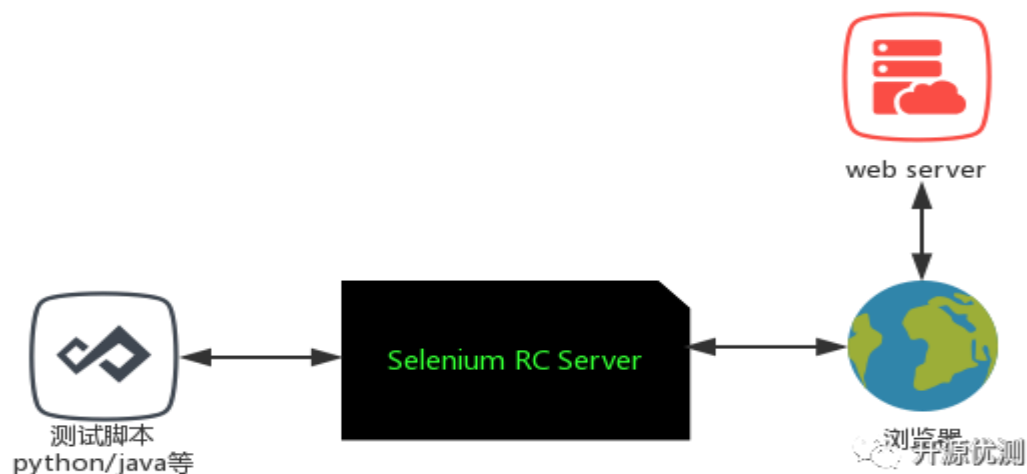
一旦 Selenium Core 被注入到浏览器中，Selenium Core 就会开始接收并转发来自 Selenium RC Server 的测试指令

当测试指令接收后，Selenium Core 就会把它们当做 js 来执行

浏览器按照 Selenium Core 的指令进行操作，并将执行结果返回给 Selenium RC Server

Selenium RC Server 将接收到的结果返回给你的测试脚本

Selenium RC Server 继续从你的测试脚本发送过来的指令中提取下一条指令，重复上述过程。



Selenium RC 架构图

2、速度

在运行速度方面，webdriver 会远远好于 Selenium RC。原因是 Webdriver 直接调用浏览器原生 API 进行驱动，而 Selenium RC 则通过 Selenium Core（javascript 实现）来间接驱动浏览器。

3、交互机制

webdriver 直接与浏览器进行交互

Selenium RC 通过 Selenium RC Server 中转才能与浏览器进行交互

4、API

Selenium RC 的 API 复杂冗余，不利于学习掌握

Webdriver 的 API 简洁，只要掌握几个常用的即可进行测试

5、支持的浏览器

Selenium RC 只能驱动可视化的浏览器

webdriver 除了驱动可视化的浏览器，还可以驱动内存模式的浏览器，比如

HtmlUnit browser , phantomjs

webdriver 的局限性

webdriver 无法及时的支持最新版本的浏览器，每次浏览器升级后，需要下载新的驱动程序。

webdriver 必须基于脚本模式开发测试用例

2.3.3 总结

webdriver 支持使用多种编程语言进行跨浏览器的 web 测试

webdriver 的强大在于支持 N 中编程语言来设计和实现测试

webdriver 执行速度更快是因为其简洁的架构

webdriver 直接驱动浏览器

webdriver 支持内存模式的浏览器

webdriver 不能实时的支持最新版浏览器

webdriver 没有内置的命令模式来自动生成测试结果

2.4 Selenium 2.0 与 Selenium 3.0 介绍

2.4.1 什么是 Selenium

Selenium 是一组 web 自动化测试工具集，它由以下几个部分构成：

- Selenium IDE(Integrated Development Environment)

这是 Firefox 浏览器的一个插件，用于录制和回放 selenium 测试脚本。

- WebDriver 和 RC

它提供了各种编程语言 API 的支持，例如 java、python、ruby、php、.net 等等，能够与不同的浏览器进行交互，驱动浏览器进行自动化测试。

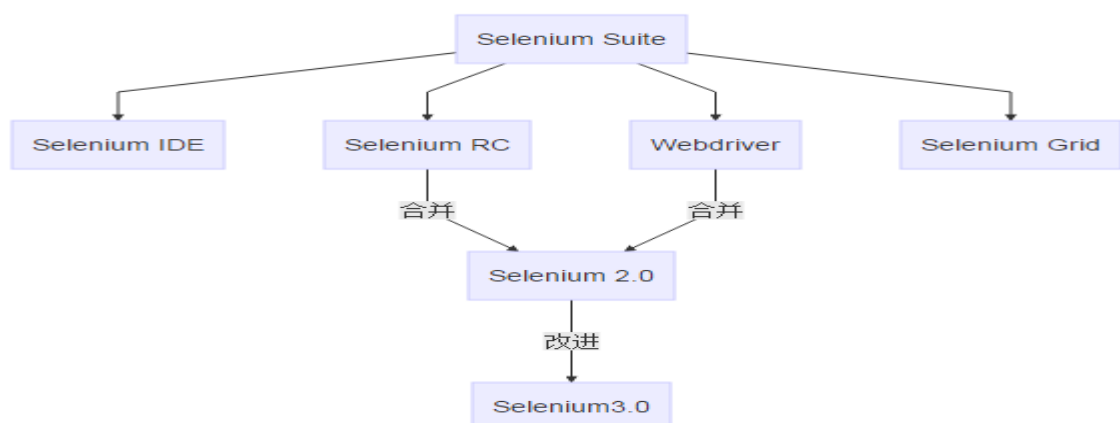
- Grid

它提供了分布式测试和并行测试的能力，能够帮助我们大幅的减少测试的执行时间。

2.4.2 什么是 Selenium 2.0

Selenium 2.0 集成了 RC 和 webdriver 来提供 web UI 级自动化测试能力。

下面我们看下其构成:



selenium_suite.png

2.4.3 什么是 Selenium 3.0

Selenium 3.0 是 selenium 最新发布版本，目前已经发布了 2 个 beta 版本出来。

下面我们一起来看看 Selenium 3.0 新增的特性：

beta 1	beta 2 (只针对 java 版本)
要求 java 版本 8+	系统属性 webdriver.firefox.marionette 被集成至 marionetter 服务或 firefox 驱动中,该版本起将忽略所有相关的 Desired Capability 设置
通过 Mozilla 的 geckodriver 驱动 Firefox	当浏览器未指定时，Grid 修复了注册时的 NPE
支持 Edge 浏览器，该驱动有 ms 提供	更新了 GeckOdriver
支持由 Apple 提供的 safari 驱动	

2.4.4 总结

selenium 3.0 有了更新的特性加入，尤其是对 Edge 和 safari 原生驱动的支持，Edge 驱动有 MS 提供，safari 原生驱动有 Apple 提供。

在最新的 Firefox 方面，开始支持 Mozilla 的 geckodriver 驱动，来驱动 Firefox 的控制。

总而言之，selenium 3.0 在支持的原生驱动方面更为丰富，在 2.0 的基础上有了更多的改进。

2.5 创建你的第一个 webdriver python 代码

2.5.1 前言

今天我们开始我们的第一个 python webdriver 自动化测试脚本。并就测试脚本进行——解释说明。

2.5.2 webdriver python 代码

本示例代码演示了使用 Ie 浏览器访问百度进行搜索测试。

HTMLTestRunner 从这里下载：

http://tungwaiyip.info/software/HTMLTestRunner_0_8_2/HTMLTestRunner.py

下载后和当前测试脚本放在同一目录。

将以下代码保存到 first_webdriver.py 中

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
from selenium import webdriver
```

```
import unittest
```

```
import HTMLTestRunner
```

```
import sys
```

```
from time import sleep
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
class BaiduTest(unittest.TestCase):
```

```
    """百度首页搜索测试用例"""
```

```
    def setUp(self):
```

```
        self.driver = webdriver.Ie()
```

```
        self.driver.implicitly_wait(30)
```

```
        self.base_url = u"http://www.baidu.com"
```

```
    def test_baidu_search(self):
```

```
        driver = self.driver
```

```
        print u"开始[case_0001]百度搜索"
```

```
        driver.get(self.base_url)
```

```
        # 验证标题
```

```
        self.assertEqual(driver.title, u"百度一下，你就知道")
```

```
        driver.find_element_by_id("kw").clear()
```

```
        driver.find_element_by_id("kw").send_keys(u"开源优测")
```

```
        driver.find_element_by_id("su").click()
```

```
        sleep(3)
```



```
# 验证搜索结果标题

self.assertEqual(driver.title, u"开源优测_百度搜索")


def tearDown(self):

    self.driver.quit()


if __name__ == '__main__':

    testunit = unittest.TestSuite()

    testunit.addTest(BaiduTest('test_baidu_search'))

    # 定义报告输出路径

    htmlPath = u"testReport.html"

    fp = file(htmlPath, "wb")

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp,

        title=u"百度测试",

        description=u"测试用例结果")

    runner.run(testunit)

    fp.close()
```

2.5.3 代码解释

总体上代码分为五大块

1. 文件保存编码及作者定义

```
#-*- coding:utf-8 -*-  
  
__author__ = u'苦叶子'
```

2. 导入相关基础模块

```
# 从 selenium 中导入 webdriver 模块  
  
from selenium import webdriver  
  
# 导入 unittest 模块，作为用例基类  
  
import unittest  
  
# 导入 html 报告生成模块，用于 html 格式报告生成  
  
import HTMLTestRunner  
  
# 导入 sys 模块  
  
import sys  
  
# 导入 sleep 模块，用于强制等待  
  
from time import sleep
```

3. 设置当前 python 运行环境为 utf8

```
# 设置当前 python 运行在 utf-8 编码下，这样你的中文就不会乱码了  
  
reload(sys)  
  
sys.setdefaultencoding("utf-8")
```

4. 定义和实现测试用例

```
# 从 unittest.TestCase 继承

class BaiduTest(unittest.TestCase):

    """百度首页搜索测试用例"""

    # 用例级初始化函数，自动执行

    def setUp(self):

        # 初始化基于 IE 浏览器的 webdriver 实例

        self.driver = webdriver.Ie()

        # 给当前 webdriver 设置全局隐性等待时间，最大 30s

        self.driver.implicitly_wait(30)

        # 设置首页 url

        self.base_url = u"http://www.baidu.com"

    def test_baidu_search(self):

        # 简单赋值，这样在本测试中后续就不用每次都写 self.driver，

        # 少写几个字符，都是为了偷懒啊😁

        driver = self.driver

        # 在控制台打印输出

        print u"开始[case_0001]百度搜索"

        # 启动浏览器，并访问首页
```

```

driver.get(self.base_url)

# 验证标题

self.assertEqual(driver.title, u"百度一下，你就知道")

# 清理搜索输入框中的数据

driver.find_element_by_id("kw").clear()

# 在搜索输入框中输入 开源优测

driver.find_element_by_id("kw").send_keys(u"开源优测")

# 单击 百度一下 按钮

driver.find_element_by_id("su").click()

# 强制等 3s

sleep(3)

# 验证搜索结果标题

self.assertEqual(driver.title, u"开源优测_百度搜索")

# 用例级清理函数，自动执行

def tearDown(self):

    # 退出 webdriver，同时关闭当前 webdrier session 下所有浏览
器窗口

    self.driver.quit()

```

5. 测试脚本主运行入口

```

# python main 函数

```

```
if __name__ == '__main__':

    # 初始化一个用例套件集

    testunit = unittest.TestSuite()

    # 往用例套件集新增一个测试

    testunit.addTest(BaiduTest('test_baidu_search'))

    # 定义报告输出路径，这里是当前目录

    htmlPath = u"testReport.html"

    # 打开测试报告文件

    fp = file(htmlPath, "wb")

    # 构建一个 HTMLTestReport 执行器

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp,

        title=u"百度测试",

        description=u"测试用例结果")

    # 运行测试集

    runner.run(testunit)

    # 关闭打开的测试报告文件

    fp.close()
```

2.5.4 运行代码

使用以下命令运行上述代码

```
python first_webdriver.py
```

可以看到：

将启动浏览器访问百度首页

在百度首页搜索框中输入了 开源优测 单击了 百度一下 按钮 显示出搜索结果

闭关了浏览器

在当前目录下 生成了 testReport.html 的测试报告文件

2.5.5 总结

最后总结下，要注意的几个关键点：

确保要启动的浏览器的驱动已经下载好，具体在哪下载请参见上一章

确保下载了 HTMLTestRunner 模块

最好自己把代码一行行敲入一遍，不要直接拷贝运行

2.6 Python 多线程 Selenium 跨浏览器测试

2.6.1 前言

在 web 测试中，不可避免的一个测试就是浏览器兼容性测试，在没有自动化测试前，我们总是苦逼的在一台或多台机器上安装 N 种浏览器，然后手工在不同的浏览器上验证主业务流程和关键功能模块功能，以检测不同浏览器或不同版本浏览器上，我们的 web 应用是否可以正常工作。



browser.png

下面我们看看怎么利用 python selenium 进行自动化的跨浏览器测试。

2.6.2 什么是跨浏览器测试

跨浏览器测试是功能测试的一个分支，用以验证 web 应用能在不同的浏览器上正常工作。

2.6.3 为什么需要跨浏览器测试

通常情况下，我们都期望 web 类应用能够被我们的用户在任何浏览器上使用。

例如，有的人喜欢用 IE 来打开开源优测 web 站点

<http://www.testingunion.com>，但有的人喜欢 firefox 或 chrome。

我们期望我们的 web 系统能在任何浏览器上正常的工作，这样能吸引更多的用户来使用。

需要跨浏览器测试的根源是：

1. 在不同浏览器字体大小不匹配
2. javascript 的实现不一样
3. css、html 的验证有所区别
4. 有的浏览器或低版本不支持 HTML5
5. 页面对齐和 div 大小问题
6. 图片位置或大小问题
7. 浏览器和操作系统间的兼容问题

以上几个方面不仅仅对布局有影响，甚至会导致功能不可用，所以我们需要进行跨浏览器测试。

2.6.4 如何执行跨浏览器测试

如果我们使用 selenium webdriver，那我们就能够自动的在 IE、firefox、chrome、等不同浏览器上运行测试用例。

为了能在同一台机器上不同浏览器上同时执行测试用例，我们需要多线程技术。

下面我们基于 python 的多线程技术来尝试同时启动多个浏览器进行 selenium 自动化测试。

```
#-*- coding:utf-8 -*-

__author__ = u'苦叶子'

from selenium import webdriver

import sys

from time import sleep

from threading import Thread

reload(sys)

sys.setdefaultencoding("utf-8")

def test_baidu_search(browser, url):

    driver = None

    # 你可以自定义这里，添加更多浏览器支持进来

    if browser == "ie":

        driver = webdriver.Ie()

    elif browser == "firefox":

        driver = webdriver.Firefox()

    elif browser == "chrome":

        driver = webdriver.Chrome()
```

```
if driver == None:

    exit()

print u"开始[case_0001]百度搜索"

driver.get(url)

print u"清除搜索中数据，输入搜索关键词"

driver.find_element_by_id("kw").clear()

driver.find_element_by_id("kw").send_keys(u"开源优测")

print u"单击 百度一下 按钮 开始搜索"

driver.find_element_by_id("su").click()

sleep(3)

print u"关闭浏览器，退出 webdriver"

driver.quit()
```

```
if __name__ == "__main__":

    # 浏览器和首页 url

    data = {

        "ie":"http://www.baidu.com",

        "firefox":"http://www.baidu.com",

        "chrome":"http://www.baidu.com"

    }
```

```
# 构建线程

threads = []

for b, url in data.items():

    t = Thread(target=test_baidu_search,args=(b,url))

    threads.append(t)

# 启动所有线程

for thr in threads:

    thr.start()
```

运行上述代码，你会发现三个浏览器都会启动开始进行百度搜索，是不是很有意思？当然上面只是简单的演示，更多更实用的能力有待挖掘。

2.6.5 总结

本文初始演示了利用 python 多线程技术来启动多个浏览器同时进行 selenium 自动化测试，通过这个示例你应该要去学习更深入的知识，和深入结合实际业务测试梳理出更合适的自动化测试业务场景。

至于如何更深入的利用 selenium 把兼容性测试做好，还有待深入研究挖掘，真正的把 selenium 的特性用好。

2.7 在 Selenium Webdriver 中使用 XPath Contains、Sibling 函数定位

2.7.1 前言

在一般情况下，我们通过简单的 xpath 即可定位到目标元素,但对于一些既没 id 又没 name，而且其他属性都是动态的情况就很难通过简单的方式进行定位了。

在这种情况下，我们需要使用 xpath1.0 内置的函数来进行定位，下面我们重点讨论一下 3 个函数：

- Contains
- Sibling

2.7.2 Contains 函数

通过 contains 函数，我们可以提取匹配特定文本的所有元素。

例如在百度首页，我们使用 contains 定位包含“新闻”文本的元素。



baidu_news.png

```
"//div/a[contains(text(), 新闻)]"
```

在 python selenium 中使用 xpath contains 定位，代码片段如下：

```
driver.find_element_by_xpath("//div/a[contains(text(), 新闻)])")
```

2.7.3 sibling 函数

通过 sibling 函数我们可以提取指定元素的所有同级元素，即获取目标元素的所有兄弟节点。

例如通过刚才“新闻”节点来定位“hao123”节点。

```
"//div/following-sibling::a[contains(text(), 新闻)]"
```

python selenium 代码片段为如下

```
driver.find_element_by_xpath(  
u"//div/a[contains(text(), '%s')]/following-sibling::*" % u"新闻")
```

通过刚才“新闻”节点来定位其所有的兄弟节点。

python selenium 代码片段如下（注意这里用的是

find_elements_by_xpath）：

```
driver.find_elements_by_xpath(  
u"//div/a[contains(text(), '%s')]/following-sibling::*" % u"新闻")
```

下面我们看一个完整的代码示例：

```
#_ coding:utf-8 _
```

```

__author__ = '苦叶子'

from selenium import webdriver

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

if __name__ == '__main__':

    driver = webdriver.Ie()

    driver.get(u"http://www.baidu.com")

    # 定位 通过 contains 定位包含 “新闻” 的元素

    new_node = driver.find_element_by_xpath(

        u"//div/a[contains(text(), '%s')]" % u"新闻")

    print new_node.text

    # 定位 “新闻” 元素的兄弟节点 “hao123”

    hao123_node = driver.find_element_by_xpath(

        u"//div/a[contains(text(), '%s')]/following-sibling::*" % u"新闻")

    print hao123_node.text

    # 定位 “新闻” 元素的所有兄弟节点

```

```
all_node = driver.find_elements_by_xpath(  
  
u"//div/a[contains(text(), '%s')]/following-sibling::*" % u"新闻")  
  
for ee in all_node:  
  
    print ee.text  
  
driver.quit()
```

2.7.4 xpath 常用函数

1. child 选取当前节点的所有子节点
2. parent 选取当前节点的父节点
3. descendant 选取当前节点的所有后代节点
4. ancestor 选取当前节点的所有先辈节点
5. descendant-or-self 选取当前节点的所有后代节点及当前节点本身
6. ancestor-or-self 选取当前节点所有先辈节点及当前节点本身
7. preceding-sibling 选取当前节点之前的所有同级节点
8. following-sibling 选取当前节点之后的所有同级节点
9. preceding 选取当前节点的开始标签之前的所有节点
10. following 选取当前节点的开始标签之后的所有节点
11. self 选取当前节点

12.attribute 选取当前节点的所有属性

13.namespace 选取当前节点的所有命名空间节点

2.7.5 总结

在本文中对 xpath 常用的 contains、sibling 函数进行了说明和代码演示，对于其他的函数建议大家自己写代码去实践，理解其原理，将会更有利于后续的自动化测试实践。

2.8 Selenium Webdriver Desired Capabilities

2.8.1 前言

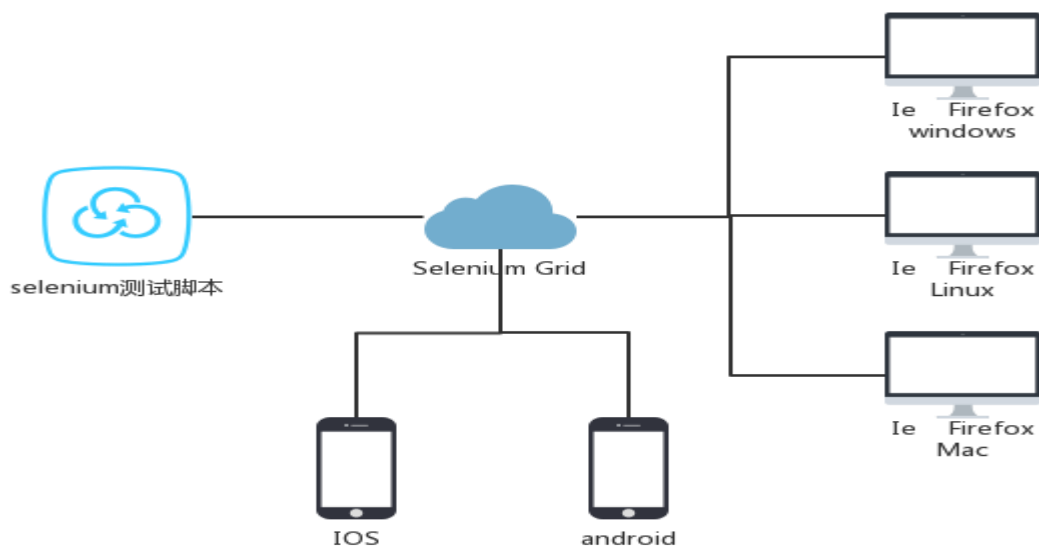
我们每一个的 selenium 测试都应该在指定的环境下运行，这个环境可以是 web 浏览器、移动设备、移动模拟器等等。

那怎么来指定我们的 selenium 测试脚本运行环境呢？

本次分享主要解决这个问题！！！！

在 python 版本的 webdriver 中，DesiredCapabilities 类为我们提供了解决方案，通过这个类，我们能够配置 webdriver 在指定的环境执行我们的测试脚本。

我们先看一张实际测试过程中会应用到的结构图，基于 selenium grid 进行分布式测试：



selenium_grid.png

在上图中，我们可以使用 windows 下 ie、firefox 或 linux 下 ie、firefox 进行测试，甚至可以设置使用 andriod 或 ios 设备下浏览器进行测试。

那怎么来进行设置呢？

我们先看一下 DesiredCapabilities 的源码

2.8.2 源码分析

DesiredCapabilities 类代码在 C:\Python27\Lib\site-packages\selenium\webdriver\common\desired_capabilities.py。

注：笔者的 python 安装在 C 盘下。

```
class DesiredCapabilities(object):
```

```
    """
```

```
        Set of default supported desired capabilities.
```

Use this as a starting point for creating a desired capabilities object for

requesting remote webdrivers for connecting to selenium server or selenium grid.

Usage Example::

```
from selenium import webdriver

selenium_grid_url = "http://198.0.0.1:4444/wd/hub"

# Create a desired capabilities object as a starting point.

capabilities = DesiredCapabilities.FIREFOX.copy()

capabilities['platform'] = "WINDOWS"

capabilities['version'] = "10"

# Instantiate an instance of Remote WebDriver with the desired
capabilities.

driver = webdriver.Remote(desired_capabilities=capabilities,

command_executor=selenium_grid_url)
```

Note: Always use '.copy()' on the DesiredCapabilities object to avoid the side

effects of altering the Global class instance.

```
"""
```

```
FIREFOX = {
```

```
    "browserName": "firefox",
```

```
    "version": "",
```

```
    "platform": "ANY",
```

```
    "javascriptEnabled": True,
```

```
    "marionette": True,
```

```
}
```

```
INTERNETEXPLORER = {
```

```
    "browserName": "internet explorer",
```

```
    "version": "",
```

```
    "platform": "WINDOWS",
```

```
    "javascriptEnabled": True,
```

```
}
```

```
EDGE = {
```

```
    "browserName": "MicrosoftEdge",
```

```
    "version": "",  
  
    "platform": "WINDOWS"  
}
```

```
CHROME = {
```

```
    "browserName": "chrome",  
  
    "version": "",  
  
    "platform": "ANY",  
  
    "javascriptEnabled": True,  
  
}
```

```
OPERA = {
```

```
    "browserName": "opera",  
  
    "version": "",  
  
    "platform": "ANY",  
  
    "javascriptEnabled": True,  
  
}
```

```
SAFARI = {
```

```
    "browserName": "safari",
```

```
    "version": "",  
  
    "platform": "MAC",  
  
    "javascriptEnabled": True,  
  
}
```

```
HTMLUNIT = {  
  
    "browserName": "htmlunit",  
  
    "version": "",  
  
    "platform": "ANY",  
  
}
```

```
HTMLUNITWITHJS = {  
  
    "browserName": "htmlunit",  
  
    "version": "firefox",  
  
    "platform": "ANY",  
  
    "javascriptEnabled": True,  
  
}
```

```
IPHONE = {  
  
    "browserName": "iPhone",
```

```
    "version": "",  
  
    "platform": "MAC",  
  
    "javascriptEnabled": True,  
  
}
```

```
IPAD = {
```

```
    "browserName": "iPad",  
  
    "version": "",  
  
    "platform": "MAC",  
  
    "javascriptEnabled": True,  
  
}
```

```
ANDROID = {
```

```
    "browserName": "android",  
  
    "version": "",  
  
    "platform": "ANDROID",  
  
    "javascriptEnabled": True,  
  
}
```

```
PHANTOMJS = {  
  
    "browserName": "phantomjs",  
  
    "version": "",  
  
    "platform": "ANY",  
  
    "javascriptEnabled": True,  
  
}
```

翻译下：

Set of default supported desired capabilities.

desired capabilities 默认支持的设置。

Use this as a starting point for creating a desired capabilities object for requesting remote webdrivers for connecting to selenium server or selenium grid.

使用该类为 selenium server 或 selenium grid 启动一个 desired capabilities 配置

对配置项进行解释

```
FIREFOX = {  
  
    "browserName": "firefox", # 浏览器名称  
  
    "version": "",           # 操作系统版本
```

```

        "platform": "ANY",          # 平台，这里可以是 windows、linux、
andriod 等等

        "javascriptEnabled": True, # 是否启用 js

        "marionette": True,        # 这个值没找对应的说明^_^ 不解释了
    }

```

2.8.3 DesiredCapabilities 示例

```

from selenium import webdriver

# 本地启动 selenium grid

selenium_grid_url = "http://127.0.0.1:4444/wd/hub"

# 创建一个 DesiredCapabilities 实例

capabilities = DesiredCapabilities.FIREFOX.copy()

capabilities['platform'] = "WINDOWS" # 指定操作系统

capabilities['version'] = "10"      # 指定操作系统版本

# 连接到远程服务进行自动化测试

driver = webdriver.Remote(desired_capabilities=capabilities,

                           command_executor=selenium_grid_url)

```


2.8.4 总结

这里的本质就是基于 selenium grid 构建分布式自动化测试，而 selenium grid 根据测试脚本构建的 DesiredCapabilities 参数来决定将您的测试脚本分发到哪台机器或设备进行测试。

2.9 基于 Excel 参数化你的 Selenium3 测试代码

2.9.1 前言

今天我们就如何使用 xlrd 模块来进行 python Selenium3 + excel 自动化测试过程中的参数化进行演示说明，以解决大家在自动化测试实践过程中参数化的疑问。

2.9.2 环境安装

xlrd 是 python 用于读取 excel 的第三方扩展包，因此在使用 xlrd 前，需要使用以下命令来安装 xlrd。

```
pip install xlrd
```

2.9.3 xlrd 基本用法

1. 导入扩展包

```
import xlrd
```

2. 打开 excel 文件

```
excel = xlrd.open_workbook(u'excelFile.xls')
```

3. 获取工作表

通过索引顺序获取

```
table = excel.sheets()[0]
```

```
table = excel.sheet_by_index(0)
```

4. 通过工作表名获取

```
table = excel.sheet_by_name(u'Sheet1')
```

5. 获取行数和列数

获取行数

```
nrows = table.nrows
```

获取列数

```
ncols = table.ncols
```

6. 获取整行或整列的值

其中 i 为行号，j 为列号

行号、列号索引从 0 开始

```
row_values = table.row_values(i)
```

```
col_values = table.col_values(j)
```

7. 获取指定单元格数据

i 行号，j 列号

```
value = table.cell(i, j).value
```

8. 例如获取第一行、第一列的数据

```
value = table.cell(0, 0).value
```

9. 循环行遍历列表数据

先获取行数

```
nrows = table.nrows
```

遍历打印所有行数据

```
for i in range(0, nrows):
```

```
    print table.row_values(i)
```

至此我们将 xlrd 基本常用的技巧和方法都一一列举完毕，下面我们一起看一下如何利用 xlrd 来实现 python Selenium3 自动化测试参数化。

2.9.4 代码示例

我们以上一章我们的第一个 python Selenium3 测试代码为蓝本，进行改造，从 excel 中读取以下格式的数据来进行测试，
请将下列表格数据存入名为 baidu_search.xlsx 的 excel 文件。

序号	搜索词	期望结果
1	开源优测	开源优测_百度搜索
2	别啊	别啊_百度搜索
3	尼玛，能不能动手分享下？	尼玛，能不能动手分享下？_百度搜索

excel_demo.png

#将以下代码保存到 first_webdriver.py 中

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
from selenium import webdriver
```

```
import unittest

import HTMLTestRunner

import sys

from time import sleep

import xlrd

reload(sys)

sys.setdefaultencoding("utf-8")


class LoadBaiduSearchTestData:

    def __init__(self, path):

        self.path = path

    def load_data(self):

        # 打开 excel 文件

        excel = xlrd.open_workbook(self.path)

        # 获取第一个工作表

        table = excel.sheets()[0]

        # 获取行数

        nrows = table.nrows
```

```
# 从第二行开始遍历数据

# 存入一个 list 中

test_data = []

for i in range(1, nrows):

    test_data.append(table.row_values(i))

# 返回读取的数据列表

return test_data
```

```
class BaiduTest(unittest.TestCase):

    """百度首页搜索测试用例"""

    def setUp(self):

        self.driver = webdriver.Firefox()

        self.driver.implicitly_wait(30)

        self.base_url = u"http://www.baidu.com"

        self.path = u"baidu_search.xlsx"

    def test_baidu_search(self):

        driver = self.driver

        print u"开始[case_0001]百度搜索"
```

```
# 加载测试数据

test_excel = LoadBaiduSearchTestData(self.path)

data = test_excel.load_data()

print data

# 循环参数化

for d in data:

    # 打开百度首页

    driver.get(self.base_url)

    # 验证标题

    self.assertEqual(driver.title, u"百度一下，你就知道")

    sleep(1)

    driver.find_element_by_id("kw").clear()

    # 参数化 搜索词

    driver.find_element_by_id("kw").send_keys(d[1])

    sleep(1)

    driver.find_element_by_id("su").click()

    sleep(1)

    # 验证搜索结果标题

    self.assertEqual(driver.title, d[2])

    sleep(2)
```

```

def tearDown(self):

    self.driver.quit()

if __name__ == '__main__':

    testunit = unittest.TestSuite()

    testunit.addTest(BaiduTest('test_baidu_search'))

    # 定义报告输出路径

    htmlPath = u"testReport.html"

    fp = file(htmlPath, "wb")

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp,

        title=u"百度测试",

        description=u"测试用例结果")

    runner.run(testunit)

    fp.close()

```

2.9.5 总结

在上文中，我们详细的描述了 xlrd 操作 excel 的各种方法和技巧，以及封装 xlrd 读取 excel 实现在 python selenium 自动化测试过程参数化相应的输入数据和期望结果。

最重要的还是需要大家自己多练习相关的代码，并能做相应的扩展，同时要去有针对性的学习对应的库，深入了解其使用方法和技巧，甚至原理。

2.10 Python Selenium 设计模式-POM

2.10.1 前言

本文就 python selenium 自动化测试实践中所需要的 POM 设计模式进行分享，以便大家在实践中对 POM 的特点、应用场景和核心思想有一定的理解和掌握。

2.10.2 为什么要用 POM

基于 python Selenium3 开始 UI 级自动化测试并不是多么艰巨的任务。**只需要定位到元素，执行对应的操作即可。**下面我们看一下这个简单的脚本实现百度搜索。

```
from selenium import webdriver

import time

driver = webdriver.Firefox()

driver.implicitly_wait(30)

# 启动浏览器，访问百度

driver.get("http://www.baidu.com")

# 定位 百度搜索框，并输入 selenium

driver.find_element_by_id("kw").send_keys("selenium")

# 定位 百度一下 按钮并单击进行搜索

driver.find_element_by_id("su").click()

time.sleep(5)

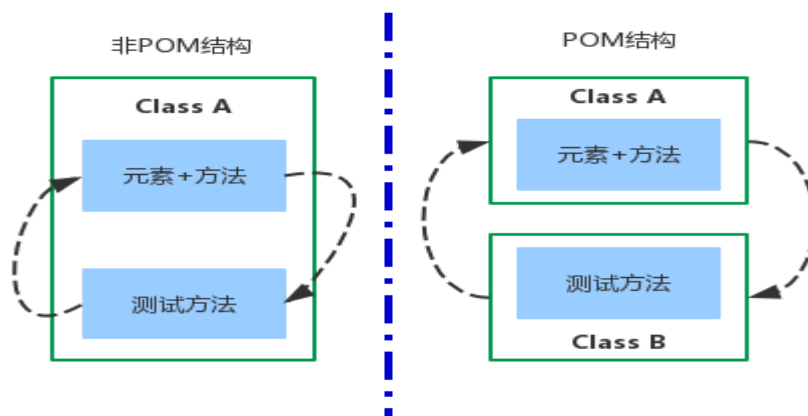
driver.quit()
```


从上述代码来看，我们所能做的就是定位到元素，然后进行键盘输入或鼠标动作。就这个小程序而已，维护起来看起来是很容易的。但随着时间的迁移，测试套件将持续的增长。脚本也将变得越来越臃肿庞大。如果变成我们需要维护 10 个页面，100 个页面，甚至 1000 个呢？那页面元素的任何改变都会让我们的脚本维护变得繁琐复杂，而且变得耗时易出错。

那怎么解决呢？

在自动化测试中，引入了 Page Object Model (POM)：页面对象模式来解决，POM 能让我们的测试代码变得可读性更好，高可维护性，高复用性。

下图为非 POM 和 POM 对比图：



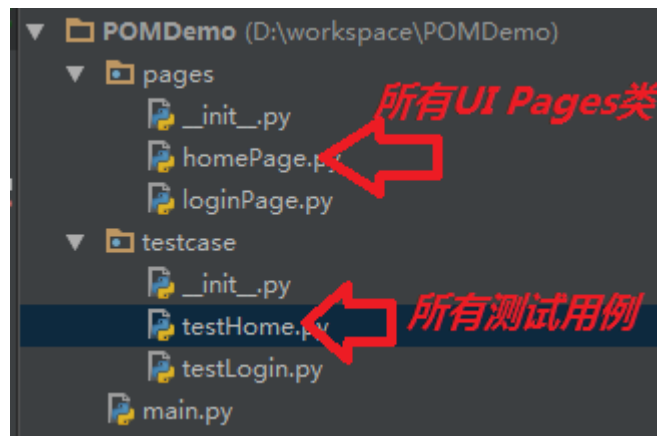
pom.png

2.10.3 POM 是什么

- 页面对象模型(POM)是一种设计模式，用来管理维护一组 web 元素集的对象库
- 在 POM 下，应用程序的每一个页面都有一个对应的 page class
- 每一个 page class 维护着该 web 页的元素集和操作这些元素的方法

- page class 中的方法命名最好根据其对应的业务场景进行，例如通常登录后我们需要等待几秒中，我们可以这样命名该方法：
`waitingForLoginSuccess()`.

下面我们看看 POM 的代码目录组织示例：



pages_dir.png

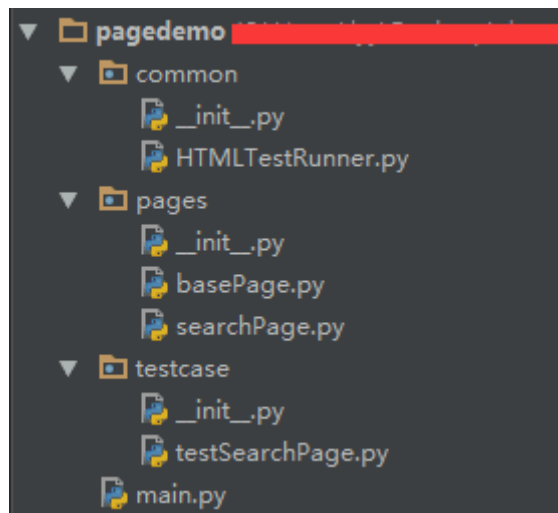
2.10.4 POM 的优势

1. POM 提供了一种在 UI 层操作、业务流程与验证分离的模式，这使得测试代码变得更加清晰和高可读性
2. 对象库与用例分离，使得我们更好的复用对象，甚至能与不同的工具进行深度结合应用
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，方法名命名为: `gotoHomePage()`,通过方法名即可清晰的知道具体的功能实现。

2.10.5 POM 实现示例

下面我们看下使用 POM 百度搜索 POM 代码示例：

看先下代码组织结构如下：



page_demo_dir.png

basePage.py 代码如下

```
# -*- coding:utf-8 -*-
```

```
__author__ = '苦叶子'
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
# pages 基类
```

```
class Page(object):
```

```
    """
```

```
        Page 基类，所有 page 都应该继承该类
```

```
    """
```

```

def __init__(self, driver, base_url=u"http://www.baidu.com"):

    self.driver = driver

    self.base_url = base_url

    self.timeout = 30


def find_element(self, *loc):

    return self.driver.find_element(*loc)


def input_text(self, loc, text):

    self.find_element(*loc).send_keys(text)


def click(self, loc):

    self.find_element(*loc).click()


def get_title(self):

    return self.driver.title


# searchPage.py 代码如下

# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import sys

```

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

from pages.basePage import Page

reload(sys)

sys.setdefaultencoding("utf-8")


# 百度搜索 page

class SearchPage(Page):

    # 元素集

    # 搜索输入框

    search_input = (By.ID, u'kw')

    # 百度一下 按钮

    search_button = (By.ID, u'su')


    def __init__(self, driver, base_url=u"http://www.baidu.com"):

        Page.__init__(self, driver, base_url)


    def gotoBaiduHomePage(self):

        print u"打开首页: ", self.base_url

        self.driver.get(self.base_url)
```

```

def input_search_text(self, text=u"开源优测"):

    print u"输入搜索关键字： 开源优测 "

    self.input_text(self.search_input, text)


def click_search_btn(self):

    print u"点击 百度一下 按钮"

    self.click(self.search_button)


# testSearchPage.py 代码如下

# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

from selenium import webdriver

from pages.searchPage import SearchPage

reload(sys)

sys.setdefaultencoding("utf-8")


# 百度搜索测试

class TestSearchPage(unittest.TestCase):

    def setUp(self):

```

```

        self.driver = webdriver.Ie()

def testSearch(self):

    driver = self.driver

    # 百度网址

    url = u"http://www.baidu.com"

    # 搜索文本

    text = u"开源优测"

    # 期望验证的标题

    assert_title = u"开源优测_百度搜索"

    print assert_title

    search_Page = SearchPage(driver, url)

    # 启动浏览器，访问百度首页

    search_Page.gotoBaiduHomePage()

    # 输入 搜索词

    search_Page.input_search_text(text)

    # 单击 百度一下 按钮进行搜索

    search_Page.click_search_btn()

    # 验证标题

    self.assertEqual(search_Page.get_title(), assert_title)

def tearDown(self):

```

```
        self.driver.quit()

# 主入口程序代码如下

# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

from common import HTMLTestRunner

from testcase.testSearchPage import TestSearchPage

reload(sys)

sys.setdefaultencoding("utf-8")


if __name__ == '__main__':

    testunit = unittest.TestSuite()

    testunit.addTest(TestSearchPage('testSearch'))

    # 定义报告输出路径

    htmlPath = u"page_demo_Report.html"

    fp = file(htmlPath, "wb")

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp,

        title=u"百度测试",

        description=u"测试用例结果")
```



```
runner.run(testunit)
```

```
fp.close()
```

按照如图所示组织代码结构，输入如上代码，执行以下命令运行，会在当前目录生成测试报告：

```
python main.py
```

2.10.6 总结

最后做个总结，所有代码请手动输入，不要直接拷贝。再次对 POM 进行小结

1. POM 是 selenium webdriver 自动化测试实践对象库设计模式
2. POM 使得测试脚本更易于维护
3. POM 通过对象库方式进一步优化了元素、用例、数据的维护组织

三、高级示例

3.1 python Selenium3 示例 - 启动不同浏览器

3.1.1 启动 firefox 浏览器

不需要下载任何驱动，原生支持 firefox，但要注意 firefox 浏览器的版本，如果出现启动 firefox 失败的情况，请降低或升级 firefox 版本。



浏览器

1、firefox 安装在默认路径，启动代码如下：

```
# -*- coding:utf-8 -*-  
  
from selenium import webdriver  
  
driver=webdriver.Firefox()  
  
# 注意 http 不可以省略  
url='http://www.baidu.com'  
  
driver.get(url)  
  
driver.close()
```

2、指定 firefox 的安装路径启动，代码如下：

```
# -*- coding:utf-8 -*-

from selenium import webdriver

import os

# firefox 实际安装路径

ffdriver = "C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe"

os.environ["webdriver.firefox.driver"] = ffdriver

driver = webdriver.Firefox(ffdriver)

# 注意 http 不可以省略

url='http://www.baidu.com'

driver.get(url)

driver.close()
```

3.1.2 启动 google 浏览器

需要下载相应的驱动，下载地址：

<http://chromedriver.storage.googleapis.com/index.html>

参考代码如下：

```
# -*- coding:utf-8 -*-

from selenium import webdriver

import os

# chromedriver.exe 实际安装路径, 笔者这里放置在 C 盘根目录

googledriver = "C:\\chromedriver.exe"
```

```
os.environ["webdriver.chrome.driver"] = googledriver
```

```
driver = webdriver.Chrome(googledriver)
```

```
# 注意 http 不可以省略
```

```
url='http://www.baidu.com'
```

```
driver.get(url)
```

```
driver.close()
```

3.1.3 启动 IE 浏览器

需要下载相应的驱动，下载地址：

<http://selenium-release.storage.googleapis.com/index.html>

参考代码如下：

```
# -*- coding:utf-8 -*-
```

```
from selenium import webdriver
```

```
import os
```

```
#iedriver.exe 实际安装路径, 笔者这里放置在 C 盘根目录
```

```
iedriver = "C:\\iedriver.exe"
```

```
os.environ["webdriver.ie.driver"] = iedriver
```

```
driver = webdriver.Chrome(iedriver)
```

```
# 注意 http 不可以省略
```

```
url='http://www.baidu.com'
```

```
driver.get(url)
```

driver.close()

3.2 python Selenium3 示例 - Page Object Model

3.2.1 前言

python Selenium3 是当前主流的 web 自动化测试框架，提供了多浏览器的支持（chrome、ie、firefox、safari 等等），同时支持多种编程语言来写用例（python、ruby、java 等等），非常容易上手，但当大家在深入应用时，会发现随着代码量的增加，感觉整个用例测试代码的维护会越来越庞大，例如：

```
1  from selenium import webdriver
2  from selenium.webdriver.common.keys import Keys
3  import unittest
4
5  class TcLogin(unittest.TestCase):
6      def setUp(self):
7
8          #webdriver
9          self.driver = webdriver.Chrome(r'C:\\chromedriver.exe')
10         self.driver.implicitly_wait(30)
11         self.base_url = "http://127.0.0.1:9000/"
12
13     def test_(self):
14         driver = self.driver
15         driver.get(self.base_url)
16
17         #enter username and password
18         driver.find_element_by_id("username").clear()
19         driver.find_element_by_id("username").send_keys("sb")
20         driver.find_element_by_id("password").clear()
21         driver.find_element_by_id("password").send_keys("dsfsfs"+Keys.RETURN)
22
23         #find dialog and check
24         dialogTitle = driver.find_element(By.XPATH, '//div/div[1]/h3')
25         self.assertEqual("Sign in", dialogTitle.text)
26
27         #find cancel button and click
28         cancelBtn = driver.find_element(By.XPATH, '//div/div[3]/button[2]')
29         cancelBtn.click()
30
31     def tearDown(self):
32         self.driver.close()
```

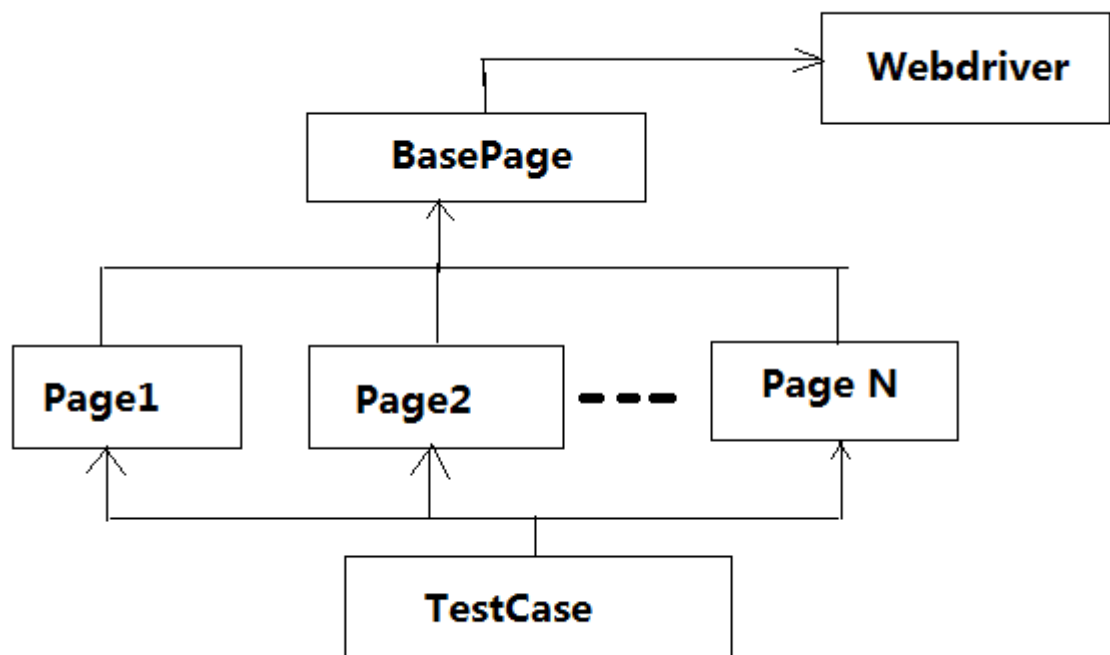
如上代码在随着进一步新增测试用例的情况会有以下几个问题：

1、易读性，一连串的 find element 会显得杂乱无章

- 2、可扩展不好：用例孤立，难以扩展
- 3、可复用性：无公共方法，很那复用
- 4、可维护性：一旦页面元素变化，需要维护修改大量的用例

3.2.2 Page 模式

基于上述问题，我们采用自动化测试的一种设计模式来进行一定层面的解决，这就是 Page 模式。什么是 Page 模式呢？它大概应该长成这样子，如图所示：



POM

Page 模式的核心要素：

- 1、抽象封装一个 BasePage 基类，基类应该拥有一个指向 webdriver 实例的属性

2、每一个 Page 都应该继承 BasePage,并通过 driver 来管理本 Page 的元素，且将 Page 才操作都封装成一个个的方法

3、TestCase 应该继成 unittest.TestCase 类，并依赖相应的 Page 类来实现相应的 test step（即测试步骤）

BasePage 代码示例如下：

```
# BaePage
```

```
class BasePage(object):
```

```
    def __init__(self, driver):
```

```
        self.driver = driver
```

登录 Page 代码示例如下：

```
class LoginPage(BasePage):
```

```
    # 登录 pange 元素维护
```

```
    username = (By.ID, "username")
```

```
    password = (By.ID, "pass")
```

```
    login_btn = (By.ID, "loginBtn")
```

```
    def set_username(self, username): # 输入用户名
```

```
        name= self.driver.find_element(*LoginPage.username)
```

```
        name.send_keys(username)
```

```
    def set_password(self, password): # 输入密码
```

```
        pwd= self.driver.find_element(*LoginPage.password)
```

```
        pwd.send_keys(password)
```

```
def click_login(self): # 单击登录

    loginbtn= self.driver.find_element(*LoginPage.login_btn)

    loginbtn.click()
```

TestCase 代码示例如下：

```
class Test_Login(unittest.TestCase):

    #Setup 初始化

    def setUp(self):

        self.driver = webdriver.Chrome(r'C:\\chromedriver.exe')

        self.driver.implicitly_wait(30)

        self.base_url = "http://127.0.0.1:9000/"

    #tearDown 清理

    def tearDown(self):

        self.driver.close()

    def test_Login(self):

        #Step1: 打开登录页

        self.driver.get(self.base_url)

        #Step2: 初始化登录 Page

        login_page = BasePage.LoginPage(self.driver)

        #Step3: 输入用户名

        login_page.set_username("admin")
```



```
#Step4: 输入密码
```

```
login_page.set_password("passwd")
```

```
#Step5: 单击登录按钮
```

```
login_page.click_login()
```

主运行程序代码：

```
if __name__ == "__main__":
```

```
    unittest.main()
```

至此整个 Page 模式演示就完成了。再来回顾下上述两种方式的代码组织，是不是 Page 模式的魅力更大？

3.2.3 结束语

Page 模式给我们提供了一种很好的设计模式，实现了用例和页面的分离，降低了耦合，提高了内聚，为后续更大规模的应用 python Selenium3 进行自动化测试提供了坚实的基础。

3.3 python Selenium3 示例 - 利用 excel 实现参数化

3.3.1 前言

在进行软件测试或设计自动化测试框架时，一个不可避免的过程就是：参数化，在利用 python 进行自动化测试开发时，通常会使用 excel 来做数据管理，利用 xlrd、xlwt 开源包来读写 excel。

3.3.2 环境安装

首先在命令行下安装 xlrd、xlwt

```
pip install xlrd
```

pip install xlwt

3.3.3 一个简单的读写示例

让我们先看一个简单的 excel 读写示例，示例代码功能，从表 1 中读取数据。

excel 数据表如图所示

	A	B	C	D
1	序号	数据1	数据2	数据3
2	1	11	sdf	fef
3	2	12	fsd	efe
4	3	13	dsfds	sdf
5	4	sds	fsdfs	sdf
6				
7				
8				
9				
10				

demo.xlsx

1、读取代码示例

```
#_*_ coding:utf-8 *_*
```

```
__author__ = '苦叶子'
```

```
import xlrd
```

```
if __name__ == '__main__':
```

```
    # excel 文件全路径
```

```
    xlPath = "C:\\Users\\lyy\\Desktop\\demo.xlsx"
```

```
    # 用于读取 excel
```

```
    xlBook = xlrd.open_workbook(xlPath)
```

```
    # 获取 excel 工作簿数
```

```
    count = len(xlBook.sheets())
```

```

print u"工作簿数为: ", count

# 获取 表 数据的行列数

table = xlBook.sheets()[0]

nrows = table.nrows

ncols = table.ncols

print u"表数据行列为(%d, %d)" % (nrows, ncols)

# 循环读取数据

for i in xrange(0, nrows):

    rowValues = table.row_values(i) # 按行读取数据

    # 输出读取的数据

    for data in rowValues:

        print data, "    ",

    print ""

```

运行效果如图



```

C:\Python27\python.exe
工作簿数为: 1
表数据行列为(5, 4)
序号      数据1      数据2      数据3
1.0      11.0      sdf      fef
2.0      12.0      fsd      efe
3.0      13.0      dsfds    sdf
4.0      sds      fsdfs    sdf
>>>

```

运行结果

2、写 excel 示例代码

```

#_*_ coding:utf-8 *_

__author__ = '苦叶子'

import xlwt

import random

if __name__ == '__main__':

    # 注意这里的 excel 文件的后缀是 xls 如果是xlsx 打开是会提示无效

    newPath = unicode("C:\\Users\\lly\\Desktop\\demo_new.xls",
"utf8")

    wtBook = xlwt.Workbook()

    # 新增一个 sheet

    sheet = wtBook.add_sheet("sheet1", cell_overwrite_ok=True)

    # 写入数据头

    headList = [u'序号', u'数据 1', u'数据 2', u'数据 3']

    rowIndex = 0

    col = 0

    # 循环写

    for head in headList:

        sheet.write(rowIndex, col, head)

        col = col + 1

    # 写入 10 行 0-99 的随机数据

```

```

for index in xrange(1, 11):

    for col in xrange(1, 4):

        data = random.randint(0,99)

        sheet.write(index, 0, index) # 写序号

        sheet.write(index, col, data) # 写数据

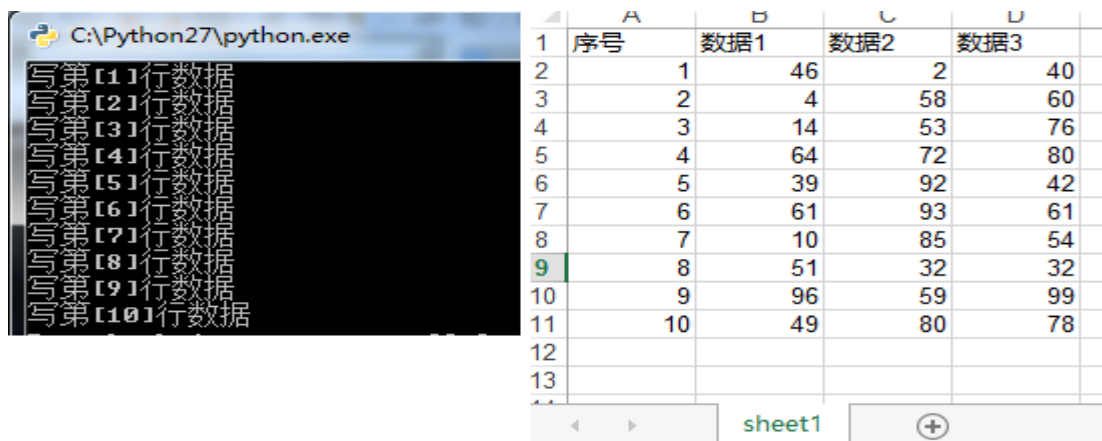
    print u"写第[%d]行数据" % index

# 保存

wtBook.save(newPath)

```

运行结果如图



	A	B	C	D
1	序号	数据1	数据2	数据3
2	1	46	2	40
3	2	4	58	60
4	3	14	53	76
5	4	64	72	80
6	5	39	92	42
7	6	61	93	61
8	7	10	85	54
9	8	51	32	32
10	9	96	59	99
11	10	49	80	78
12				
13				

写 excel 结果

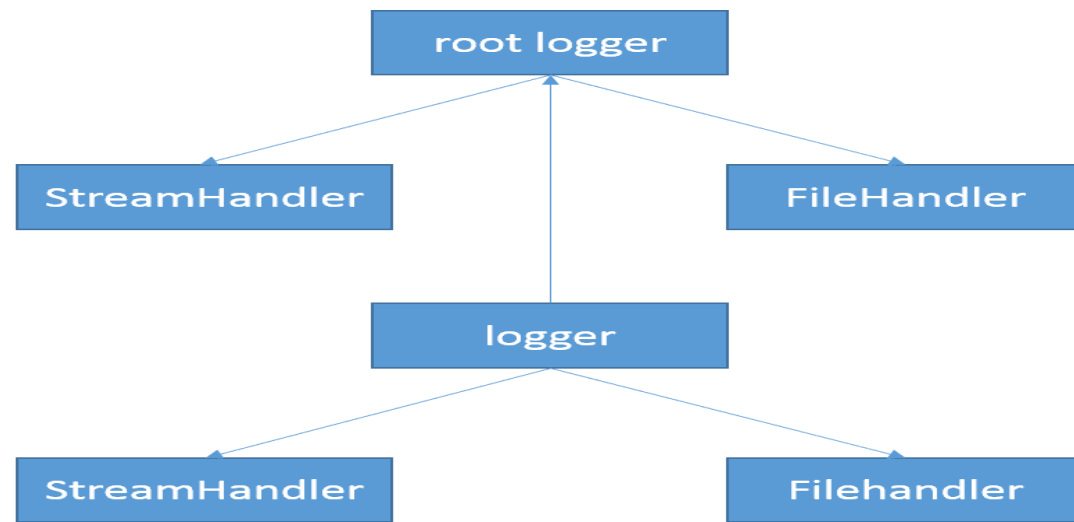
3.3.4 结束语

这里只是简单的对 xlrd、xlwt 模块的应用演示，对于实际做自动化测试过程中，需要封装一个通用的 excel 解析类，以便提高复用性和降低维护成本。

在实际应用中，我们通常需要对表格进行合并、样式设置等等系列动作，请参考官方文档，进行更深入的学习研究

python excel 官网： <http://www.python-excel.org/>

3.4 python Selenium3 示例 - 日志管理



logger 继承图

3.4.1 前言

在自动化测试实践过程中，必不可少的就是进行日志管理，方便调试和生产问题追踪，python 提供了 logging 模块来进行日志的管理。下面我们就 logging 模块的学习和使用进行一个层层推进演示学习。

Python 的 logging 模块提供了通用的日志系统，可以方便第三方模块或应用使用。这个模块提供了不同的日志级别，并可以采用不同的方式进行日志记录，比如文件，HTTP GET/POST，SMTP，socket 等等，甚至可以自定义实现具体的日志记录方式。

logging 模块与 java 的 log4j 的机制是一样的，只是具体的语言实现细节有些不同。python logging 模块提供了 logger、handler、filter、formatter 等基础类。

1、logger：提供日志接口，供应用程序调用。logger 最常用的操作有两大类：配置和发送日志消息。

2、handler：将日志记录发送到合适的目的，比如文件、socket 等等。一个 logger 对象可以通过 addhandler 方法添加 0 到 N 个 handler，每个 handler 又可以定义不同的日志级别，以实现日志分级过滤。

3、filter：提供了一种优雅的方式决定一个日志记录是否发送到 handler。

4、formatter：指定日志记录的输出格式。formatter 的构造方法需要两个参数：消息的格式字符串和日期字符串，这两个参数是可选的。

默认情况下，logging 将日志输出至 console，日志级别为 WARNING。

logging 中按日志级别大小关系为 CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET，当然也可以自定义日志级别。

3.4.2 简单日志

下面我们看一下一个简单的日志示例，将日志记录输出到 console：

```
#-*- coding:utf-8 -*-

import logging

if __name__ == '__main__':

    logging.debug(u'这是 bug 级别日志记录')

    logging.info(u'这是提示信息级别日志记录')

    logging.warning(u'这是警告级别日志记录')
```

在 console 中将输出一下信息：

WARNING:root:这是警告级别日志记录

为什么只输出了一条呢？因为 logging 默认情况下的日志输出级别是：

WANRING

3.4.3 日志格式和级别控制

接下来我们看看如何控制日志的输出格式和日志级别。代码示例如下：

```
#-*- coding:utf-8 -*-

import logging

if __name__ == '__main__':

    logging.basicConfig(level=logging.DEBUG, # 日志级别设置

        format="%(asctime)s %(filename)s

        [line: %(lineno)d] %(levelname)s %(message)s",

        datefmt='%a, %d %b %Y %H:%M:%S',

        filename='mylog.log',

        filemode='w')

    logging.debug(u'这是 debug 级别日志记录')

    logging.info(u'这是信息级别日志记录')

    logging.warning(u'这是警告级别日志记录')
```

在当前目录下 mylog.log 文件中的内容为：

Mon, 20 Mar 2017 16:21:28 log.py [line: 14] DEBUG 这是 debug 级别日志记录

Mon, 20 Mar 2017 16:21:28 log.py [line: 15] INFO 这是信息级别日志记录

Mon, 20 Mar 2017 16:21:28 log.py [line: 16] WARNING 这是警告级别日志记录

logging.basicConfig 函数各参数说明

filename：指定日志输出文件名

filemode：和 file 函数的意义相同，指定日志文件的打开模式，‘w 或 a’

format：指定日志输出格式和内容，format 可以输出很多有用的信息，如上例所示：

%(levelno)s: 打印日志级别的数值

%(levelname)s: 打印日志级别名称

%(pathname)s: 打印当前执行程序的路径，其实就是 sys.argv[0]

%(filename)s: 打印当前执行程序名

%(funcName)s: 打印日志的当前函数

%(lineno)d: 打印日志的当前行号

%(asctime)s: 打印日志的时间

%(thread)d: 打印线程 ID

%(threadName)s: 打印线程名称

%(process)d: 打印进程 ID

%(message)s: 打印日志信息

datefmt : 指定时间格式 , 同 time.strftime()

level : 指定日志级别 , 默认为 logging.WARNING

stream : 指定日志的输出流 , 可以指定输出到 sys.stderr, sys.stdout 或文件 , 默认输出到 sys.stderr , 当 stream 和 filename 同时指定时 , stream 被忽略。

3.4.4 日志输入定向

下面我们来看看如何把日志同时输出到 console 和文件中 , 代码示例如下 :

```
#-*- coding:utf-8 -*-

import logging

if __name__ == '__main__':

    logging.basicConfig(level=logging.DEBUG, # 日志级别设置

        format="%(asctime)s %(filename)s

[line: %(lineno)d] %(levelname)s %(message)s",

        datefmt='%a, %d %b %Y %H:%M:%S', filename='mylog.log',

        filemode='w')

    #####

    # 定义一个 StreamHandler , 将 info 级别的或更高级别的日志输出到标

    错错误

    # 并将其添加到当前的日志处理对象

    console = logging.StreamHandler()
```

```
console.setLevel(logging.INFO)

formatter = logging.Formatter('%(name)-12s: %(levelname)-
8s %(message)s')

console.setFormatter(formatter)

logging.getLogger('').addHandler(console)

#####

logging.debug(u"这是 debug 日志记录")

logging.info(u'这是 info 日志记录')

logging.warning(u'这是 warning 日志记录')
```

在 console 中输出以下日志记录：

root: INFO 这是 info 日志记录

root: WARNING 这是 warning 日志记录

在当前目录下 mylog.log 文件中内容为：

Mon, 20 Mar 2017 17:32:43 log.py [line: 26] DEBUG 这是 debug 日志记录

Mon, 20 Mar 2017 17:32:43 log.py [line: 27] INFO 这是 info 日志记录

Mon, 20 Mar 2017 17:32:43 log.py [line: 28] WARNING 这是 warning 日志记录

在本示例中实现了根据不同需要，将不同级别的日志重定向输出至不同的目标。

3.4.5 日志配置

在上述所有的示例中，日志的配置都是在代码中实现，但在实际的应用过程中，我们一般都需要动态的配置日志信息，或是满足自定义的需要，下面我们就自定义日志配置进行示例演示：

定义一个配置文件，这里命名为 logger.conf，为标准的 INI 格式的文件，内容如下

```
#####
```

下面定义了三个 logger： root,demo01,demo01

```
[loggers]
```

```
keys=root,demo01,demo01
```

```
[logger_root]
```

```
level=DEBUG
```

```
handlers=hand01,hand02
```

```
[logger_demo01]
```

```
handlers=hand01,hand02
```

```
qualname=demo01
```

```
propagate=0
```

```
[logger_demo02]
```

```
handlers=hand01,hand03
```

```
qualname=demo02
```

```
propagate=0
```

```
#####
```

```
下面定义了三个 handler : hand01,hand02,hand03
```

```
[handlers]
```

```
keys=hand01,hand02,hand03
```

```
[handler_hand01]
```

```
class=StreamHandler
```

```
level=INFO
```

```
formatter=form02
```

```
args=(sys.stderr,)
```

```
[handler_hand02]
```

```
class=FileHandler
```

```
level=DEBUG
```

```
formatter=form01
```

```
args=('mylog.log', 'a')
```

```
[handler_hand03]
```

```
class=handlers.RotatingFileHandler
```

```
level=INFO
```

```
formatter=form02
```

```
args=('mylog.log', 'a', 10*1024*1024, 5)
```

```
#####
```

下面定义了两种 formatter : form01,form02

```
[formatters]
```

```
keys=form01,form02
```

```
[formatter_form01]
```

```
format=%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s
```

```
datefmt=%a, %d %b %Y %H:%M:%S
```

```
[formatter_form02]
```

```
format=%(name)-12s: %(levelname)-8s %(message)s
```

```
datefmt=
```

使用 demo01 logger 代码示例 :

```
#!/usr/bin/python3 # coding:utf-8
```

```
import logging
```

```
import logging.config
```

```
if __name__ == '__main__':
```

```
    logging.config.fileConfig("logger.conf")
```

```
    logger = logging.getLogger("demo01")
```

```
    logger.debug(u'这是 demo01 debug 日志记录')
```

```
    logger.info(u'这是 demo01 info 日志记录')
```

```
logger.warning(u'这是 demo01 warning 日志记录')
```

下面是使用 demo02 logger 代码示例：

```
#-*- coding:utf-8 -*-
```

```
import logging
```

```
import logging.config
```

```
if __name__ == '__main__':
```

```
    logging.config.fileConfig("logger.conf")
```

```
    logger = logging.getLogger("demo02")
```

```
    logger.debug(u'这是 demo02 debug 日志记录')
```

```
    logger.info(u'这是 demo02 info 日志记录')
```

```
    logger.warning(u'这是 demo02 warning 日志记录')
```

3.4.6 结束语

本文从日志的基本应用到更高级的应用方式层层推进进行演示，当然了在实际的自动化测试实践中，还需要对 logging 模块进行更高级的封装以提高其复用性，达成高可用的目的。对于测试人员而言更需要加强编程基本功，提升测试技术能力，更加灵活的应用各种基础技术。

3.5 python Selenium3 示例 - 同步机制

3.5.1 前言

在使用 python Selenium3 进行自动化测试实践的过程中，经常会遇到元素定位不到，弹出框定位不到等等各种定位不到的情况，在大多数的情况下，无非是以下两种情况：

- 1、有 frame 存在，定位前，未 switch 到对应的 frame 内
- 2、元素未加载完毕（从界面看已经显示），但 DOM 树还在 load 状态或在加载 js

那对于这类情况，怎么解决呢？

通俗的讲法：等待。

高大上点：解决自动化测试代码与浏览器加载渲染之间的同步问题。

下面我们分段讲述各种处理方式：

3.5.2 强制等待

这种方式简单粗暴直接有效，不足的地方就是不够灵活。下面看下代码片段：

```
#_*_ coding:utf-8 *_*  
  
__author__ = '苦叶子'  
  
from selenium import webdriver  
  
from time import sleep # 注意  
  
if __name__ == '__main__':  
  
    driver = webdriver.Firefox()  
  
    driver.get('http://www.testingunion.com')  
  
    sleep(3) # 强制等待 3s 在执行下一步  
  
    print u"当前 url: ", driver.current_url  
  
    sleep(3) # 强制等待 3s 在执行下一步  
  
    driver.quit()
```


注： 请注意加粗有删除线的代码行，用于实现强制等待

3.5.3 隐性等待

webdriver 提供了基础的同步方法，隐性等待 `implicitly_wait(xx)`，该方式的含义是：不论业务代码运行在那一步，都需要等待 webdriver xx 秒，如果在等待的 xx 秒内 webdriver 完成了对应的动作，则业务代码和 webdriver 都正常继续执行；如果超过了 xx 秒，webdriver 还未完成对应的动作，则业务代码继续执行，而 webdriver 则会抛出异常（例如 timeout 或元素未找到等等异常），请看代码实现片段：

```
#_*_coding:utf-8_*_  
  
__author__ = '苦叶子'  
  
from selenium import webdriver  
  
if __name__ == '__main__':  
  
    driver = webdriver.Firefox()  
  
    driver.implicitly_wait(30) # 隐性等待，最长等 30s  
  
    driver.get('http://www.testingunion.com')  
  
    print u"当前 url： ", driver.current_url  
  
    driver.quit()
```

注：上述代码中加粗删除线的代码通过调用 webdriver 提供的标准隐性等待方式来实现一种同步机制。其设置的是一个最长等待时间，如果在规定的时间未完成，则进入下一步。

不足：在实践中，通常我们需要操作的元素已经显示出来，但因网络或其他因素，浏览器一直处于加载个别 js 或图片或其他资源时，隐性等待模式下，这时会依旧处于等待状态直至页面全部加载完毕才能进入下一步。那有没有更好的办法呢？当然是有的，请参见下一方式。

重要：隐性等待是全局性质的，只需在 driver 实例化后，设置一次即可。

在实践中，经常见到新手把隐性等待当做 sleep 来使用，在每个步骤后都用一次。

3.5.4 显性等待

更为强大的方式是显性等待来实现同步机制，需要 WebDriverWait 类，辅以 until()或 until_not()方法，根据判断条件进行灵活的同步，它的主要机制是：程序在规定的时间内每个 xx 秒看一下判断条件是否成立，如果成立则执行下一步，否则继续等待，直至超过设置的最长时间，然后抛出异常。请看具体的代码片段：

```
#_*_coding:utf-8_*
```

```
__author__ = '苦叶子'
```

```
from selenium import webdriver
```

```
from selenium.webdriver.support.wait import WebDriverWait
```

```
from selenium.webdriver.support import expected_conditions as EC
```

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

```
if __name__ == '__main__':
```

```
    driver = webdriver.Firefox()
```

```

# 隐性等待和显性等待可以同时用，

# 要注意的是：最大等待时间取决两者之间的大值

driver.implicitly_wait(10)

driver.get('http://www.testingunion.com')

locator = (By.LINK_TEXT, u'webdriver')

try:

    # 在最长 20s 内，每个 0.5 秒去检查 locator 是否存在，

    # 如果存在则进入下一步

    WebDriverWait(driver, 20,

0.5).until(EC.presence_of_located(locator))

    # 提取该文本对应的 url，并打印出来

    print driver.find_element_by_link_text(u'webdriver'

    ).get_attribute('href')

finally:

    print u"异常了"

driver.quit()

```

在本示例中，我们设置了隐性等待和显性等待，在其他的操作中隐性等待起决定性作用，在示例中的 `WebDriverWait` 设置了显性等待的地方，则显性等待起决定性作用，但要注意的是：最长等待时间取决于两者之间的大值，所以这里显性等待的最长时间为 20s。

在这里我们主要用到了 WebDriverWait 类和 expected_conditions 模块，让我们近距离的接触下它们。

3.5.5 WebDriverWait 类

定义实现在 wait 模块中，实现了 webdriver 的显性等待机制，先看下它有哪些参数和方法：

```
selenium.webdriver.support.wait.WebDriverWait ( 类 )
```

```
__init__(self,
```

```
driver, # 传入实例化的 webdriver 对象
```

```
timeout, # 超时时间，等待的最大时间（需考虑同时考虑隐性等待时间）
```

```
poll_frequency=POLL_FREQUENCY, # 调用 until 或 until_not 方法的间隔  
时间，上例为 0.5s
```

```
ignored_exceptions=None #指定忽略的异常，如果指定了要忽略的异常类  
型，则在调用 until 或 until_not 过程中，捕获该类异常时不中断代码，继续等  
待。默认只有 NoSuchElementException  
)
```

```
until(self,
```

```
method, # 在等待期间，每个一段时间调用这个传入的方法，直到返回值为  
false
```

```
message="" # 如果超时，则抛出 TimeoutException，将 message 传入给异  
常
```

)

until_not 与 until 相反，until 是当某个元素满足某种条件时（出现、存在等等）继续执行；until_not 则是当某个元素不满足某种条件时继续执行，参数含义相同

特别注意：

很多时候大家在使用 until 或 until_not 时，会将一个 WebElement 对象传入给 method，如下：

```
WebDriverWait(driver, 10).until(driver.find_element_by_id('kw'))
```

这是错误的用法

这里的参数一定要是可调用的，这个对象一定要有__call__()方法，否则会抛出异常：

```
TypeError: 'xxx' object is not callable
```

在这里，也可以用 Selenium3 提供的 expected_conditions 模块中提供的各种条件，也可用 WebElement 中的 is_displayed()，is_selected()，is_enabled()方法或是自己封装的方法均可。下面我们再看看 Selenium3 提供了哪些条件，如图所示：



条件判断

结束语

本文就 python Selenium3 三种同步解决方式进行了较为详细的说明，这是使用 Selenium3 进行自动化测试实践的必备技能，希望对大家有所帮助，有任何问题请关注公众号号，直接回复消息进行交流。

3.6 python Selenium3 示例 - SSL 处理

3.6.1 前言

随着现在站点对安全的要求越来越高，越来越多的企业网站接入了 https，随着 https 的大规模应用，我们在使用 python Selenium3 进行自动化测试时，也要面临的挑战。

3.6.2 面临的问题

在实际的自动化测试实践中，因为越来越多的站点接入 https，使得我们原有的 python Selenium3 自动化测试代码进行测试时，浏览器总是报安全问题，即便在浏览器选项中将被测网址加入信任网址也没用。

一般情况下，我们访问 http 站点时的代码如下：

```
driver = webdriver.Firefox()
```

```
driver.get(u'http://www.testingunion.com')
```

一般情况下，这样处理是正常，但如果目标 url 是 HTTPS 访问模式，则浏览器会提示安全问题或是非信任站点。

在不同的浏览器上显示的提示如图所示（这里以英文版的浏览器为准）：



浏览器 SSL 提示

我们看一下 IE 的解决方案，对 ie 浏览器而言，需要添加 Desired Capabilities 的 acceptSslCerts 选项为 True，代码如下：

的

```

#*_ coding:utf-8 *_

__author__ = '苦叶子'

from selenium import webdriver

if __name__ == '__main__':

    capabilities = webdriver.DesiredCapabilities().INTERNETEXPLORER

    capabilities['acceptSslCerts'] = True

    driver = webdriver.Ie(capabilities=capabilities)

    driver.get(u'https://cacert.org/')

    print driver.title


    driver.quit()

```

对于 firefox 浏览器则需要添加 FirefoxProfile()的 accept_untrusted_certs 的选项为 True , 示例代码如下 :

```

#*_ coding:utf-8 *_

__author__ = '苦叶子'

from selenium import webdriver

if __name__ == '__main__':

    profile=webdriver.FirefoxProfile()

    profile.accept_untrusted_certs=True

    driver=webdriver.Firefox(firefox_profile=profile)

```



```
driver.get(u'https://cacert.org/')
```

```
driver.close()
```

对于 chrome 浏览器则需要添加 ChromeOptions()的--ignore-certificate-errors 选项为 True , 示例代码如下 :

```
#_*_ coding:utf-8 _*_
```

```
__author__ = '苦叶子'
```

```
from selenium import webdriver
```

```
if __name__ == '__main__':
```

```
    options=webdriver.ChromeOptions()
```

```
    options.add_argument('--ignore-certificate-errors')
```

```
    driver=webdriver.Chrome(chrome_options=options)
```

```
    driver.get(u'https://cacert.org/')
```

```
    driver.close()
```

3.6.3 结束语

对于在利用上述方式针对不同浏览器处理 SSL 时 , 可能还会碰到还是处理不了的情况 , 比如提示证书损坏、无效等等 ; 如果出现这类情况 , 请联系网站管理员更新 SSL 证书。

3.7 python Selenium3 示例 - email 发送

3.7.1 前言

在进行日常的自动化测试实践中，我们总是需要将测试过程中的记录、结果等等相关信息通过自动的手段发送给相关人员。python 的 smtplib、email 模块为我们提供了很好的 email 发送等功能的实现。

3.7.2 纯文本邮件

在通常情况下，我们需要发送大量的纯文本类的邮件通知，或是发送概要性测试报告时，会用到此类发送方式，示例代码如下：

```
#-*- coding:utf-8 -*-

__author__ = u'苦叶子'

import smtplib

from email.mime.text import MIMEText

from email.header import Header

if __name__ == '__main__':

    sender = u'sender@163.com' # 发送人邮件地址

    receiver = u'receiver@163.com' # 接收人邮件地址

    subject = u'python email 文本邮件发送测试'

    smtpserver = u'smtp.163.com' # smtp 服务

    username = u'testname' # 发送人邮件用户名或专用于 smtp 账户用户名
```

```
password = u'testpassword' # 发送人邮件密码或专用于 smtp 账户的密码
```

```
msg = MIMEText(u'你好', 'text', 'utf-8') # 文本格式邮件 正文内容
```

```
msg['Subject'] = Header(subject,'utf-8') # 邮件标题
```

```
smtp = smtplib.SMTP() # 初始化一个 smtp 对象
```

```
smtp.connect('smtp.163.com') # 连接至 smtp 服务器
```

```
smtp.login(username, password) # 登录 smtp 服务
```

```
smtp.sendmail(sender, receiver, msg.as_string())# 发送邮件
```

```
smtp.quit() # 发送完成后关闭连接
```

3.7.3 HTML 形式的邮件

通常情况下，我们经常生成 html 格式的测试报告或记录，如果采用文本邮件方式发送，则 html 格式的报告或记录会将 html 标签也显示出来，那么为了让邮件接收者能够正常的看到 html 格式的报告，则需要在邮件发送时，对相应的参数进行配置，以便邮件客户端能正常解析 html 格式的邮件，示例如下：

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
import smtplib
```

```
from email.mime.text import MIMEText
```

```
from email.header import Header
```

```
if __name__ == '__main__':
```

```
    sender = u'sender@163.com' # 发送人邮件地址
```

```

receiver = u'receiver@163.com' # 接收人邮件地址

subject = u'python email HTML 形式邮件发送测试'

smtpserver = u'smtp.163.com' # smtp 服务

username = u'testname' # 发送人邮件用户名或专用于 smtp 账户用户
名

password = u'testpassword' #发送人邮件密码或专用于 smtp 账户的密
码

msg = MIMEText(u'<html> <h1>你好，这是 html 格式的邮件，哇咋
咋</h1> </html>', 'html', 'utf-8') # html 格式邮件

msg['Subject'] = Header(subject,'utf-8') # 邮件标题

smtp = smtplib.SMTP()# 初始化一个 smtp 对象

smtp.connect('smtp.163.com') # 连接至 smtp 服务器

smtp.login(username, password) # 登录 smtp 服务

smtp.sendmail(sender, receiver, msg.as_string()) # 发送邮件

smtp.quit() # 发送完成后关闭连接

```

3.7.4 带附件的邮件

文本和 html 格式的邮件能满足您的需要嘛？仔细回顾下，测试过程中是不是还有很多的附件要进行发送？在自动化测试过程中是不是有很多截图？等等....

是的，我们还需要发送带附件的邮件来满足我们日常的测试需要，下面看看带附件的邮件发送示例:

```
#-*- coding:utf-8 -*-
```

```

__author__ = u'苦叶子'

import smtplib

from email.mime.text import MIMEText

from email.header import Header

if __name__ == '__main__':

    sender = u'sender@163.com' # 发送人邮件地址

    receiver = u'receiver@163.com' # 接收人邮件地址

    subject = u'python email 附件邮件发送测试'

    smtpserver = u'smtp.163.com' # smtp 服务

    username = u'testname' # 发送人邮件用户名或专用于 smtp 账户用户
名

    password = u'testpassword' # 发送人邮件密码或专用于 smtp 账户的密
码

    msg = MIMEText(open('C:\\1.jpg','rb').read(),'base64','utf-8') # 读取附
件

    msg['Subject'] = Header(subject,'utf-8')

    #构造附件

    att = MIMEText(open('C:\\1.jpg','rb').read(),'base64','utf-8') # 读取附
件

    att["Content-Type"] = 'application/octet-stream'

    att["Content-Disposition"] = 'attachment; filename="1.jpg"'

```

```

msg.attach(att) # 关联附件

#####

smtp = smtplib.SMTP()

smtp.connect('smtp.163.com')

smtp.login(username, password)

smtp.sendmail(sender, receiver, msg.as_string())

smtp.quit()

```

3.7.5 群发邮件

在上述几个示例中，所有的邮件接收都是单个人，实际的应用中，我们则需要给一群人进行邮件发送，下面看看示例：

```

#-*- coding:utf-8 -*-

__author__ = u'苦叶子'

import smtplib

from email.mime.text import MIMEText

from email.header import Header

if __name__ == '__main__':

    # 发送人邮件地址

    sender = u'sender@163.com'

    # 群发接收人邮件地址 !!!!!

    receiver = [u'receiver@163.com', u'**@xx.com', u'**@yy.com']

```

```
# 邮件标题

subject =u'python email 群发邮件发送测试'

# smtp 服务

smtpserver =u'smtp.163.com'

# 发送人邮件用户名或专用于 smtp 账户用户名

username =u'testname'

# 发送人邮件密码或专用于 smtp 账户的密码

password =u'testpassword'

# 文本格式邮件 正文内容

msg = MIMEText(u'你好群发','text','utf-8')

# 邮件标题

msg['Subject'] = Header(subject,'utf-8')

# 初始化一个 smtp 对象

smtp = smtplib.SMTP()

# 连接至 smtp 服务器

smtp.connect('smtp.163.com')

# 登录 smtp 服务

smtp.login(username, password)

# 发送邮件

smtp.sendmail(sender, receiver, msg.as_string())
```

```
# 发送完成后关闭连接
```

```
smtp.quit()
```

3.7.6 综合示例

在上述所有的示例都是按功能分类来进行一一演示，接下来的示例，则是包含了上述所有功能：

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
import smtplib
```

```
from email.mime.multipart import MIMEMultipart
```

```
from email.mime.text import MIMEText
```

```
from email.image import MIMEImage
```

```
if __name__ == '__main__':
```

```
    # 定义一些连接数据
```

```
    sender = u"DeepTest@xx.com"
```

```
    receiver = [u'xxx@xx.com', u'xx@yy.com']
```

```
    subject = u"邮件综合示例"
```

```
    username = u"username"
```

```
    password = u"password"
```

```
    # 创建 message
```

```
    msg = MIMEMultipart('alternative')
```



```

msg['Subject'] = u"测试"

# 发送内容

text = u"你好，这是文本内容"

html = u"""

    测试报告

    测试结果概述

    """

# 添加 MIME 类型

partText = MIMEText(text, u'plain')

partHTML = MIMEText(html, u'html')

msg.attach(partText)

msg.attach(partHTML)

#构造附件

attach = MIMEText(open('c:\\demo.jpg').read(),'base64', 'utf-8')

attach['Content-Type'] = 'application/octet-stream'

attach['Content-Disposition'] = 'attachment;filename="demo.jpg"'

msg.attach(attach)

# 发送邮件

smtp = smtplib.SMTP()

smtp.connect('smtp.163.com')

```

```
smtp.login(username,password)

smtp.sendmail(sender, receiver, msg.as_string())

smtp.quit()
```

3.7.7 结束语

本文从文本邮件、html 格式邮件、附件邮件以及三者综合一起使用的方式阐述了利用 python email 模块进行邮件发送。

3.8 python Selenium3 示例 - 生成 HTMLTestRunner 测试报告

3.8.1 前言

在 python Selenium3 自动化测试过程中，一个合适的报告是必须的，而 HTMLTestRunner 模块为我们提供了一个很好的报告生成功能。

3.8.2 [什么是 HTMLTestRunner](#)

HTMLTestRunner 是 Python 标准库的 unittest 模块的一个扩展。它生成优美的 HTML 格式测试报告

3.8.3 [HTMLTestRunner 安装](#)

下载地址：<https://pypi.python.org/pypi/HTMLTestRunner>

放在当前 python 项目中，当做自己的一个模块。

3.8.4 [应用示例](#)

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
from selenium import webdriver
```

```

import unittest

import HTMLTestRunner

import sys

from time import sleep

reload(sys)

sys.setdefaultencoding("utf-8")

class BaiduTest(unittest.TestCase):

    """百度首页搜索测试用例"""

    def setUp(self):

        self.driver = webdriver.Chrome()

        self.driver.implicitly_wait(30)

        self.base_url = u"http://www.baidu.com"

    def test_baidu_search(self):

        driver = self.driver

        print u"开始[case_0001]百度搜索"

        driver.get(self.base_url)

        # 验证标题

        self.assertEqual(driver.title, u"百度一下，你就知道")

        driver.find_element_by_id("kw").clear()

        driver.find_element_by_id("kw").send_keys(u"开源优测")

```

```

        driver.find_element_by_id("su").click()

        sleep(3)

        # 验证搜索结果标题

        self.assertEqual(driver.title, u"开源优测_百度搜索")

    def tearDown(self):

        self.driver.quit()

if __name__ == '__main__':

    testunit = unittest.TestSuite()

    testunit.addTest(BaiduTest('test_baidu_search'))

    # 定义报告输出路径

    htmlPath = u"c:\\testReport.html"

    fp = file(htmlPath, "wb")

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp, title=u"百度
测试", description=u"测试用例结果")

    runner.run(testunit)

    fp.close()

```

3.8.5 报告效果

百度测试

Start Time: 2017-03-21 22:15:45
Duration: 0:00:27.587000
Status: Pass 1

测试用例结果

Show [Summary](#) [Failed](#) [All](#)

Test Group/Test case	Count	Pass	Fail	Error	View
BaiduTest: 百度首页搜索测试用例	1	1	0	0	Detail
test_baidu_search	<div><div>pass</div><div>pt1.1: 开始[case_0001]百度搜索</div></div>				
Total	1	1	0	0	

四、单元测试

4.1 基于 unittest 集成你的 Selenium3 测试

4.1.1 前言

python 单元测试框架 (The Python Unit Testing Framework) 简称 PyUnit , 是 JUnit 的 python 版本 , 自 python2.1 版本后 , PyUnit 已经成为了 Python 的标准库。下面我们就如何把 unittest 应用到 python Selenium3 自动化测试中进行分享。

4.1.2 测试用例

单元测试是由一系列的测试用例(Test Cases)构成。测试用例是被设置用来检测独立场景的集合。在 PyUnit 中 , unittest 模块中的 TestCase 类代表测试用例。

TestCase 类的实例是可以完全运行所有的方法和可选的初始化(setUp)及清理 (tearDown) 方法的对象。

TestCase 实例的测试代码必须是包含一个或多个测试方法 , 简单说 , 它可以单独运行或与其他任意数量的用例共同运行

4.1.3 简单示例

将以下代码保存到 first_webdriver.py 中

```
#-*- coding:utf-8 -*-
```

```
__author__ = u'苦叶子'
```

```
from selenium import webdriver
```

```
import unittest
```

```
import HTMLTestRunner

import sys

from time import sleep

reload(sys)

sys.setdefaultencoding("utf-8")


class BaiduTest(unittest.TestCase):

    """百度首页搜索测试用例"""

    def setUp(self):

        self.driver = webdriver.Ie()

        self.driver.implicitly_wait(30)

        self.base_url = u"http://www.baidu.com"


    def test_baidu_search(self):

        driver = self.driver

        print u"开始[case_0001]百度搜索"

        driver.get(self.base_url)


        # 验证标题

        self.assertEqual(driver.title, u"百度一下，你就知道")
```

```

        driver.find_element_by_id("kw").clear()

        driver.find_element_by_id("kw").send_keys(u"开源优测")

        driver.find_element_by_id("su").click()

        sleep(3)

        # 验证搜索结果标题

        self.assertEqual(driver.title, u"开源优测_百度搜索")

    def tearDown(self):

        self.driver.quit()

if __name__ == '__main__':

    testunit = unittest.TestSuite()

    testunit.addTest(BaiduTest('test_baidu_search'))

    # 定义报告输出路径

    htmlPath = u"testReport.html"

    fp = file(htmlPath, "wb")

    runner = HTMLTestRunner.HTMLTestRunner(stream=fp,

        title=u"百度测试",

        description=u"测试用例结果")

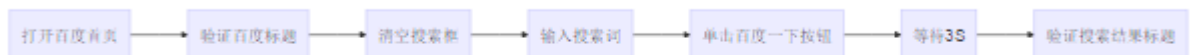
    runner.run(testunit)

    fp.close()

```


4.1.4 关键代码说明

1. 在上例代码中，我们的测试用例 BaiduTest 继承了 unittest.TestCase。
2. 在初始化方法 setUp 中，初始化了 webdriver 示例和隐性等待设置，并初始化了百度首页 url。该方法自动执行。
3. 在清理方法中 tearDown 中，退出了 webdriver。该方法自动执行。
4. test_baidu_search 方法是我们主要的测试方法，在该方法中进行了一下动作。



u1.png

4.1.5 主入口说

1. 先定义一个测试套件集，然后将用例集添加至套件中。
2. 我们使用了 unittest 第三方测试报告模块 HTMLTestRunner (该模块不是标准库，需要自己去下载，请参考前面几篇的文章)，用以自动生产 HTML 格式的测试报告。

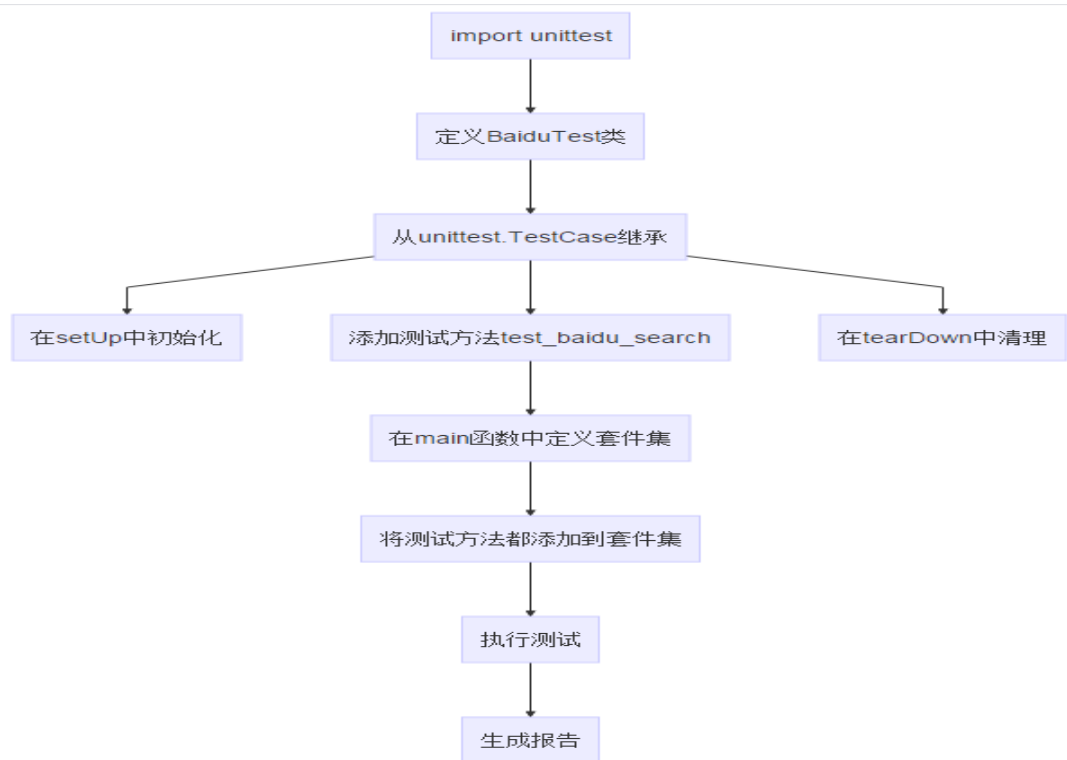
如何运行上述代码

将上述代码保存至 first_unit_test.py 中

在命令行中输入以下命令

```
python first_unit_test.py
```

4.1.6 代码组织说明



u2.png

4.1.7 总结

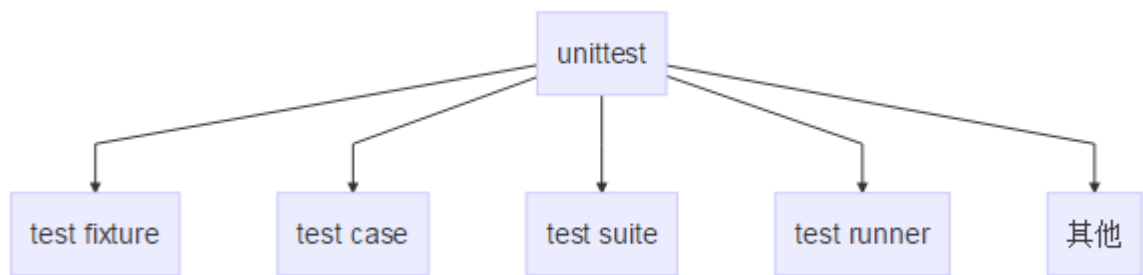
今天我们就 unittest 怎么和 selenium 测试进行结合做了初步的分享，请大家深入理解该示例，并实际动手练习代码。

4.2 python unittest 使用基本过程

4.2.1 前言

unittest 是 python 的标准的单元测试框架，能够很好的和自动化测试相结合，并有独立的测试报告框架。

unittest 提供了一系列类让测试变得更加容易，下面我们看下 unittest 的主要构成部分



unittest-1.png

1. test fixture

用于初始化、清理等动作。在 selenium 测试中，我们可以用来做 webdriver 的初始化等等

2 testcase

测试用例，unittest 的最小单元。用以对指定输入的返回结果进行检测。在 unittest 中提供了 TestCase 基类，用来创建新的测试用例类。

3 test suite

测试套件，一系列测试用例或测试套件的集合。在 unittest 中由 TestSuite 类实现。

4 test runner

测试执行器，负责用例执行并生成测试报告，在 unittest 中提供了命令行模式和 GUI 模式来执行。

4.2.2 unittest 使用过程

下面一步步的展示如何使用 unittest 来测试。

1. 导入 unittest 模块

```
import unittest
```

2 定义一个被测试函数

```
def add(a, b):
```

```
    return a + b
```

3 创建一个 unittest.TestCase 子类

```
class demoTest(unittest.TestCase):
```

```
    pass
```

4 在 demoTest 新增一个测试方法:test_add_4_5, 测试方法名称必须以 ==test==开始

```
class demoTest(unittest.TestCase):
```

```
    def test_add_4_5(self):
```

```
        pass
```

5 在新增的测试方法 test_add_4_5 中添加断言验证

```
class demoTest(unittest.TestCase):
```

```
    def test_add_4_5(self):
```

```
        self.assertEqual(add(4,5),9)
```

6 最后, 在 main 函数中调用 unittest 的 main 方法启动测试,最终整个代码如下:

```
# _*_ coding:utf-8 _*_
```

```
__author__ = '苦叶子'
```

```
import unittest
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
# 被测函数
```

```
def add(a, b):
```

```
    return a + b
```

```
# 测试用例
```

```
class demoTest(unittest.TestCase):
```

```
    def test_add_4_5(self):
```

```
        self.assertEqual(add(4,5),9)
```

```
# 主函数
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```

1. 代码保存至 demoTest.py 中，执行下属命令

```
python demoTest.py
```

运行结果如下：

Run 1 testin 0.000s

OK

1. 下表信息是在运行 unittest 是可能的输出信息

序号	描述
1	OK 表示测试通过
2	FAIL 表示测试不通过，控制台输出断言异常信息
3	ERROR 表示测试异常

4.4.3 unittest 命令

unittest 提供了丰富的命令选项来控制 unittest 测试，下面我们简单的列举如下：

使用方法

python -m unittest 选项

示例,查看帮助信息

python -m unittest -h

序号	选项 & 描述
1	-h, --help 显示帮助信息

序号	选项 & 描述
2	-v, --verbose 查完整的测试结果输出信息
3	-q, --quiet 查看最小测试结果输出信息
4	-f, --failfast 在第一次遇到失败时，停止测试
5	-c, --catch 捕获 control-C 并显示结果
6	-b, --buffer 将 stdout, stderr 信息输出到 buffer 中

4.2.4 总结

本次主要就 python unittest 的基本使用进行了分享，非常的简单，期望大家能掌握起基本原理，再此基础上进一步扩展应用于实战中。后续逐步分享 unittest 相关技术。

4.3 python unittest 之关键 API 说明及示例

4.3.1 前言

本次就 python unittest 单元测试框架的一些基本的、常用的 API 进行分享，以便大家后续更深入的熟悉和应用 unittest。

4.3.2 TestCase 类 API

TestCase 类实例化的对象是最小的可测单元颗粒。它维护着一组测试方法以及为测试方法所提供的初始化方法和清理方法。

下面我们一起看一下定义在 TestCase 类中常用的方法

1. setUp()

初始化函数，在所有的测试方法调用之前调用（自动调用）

1. tearDown()

清理函数，在所有的测试方法调用之后调用（自动调用）

1. setUpClass()

类初始化方法，在单个类中的所有测试方法调用之前调用

1. tearDownClass()

类清理方法，在单个类中的所有测试方法调用之后调用

1. run(result=None)

运行测试，并返回测试结果（返回值为对象）

1. skipTest(reason)

在测试方法或 setUp 调用该方法可跳过当前测试

1. debug()

以不采集测试结果方式运行测试

1. shortDescription()

返回一行描述的测试结果信息

4.3.3 TestSuite 类 API

在 python unittest 中，提供了一套非常不错的用例组织机制（TestSuite）：

用来组织系列 TestCase 构建测试套件。

下面一起来看看如何一步步的创建套件并运行它

1. 创建 TestSuite 实例对象


```
suite = unittest.TestSuite()
```

1. 添加 TestCase 对象至套件中

```
suite.addTest(testcase class)
```

1. 添加 TestCase 对象至套件中方法 2

```
suite = unittest.makeSuite(testcase class)
```

1. 添加测试方法至套件中

```
suite.addTest(testcaseclass("testmethod"))
```

1. 使用 TextTestRunner 创建一个运行器

```
runner = unittest.TextTestRunner()
```

1. 运行

```
runner.run(suite)
```

下面看看 TestSuite 类的 API 说明，以便进一步理解上述过程的细节。

1. addTest()

新增一个测试方法到套件中

1. addTests()

新增多个测试方法到套件中

1. run()

运行套件中关联的测试方法，并返回测试结果（返回值为测试结果对象）

1. debug()

运行套件中关联的测试方法，但不搜集测试结果

1. countTestCases()

返回当前测试对象的测试方法数

4.3.4 TestSuite 应用示例

下面我用看一个 TestSuite 应用基本示例

```
# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class suiteTest(unittest.TestCase):

    def setUp(self):

        self.a = 10

        self.b = 20

    def testadd(self):

        # 验证加法

        result = self.a + self.b

        self.assertTrue(result == 100)
```

```

def testsub(self):

    # 验证减法

    result = self.a - self.b

    self.assertTrue(result == -10)


# 定义 suite

def suite():

    suite = unittest.TestSuite()

    # 添加测试方法

    suite.addTest(suiteTest("testadd"))

    suite.addTest(suiteTest("testsub"))

    ## 或用以下方式添加测试方法

    # suite.addTest(unittest.makeSuite(suiteTest))

    return suite


if __name__ == '__main__':

    runner = unittest.TextTestRunner()

    test_suite = suite()

    runner.run(test_suite)

```

将上述代码保存至 test_suite.py 中，在命令行中执行下属命令，运行

```
python test_suite.py
```

结果如下图：

```
F.
=====
FAIL: testadd (<__main__.suiteTest>)
-----
Traceback (most recent call last):
  File "C:\Users\lyy\Desktop\demo\testSuite.py", line 19, in testadd
    self.assertTrue(result == 100)
AssertionError: False is not true
-----

Ran 2 tests in 0.001s

FAILED (failures=1)
>>> _
```

testsuite.png

4.3.5 TestLoader 类 API

TestLoader 类提供了从类或模块级别来创建 test suites 的能力。

下面我们看一下简单的代码示例：

```
# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

reload(sys)

sys.setdefaultencoding("utf-8")
```

测试用例 1

```
class demo1Test(unittest.TestCase):  
  
    def setUp(self):  
  
        self.a = 10  
  
        self.b = 20  
  
    def testadd(self):  
  
        # 验证加法  
  
        result = self.a + self.b  
  
        self.assertTrue(result == 100)  
  
    def testsub(self):  
  
        # 验证减法  
  
        result = self.a - self.b  
  
        self.assertTrue(result == -10)
```

测试用例 2

```
class demo2Test(unittest.TestCase):  
  
    def setUp(self):  
  
        self.a = 1
```

```
self.b = 2
```

```
def testadd(self):
```

```
    # 验证加法
```

```
    result = self.a + self.b
```

```
    self.assertTrue(result == 10)
```

```
def testsub(self):
```

```
    # 验证减法
```

```
    result = self.a - self.b
```

```
    self.assertTrue(result == -1)
```

```
if __name__ == '__main__':
```

```
    # 用例列表
```

```
    testlist = [demo1Test, demo2Test]
```

```
    testload = unittest.TestLoader()
```

```
    # 构建 test suite
```

```
    test_Suite = []
```

```
    for testcase in testlist:
```

```
        testSuite = testload.loadTestsFromTestCase(testcase)
```

```
test_Suite.append(testSuite)

newSuite = unittest.TestSuite(test_Suite)

# 运行测试

runner = unittest.TextTestRunner()

runner.run(newSuite)
```

其他方式说明，这里就不一一示例说明了。

1. loadTestsFromTestCase()

从指定的 TestCase 构建一个 TestSuite 对象，该对象包含了 TestCase 中所有的测试方法

1. loadTestsFromModule()

从指定的模块中构建一个 TestSuite 对象，该对象包含了模块中所有的测试方法

注：模块，一个个的.py 文件，这些.py 文件里有一个个继承至 unittest.TestCase 的类

1. loadTestsFromName()

从特定的字符串构建一个 TestSuite 对象

1. discover()

从指定目录，并递归其子目录，查找所有的测试模块，构建 TestSuite。

4.3.6 TestResult 类

在 unittest 中通过该类提供了测试结果信息。下面对 TestResult 的一些常用方法进行说明。

1. Errors

返回所有的因异常抛出导致的错误信息

1. Failures

返回所有的因断言失败的信息

1. Skipped

返回所有因某些原因导致跳过的测试信息

1. wasSuccessful()

如果所有测试都 passed 则返回 True , 否则返回 False

1. stop()

取消所有正在执行的测试

1. startTestRun()/stopTestRun()

自己去尝试下^_^

1. testsRun

返回截止至当前的执行数

1. Buffer

如果设置为 True, 控制 stdout/stderr 信息是否缓存

这里就不写示例, 请直接参考标准文档

4.3.7 总结

本次就 unittest 中的 TestCase、TestSuite、TestResult、TestLoader 关键类进行了分享, 并展示了关键使用实例。

4.4 python unittest 之断言及示例

4.4.1 前言

python unittest 单元测试框架提供了一整套内置的断言方法。

1. 如果断言失败，则抛出一个 AssertionError,并标识该测试为失败状态
2. 如果异常，则当做错误来处理

注意：以上两种方式的区别

3. 如果成功，则标识该测试为成功状态

下面我们看下在 unittest 框架中定义了哪几类断言方法:

1. 基本的 Boolean 断言，即：要么 True，要么 False 的验证
2. 简单比较断言，例如比较 a,b 两个变量的值
3. 复杂断言

4.4.2 基本断言方法

基本的断言方法提供了测试结果是 True 还是 False。所有的断言方法都有一个 msg 参数，如果指定 msg 参数的值，则将该信息作为失败的错误信息返回。

序号	断言方法	断言描述
1	assertEqual(arg1, arg2, msg=None)	验证 arg1=arg2，不等则 fail
2	assertNotEqual(arg1, arg2, msg=None)	验证 arg1 != arg2, 相等则 fail

序号	断言方法	断言描述
3	assertTrue(expr, msg=None)	验证 expr 是 true , 如果为 false , 则 fail
4	assertFalse(expr,msg=None)	验证 expr 是 false , 如果为 true , 则 fail
5	assertIs(arg1, arg2, msg=None)	验证 arg1、arg2 是同一个对象 , 不是则 fail
6	assertIsNot(arg1, arg2, msg=None)	验证 arg1、arg2 不是同一个对象 , 是则 fail
7	assertIsNone(expr, msg=None)	验证 expr 是 None , 不是则 fail
8	assertIsNotNone(expr, msg=None)	验证 expr 不是 None , 是则 fail
9	assertIn(arg1, arg2, msg=None)	验证 arg1 是 arg2 的子串 , 不是则 fail
10	assertNotIn(arg1, arg2, msg=None)	验证 arg1 不是 arg2 的子串 , 是则 fail
11	assertIsInstance(obj, cls, msg=None)	验证 obj 是 cls 的实例 , 不是则 fail

序号	断言方法	断言描述
12	assertNotIsInstance(obj, cls, msg=None)	验证 obj 不是 cls 的实例，是则 fail

看一下上述断言简单的代码示例

```
# _*_ coding:utf-8 _*_
```

```
__author__ = '苦叶子'
```

```
import unittest
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
class demoTest(unittest.TestCase):
```

```
    def test1(self):
```

```
        self.assertEqual(4 + 5,9)
```

```
    def test2(self):
```

```
        self.assertNotEqual(5 * 2,10)
```

```
    def test3(self):
```

```
        self.assertTrue(4 + 5 == 9,"The result is False")
```

```
def test4(self):

    self.assertTrue(4 + 5 == 10,"assertion fails")


def test5(self):

    self.assertIn(3,[1,2,3])


def test6(self):

    self.assertNotIn(3, range(5))


if __name__ == '__main__':

    unittest.main()
```

将上述代码保存至 demoAssert.py 中，运行以下命令

```
python demoAssert.py
```

具体结果请看运行结果即可，这里不做一一分解了。

4.4.3 比较断言

unittest 框架提供的第二种断言类型就是比较断言。

下面我们看下各种比较断言：

1. `assertAlmostEqual (first, second, places = 7, msg = None, delta = None)`

验证 first 约等于 second。

places: 指定精确到小数点后多少位，默认为 7

1. `assertNotAlmostEqual (first, second, places, msg, delta)`

验证 first 不约等于 second。

places: 指定精确到小数点后多少位，默认为 7

==注：在上述的两个函数中，如果 delta 指定了值，则 first 和 second 之间的差值必须 \leq delta==

1. `assertGreater (first, second, msg = None)`

验证 $\text{first} > \text{second}$ ，否则 fail

1. `assertGreaterEqual (first, second, msg = None)`

验证 $\text{first} \geq \text{second}$ ，否则 fail

1. `assertLess (first, second, msg = None)`

验证 $\text{first} < \text{second}$ ，否则 fail

1. `assertLessEqual (first, second, msg = None)`

验证 $\text{first} \leq \text{second}$ ，否则 fail

1. `assertRegexMatches (text, regex, msg = None)`

验证正则表达式 regex 搜索==匹配==的文本 text。

regex：通常使用 `re.search()`

1. `assertNotRegexMatches (text, regex, msg = None)`

验证正则表达式 regex 搜索==不匹配==的文本 text。

regex：通常使用 `re.search()`

下面看一个简单的示例

```
# -*- coding:utf-8 -*-  
  
__author__ = '苦叶子'  
  
import unittest  
  
import math  
  
import re  
  
import sys  
  
reload(sys)  
  
sys.setdefaultencoding("utf-8")  
  
class demoTest(unittest.TestCase):  
  
    def test1(self):  
  
        self.assertAlmostEqual(22.0/7,3.14)  
  
    def test2(self):  
  
        self.assertNotAlmostEqual(10.0/3,3)  
  
    def test3(self):  
  
        self.assertGreater(math.pi,3)
```

```
def test4(self):

    self.assertNotRegexMatches("Tutorials Point (I) Private
    Limited","Point")

if __name__ == '__main__':

    unittest.main()
```

将上述代码保存至 demoAssert2.py 中，运行以下命令

```
python demoAssert2.py
```

具体结果请看运行结果即可，这里不做一一分解了。

4.4.4 复杂断言

unittest 框架提供的第三种断言类型，可以处理元组、列表、字典等更复杂的数据类型。

序号	断言方法	断言描述
1	assertListEqual (list1, list2, msg = None)	验证列表 list1、list2 相等，不等则 fail，同时报错信息返回具体的不同的地方
2	assertTupleEqual (tuple1, tuple2, msg = None)	验证元组 tuple1、tuple2 相等，不等则 fail，同时报错信息返回具体的不同的地方
3	assertSetEqual (set1, set2, msg = None)	验证集合 set1、set2 相等，不等则 fail，同时报错信息返回具体的不同的地方

序号	断言方法	断言描述
4	assertDictEqual (expected, actual, msg = None)	验证字典 expected、actual 相等，不等则 fail，同时报错信息返回具体的不同的地方

下面看下具体的示例代码:

```
# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

class demoTest(unittest.TestCase):

    def test1(self):

        self.assertEqual([2,3,4], [1,2,3,4,5])

    def test2(self):

        self.assertTupleEqual((1*2,2*2,3*2), (2,4,6))

    def test3(self):

        self.assertDictEqual({1:11,2:22},{3:33,2:22,1:11})
```



```
if __name__ == '__main__':
```

```
    unittest.main()
```

将上述代码保存至 demoAssert2.py 中，运行以下命令

```
python demoAssert3.py
```

具体结果请看运行结果即可，这里不做一一分解了。

4.4.5 总结

本次从 python unittest 提供的三种标准的断言方法进行了分享，要更好的掌握这些断言，需要去做扎实的练习。才能确保后续用的时候能更好的应用。

4.5 python unittest 之异常测试

4.5.1 前言

在 python unittest 测试框架中，提供了一系列的方法用于验证程序的异常。

下面和我一起看看在 unittest 中，如何使用异常验证来进行断言，完成对应的测试工作

4.5.2 assertRaises(exception, callable, *args*, **kwds*)

参数说明:

assertRaises(

exception, # 待验证异常类型

callable, # 待验证方法

*args, # 待验证方法参数

```
**kwds # 待验证方法参数(dict 类型)
```

```
)
```

功能说明：

验证异常测试，验证异常（第一个参数）是当调用待测试函数时，在传入相应的测试数据后，如果测试通过，则表明待测试函数抛出了预期的异常，否则测试失败。

下面我们通过一个示例来进行演示，如果验证做除法时抛出除数不能为 0 的异常 ZeroDivisionError。

```
# -*- coding:utf-8 -*-
```

```
__author__ = '苦叶子'
```

```
import unittest
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
# 除法函数
```

```
def div(a, b):
```

```
    return a/b
```

测试用例

```
class demoRaiseTest(unittest.TestCase):
```

```
    def test_raise(self):
```

```
        self.assertRaises(ZeroDivisionError, div, 1, 0)
```

主函数

```
if __name__ == '__main__':
```

```
    unittest.main()
```

test_raise 方法使用了 assertRaises 方法来断言验证 div 方法除数为零时抛出的异常。

运行 python raise_demo.py 结果如下

.

Ran 1 test in 0.000s

OK

你还可以尝试调整下数据，如下：

```
def test_raise(self):
```

```
    self.assertRaises(ZeroDivisionError, div, 1,1)
```

执行结果如下:

F

=====

FAIL: test_raise (__main__.demoRaiseTest)

Traceback (most recent call last):

File "raise_demo.py", line 18, in test_raise

self.assertRaises(ZeroDivisionError, div, 1,1)

AssertionError: ZeroDivisionError not raised

Ran 1 test in 0.000s

4.5.3 assertRaisesRegexp(exception, regexp, callable, args, *kwds)

这里就不对参数进行说明了，该方法使用正则表达式方式来匹配异常断言，能更加灵活，实用更多的场景。

通常 regexp 参数是一个正常表达式，或包含正则表达式的字符串（使用 re.search()函数）

下面跟我一下看下 assertRaisesRegexp 的实际应用示例

```
# -*- coding:utf-8 -*-
```

```
__author__ = '苦叶子'

import unittest

import re

import sys

reload(sys)

sys.setdefaultencoding("utf-8")


# 除法函数

def div(a, b):

    return a/b


# 测试用例

class demoRaiseTest(unittest.TestCase):

    def test_raise_regexp(self):

        self.assertRaisesRegexp(

            ZeroDivisionError, "integer division or modulo by zero", div, 2,

0)


# 主函数

if __name__ == '__main__':
```

```
unittest.main()
```

运行 `python test_raise_regexp.py` 上述示例，结果如下：

```
.
```

```
-----
```

```
Ran 1 test in 0.001s
```

```
OK
```

我们修改下除数，把

```
self.assertRaisesRegexp(
    ZeroDivisionError, "integer division or modulo by zero", div, 2,
    0)
```

改为：

```
self.assertRaisesRegexp(
    ZeroDivisionError, "integer division or modulo by zero", div, 2,
    2)
```

则执行失败，因为匹配失败了。

```
F
```

```
=====
```

```
FAIL: test_raise_regexp (__main__.demoRaiseTest)
```

```
-----
```

Traceback (most recent call last):

File "test_raise_regexp.py", line 19, in test_raise_regexp

ZeroDivisionError, "integer division or modulo by zero",

AssertionError: ZeroDivisionError not raised

Ran 1 test in 0.001s

FAILED (failures=1)

4.5.4 总结

今天我们就如何对被测对象的抛出的异常进行断言验证，做了基本的说明和示例，大家可以基于上述示例进行修改理解。深入掌握异常断言原理和方法。

4.6 python unittest 之加载及跳过测试方法和示例

4.6.1 前言

在 python unittest 框架中，内置了用例加载及跳过的标准函数。

其加载用例通过 TestLoader 类实现，而跳过测试方法则通过 unittest.skip()类实现。下面我们一起来学习下。

4.6.2 TestLoader 加载用例

TestLoader 类有一个 discover()函数，简洁的实现了从指定顶层目录、模块等加载测试用例。

将下述代码保存至==test==_discover.py 中

```
# -*- coding:utf-8 -*-
```

```
__author__ = '苦叶子'
```

```
import unittest
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
# 被测函数
```

```
def add(a, b):
```

```
    return a + b
```

```
# 测试用例
```

```
class demoTest(unittest.TestCase):
```

```
    def test_add_4_5(self):
```

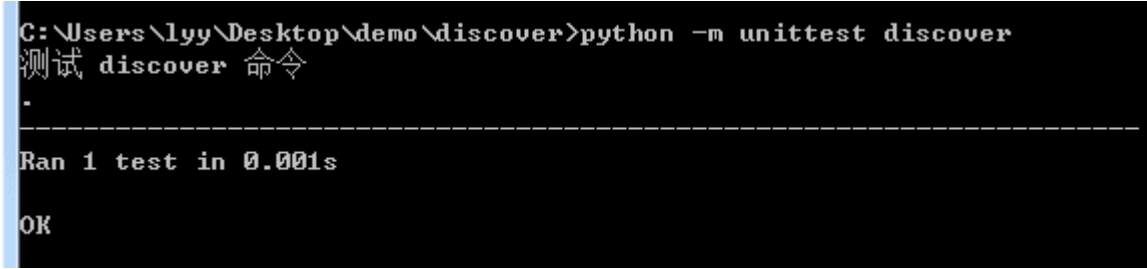
```
        print u"测试 discover 命令"
```


self.assertEqual(add(4,5),9)

切换至==test==_disvoover.py 所在目录，在命令行中执行一下命令:

python -m unittest discover

执行结果如下:



discover.png

通过该命令，unittest 测试框架会尝试在当前目录及其子目录加载所有满足要求的测试（==注意：所有的用例模块应该以 test 开头进行命名==，默认加载 test 开头的模块）

下面我们一起看下其他相关的命令

序号	命令选项	描述
1	-v, --verbose	详细输出
2	-s, --start-directory	启动目录（默认为当前目录）
3	-p, --pattern	匹配加载的测试文件(默认匹配 test*.py)
4	-t, --top-level-directory	顶层目录（默认同--start-directory）

例如我们指定 C:\test 下，匹配 assert 开头的所有测试模块

python -m unittest -v -s "c:\\test" -p "assert*.py"

这个命令将加载 C:\test 目录下所有 assert 开头的测试模块中的测试方法

4.6.3 unittest.skip 跳过测试方法

python unittest 测试框架从 python2.7 开始支持设置跳过指定的测试方法或是跳过满足某种条件的测试用例。

下面我看一个强制跳过指定的测试用例的示例：

```
# -*- coding:utf-8 -*-

__author__ = '苦叶子'

import unittest

import sys

reload(sys)

sys.setdefaultencoding("utf-8")

# 被测函数

def add(a, b):

    return a + b

class demoSkipTest(unittest.TestCase):

    @unittest.skip(u"强制跳过示例")

    def test_add(self):
```

```
self.assertEqual(add(4,5), 9)
```

```
def test_add_2(self):
```

```
    self.skipTest("强制跳过示例 2")
```

```
    self.assertEqual(add(4,5), 9)
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```

将上述代码保存至 demoskip.py,运行下述命令：

```
python demoskip.py
```

结果如下:

```
C:\Users\lyy\Desktop\demo>python demoskip.py
ss
-----
Ran 2 tests in 0.001s

OK (skipped=2)
```

demoskip.png

下面我们一起看下各种 skip 方法说明

序号	方法	说明
1	unittest.skip(reason)	强制跳转。reason 是跳转原因
2	unittest.skipIf(condition, reason)	条件跳转，如果 condition 是 True 则跳转

序号	方法	说明
3	<code>unittest.skipUnless(condition, reason)</code>	除非 condition 为 True , 才进行调整
4	<code>unittest.expectedFailure()</code>	标记该测试预期为失败 , 如果该测试方法运行失败, 则该测试不算做失败

下面我们一起看下各种方式实践的示例

```
# -*- coding:utf-8 -*-
```

```
__author__ = '苦叶子'
```

```
import unittest
```

```
import sys
```

```
reload(sys)
```

```
sys.setdefaultencoding("utf-8")
```

```
class demoSkipTest(unittest.TestCase):
```

```
    a = 50
```

```
    b = 20
```

```
    def test_add(self):
```

```
        """加法"""
```

```
        result = self.a + self.b
```

```
        self.assertEqual(result, 40)
```

```
@unittest.skipIf(a>b, u"a>b 就跳过")
```

```
def test_sub(self):
```

```
    """减法"""
```

```
    result = self.a - self.b
```

```
    self.assertTrue(result == -30)
```

```
@unittest.skipUnless(b==0, u"除数为 0 , 则跳转")
```

```
def test_div(self):
```

```
    """除法"""
```

```
    result = self.a / self.b
```

```
    self.assertTrue(result == 1)
```

```
@unittest.expectedFailure
```

```
def test_mul(self):
```

```
    """乘法"""
```

```
    result = self.a * self.b
```

```
    self.assertTrue(result == 0)
```

```
if __name__ == "__main__":
```

```
unittest.main()
```

将上述代码保存至 demo_skip_test.py,运行结果如下

```
C:\Users\lyy\Desktop\demo>python demo_skip_test.py
Fsxs
=====
FAIL: test_add (__main__.demoSkipTest)
鐳狗砵
-----
Traceback (most recent call last):
  File "demo_skip_test.py", line 17, in test_add
    self.assertEqual(result, 40)
AssertionError: 70 != 40
-----
Ran 4 tests in 0.001s

FAILED (failures=1, skipped=2, expected failures=1)
```

allskip.png

4.6.4 总结

本次就用例加载及如何跳过某些测试方法或用例进行了分享，大家可以基于上述示例代码进行改造学习和实践。

更多精彩内容请关注公众号：

