

《selenium2 python 自动化测试实战》

序

自动化测试，一个现在被炒的火热的词；各大公司都在嚷嚷着要上自动化测试的项目，都在招聘各种自动化测试人员.....

非常荣幸的受作者邀请来帮忙写这个序，诚惶诚恐，何德何能？

不记得何时开始认识的作者了。当初只是作为一个自学者混迹于各个技术群中，后来发现几乎每个群里每天充斥着大量的垃圾信息，QQ 不停的闪动，看吧？！都是无用的信息，不看吧？！却又怕错过些什么。后来自己着手建立了一个群，期望能按着自己的想法来建立一个平台，就有了后来的相识吧。

作者，是一个勤奋，主动积极，乐于实践钻研的人，所以，就有了这本书的存在；他将我们曾一起讨论过的东西，以及自己实践钻研的收获，都做了一一收入。

本书，主要是面向编程基础较弱的人，但也同时适合有一定技术储备的人学习 selenium。

对于编程基础较低初学者，适合通篇阅读，过程中可以学习和接触到很多旁枝侧节的知识，这些都是做好 web 自动化所有需要的知识；对于有一定技术储备，只是为了学习 selenium 的人而言，你大可根据目录，把它当成手册直接阅读你需要的东西。

这不是一本编程语言和技巧的书籍，虽然书中涉及了很多 python 知识，以及其他的技术知识。它更多的是充当“布道者”的角色，通过大量的实例，传达一种思维模式：如何利用 python+selenium 组建起生产应用的 web 自动化测试。

这本书也不能帮你成为高大上的编程大牛，或者自动化测试的行家。但是，它可以引领你迈入 web 自动化测试的领域。

师傅领进门，修行靠个人；一切的一切都还是要靠你自己去多多实践，不是有一句名言么？实践是检验真理的唯一标准！

Mark Rabbit

前言

2013年即将结束，不知读者在这一年中都收获了那些。在这一年的最后一天班，我怀着激动的心情来写这本电子书的前言，在这本电子书的整理过程中，虽然舍弃了很多享受生活的时间，但从中我也收获了很多。

自从开始从事软件测试工作开始，我就深深的喜欢上了这个职业。对我来说软件测试不单单是一份为了赚钱的工作，它同样也是我生活的一部分，我从中找到了自我的价值。从开始在博客园写博客时，自我的价值开始被放大，我只多了一点分享精神。

从开始从事软件工作时就知道 selenium 这个自动化工具，网上找来资料学习，学会了用 selenium IDE 录制脚本，学会了简单搭建 java +selenium RC 的环境，写一个简单的自动化脚本。后来，换了城市换了工作，一直于忙于工作和其它技术的学习，中间间隔了一年多没有再接触 selenium 。

直到2013年年初换了新工作后工作稍微轻松，业余时间开始学习 python 语言，然后就喜欢上了这门语言，由于所测试的是 web 产品，所以，就考虑通过 python + selenium 将产品自动化起来。关于 python + selenium 的资料除了官方的一份 API 并不多，我们更容易找到的是 java + selenium 的资料。对我来说学习的过程也比较缓慢，后来有幸认识了 MarkRabbit ，他在 python + selenium 方面有着比较丰富的实践经验。webdriver API 对种元素的定位和操作有着不少知识点，我每学会使用一个知识点整理一篇博客。后来，积累了十几篇博客出来。为了便于阅读我就整理成了一份 PDF 上传到了 CSDN 上面。

在 MarkRabbit 的一路指点下，我又开始学习 pyhon unittest 单元测试框架，通过 python 脚本批量执行测试用例等，然后整理出来第二版的内容。在此过程中得到了不少同学的反馈，自己的自动化测试水平在不断的学习实践中得到了长足的进步。后来，开始对脚本做参数化，引入 HTMLTestRunner 测试报告以及对测试结构调整。整理出了第三版。

MarkRabbit 趁周末休息的时间向我展示他们目前的 python + selenium 测试框架，我非常兴奋，同时也觉得这个技术非常有用，于是决定整理一本完整书出来，市面上关于 selenium 的书大多翻译官方文档，对 selenium 的讲解也泛泛之谈，并没有真正通过编程的方式来帮助读者真正的去实施自动化。联系了一位人民邮电出版社的编辑，获得了一份编书的规范，当时并没有约稿。这对我来说是一次新尝试，我想自己真能写出来再说。

有了这个想法之后，我每天像打了鸡血一样活在兴奋当中，坐车和睡觉前也在思考书中的技术点。后来，乙醇告诉我编辑成书比较麻烦，不断的修改也是非常头痛的事情，而我没有精力反复做这些，由于自身水平的局限，我的更多精力是在技术点学习上。后来，改变了想法以电子书的形式展现给大家，这样我的编写过程随意了许多，我要做就是简单易懂告诉这是怎么回事，如何去实现。

全书的结构：

全书共分11章，第一章是基础，了解 selenium 家谱，各种组件之间的关系以及一些必备知识。第二章告诉如何开始用 python IDLE 写程序以及自动化测试环境的搭建。第三章是 webdriver API ，我花了相当多时间对原先的文档，冗余的地方进行压缩，并且增加了许多新的知识点。第四、五两章介绍自动化测试

模型，以及如何设计自动化测试用例。第六、七、八章的知识点关联性比较大，帮助读者搭建一个实例的测试结构，读者可以在此基础上扩展和优化。第九章介绍 selenium grid 如何多台平多浏览器的执行测试用例。第十章 带领读者了解形为驱动开发框 lettuce ，第十一章通过 git 来管理自己的测试用例。

本书的特点：

本书内容由浅入深，章节的安排也符合全读者的学习曲线，所有涉及到 python 语言的地方都有详细的介绍。这是一本自动化测试书，这也是一本 python 编程书。希望通过本书的学习，你不仅仅只是掌握一个自动化测试技术，使你的编程水平也有长足的进步，从此摆脱纯手工测试，向“测试开发”人员转型，向高薪挑战。

本书中的不足：

当然，本书也存在许多不足，python 的多线程技术，selenium grid 并发测试技术对作者来说也是难点，作者水平有限，无法讲解的很透彻。git 工具的使用，分支的使用，以及本地搭建 git 环境，由于时间仓促未能编写（只能后续版本完善），本书所提供的代码随着时间环境变化，有些在运行过程中会出现各种各样的错误，请读者遇到错误多调试多思考，欢迎向作者反馈书中的错误。

2014.1.24
虫师

目录

序.....	2
前言.....	3
目录.....	5
第一章 自动化测试基础.....	9
第一节 软件测试分类.....	9
第二节 什么样的项目适合自动化测试.....	13
第三节 自动化测试及工具简述.....	14
第四节 selenium 工具介绍.....	15
第五节 前端技术介绍.....	17
第六节 前端工具介绍.....	20
第七节 自动化测试语言的选择.....	22
第二章 python webdriver 环境搭建.....	24
第一节 环境搭建.....	24
第二节 使用 IDLE 来编写 python.....	26
第三节 第一个自动化脚本.....	28
第四节 安装浏览器驱动.....	29
第三章 python webdriver API.....	31
第一节、浏览器的操作.....	31
3.1.1、浏览器最大化.....	31
3.1.2、设置浏览器宽、高.....	32
3.1.3、控制浏览器前进、后退.....	32
第二节 简单对象的定位.....	34
3.2.1 id 和 name 定位.....	35
3.2.2 tag name 和 class name 定位.....	36
3.2.3 link text 与 partial link text 定位.....	37
3.2.4 XPath 定位.....	37
3.2.5 CSS 定位.....	40
第三节 操作测试对象.....	43
3.3.1、登录实例.....	44
3.3.2 WebElement 接口常用方法.....	45
第四节 鼠标事件.....	46
第五节 键盘事件.....	50
第六节 打印信息.....	52
第七节 设置等待时间.....	54
第八节 定位一组对象.....	57
第九节 层级定位.....	60
第十节 定位 frame 中的对象.....	64

第十一节 对话框处理.....	67
第十二节 浏览器多窗口处理.....	68
第十二节 alert/confirm/prompt 处理.....	71
第十三节 下拉框处理.....	73
第十四节 分页处理.....	75
第十五节 上传文件.....	77
第十六节 下载文件.....	79
第十七节 调用 JavaScript.....	80
第十八节、控制浏览器滚动条.....	83
第十九节 cookie 处理.....	85
3.19.1 打印 cookie 信息.....	86
3.19.2、对 cookie 操作.....	86
第二十节 获取对象的属性.....	88
第二十一节 验证码问题.....	89
第二十二节 weddriver 原理.....	90
第四章 自动化测试模型.....	91
第一节、自动化测试模型介绍.....	92
4.1.1 线性测试.....	92
4.1.2 模块化与类库.....	93
4.1.3 数据驱动.....	94
4.1.4 关键字驱动.....	95
第二节、登录模块化.....	97
第三节、数据驱动（参数化）.....	103
第五章 自动化测试用例设计.....	111
第一节、手工测试用例与自动化测试用例.....	111
第二节、测试类型.....	113
第三节、python 异常断言.....	115
第四节、webdriver 错误截图.....	119
第五节、自动化测试用例设计实例.....	120
5.5.1 登陆用例实例.....	121
5.5.2 添加文件用例实例.....	123
5.5.3 删除文件用例实例.....	125
5.5.4 重命名文件用例实例.....	127
第六章 引入 unittest 单元测试框架.....	129
第一节、selenium IDE 介绍.....	129
6.1.1 selenium IDE 安装.....	129
6.1.2 selenium IDE 界面介绍.....	130
6.1.3 selenium IDE 录制脚本.....	132
6.1.4 selenium IDE 编辑脚本.....	133
第二节、引入 unittest 框架.....	136
第三节、unittest 单元测试框架解析.....	140
第四节、批量执行测试用例.....	146
第七章 引入测试报告与结构优化.....	150
第一节、生成 HTMLTestRunner 测试报告.....	150
第二节、测试套件.....	155

7.2.1、测试套件实例.....	155
7.2.2、整合 HTMLTestRunner 测试报告.....	159
7.2.3、易读的测试报告.....	160
7.2.3、报告文件名取当前时间.....	163
第三节、结构改进.....	164
7.3.1、all_tests.py 文件移出来.....	164
7.3.2、__init__.py 文件解析.....	166
7.3.3、把公共模块文件移进去.....	167
第四节、用例的读取.....	168
7.3.1、改进用例的读取.....	168
7.3.2、discover 解决用例的读取.....	171
第八章 自动化测试高级应用.....	175
第一节、自动发邮件功能.....	175
8.1.1、文件形式的邮件.....	176
8.1.2、HTML 形式的邮件.....	177
8.1.3、获取测试报告.....	178
8.1.4、整合自动发邮件功能.....	180
第二节、python 多进程/线程基础.....	183
8.2.1、单线程.....	184
8.2.2、thread 模块.....	185
8.2.3、threading 模块.....	188
8.2.4、multiprocessing 模块.....	192
8.2.5、Pipe 和 queue.....	194
第三节、多进程执行测试用例.....	196
第四节、定时任务.....	200
8.4.1、程序控制时间执行.....	200
8.4.2、windows 添加任务计划.....	202
8.4.3、linux 实现定时任务.....	207
第五节、WebDriver 方法二次封装.....	215
第九章 selenium grid2 分布式执行测试用例.....	220
第一节、selenium1 与 2 工作原理.....	220
第二节、selenium server 环境配置.....	224
第三节、selenium Grid 工作原理.....	227
第四节、selenium Grid 应用.....	229
9.4.1、多浏览器执行用例.....	229
9.4.2、多节点执行用例.....	232
9.4.3、分布式并行运行脚本.....	237
第十章 行为驱动开发 BDD 框架 lettuce 入门.....	241
第一节、安装与例子.....	242
第二节、lettuce 解析.....	244
第三节、添加测试场景.....	248
第四节、lettuce 的目录结构与执行过程.....	252
第五节、lettuce webdriver 自动化测试.....	254
第十一章 git/getcafe 管理自动化测试项目.....	257
第一节、Git 搭建.....	257

第二节、提交代码.....	261
第三节、更新代码.....	268
附录.....	274
UliPad--python 开发利器环境搭建.....	275
Sublime--强大好用的代码编辑器.....	276
sublime 使用技巧.....	277
参考.....	282

第一章 自动化测试基础

第一节 软件测试分类

关于软件测试领域名词颇多，发现有许多测试新手混淆概念，从不同的角度可以将软件测试有不同的分类的方法；所以，这里汇总常见软件测试的相关名词，对软件测试领域有个概括的了解。

根据项目流程阶段划分软件测试

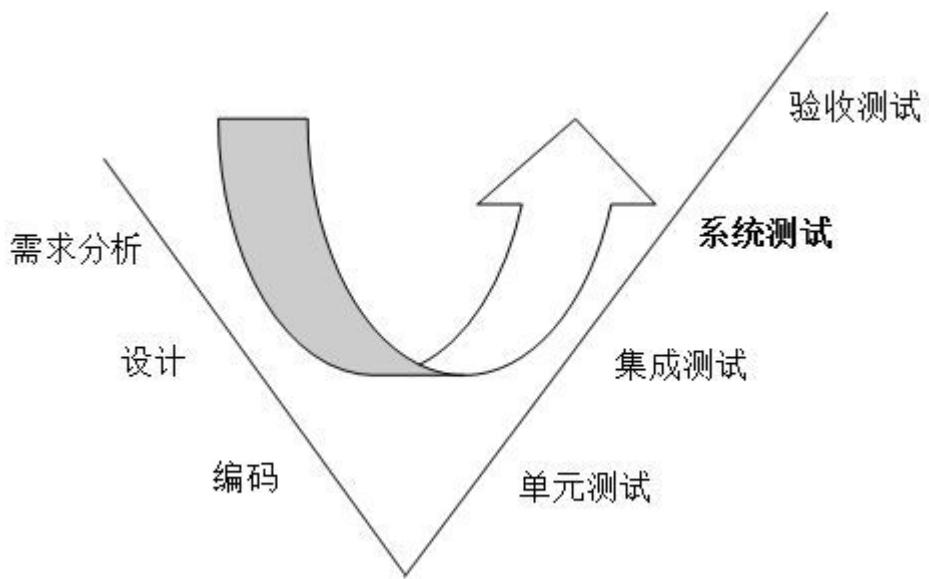


图1.1

上图是一个典型“V”模型软件开发流程，那么各项软件测试工作是在项目开发流程中循序渐进进行的。下面将介绍各个阶段测试的含义。

单元测试：单元测试（或模块测试）是对程序中的单个子程序或具有独立功能的代码段进行测试的过程。

集成测试：集成测试是单元测试的基础上，将通过单元模块组装成系统或子系统，再进行测试，重点是检查模块之间的接口是否正确。

系统测试：系统测试是针对整个产品系统进行的测试，验证系统是否满足了需求规格的定义，以及软件系统的正确性和性能等是否满足其规约所指定的要求。

验收测试：验收测试是部署软件之前的最后一个测试操作。验收测试的目的是确保软件准备就绪，向软件购买者展示该软件系统满足其用户的需求。

白盒测试、黑盒测试、灰盒测试

白盒测试与黑盒测试，主要是根据在软件测试工作中对软件代码的可见程度进行的划分；这也是软件测试领域中最基本的概念。

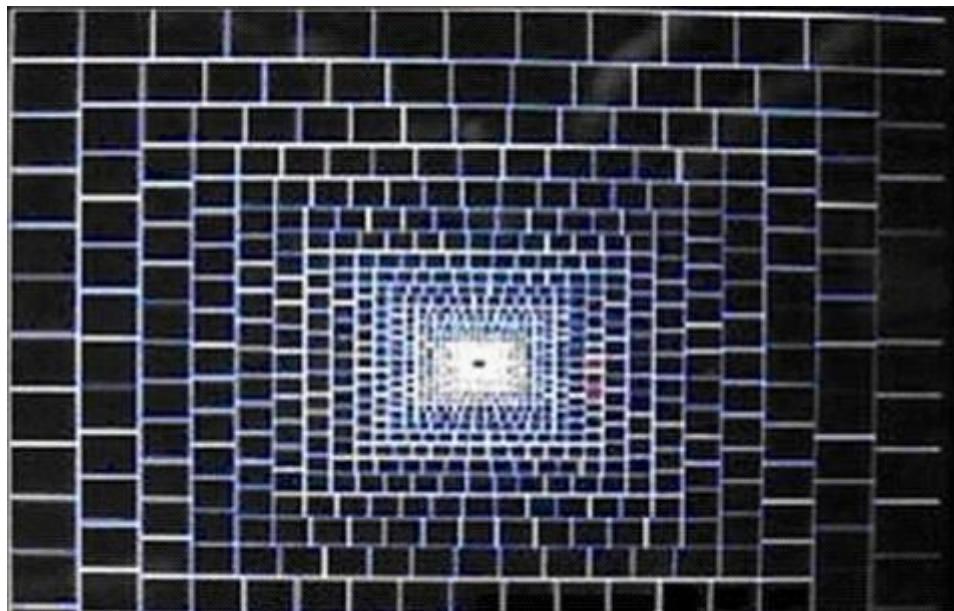


图1.2

黑盒测试：

黑盒测试，指的是把被测的软件看作是一个黑盒子，我们不去关心盒子里面的结构是什么样子的，只关心软件的输入数据和输出结果。

它只检查程序呈现给用户的功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息。黑盒测试着眼于程序外部结构，不考虑内部逻辑结构，主要针对软件界面和软件功能进行测试。

白盒测试：

白盒测试，指的是把盒子打开，去研究里面的源代码和程序执行结果。

它是按照程序内部的结构测试程序，通过测试来检测产品内部动作是否按照设计规格说明书的规定正常进行，检验程序中的每条通路是否都能按预定要求正确工作。

灰盒测试：

灰盒测试介于黑盒测试与白盒测试之间。

可以这样理解，灰盒测试关注输出对于输入的正确性，同时也关注内部表现，但这种关注不象白盒那样详细、完整，只是通过一些表征性的现象、事件、标志来判断内部的运行状态，有时候输出是正确的，但内部其实已经错误了，这种情况非常多，如果每次都通过白盒测试来操作，效率会很低，因此需要采取这样的一种灰盒测试的方法。

功能测试与性能测试

从对软件的不同测试点可以划分为功能测试与性能测试。

功能测试

功能测试检查实际的功能是否符合用户的需求。测试的大部分工作也是围绕软件的功能进行，设计软件的目的也就是满足客户对其功能的需求。如果偏离的这个目的任何测试工作都是没有意义的。

功能测试又可以细分为很多种：逻辑功能测试、界面测试、易用性测试、安装测试、兼容性测试等。

性能测试

性能测试是通过自动化的测试工具模拟多种正常、峰值以及异常负载条件来对系统的各项性能指标进行测试。

软件的性能包括很多方面，主要有时间性能和空间性能两种。

时间性能：主要是指软件的一个具体的响应时间。比如一个登录所需要的时间，一个交易所需要的时间等。当然，抛开具体的测试环境，来分析一次事务的响应时间是没有任何意义的。需要搭建一个具体且独立的测试环境。

空间性能：主要指软件运行时所消耗的系统资源，比如硬件资源，CPU、内存，网络带宽消耗等。

手工测试与自动化测试

从对软件测试工作的自动化程度可以划分为手工测试与自动化测试。

手工测试：

手工测试就是由人去一个一个的去执行测试用例，通过键盘鼠标等输入一些参数，查看返回结果是否符合预期结果。

手工测试并不非专业术语，手工测试通常是指我们在系统测试阶段所进行的功能测试，为了更明显的与自动化测试进行区分，所以这里使用了手工测试。

自动化测试

自动化测试是把以人为驱动的测试行为转化为机器执行的一种过程。通常，在设计了测试用例并通过评审之后，由测试人员根据测试用例中描述的规程一步步执行测试，得到实际结果与期望结果的比较。在此过程中，为了节省人力、时间或硬件资源，提高测试效率，便引入了自动化测试的概念。

自动化测试又可分为：**功能自动化测试与性能自动化测试**。

我们一般所说的自动化测试就是指功能自动化测试，通过相关的测试技术，通过编码的方式用一段程序来测试一个软件的功能，这样就可以重复执行程序来进行重复的测试。如果一个软件一小部分发生改变，我们只要修改一部分自动化测试代码，就可以重复的对整个软件进行功能测试；从而大大的提高了测试效率。

性能自动化测试，当然，除了早期阶段，现在的性能测试工作都是通过性能测试工具辅助完成的。通过工具可以模拟成千上万的用户向系统发送请求，用来验证系统的处理能力。

冒烟测试、回归测试、随机测试

这三种测试出现在软件功能测试周期中，既不算具体明确的测试阶段也不是具体的测试方法。

冒烟测试：

是指在对一个新版本进行系统大规模的测试之前，先验证一下软件的基本功能是否实现，是否具备可测性。

引入到软件测试中，就是指测试小组在正规测试一个新版本之前，先投入较少的人力和时间验证一个软件的主要功能，如果主要功能都没有实现，则打回开发组重新开发。这样做的好处是可以节省大量的时间成本和人力成本。

回归测试：

回归测试是指修改了旧代码后，重新进行测试以确认修改后没有引入新的错误或导致其他代码产生错误。

回归测试一般是在进行软件的第二轮测试开始的，验证第一轮中发现的问题是否得到修复。当然，回归也是一个循环的过程，如果回归的问题通不过，则需要开发人员修改后再次进行回归，直到通过为止。

随机测试：

是指测试中的所有输入数据都是随机生成的，其目的是模拟用户的真实操作，并发现一些边缘性的错误。

随机测试可以发现一些隐蔽的错误，但是也有很多缺点，比如测试不系统，无法统计代码覆盖率和需求覆盖率，发现的问题难以重现。一般是放在测试的最后执行。其实随机测试更专业的升级版叫 探索性测试

其他测试

探索性测试

探索性测试可以说是一种测试思维技术。它没有很多实际的测试方法、技术和工具，但是却是所有测试人员都应该掌握的一种测试思维方式。探索性强调测试人员的主观能动性，抛弃繁杂的测试计划和测试用例设计过程，强调在碰到问题时及时改变测试策略。

安全测试

安全测试是在 IT 软件产品的生命周期中，特别是产品开发基本完成到发布阶段，对产品进行检验以验证产品符合安全需求定义和产品质量标准的过程。

安全测试也在越来越受到企业的关注和重视，因为由于安全性问题造成的后果是不可估量的。尤其对于互联网产品最容易遭受各种安全攻击。



第二节 什么样的项目适合自动化测试

虽然，在你拿到这本书时已经对要测试的项目做了一些分析和考量，但笔者还是有必要在这里啰嗦一下不是所有项目有适合实施自动化测试的，以免读者对项目实施自动化过程中发现困难重重，浪费了大量的人力和时间而没有得到应有的收益。

- 1、任务测试明确，不会频繁变动
- 2、每日构建后的测试验证
- 3、比较频繁的回归测试
- 4、软件系统界面稳定，变动少
- 5、需要在多平台上运行的相同测试案例、组合遍历型的测试、大量的重复任务
- 6、软件维护周期长
- 7、项目进度压力不太大
- 8、被测软件系统开发比较规范，能够保证系统的可测试性
- 9、具备大量的自动化测试平台
- 10、测试人员具备较强的编程能力

当然，并非以上 10 条都具备有情况下才能开展测试工作。这里就需要读者做综合的权衡。在我

们普遍的经验中，只要满足三个条件就可以对项目开展自动化测试：

软件需求编程不频繁

测试脚本的稳定性决定了自动化测试的维护成本。如果软件需求变动过于频繁，测试人员需要根据变动的需求来更新测试用例以及相关的测试脚本，而脚本的维护本身就是一个代码开发的过程，需要修改、调试，必要的时候还要修改自动化测试的框架，如果所花费的成本不低于利用其节省的测试成本，那么自动化测试便是失败的。

项目中的某些模块相对稳定，而某些模块需求变动性很大。我们便可对相对稳定的模块进行自动化测试，而变动较大的仍是用手工测试。

项目周期较长

由于自动化测试需求的确定、自动化测试框架的设计、测试脚本的编写与调试均需要相当长的时间来完成。这样的过程本身就是一个测试软件的开发过程，需要较长的时间来完成。如果项目的周期比较短，没有足够的时间去支持这样一个过程，那么自动化测试便成为笑谈。

自动化测试脚本可重复使用

自动化测试脚本的重复使用要从三个方面来考量，一方面所测试的项目之间是否很大的差异性（如 C/S 系统和 B/S 系统的差异）；所选择的测试工具是否适应这种差异；最后，测试人员是否有能力开发出适应这种差异的自动化测试框架。

第三节 自动化测试及工具简述



自动化测试的概念有广义与狭义之分；广义上来讲所有借助工具来进行软件测试都可以称为自动化测试；狭义上来讲，主要指基于 UI 层的自动化测试；除此之外还有基代码编写阶段的单元自动化测试，基本集成测试阶段的接口自动化测试。

注意：如果没有特别说明，本文所说的“自动化测试”均指基于“UI 的功能自动化测试”。

目前市面上的自动化测试工具非常多，下面几款是比较常见的自动化测试工具。

QTP

QTP 是 HP Quick Test Professional software 的简称，是一种企业级的自动测试工具。提供了强大易用的录制回放功能。支持 B/S 与 C/S 两种架构的软件测试。是目前主流的自动化测试工具。

Robot Framework

Robot Framework 是一款 python 编写的功能自动化测试框架。具备良好的可扩展性，支持关键字驱动，可以同时测试多种类型的客户端或者接口，可以进行分布式测试执行。

watir

<http://fnng.cnblogs.com>

Watir 全称是“Web Application Testing in Ruby”。它是一种基于 Web 模式的自动化功能测试工具。watir 是一个 ruby 语言库，使用 ruby 语言进行脚本开发。

selenium

Selenium 也是一个用于 Web 应用程序测试的工具，支持多平台、多浏览、多语言去实现自动化测试。目前在 web 自动化领域应用越来越广泛。

当然，除了上面所列自动化测试工外，根据不同的应用还有很多商业的、开源的以及公司自己开发的自动化测试工具。

第四节 selenium 工具介绍

什么是 selenium?

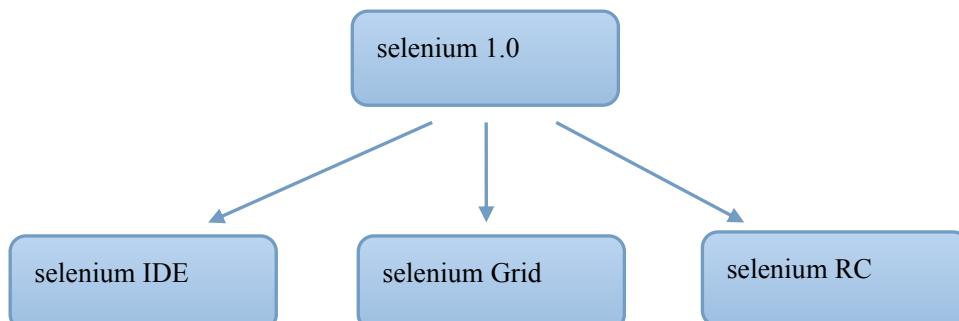
selenium 自动化测试浏览器，它主要是用于 Web 应用程序的自动化测试，但肯定不只局限于此，同时支持所有基于 web 的管理任务自动化。



selenium 的特点：

- 开源，免费
- 多浏览器支持：FireFox、Chrome、IE、Opera
- 多平台支持：linux、windows、MAC
- 多语言支持：java、python、ruby、php、C#、JavaScript
- 对 web 页面有良好的支持
- 简单（API 简单）、灵活（用开发语言驱动）
- 支持分布式测试用例执行

selenium 经历了两个版本，selenium 1.0 和 selenium 2.0，selenium 也不是简单一个工具，而是由几个工具组成，每个工具都有其特点和应用场景。



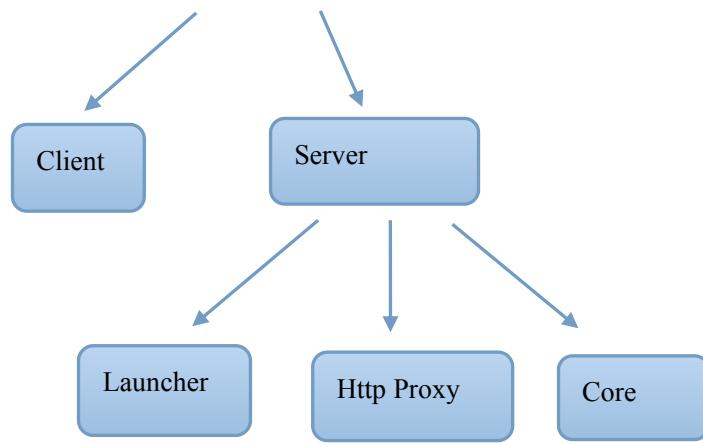


图 1.3

selenium IDE

selenium IDE 是嵌入到 Firefox 浏览器中的一个插件，实现简单的浏览器操作的录制与回放功能。那么什么情况下用到它呢？

快速的创建 bug 重现脚本，在测试人员的测试过程中，发现了 bug 之后可以通过 IDE 将重现的步骤录制下来，以帮助开发人员更容易的重现 bug。

IDE 录制的脚本可以转换成多种语言，从而帮助我们快速的开发脚本，关于这个功能后而用到时再详细介绍。

selenium Grid

Selenium Grid 是一种自动化的测试辅助工具，Grid 通过利用现有的计算机基础设施，能加快 Web-app 的功能测试。利用 Grid，可以很方便地同时在多台机器上和异构环境中并行运行多个测试事例。其特点为：

- 并行执行
- 通过一个主机统一控制用例在不同环境、不同浏览器下运行。
- 灵活添加变动测试机

selenium RC

selenium RC 是 selenium 家族的核心工具，selenium RC 支持多种不同的语言编写自动化测试脚本，通过 selenium RC 的服务器作为代理服务器去访问应用从而达到测试的目的。

selenium RC 使用分 Client Libraries 和 selenium Server，Client Libraries 库主要主要用于编写测试脚本，用来控制 selenium Server 的库。

Selenium Server 负责控制浏览器行为，总的来说，Selenium Server 主要包括 3 个部分：Launcher、Http Proxy、Core。其中 Selenium Core 是被 Selenium Server 嵌入到浏览器页面中的。其实 Selenium Core 就是一堆 JS 函数的集合，就是通过这些 JS 函数，我们才可以实现用程序对浏览器进行操作。Launcher 用于启动浏览器，把 selenium Core 加载到浏览器页面当中，并把浏览器的代理设置为 Selenium Server 的 Http Proxy。

selenium 2.0

搞清了 selenium 1.0 的家族关系，selenium 2.0 是把 WebDriver 加入到了这个家族中；简单用公式表示为：

```
selenium 2.0 = selenium 1.0 + WebDriver
```

需要强调的是，在 selenium 2.0 中主推的是 WebDriver，WebDriver 是 selenium RC 的替代品，因为 selenium 为了向下兼容性，所以 selenium RC 并没有彻底抛弃，如果你使用 selenium 开发一个新自动化测试项目，强烈推荐使用 WebDriver。那么 selenium RC 与 webdriver 主要有什么区别呢？

selenium RC 在浏览器中运行 JavaScript 应用，使用浏览器内置的 JavaScript 翻译器来翻译和执行 selenese 命令（selenese 是 selenium 命令集合）。

WebDriver 通过原生浏览器支持或者浏览器扩展直接控制浏览器。WebDriver 针对各个浏览器而开发，取代了嵌入到被测 Web 应用中的 JavaScript。与浏览器的紧密集成支持创建更高级的测试，避免了 JavaScript 安全模型导致的限制。除了来自浏览器厂商的支持，WebDriver 还利用操作系统级的调用模拟用户输入。

selenium 与 WebDriver 原先属于两个不同的项目，WebDriver 的创建者 Simon Stewart 早在 2009 年八月的一份邮件中解释了项目合并的原因。

为何把两个项目合并？部分原因是 WebDriver 解决了 Selenium 存在的缺点（比如，能够绕过 JS 沙箱。我们有出色的 API），部分原因是 Selenium 解决了 WebDriver 存在的问题（例如支持广泛的浏览器），部分原因是因为 Selenium 的主要贡献者和我都觉得合并项目是为用户提供最优秀框架的最佳途径。

第五节 前端技术介绍

由于 selenium 基于 web 的自动化测试技术，我们的要操作的对象是页面，所以有必要对前端的技术和工具做一个简单的介绍。

HTML 简介

HTML (Hyper Text Markup Language) 中文为超文本标记语言，HTML 是网页的基础，它并不是一种编程语言，而是一种标记语言（一套标记标签），但我们可以在此标签中嵌入各种前端脚本语言，如 VBScript、JavaScript 等。下面是一个简单的 HTML 页面：

```
<html>

    <title>标题</title>

    <body>

        <h1>正文</h1>

    </body>

</html>
```

<html> 与 </html> 之间的文本描述网页

<title> 与</title> 之间的内容显示在浏览器的标题栏

<body> 与 </body> 之间的文本是可见的页面内容

<h1> 与 </h1> 之间的文本被显示为正文， h1 为页面中的一号字体

现在我们通过浏览器打开任意一个页面，在页面上右键菜单选择“查看网页源代码”，在复杂的前端代码中你依然可以找到 HTML 的身影。

当然了，HTML 还定义了其它许多功能，请参考其它资料进行学习。

JavaScript 简介



JavaScript 是一种由 **Netscape** 公司的 **LiveScript** 发展而来的前端脚本语言（脚本语言是一种轻量级的语言），是一种解释性语言（代码执行不需要预编译）；被设计用来向 **HTML** 页面添加交互行为，通常被直接嵌入到 **HTML** 页面。

如果要在 **HTML** 页面中使用 **JavaScript**，我们需要使用<script>标签，同时使用 **type** 属性来定义脚本语言：

```
<html>

    <body>

        <script type="text/javascript">

            document.write("Hello World!");

        </script>

    </body>
```

```
</html>
```

通过`<script type="text/javascript">` 和`</script>` 就可以告诉浏览器 JavaScript 脚本从何处开始，到何处结束。使用 `document.write()` 可以向文档输出写内容。

XML 简介

XML 是指扩展标记语言，是标准通用标记语言的一个子集；与 HTML 类似，但它并非 HTML 的替代品，它们为不同的目的而设计；HTML 被设计用来显示数据，其焦点是数据的外观。XML 被设计为传输和存储数据，其焦点是数据的内容。

下面是一个简单的 XML

```
<?xml version="1.0"?>

<note>

    <to>George</to>

    <from>John</from>

    <heading>Reminder</heading>

    <body>Don't forget the meeting!</body>

</note>
```

`<?xml version="1.0"?>` 一个应该包含 XML 的声明，它定义了 XML 文档的版本号。

`<note></note>` 定义了文档里的第一个元素，也叫根元素。

`<to></to>`、`<from></from>`、`<heading></heading>`、`<body></body>` 为根元素的子元素，他们分别包含了发送者与接收者的信息。这个 XML 文档仅仅是用标签包装了纯粹的信息，我们需要编写软件或程序，才能传送、接收和显示出这个文档。

XML 允许我们自己定义标签，上例中的标签没有在任何 XML 标准中定义过，如`<to>` 和`<from>`，这些标签是由我们自己定义的。

上面只是简单的介绍了 HTML 、JavaScript 以及 XML 等前端技术，Web 自动化测试就是与前端技术打交道，所以，了解前端技术有助于我们顺利的进行 web 自动化测试工作，笔者推荐去 w3school 网站进一步学习和掌握这些技术。

第六节 前端工具介绍

FireBug

FireBug 是 FireFox 浏览器下的一套开发类插件，相信很多同学对这款前端工具并不陌生。它集 HTML 查看和编辑、Javascript 控制台、网络状况监视器、cookie 查看于一体，是开发 JavaScript、CSS、HTML 和 Ajax 的得力助手。



图 1.4

我们可以通过他方便的查看页面上的元素，从而根据其属性进行定位。在前 web 自动化测试工作中，此工具必不可少。

 安装方式：firefox 浏览器的菜单栏中选择 tools (工具)--->add-ons Manager(添加组件)，搜索 FireBug；对搜索到的插件进行安装，再次重启浏览器即可使用。

FirePath

FirePath 是 FireBug 插件扩展的一个开发工具，用来编辑、检查和生成的 XPath 1.0 表达式、CSS 3 选择器以及 jQuery 选择器。可以快速度的帮助我们通过 xPath 和 CSS 来定位页面上的元素。

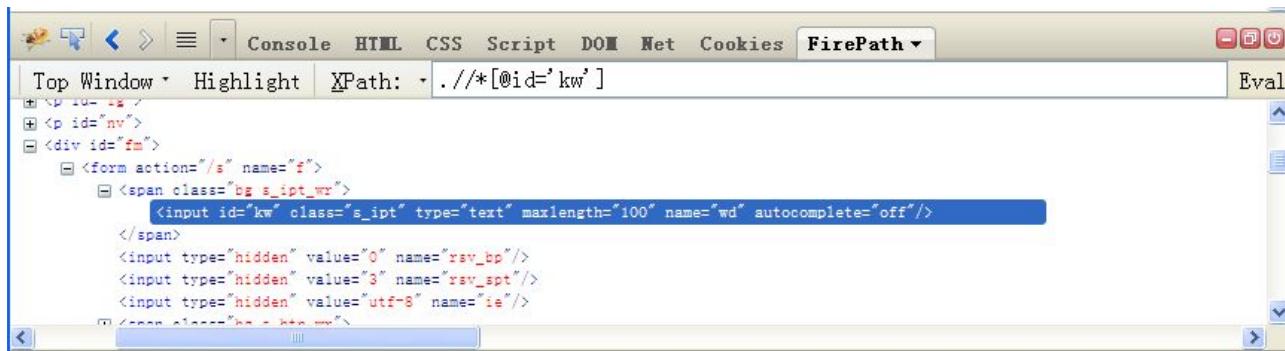


图 1.5

当通过 FireBug 的鼠标箭头选择一个页面元素后，FirePath 输入框将给出 XPath 的表达式，快速的帮助我们定位。注意：我们可以点击“XPath:”按钮切换到 CSS 定位方式，从而获得一个元素的 CSS 定位方式。FirePath 的安装方式与 FireBug 类似。

chrome 和 IE 的开发人员工具

chrome 和 IE 浏览器同样也提供了类似 FireBug 的开发人员工具，可以帮助我们定位页面元素。

chrome 浏览器默认自带 chrome 开发者工具，浏览器右上角的小扳手，在下拉菜单中选择“工具”--“开发者工具”即可打开，更为快捷的是通过 Ctrl+Shift+I 或 F12 打开。

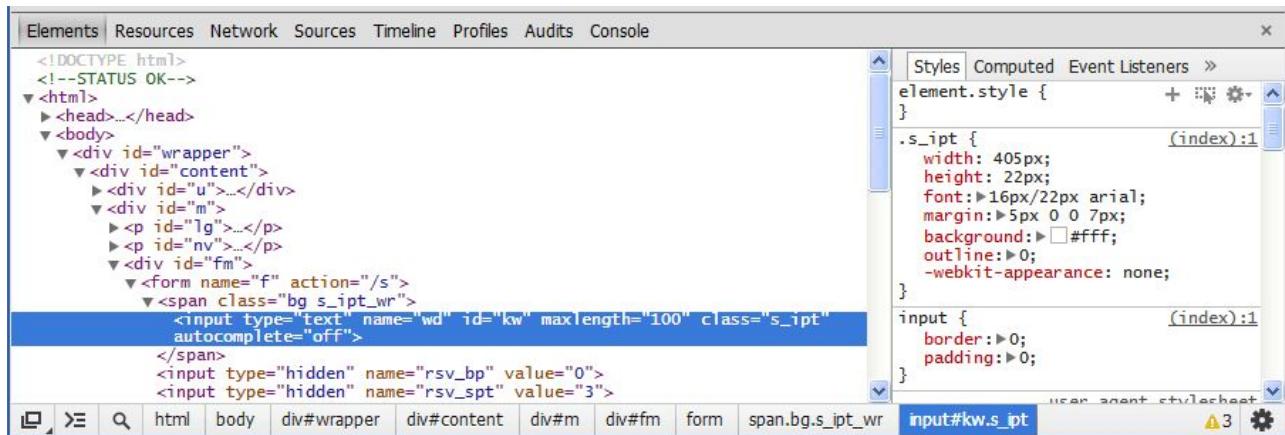


图 1.6

IE 浏览器从 IE8 版本开始，加入了开发人员工具，使用它也非常方便，通过菜单栏“工具”---“开发人员工具”或者通过快捷键 F12 即可打开。值得一提的是，它提供了浏览器的兼容模式，我们可以选择浏览器模式切换到 IE7 模式，IE 9/10 同样提供向下兼容模式到 IE7，这将非常方便的帮助我们测试 IE 浏览器的兼容性。

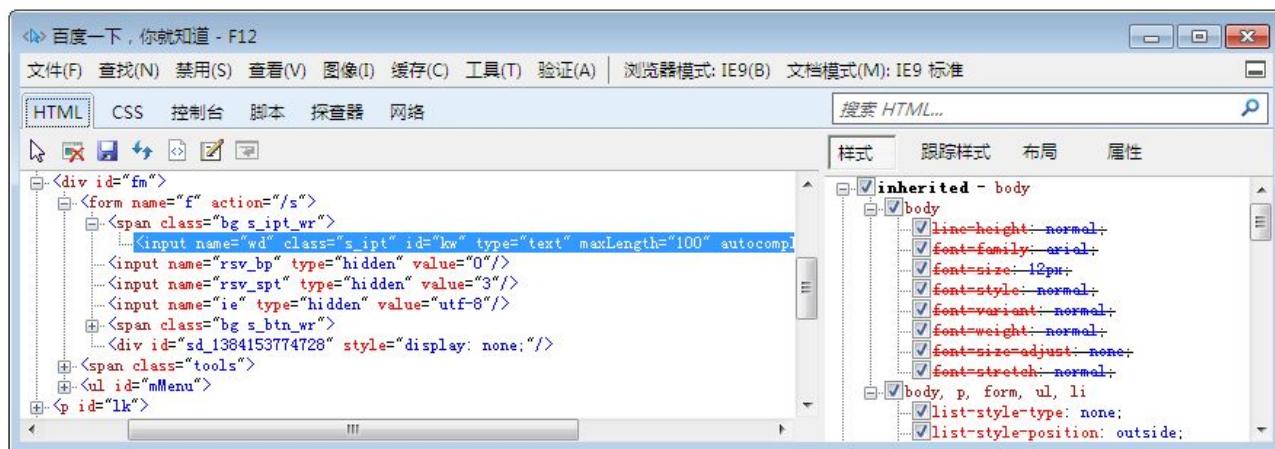


图 1.7

第七节 自动化测试语言的选择

通过前面的介绍，我们了解到 selenium webdriver 支持多种语言的开发，java、python、ruby、php、C#、JavaScript 等，那么我们应该选择哪一种语言结合 selenium webdriver 进行开发呢？这里笔者给出一点自己的看法。

有同学说我们公司的软件是用某种语言开发，自动化也要选某语言；其实从本质上来说，使用软件开发语言和自动化开发语言没有任何联系。所以，在选择语言进行自动化测试时不要有这方面的顾虑。从个人来讲，自动化测试所用到代码与开发人员相同，自己的编码能力一般没开发人员强，会糟鄙视，从而也降低了自身的不可替代性。

选择与开发相同的语言当然也有有利的一面，测试人员通过自动化测试的实践，提高了自己的编码能力，也有助于其它测试工作的进行，比如，协助开发人员定位代码级的 bug，协助开发人员进行接口测试等。

本书并没有向前面几本 selenium 书选用应用更为广泛的 java、C#，而是选用了 python，主要有以下几个方面考虑。

对于编程能力比较弱的初学者来说，python 与 ruby 等语言更容易学习和使用。通过自动化测试技术的实践，我们不仅掌握了自动化测试技术，从而也掌握一门语法简单且功能强大的脚本语言。（本书中对涉及到的 python 知识都会做详细的讲解，所以没有 python 基础的同学完全不用担心），那为什么不选 ruby 而选 python 呢？从笔者角度来看，python 语除了在自动化测试领域有出色的表现外，在系统编程，网络编程，web 开发，GUI 开发，科学计算，游戏开发等多个领域应用非常广泛，而且具有非常良好的社区支持。也就是说学习和掌握 python 编程，其实是为你打开了一道更广阔的大门。ruby 是一个“魔法”语

言，时常会给你带来很多惊喜，python 的宗旨是使处理问题变得更简单，而且格式严谨，在协同编程时不容易产生混乱。所以，综合考虑笔者认为 python 更适合测试菜鸟的养成计划。

那么对于有编程经验的同学，学习 python 对你来说几乎没有任何成本，你完全可以在很短的时间内学习和使用 python 处理问题，有一个看上去还不错的一门语言，为什么不去尝试使用一下呢！？当然，对于非常“专一”的同学，只愿意选择自己熟悉的语言，而不愿意尝试使用新语言，那么本更多的是传递你处理问题的思路，虽然编程语言的语法有差异，但仍然可以对你的自动化工作提供解决问题的思路。

虽然本书中涉及到 python 的知识都会进行讲解，但为了你能系统全面的使用 python 语言，笔者建议准备好一本 python 基础教程在身边，以便有疑问的地方随时翻阅学习。

第二章 python webdriver 环境搭建

有了上一节的基础后，下面我们就开始动手搭建自己的自动化测试环境。这也是我们实施自动化测试的准备工作，本书选用 python 语言，如果读者选用的其它语言，请参考其它资料进行环境的搭建。

第一节 环境搭建

准备工具如下：

下载 python 【python 开发环境】

<http://python.org/getit/>

下载 setuptools 【python 的基础包工具】

<http://pypi.python.org/pypi/setuptools>

下载 pip 【python 的安装包管理工具】

<https://pypi.python.org/pypi/pip>

要想使用 python 语言开发，首先需要 python 开发环境，需要说明的是 python 目前最新版本分：2.7.x 和3.3.x（简称 python 2 和 python 3）；python 3 并非完全的向下兼容 python 2，语法上也有较大的差异。python 3在性能上更加优秀，但由于 python 2多年的发展，大量的类库、框架是基于 python 2，所以，目前两个版本都在维护更新。笔者推荐新手从 python 2开始学习 python，因为有丰富的资料、类库和框架给我们学习和使用。当然，随着时间的推移，python 3 才是 python 发展的未来。

setuptools 是 python 的基础包工具，可以帮助我们轻松的下载，构建，安装，升级，卸载 python 的软件包。

pip 是 python 软件包的安装和管理工具，有了这个工具，我们只需要一个命令就可以轻松的 python 的

任意类库。

windows 环境安装

第一步、安装 python 的开发环境包，选择需要安装路径进行安装，笔者下载的是目前最新的 python2.7.5 版本，安装目录为：C:\Python27。

第二步、安装 setuptools 通过前面提供的 setuptools 的连接，拖动页面到底部找到，setuptools-1.3.2.tar.gz 文件（版本随着时间版本会有更新），对文件进行解压，找到 ez_install.py 文件，进入 windows 命令提示（开始--运行--cmd 命令，回车）下执行 ez_install.py：

```
C:\setuptools-1.3>python ez_install.py
```

如果提示 python 不是内部或外部命令！别急，去添加一下 python 的环境变量吧！桌面“我的电脑”右键菜单-->属性-->高级-->环境变量-->系统变量中的 Path 为：

变量名： PATH

变量值： ;C:\Python27

第三步、安装 pip，通过上面提供的链接下载 pip-1.4.1.tar.gz（版本随着时间版本会有更新），我默认解压在了 C:\pip-1.4.1 目录下，打开命令提示符（开始--运行--cmd 命令，回车）进入 C:\pip-1.4.1 目录下输入：

```
C:\pip-1.4.1 > python setup.py install
```

再切换到 C:\Python27\Scripts 目录下输入：

```
C:\Python27\Scripts > easy_install pip
```

第四步、安装 selenium，如果是电脑处于联网状态的话，可以直接在 C:\Python27\Scripts 下输入命令安装：

```
C:\Python27\Scripts > pip install -U selenium
```

如果没联网，可以通过下载安装：

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

下载 selenium 2.33.0（目前的最新版本），并解压把整个目录放到 C:\Python27\Lib\site-packages 目录下。

linux 环境安装

下面以 unbuntu 为例进行安装，其它版本的 linux 可能会有所差异，在绝大多数 linux 和 UNIX 系统安装中（包括 Mac OS X），Python 的解释器就已经存在了。我们需要做的就是打开终端，输入 python 命令

令进行验证，这里不再介绍 python 的安装。

第一步、安装：setuptools

```
root@fnngj-H24X:~# apt-get install python-setuptools
```

第二步、安装 pip

下载 pip 安装文件，切换到文件目录，对其进行解压：

```
root@fnngj-H24X:/home/user/python# tar -zxvf pip-1.4.1.tar.gz
```

切换到解压目录：

```
root@fnngj-H24X:/home/user/python# cd pip-1.4.1/
```

进行 pip 的安装

```
root@fnngj-H24X:/home/user/python/pip-1.4.1# python setup.py install
```

第三步、安装 selenium

```
root@fnngj-H24X:/home/user/python/pip-1.4.1# pip install -U selenium
```

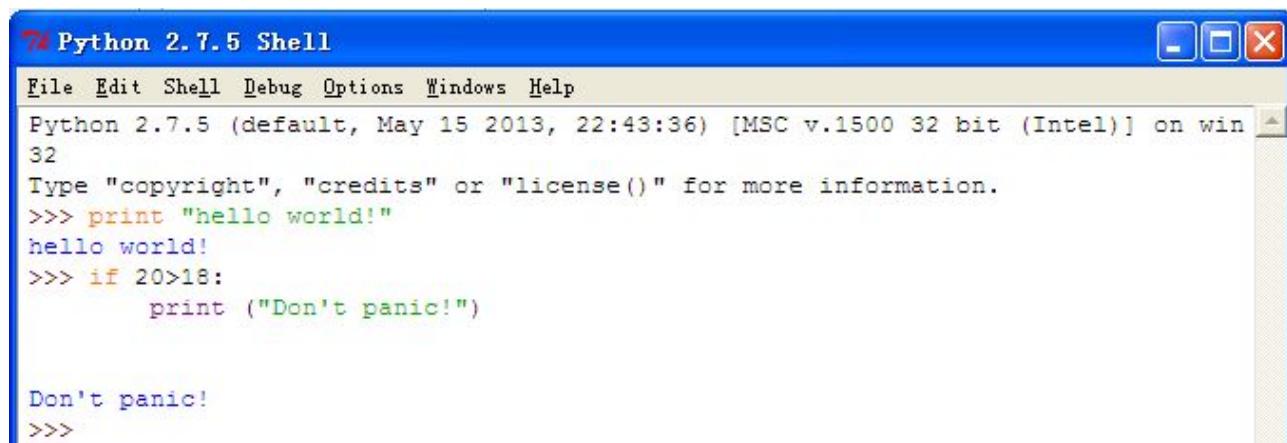
第二节 使用 IDLE 来编写 python

相信上面的配置过程已经让不少新手感到非常繁琐，万事开头难，我们有必要花一点时间在环境的配置上，因为环境的搭建是后面实施自动化测试的前提。

如果笔者是第一次接触 python 语言且编程能力薄弱，那么笔者建议使用 python 自带的 IDLE 来编写脚本。为了更好的通过 IDLE 帮助们编写 python+webdriver 脚本，我们需要先了解一下 IDLE。

IDLE 提供了一个功能完备的代码编辑器，允许你在这个编辑器中编写代码，另外还有一个 python shell (python 的交互模式)，可以在其中试验运行代码。

第一次启动 IDLE 时，会显示“三个大于号”提示符 (>>>)，可以在这里输入代码。python shell 得到你的代码语句后会立即执行，并在屏幕上显示生成的结果。如图 2.1



```

74 Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello world!"
hello world!
>>> if 20>18:
    print ("Don't panic!")

Don't panic!
>>>

```

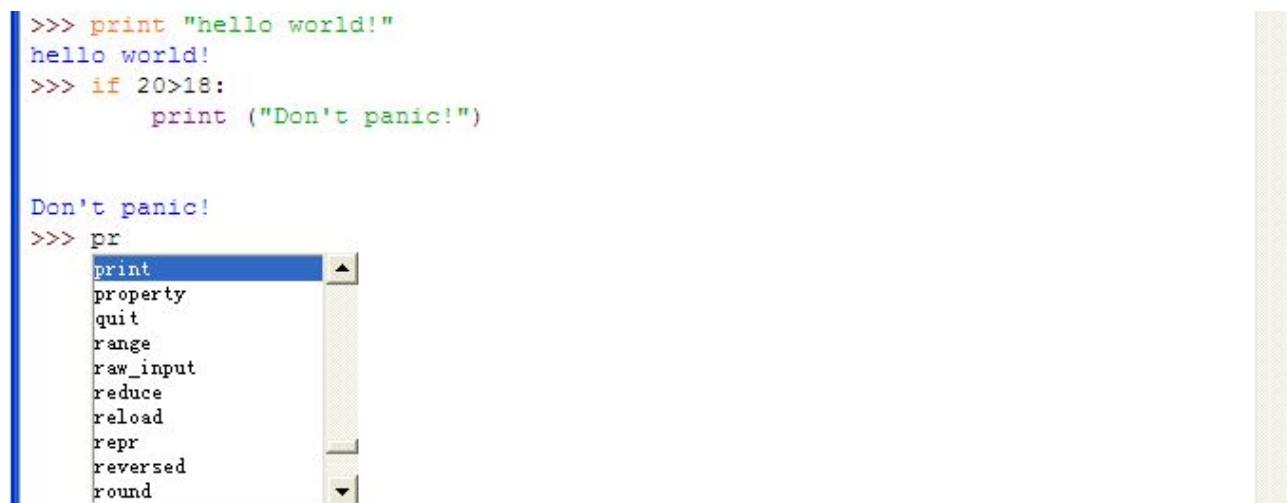
图 2.1

有效地使用 IDLE

IDLE 提供了大量的特性，不过只需了解其中一小部分就能很好地使用 IDLE。

TAB 完成：

先键入一些代码，然后按下 TAB 键。IDLE 会提供一些建议，帮助你完成这个语句。



```

>>> print "hello world!"
hello world!
>>> if 20>18:
    print ("Don't panic!")

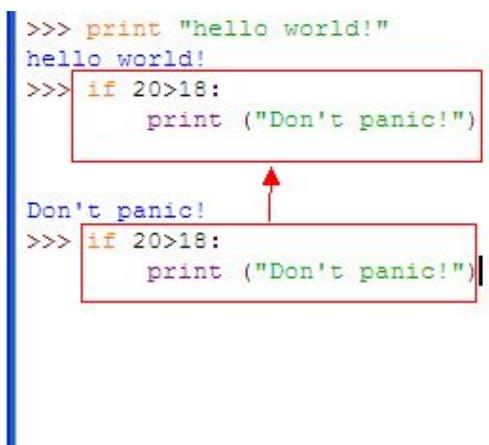
Don't panic!
>>> pr
    print
property
quit
range
raw_input
reduce
reload
repr
reversed
round

```

图 2.2

回退代码语句：

按下 Alt+P，可回退到 IDLE 中之前输入的代码语句，或者按下 Alt+ N 可以移至下一个代码语句。
如图 2.x 按 Alt+P 回退到上一次编辑的代码。



```

>>> print "hello world!"
hello world!
>>> if 20>18:
    print ("Don't panic!")

Don't panic!
>>> if 20>18:
    print ("Don't panic!")

```

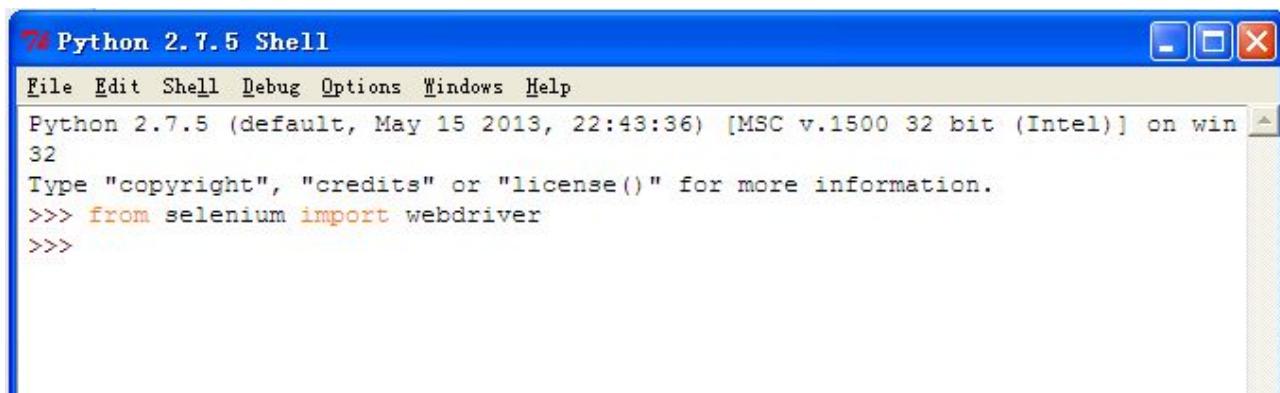
图 2.3

第三节 第一个自动化脚本

有了上面的环境，你一定很迫切想要编写并运行一个自动化脚本，下面就来体验一下 python 与 webdriver 结合之后编写的脚本是多么简洁：

如果是 windows 用户，在开始菜单找到 python 目录，打开 IDLE (python GUI) 程序，启动的是一个交互模式。可以输入：from selenium import webdriver

上面的命令为导入 selenium 的相关包，如果回车后没有报错表示我们的 selenium 安装是成功的。



```

Python 2.7.5 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.5 (default, May 15 2013, 22:43:36) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> from selenium import webdriver
Imported selenium
>>>

```

图 2.4

下面通过选择菜单栏 File--->New Windows 或通过快捷键 Ctrl+N 打开新的窗口。输入以下代码：

```

# coding = utf-8
from selenium import webdriver

browser = webdriver.Firefox()

```

```

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

browser.find_element_by_id("kw").send_keys("selenium")
browser.find_element_by_id("su").click()

browser.quit()

```

输入完成后命令为 baidu.py 保存，按 F5 快捷键运行脚本，将看到脚本启动 Firefox 浏览器进入百度页，输入“selenium” 点击搜索按钮，最后关闭浏览器的过程。（这里默认读者已经安装了 Firefox 浏览器）

我们后面的脚本也将会在这个编辑器下完成，在你还没找到更好的编辑器之前。

代码解析：

```
# coding = utf-8
```

为了防止乱码问题，以及方便的在程序中添加中文注释，把编码统一成 UTF-8。

```
from selenium import webdriver
```

导入 selenium 的 webdriver 包，只有导入 webdriver 包我们才能使用 webdriver API 进行自动化脚本的开发。**import 所引入包，更专业的叫法为：模组（modules）**

```
browser = webdriver.Firefox()
```

需要将控制的 webdriver 的 Firefox 赋值给 browser；获得了浏览器对象才可以启动浏览器，打开网址，操作页面严肃，Firefox 是默认已经在 selenium webdriver 包里了，所以可以直接调用。当然也可以调用 Ie 或 Chrome ，不过要先安装相关的浏览器驱动才行。

```
browser.get("http://www.baidu.com")
```

获得浏览器对象后，通过 get() 方法，可以向浏览器发送网址。

```
browser.find_element_by_id("kw").send_keys("selenium")
```

关于页面元素的定位后面将会详细的介绍，这里通过 id=kw 定位到百度的输入框，并通过键盘方法 send_keys() 向输入框里输入 selenium 。多自然语言呀！

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

这一步通过 id=su 定位的搜索按钮，并向按钮发送单击事件（ click() ）。

```
browser.quit()
```

退出并关闭窗口的每一个相关的驱动程序。

第四节 安装浏览器驱动



WebDriver 支持 Firefox (FirefoxDriver)、IE (InternetExplorerDriver)、Opera (OperaDriver) 和 Chrome (ChromeDriver)。对 Safari 的支持由于技术限制在本版本中未包含，但是可以使用 SeleneseCommandExecutor 模拟。它还支持 Android (AndroidDriver) 和 iPhone (iPhoneDriver) 的移动应用测试。它还包括一个基于 HtmlUnit 的无界面实现，称为 HtmlUnitDriver。

各个浏览器驱动下载地址：

<https://code.google.com/p/selenium/downloads/list>

安装 chrome 浏览器驱动，下载 ChromeDriver_win32.zip(根据自己系统下载不同的版本驱动)，解压得到 chromedriver.exe 文件放到环境变量 Path 所设置的目录下，如果前面我们已经将 (C:\Python27) 添加到了环境变量 Path 所设置的目录，可以将 chromedriver.exe 放到 C:\Python27\目录下。

安装 IE 浏览器驱动，下载 IEDriverServer_Win32_x.x.zip, 将解压得到 IEDriverServer.exe，同样放置到 C:\Python27\目录下。

liunx 系统下，同样下载系统对应的浏览器驱动，并将驱动放置到环境变量 Path 所设置的目录下，这里不再详细介绍。

安装完成后可以用 IE 和 chrome 来替换 firefox 运行上面的例子。

```
browser = webdriver.Firefox()
```

替换为：

```
browser = webdriver.Ie()
```

或

```
browser = webdriver.Chrome()
```

如果程序能调用相应的浏览器运行，说明我们的浏览器驱动安装成功。

OperaDriver 是 WebDriver 厂商 Opera Software 和志愿者开发了对于 Opera 的 WebDriver 实现。安装方式与 IE、chrome 有所不同；请参考其它文档进行安装。

第三章 python webdriver API

这一章将详细的讲解基于 python 的 webdriver API, 笔者更愿意读者自己去查询 webdriver API 中各种操作方法的使用, 为了保持本书由浅入深的完整性, 本章将用相当有篇幅介绍基于 python 语言的 webdriver 对种操作的使用。通过本章的学习, 我们掌握 web 页面上各种元素、弹窗的定位与操作, 以及浏览器 cookie 的操作, JavaScript 的调用等问题。

第一节、浏览器的操作

3.1.1、浏览器最大化

在统一的浏览器大小下运行用例, 可以比较容易的跟一些基于图像比对的工具进行结合, 提升测试的灵活性及普遍适用性。比如可以跟 sikuli 结合, 使用 sikuli 操作 flash。

```
#coding=utf-8
from selenium import webdriver

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

print "浏览器最大化"
driver.maximize_window() #将浏览器最大化显示
driver.quit()
```

3.1.2、设置浏览器宽、高

在不同的浏览器大小下访问测试站点，对测试页面截图并保存，然后观察或使用图像比对工具对被测页面的前端样式进行评测。比如可以将浏览器设置成移动端大小(320x480)，然后访问移动站点，对其样式进行评估；

```
#coding=utf-8

from selenium import webdriver


driver = webdriver.Firefox()
driver.get("http://m.mail.10086.cn")

#参数数字为像素点

print "设置浏览器宽480、高800显示"

driver.set_window_size(480, 800)

driver.quit()
```

3.1.3、控制浏览器前进、后退

浏览器上有一个后退、前进按钮，对于浏览网页的人是比较方便的；对于 web 自动化测试来说是一个比较难模拟的操作；webdriver 提供了 back() 和 forward() 方法，使实现这个操作变得非常简单。

```
#coding=utf-8

from selenium import webdriver
import time


driver = webdriver.Firefox()

#访问百度首页

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

print "now access %s" %(first_url)
```

```

driver.get(first_url)

#访问新闻页面
second_url='http://news.baidu.com'
print "now access %s" %(second_url)
driver.get(second_url)

#返回（后退）到百度首页
print "back to %s "%(first_url)
driver.back()

#前进到新闻页
print "forward to %s"% (second_url)
driver.forward()

driver.quit()

```

为了使脚本的执行过程看得更清晰，在每一步操作上都加了 print 来打印当前的 URL 地址。

运行结果如下：

```

>>> ===== RESTART =====
>>>
now access http://www.baidu.com
now access http://news.baidu.com
back to http://www.baidu.com
forward to http://news.baidu.com

```

实际测试中，这两个功能平时很少被使用，笔者所能想到的场景就是几个页面来回跳转，但又不想用 get url 的情况下。

python 基础知识补充：

下面打开 python shell 做以下练习：

```
>>> name = 'huhu'
```

```

>>> age = 26

>>> print "my name is %s" %name
my name is huhu

>>> print "my age is %d" %age
my age is 26

>>> print "my name is %d" %name

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print "my name is %d" %name
TypeError: %d format: a number is required, not str

>>> print "my name is %s ,age is %d" %(name,age)
my name is huhu ,age is 26

```

在 python 2 中使用 print 语句进行打印输出,如果是字符信息的话需要对打印的信息加单引号 (‘’) 或双引号 (“”),它们本质上没有任何区别,不过使用引号时必须要成对出现。

上面的例子中我们定义一个字符串变量 name 和一个数据变量 age,要想在 print 打印字符串中引用这两个变量就要用到“格式化字符串”的东西,在 print 打印字符串中指定变量类型,“%s”表示输出的类型为字符串,“%d”表示输出类型为整型数字。

name 为字符串类型,我们用%d 来指定输出类型就会报错。如果我们不确定变量类型的话可以使用%r,它的含义是“不管什么都打印出来”。

第二节 简单对象的定位



对象(元素)的定位和操作是自动化测试的核心部分,其中操作又是建立在定位的基础上的,因此元素定位就显得非常重要。(本书中用到的[对象与元素同为一个事物](#))

一个对象就像是一个人,他会有各种的特征(属性),如比我们可以通过一个人的身份证号、姓名或者他的住址找到这个人。那么一个元素也有类似的属性,我们可以通过这种唯一区别于其它元素的属性来定位这个元素。当然,除了要操作元素时需要定位元素外,有时候我们只是为了获得元素的属性(class 属性, name 属性)、text 或数量也需要定位元素。

webdriver 提供了一系列的元素定位方法,常用的有以下几种

- id
- name
- class name
- tag name
- link text
- partial link text
- xpath
- css selector

分别对应 python webdriver 中的方法为：

```
find_element_by_id()
find_element_by_name()
find_element_by_class_name()
find_element_by_tag_name()
find_element_by_link_text()
find_element_by_partial_link_text()
find_element_by_xpath()
find_element_by_css_selector()
```

3.2.1 id 和 name 定位

id 和 name 是我们最常用的定位方式，因为大多数元素都有这两个属性，而且在对控件的 id 和 name 命名时一般使其有意义也会取不同的名字。通过这两个属性使我们找一个页面上的属性变得相当容易。

```
<input id="gs_htif0" class="gsfi" aria-hidden="true" dir="ltr">
<input type="submit" name="btnK" jsaction="sf.chk" value="Google 搜索">
<input type="submit" name="btnI" jsaction="sf.lck" value="手气不错 ">
```

通过元素中所带的 id 和 name 属性对元素进行定位：

```
id="gs_htif0"
find_element_by_id("gs_htif0")
name="btnK"
find_element_by_name("btnK")
```

```
name=" btnI"
find_element_by_name("btnI")
```



3.2.2 tag name 和 class name 定位

不是所有的前端开发人员都喜欢为每一个元素添加 id 和 name 两个属性，但除此之外你一定发现了一个元素不单单只有 id 和 name，它还有 class 属性；而且每个元素都会有标签。

```
<div id="searchform" class="jhp_big" style="margin-top:-2px">
<form id="tsf" onsubmit="return name='f' method="GET" action="/search">
<input id="kw" class="s_ipt" type="text" name="wd" autocomplete="off">
```

通过元素中带的 class 属性对元素进行定位：

```
class=" jhp_big"
find_element_by_class_name("jhp_big")
class=" s_ipt"
find_element_by_class_name("s_ipt")
```

通过 tag 标签名对对元素进行定位：

```
<div>
find_element_by_tag_name("div")
<form>
find_element_by_tag_name("form")
<input>
find_element_by_tag_name("input")
```

tag name 定位应该是所有定位方式中最不靠谱的一种了，因为在一个页面中具有相同 tag name 的元素极其容易出现。

3.2.3 link text 与 partial link text 定位

有时候需要操作的元素是一个文字链接，那么我们可以通过 link text 或 partial link text 进行元素定位。

```
<a href="http://news.baidu.com" name="tj_news">新闻</a>
<a href="http://tieba.baidu.com" name="tj_tieba">贴吧</a>
<a href="http://zhidao.baidu.com" name="tj_zhidao">一个很长的文字连接</a>
```

通过 link text 定位元素：

```
find_element_by_link_text("新闻")
find_element_by_link_text("贴吧")
find_element_by_link_text("一个很长的文字连接")

通 partial link text 也可以定位到上面几个元素：
find_element_by_partial_link_text("新")
find_element_by_partial_link_text("吧")
find_element_by_partial_link_text("一个很长的")
```

当一个文字连接很长时，我们可以只取其中的一部分，只要取的部分可以唯一标识元素。一般一个页面上不会出现相同的文件链接，通过文字链接来定位元素也是一种简单有效的定位方式。

下面介绍 xpath 与 CSS 定位相比上面介绍的方式来说比较难理解，但他们的灵活性与定位能力比上面的方式要强大。

3.2.4 XPath 定位

XPath 是一种在 XML 文档中定位元素的语言。因为 HTML 可以看做 XML 的一种实现，所以 selenium 用户可是使用这种强大语言在 web 应用中定位元素。

XPath 扩展了上面 id 和 name 定位方式，提供了很多种可能性，比如定位页面上的第三个复选框。

```
<html class="w3c">
<body>
    <div class="page-wrap">
        <div id="hd" name="q">
```

```
<form target="_self" action="http://www.so.com/s">
    <span id="input-container">
        <input id="input" type="text" x-webkit-speech="" autocomplete="off"
suggestwidth="501px" >
```

我们看到的是一个有层级关系页面，下面我看看如果用 xpath 来定位最后一个元素。

用绝对路径定位：

```
find_element_by_xpath("/html/body/div[2]/form/span/input")
```

当我们所要定位的元素很难找到合适的方式时，都可以通这种绝对路径的方式位，缺点是当元素在很多级目录下时，我们不得不要写很长的路径，而且这种方式难以阅读和维护。

相对路径定位：

```
find_element_by_xpath("//input[@id='input']") #通过自身的 id 属性定位
```

```
find_element_by_xpath("//span[@id='input-container']/input") #通过上一级目录的 id 属性定位
```

```
find_element_by_xpath("//div[@id='hd']/form/span/input") #通过上三级目录的 id 属性定位
```

```
find_element_by_xpath("//div[@name='q']/form/span/input")#通过上三级目录的 name 属性定位
```

通过上面的例子，我们可以看到 XPath 的定位方式非常灵活和强大的，而且 XPath 可以做布尔逻辑运算，例如：//div[@id='hd' or @name='q']

当然，它的缺陷也非常明显：1、性能差，定位元素的性能要比其它大多数方式差；2、不够健壮，XPath 会随着页面元素布局的改变而改变；3. 兼容性不好，在不同的浏览器下对 XPath 的实现是不一样的。

通过我们第一章中介绍的 firebug 的 HTML 和 firePath 可以非常方便的通过 XPath 方式对页面元素进行定位。

打开 firefox 浏览器的 firebug 插件，点击插件左上角的鼠标箭头，再点击页面上的元素，firebug 插件的 HTML 标签页将看到页面代码，鼠标移动到元素的标签上（如图图3.1，<input>）将显示当前元素的绝对路径。

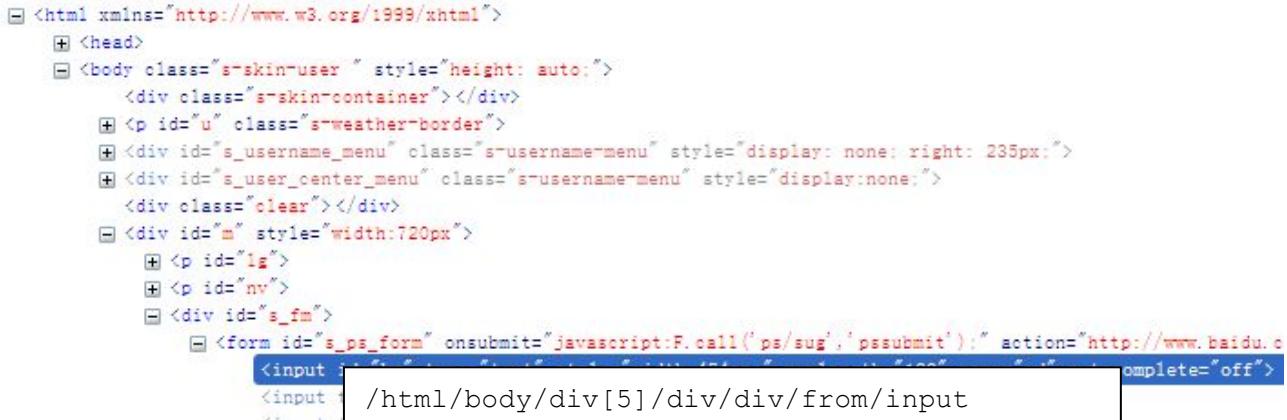


图3.1

或者直接在元素上右击弹出快捷菜单，选择 Copy XPath，将当前元素的 XPath 路径拷贝到脚本本中（如图3.2）。

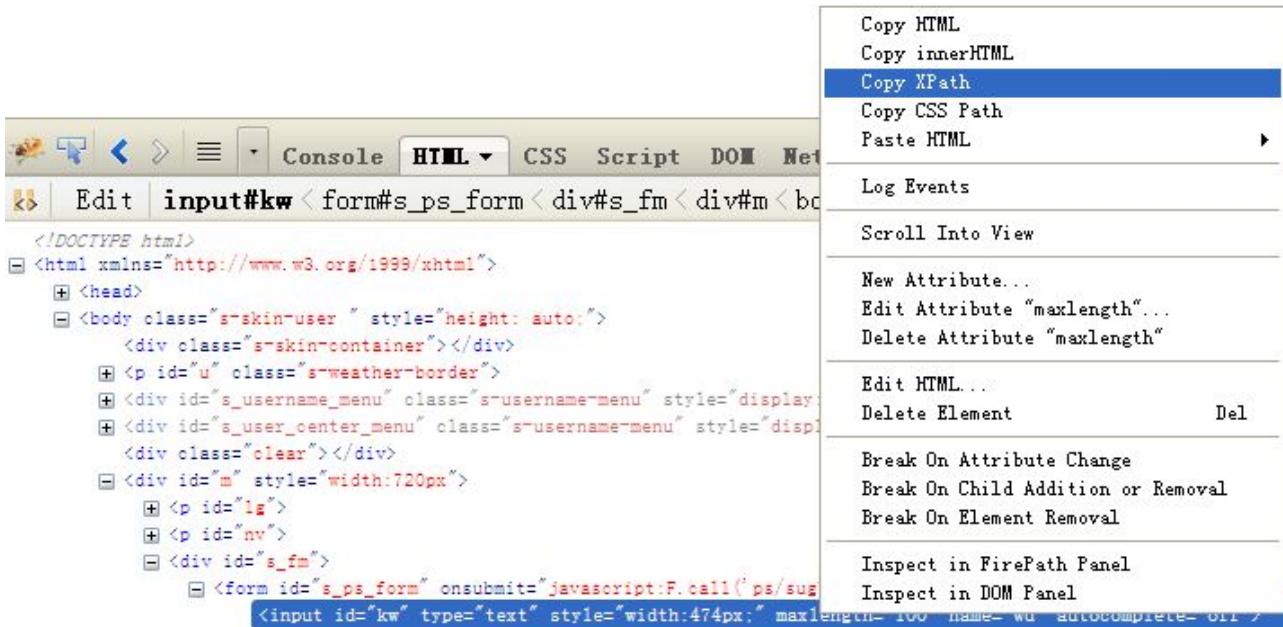


图3.2

firePath 工具的使用就更加方便和快捷了，选中元素后，直接在 XPath 的输入框中显示当前元素的 XPath 的定位信息（如图3.3）。

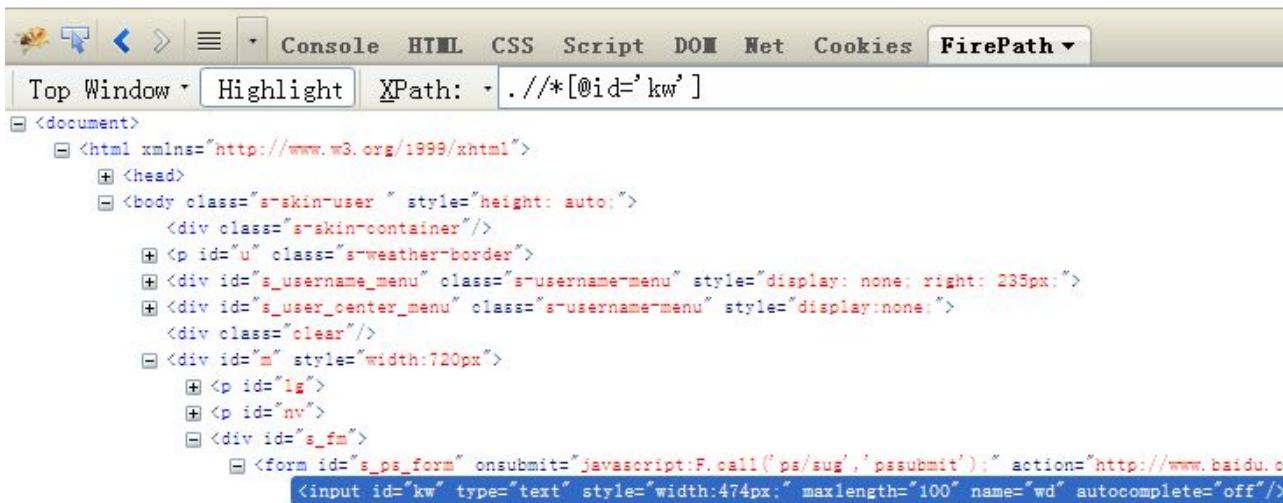


图3.3

3.2.5 CSS 定位

CSS (Cascading Style Sheets) 是一种语言，它被用来描述 HTML 和 XML 文档的表现。CSS 使用选择器来为页面元素绑定属性。这些选择器可以被 selenium 用作另外的定位策略。

CSS 可以比较灵活选择控件的任意属性，一般情况下定位速度要比 XPath 快，但对于初学者来说比较难以学习使用，下面我们就详细的介绍 CSS 的语法与使用。

CSS 选择器的常见语法：

*	通用元素选择器，匹配任何元素
E	标签选择器，匹配所有使用 E 标签的元素
.info	class 选择器，匹配所有 class 属性中包含 info 的元素
#footer	id 选择器，匹配所有 id 属性等于 footer 的元素
E,F	多元素选择器，同时匹配所有 E 元素或 F 元素，E 和 F 之间用逗号分隔
E F	后代元素选择器，匹配所有属于 E 元素后代的 F 元素，E 和 F 之间用空格分隔
E > F	子元素选择器，匹配所有 E 元素的子元素 F
E + F	毗邻元素选择器，匹配紧随 E 元素之后的同级元素 F（只匹配第一个）
E ~ F	同级元素选择器，匹配所有在 E 元素之后的同级 F 元素
E[att='val']	属性 att 的值为 val 的 E 元素（区分大小写）
E[att^='val']	属性 att 的值以 val 开头的 E 元素（区分大小写）
E[att\$='val']	属性 att 的值以 val 结尾的 E 元素（区分大小写）
E[att*='val']	属性 att 的值包含 val 的 E 元素（区分大小写）
E[att1='v1'][att2*='v2']	属性 att1 的值为 v1，att2 的值包含 v2（区分大小写）
E:contains('xxxx')	内容中包含 xxxx 的 E 元素
E:not(s)	匹配不符合当前选择器的任何元素

例如下面一段代码：

```
<div class="formdiv">
<form name="fnfn">
<input name="username" type="text"></input>
<input name="password" type="text"></input>
<input name="continue" type="button"></input>
<input name="cancel" type="button"></input>
<input value="SYS123456" name="vid" type="text">
<input value="ks10cf6d6" name="cid" type="text">
</form>
<div class="subdiv">
<ul id="recordlist">
<p>Heading</p>
<li>Cat</li>
<li>Dog</li>
<li>Car</li>
```

```
<li>Goat</li>
</ul>
</div>
</div>
```



通过 CSS 语法进行匹配的实例：

locator	匹配
css=div	<div class="formdiv">
css=div.formdiv	
css=#recordlist	<ul id="recordlist">
css=ul#recordlist	
css=div.subdiv p	<p>Heading</p>
css=div.subdiv > ul > p	
css=form + div	<div class="subdiv">
css=p + li	二者定位到的都是 Cat
css=p ~ li	但是 storeCssCount 的时候，前者得到 1，后者得到 4
css=form > input[name=username]	<input name="username">
css=input[name\$id][value^=SYS]	<input value="SYS123456" name="vid" type="hidden">
css=input:not([name\$id][value^=SYS])	<input name="username" type="text"></input>
css=li:contains('Goa')	Goat
css=li:not(contains('Goa'))	Cat

css 中的结构性定位

结构性定位就是根据元素的父子、同级中位置来定位，css3标准中有定义一些结构性定位伪类如 nth-of-type, nth-child，但是使用起来语法很不好理解，这里就不做介绍了。

Selenium 中则是采用了来自 Sizzle 的 css3定位扩展，它的语法更加灵活易懂。

Sizzle Css3的结构性定位语法：

E:nth(n) E:eq(n)	在其父元素中的 E 子元素集合中排在第 n+1 个的 E 元素 (第一个 n=0)
E:first	在其父元素中的 E 子元素集合中排在第 1 个的 E 元素
E:last	在其父元素中的 E 子元素集合中排在最后 1 个的 E 元素
E:even	在其父元素中的 E 子元素集合中排在偶数位的 E 元素 (0,2,4...)

E:odd	在其父元素中的 E 子元素集合中排在奇数的 E 元素 (1,3,5...)
E:lt(n)	在其父元素中的 E 子元素集合中排在 n 位之前的 E 元素 (n=2,则匹配 0,1)
E:gt(n)	在其父元素中的 E 子元素集合中排在 n 位之后的 E 元素 (n=2,在匹配 3,4)
E:only-child	父元素的唯一一个子元素且标签为 E
E:empty	不包含任何子元素的 E 元素, 注意, 文本节点也被看作子元素

例如下面一段代码:

```
<div class="subdiv">
    <ul id="recordlist">
        <p>Heading</p>
        <li>Cat</li>
        <li>Dog</li>
        <li>Car</li>
        <li>Goat</li>
    </ul>
</div>
```

匹配示例:

locator	匹配
css=ul > li:nth(0)	Cat
css=ul > *:nth(0)	<p>Heading</p>
css=ul > :nth(0)	
css=ul > li:first	Cat
css=ul > :first	<p>Heading</p>
css=ul > *:last	Goat
css=ul > li:last	
css=ul > li:even	Cat, Car
css=ul > li:odd	Dog, Goat
css=ul > :even	<p>Heading</p>
css=ul > p:odd	[error] not found
css=ul > li:lt(2)	Cat
css=ul > li:gt(2)	Goat
css=ul > li:only-child	
css=ul > :only-child	[error] not found (ul 没有 only-child)
css=ul > *:only-child	
css=div.subdiv > :only-child	<ul id="recordlist"> ...

Sizzle Css3还提供一些直接选取 form 表单元素的伪类:

:input: Finds all input elements (includes textareas, selects, and buttons).

:text, :checkbox, :file, :password, :submit, :image, :reset, :button: Finds the input element with the specified input type (:button also finds button elements).

下面是一些 XPATH 和 CSS 的类似定位功能比较（缺乏一定的严谨性）。

定位方式	XPath	CSS
标签	//div	div
By id	//div[@id='eleid']	div#eleid
By class	//div[@class='eleclass'] //div[contains(@class,'eleclass')]	div.eleclass
By 属性	//div[@title='Move mouse here']	div[title=Move mouse here] div[title^=Move] div[title\$=here] div[title*=mouse]
定位子元素	//div[@id='eleid']/* //div/h1	div#eleid >* div#eleid >h1
定位后代元素	//div[@id='eleid']/h1	div h1
By index	//li[6]	li:nth(5)
By content	//a[contains(.,'Issue 1164')]	a:contains(Issue 1164)

通过对比，我们可以看到，CSS 定位语法比 XPath 更为简洁，定位方式更多灵活多样；不过对 CSS 理解起来要比 XPath 较难；但不管是从性能还是定位更复杂的元素上，CSS 优于 XPath，笔者更推荐使用 CSS 定位页面元素。

关于自动化的定位问题

自动化测试的元素定位一直是困扰自动化测试新手的一个障碍，因为我们在自动化实施过程中会碰到各式各样的对象元素。虽然 XPath 和 CSS 可以定位到复杂且比较难定位的元素，但相比较用 id 和 name 来说增加了维护成本和学习成本，相比较来说 id/name 的定位方式更直观和可维护，有新的成员加入的自动化时也增加了人员的学习成本。所以，测试人员在实施自动化测试时一定要做好沟通，规范前端开发人员对元素添加 id/name 属性，或者自己有修改 HTML 代码的权限。

第三节 操作测试对象

前面讲到了不少知识都是定位对象，定位只是第一步，定位之后需要对这个对象进行操作。鼠标点击呢还是键盘输入，这要取决于我们定位的对象所支持的操作。

一般来说，所有有趣的操作与页面交互都将通过 WebElement 接口，包括上一节中介绍的对象定位，以及本节中需要介绍的常对象操作。

webdriver 中比较常用的操作元素的方法有以下几个：

- clear 清除元素的内容，如果可以的话
- send_keys 在元素上模拟按键输入
- click 单击元素
- submit 提交表单

3.3.1、登录实例

下面以快播私有云登录实例来展示常见元素操作的使用：

```
#coding=utf-8

from selenium import webdriver


driver = webdriver.Firefox()

driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")



driver.find_element_by_id("user_name").clear()

driver.find_element_by_id("user_name").send_keys("username")

driver.find_element_by_id("user_pwd").clear()

driver.find_element_by_id("user_pwd").send_keys("password")

driver.find_element_by_id("dl_an_submit").click()

#通过 submit() 来提交操作

#driver.find_element_by_id("dl_an_submit").submit()




driver.quit()
```

`clear()` 用于清除输入框的默认内容

比如登录框内一般默认会有“账号”“密码”等提示信息，如果直接输入内容，和可能会与输入框的默认提示信息拼接，从而造成输入信息的错误；这时 clear () 将变得非常有用。

send_keys("xx") 用于在一个输入框里输入 xx 内容

python 是个容易出现编码问题的语言，有时候当我们在 send_keys() 方法中输入中文时，然后脚本在运行时就报编码错误，这个时候我们可以在脚本开头声明编码为 utf-8，然后在中文字符的前面加个小 u 就解决了（表示转成 python Unicode 编码）：

```
#coding=utf-8
```

```
send_keys(u"中文内容")
```

需要注意的是 utf-8 并不是万能的，如果 utf-8 不能解决，可以尝试将编码声明为 GBK；关于 python 的编码问题，请参考 python 相关书籍。

click() 用于单击一个按钮

其实 click() 方法不仅仅用于点击一个按钮，可以单击任何可以点击的元素，文字/图片连接，按钮，下拉按钮等。

submit() 提交表单

从上面有例子，我们可看到可以使用 submit() 方法来代替 click() 对输入的信息进行提交，在有些情况下两个方法可以相互使用； submit() 要求提交对象是一个表单，更强调对信息的提交。click() 更强调事件的独立性。（比如，一个文字链接就不能用 submit() 方法。）

3.3.2 WebElement 接口常用方法

WebElement 接口除了我们前面介绍的方法外，它还包含了别一些有用的方法。下面，我们例举例几个比较有用的方法。

size

返回元素的尺寸。例：

```
#返回百度输入框的宽高
size=driver.find_element_by_id("kw").size
print size
```

text

获取元素的文本，例：

```
#返回百度页面底部备案信息
```

```
text=driver.find_element_by_id("cp").text
print text
```

`get_attribute(name)`

获得属性值。例：

```
#返回元素的属性值，可以是 id、name、type 或元素拥有的其它任意属性
attribute=driver.find_element_by_id("kw").get_attribute('type')
print attribute
```

需要说明的是这个方法在定位一组时将变得非常有用，后面将有运行的实例。

`is_displayed()`

设置该元素是否用户可见。例：

```
#返回元素的结果是否可见，返回结果为 True 或 False
result=driver.find_element_by_id("kw").is_displayed()
print result
```

WebElement 接口的其它更多方法请参考 webdriver API。

第四节 鼠标事件

前面例子中我们已经学习到可以用 `click()` 来模拟鼠标的单击操作，而我们在实际的 web 产品测试中

发现，有关鼠标的操作，不单单只有单击，有时候还要和到右击，双击，拖动等操作，这些操作包含在 `ActionChains` 类中。

`ActionChains` 类鼠标操作的常用方法：

- `context_click()` 右击
- `double_click()` 双击
- `drag_and_drop()` 拖动
- `move_to_element()` 鼠标悬停在一个元素上
- `click_and_hold()` 按下鼠标左键在一个元素上

鼠标右击操作

context_click() 右键点击一个元素。



图3.4

如图3.x4,假如一个web应用的列表文件提供了右击弹出快捷菜单的操作。可以通过context_click()方法模拟鼠标右键，参考代码如下：

```
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains
...
#定位到要右击的元素
right = driver.find_element_by_xpath("xx")
#对定位到的元素执行鼠标右键操作
ActionChains(driver).context_click(right).perform()
....
```



from selenium.webdriver.common.action_chains import ActionChains

这里需要注意的是，在使用 ActionChains 类下面的方法之前，要先将包引入。

ActionChains(driver)

driver: webdriver 实例执行用户操作。

ActionChains 用于生成用户的行为；所有的行为都存储在 actionchains 对象。通过 perform() 执行存储的行为。

perform()

执行所有 ActionChains 中存储的行为。perfrome() 同样也是 ActionChains 类提供的的方法，通常与 ActionChains () 配合使用。

除了鼠标右击之外，ActionChains 类还提供了其它的比较复杂的鼠标方法。

鼠标双击操作

`double_click(on_element)`

双点击页面元素。例如：

```
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains
...
#定位到要双击的元素
double = driver.find_element_by_xpath("xxx")
#对定位到的元素执行鼠标双击操作
ActionChains(driver).double_click(double).perform()
```

对于操作系统的操作来说，双击使用相当频繁，但对于 web 应用来说双击的用户比较少，笔者唯一能想到想的场景是地图程序（如 百度地图），可以通过双击鼠标放大地图。

鼠标拖放操作

`drag_and_drop(source, target)`

在源元素上按下鼠标左键，然后移动到目标元素上释放。

`source`: 鼠标按下的源元素。

`target`: 鼠标释放的目标元素。

鼠标拖放操作的参考代码如下：

```
#引入 ActionChains 类
from selenium.webdriver.common.action_chains import ActionChains
...
#定位元素的原位置
element = driver.find_element_by_name("xxx")
#定位元素要移动到的目标位置
target = driver.find_element_by_name("xxx")
```

```
#执行元素的移动操作  
ActionChains(driver).drag_and_drop(element, target).perform()
```

鼠标移动上元素上

```
move_to_element()
```

模拟鼠标移动到一个元素上。

```
#引入 ActionChains 类  
  
from selenium.webdriver.common.action_chains import ActionChains  
  
...  
  
#定位到鼠标移动到上面的元素  
  
above = driver.find_element_by_xpath("xxx")  
  
#对定位到的元素执行鼠标移动到上面的操作  
  
ActionChains(driver).move_to_element(above).perform()
```

按下鼠标左键

```
click_and_hold()
```

按住鼠标左键在一个元素。

```
#引入 ActionChains 类  
  
from selenium.webdriver.common.action_chains import ActionChains  
  
...  
  
#定位到鼠标按下左键的元素  
  
left=driver.find_element_by_xpath("xxx")  
  
#对定位到的元素执行鼠标左键按下的操作  
  
ActionChains(driver).click_and_hold(left).perform()
```

第五节 键盘事件



我们在实际的测试工作中，有时候我们在测试时需要使用 tab 键将焦点转移到下一个元素，用于验证元素的排序是否正确。webdriver 的 Keys() 类提供键盘上所有按键的操作，甚至可以模拟一些组合键的操作，如 Ctrl+A , Ctrl+C/Ctrl+V 等。在某些更复杂的情况下，还会出现使用 send_keys 来模拟上下键来操作下拉列表的情况。

```
#coding=utf-8

from selenium import webdriver

#引入 Keys 类包

from selenium.webdriver.common.keys import Keys

import time


driver = webdriver.Firefox()

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



#输入框输入内容

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

time.sleep(3)


#删除多输入的一个 m

driver.find_element_by_id("kw").send_keys(Keys.BACK_SPACE)

time.sleep(3)


#输入空格键+“教程”

driver.find_element_by_id("kw").send_keys(Keys.SPACE)

driver.find_element_by_id("kw").send_keys(u"教程")

time.sleep(3)

#ctrl+a 全选输入框内容

driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'a')
```

```

time.sleep(3)

#ctrl+x 剪切输入框内容

driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'x')

time.sleep(3)

#输入框重新输入内容，搜索

driver.find_element_by_id("kw").send_keys(Keys.CONTROL, 'v')

time.sleep(3)

#通过回车键来代替点击操作

driver.find_element_by_id("su").send_keys(Keys.ENTER)

time.sleep(3)

driver.quit()

```

需要说明的是上面脚本没什么实际意义，但向我们展示了组合键及键盘按键的用法。为了使用脚本的运行过程更我们可以看得更加清晰，在每一步操作之后都加上了三秒的休眠时间 time.sleep(), 后面会再介绍 time.sleep()方法的使用。

```
from selenium.webdriver.common.keys import Keys
```

在使用键盘按键方法前需要先导入 keys 类包。

下面经常使用到的键盘操作：

```

send_keys(Keys.BACK_SPACE)    删除键 (BackSpace)

send_keys(Keys.SPACE)    空格键 (Space)

send_keys(Keys.TAB)    制表键 (Tab)

send_keys(Keys.ESCAPE)    回退键 (Esc)

send_keys(Keys.ENTER)    回车键 (Enter)

send_keys(Keys.CONTROL, 'a')    全选 (Ctrl+A)

send_keys(Keys.CONTROL, 'c')    复制 (Ctrl+C)

```

```
send_keys(Keys.CONTROL, 'x') 剪切 (Ctrl+X)
send_keys(Keys.CONTROL, 'v') 粘贴 (Ctrl+V)
```

Keys 类所提供的按键请查阅 webdriver API.

第六节 打印信息



当我们要设计功能测试用例时，一般会有预期结果，有些预期结果是由测试人员通过肉眼进行判断的。因为自动化测试运行过程是无人值守，一般情况下，脚本运行成功，没有异常信息就标识用户执行成功。当然，这还不足以证明一个用例确实是执行成功的。所以我们需要获得更多的信息来证明用例执行结果确实是成功的。

通常我们可以通过获得页面的 title 、 URL 地址，页面上的标识性信息（如，登录成功的“欢迎，xxx”信息）来判断用例执行成功。

在实际测试中，访问 1 个页面然后判断其 title 是否符合预期是很常见的一个用例，假如一个页面的 title 应该是“快播私有云”，那么用例可以这样描述：访问该页面，判断页面 title 是否等于“快播私有云”。

获取当前 URL 也是非常重要的一个操作，在某些情况下，你访问一个 URL，这时系统会自动对这个 URL 进行跳转，这就是所谓的“重定向”。一般测试重定向的方法是访问这个 URL，然后等待页面重定向完毕之后，获取当前页面的 URL，判断该 URL 是否符合预期。如果页面的 URL 返回不正确，而表示当前操作没有进行正常的跳转。

下面通过快播私有云登录实例进行讲解：

```
#coding=utf-8

from selenium import webdriver


driver = webdriver.Firefox()

driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F%2Fwebcloud
.kuaibo.com%2F")

#登录

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

```
driver.find_element_by_id("user_name").send_keys("username")
driver.find_element_by_id("user_pwd").clear()
driver.find_element_by_id("user_pwd").send_keys("password")
driver.find_element_by_id("dl_an_submit").click()
```

#获得前面 title， 打印

```
title = driver.title
print title
```

#拿当前 URL 与预期 URL 做比较

```
if title == u"快播私有云":
    print "title ok!"
else:
    print "title on!"
```

#获得前面 URL， 打印

```
now_url = driver.current_url
print now_url
```

#拿当前 URL 与预期 URL 做比较

```
if now_url == "http://webcloud.kuaibo.com/":
    print "url ok!"
else:
    print "url on!"
```

#获得登录成功的用户， 打印

```
now_user=driver.find_element_by_xpath("//div[@id='Nav']/ul/li[4]/a[1]/span")
.text
```

```
print now_user

driver.quit()
```

运行结果：

```
>>> ===== RESTART =====

>>>

快播私有云

title ok!

http://webcloud.kuaibo.com/

url ok!

虫师
```

本例中涉及到新的方法如下：

[title](#)

返回当前页面的标题

[current_url](#)

获取当前加载页面的 URL

在上面的例子中我们用到了 python 的 if 判断语句，与其它语言没有差异，python 的 if 语句块用冒号（：）表示后面需要执行的语句。



第七节 设置等待时间

有时候为了保证脚本运行的稳定性，需要脚本中添加等待时间。

[sleep\(\)](#): 设置固定休眠时间。python 的 time 包提供了休眠方法 sleep()，导入 time 包后就可以使用 sleep() 进行脚本的执行过程进行休眠。

[implicitly_wait\(\)](#): 是 webdirver 提供的一个超时等待。隐的等待一个元素被发现，或一个命令完成。如果超出了设置时间的则抛出异常。

[WebDriverWait\(\)](#): 同样也是 webdirver 提供的方法。在设置时间内，默认每隔一段时间检测一次当前页面元素是否存在，如果超过设置时间检测不到则抛出异常。

下面通过实例来展示方法的具体使用：

```
#coding=utf-8

from selenium import webdriver

#导入 WebDriverWait 包

from selenium.webdriver.support.ui import WebDriverWait

#导入 time 包

import time


driver = webdriver.Firefox()

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



#WebDriverWait()方法使用

element=WebDriverWait(driver, 10).until(lambda driver : driver.find_element_by_id("kw"))

element.send_keys("selenium")


#添加智能等待

driver.implicitly_wait(30)

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


#添加固定休眠时间

time.sleep(5)


driver.quit()

sleep()
```

sleep()方法以秒为单位，假如休眠时间为 1 秒，可以用小数表示。

```
import time

.....


time.sleep(5)
```

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

当然，也可以直接导入 sleep()方法，使脚本中的引用更简单

```
from time import sleep
...
sleep(3)
sleep(30)
```

[implicitly_wait\(\)](#)

implicitly_wait()方法比 sleep() 更加智能，后者只能选择一个固定的时间的等待，前者可以在一个时间范围内智能的等待。

[WebDriverWait\(\)](#)

详细格式如下：

[WebDriverWait\(driver, timeout, poll_frequency=0.5, ignored_exceptions=None\)](#)

driver - WebDriver 的驱动程序(Ie, Firefox, Chrome 或远程)

timeout - 最长超时时间， 默认以秒为单位

poll_frequency - 休眠时间的间隔（步长）时间， 默认为 0.5 秒

ignored_exceptions - 超时后的异常信息， 默认情况下抛 NoSuchElementException 异常。

实例：

```
from selenium.webdriver.support.ui import WebDriverWait
...
element = WebDriverWait(driver, 10).until(lambda x: x.find_element_by_id("someId"))
is_disappeared = WebDriverWait(driver, 30, 1, (ElementNotVisibleException)).
    until_not(lambda x: x.find_element_by_id("someId").is_displayed())
```

WebDriverWait()一般由 unit()或 until_not()方法配合使用，下面是 unit()和 until_not()方法的说明。

[until\(method, message=''\)](#)

调用该方法提供的驱动程序作为一个参数，直到返回值不为 False。

[until_not\(method, message=''\)](#)

调用该方法提供的驱动程序作为一个参数，直到返回值为 False。

第八节 定位一组对象

webdriver 可以很方便的使用 `find_element` 方法来定位某个特定的对象，不过有时候我们却需要定位一组对象，WebElement 接口同样提供了定位一组元素的方法 `find_elements`。

定位一组对象一般用于以下场景：

- 批量操作对象，比如将页面上所有的 checkbox 都勾上
- 先获取一组对象，再在这组对象中过滤出需要具体定位的一些对象。比如定位出页面上所有的 checkbox，然后选择最后一个。

checkbox.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
<title>Checkbox</title>
<script type="text/javascript" async="" src="https://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<link href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet" />
<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>
</head>
<body>
<h3>checkbox</h3>
<div class="well">
<form class="form-horizontal">
<div class="control-group">
<label class="control-label" for="c1">checkbox1</label>
<div class="controls">
<input type="checkbox" id="c1" />
</div>
</div>
<div class="control-group">
<label class="control-label" for="c2">checkbox2</label>
<div class="controls">
<input type="checkbox" id="c2" />
</div>
</div>
<div class="control-group">
```

```

<label class="control-label" for="c3">checkbox3</label>
<div class="controls">
    <input type="checkbox" id="c3" />
</div>
</div>
</form>
</div>
</body>
</html>

```

将这段代码保存复制到记事本中，将保存成 `checkbox.html` 文件。（注意，这个页面需要和我们的自动化脚本放在同一个目录下，否则下面的脚本将指定 `checkbox.html` 的所在目录）

通过浏览器打开 `checkbox.html`，将看到以下页面：

checkbox

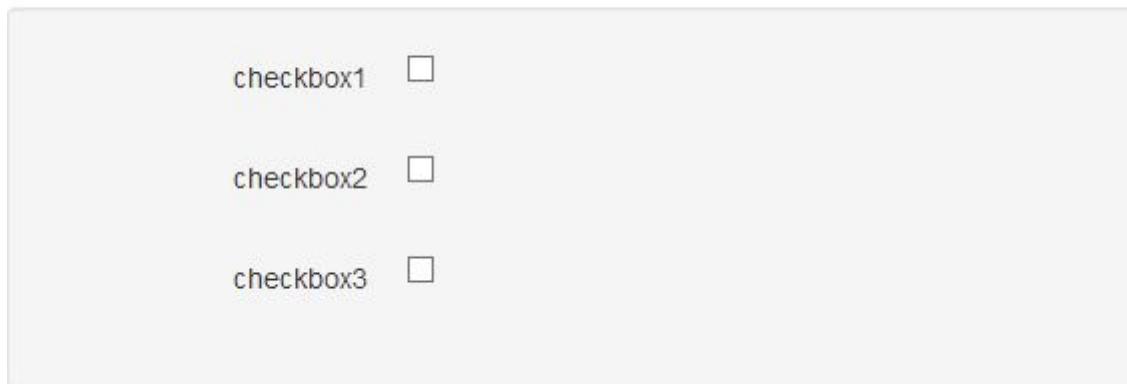


图 3.5

通过图 3.4 可以看到页面提供了三个复选框和两个单选按钮。下面通过脚本来单击勾选三个复选框。

```

# -*- coding: utf-8 -*-
from selenium import webdriver
import os

driver = webdriver.Firefox()

file_path = 'file:///+' + os.path.abspath('checkbox.html')

driver.get(file_path)

# 选择页面上所有的 tag name 为 input 的元素
inputs = driver.find_elements_by_tag_name('input')

```

```
#然后从中过滤出 type 为 checkbox 的元素，单击勾选
```

```
for input in inputs:
    if input.get_attribute('type') == 'checkbox':
        input.click()

driver.quit()
```

```
import os
```

```
os.path.abspath()
```

os 模块为 python 语言标准库中的 os 模块包含普遍的操作系统功能。主要用于操作本地目录文件。path.abspath()方法用于获取当前路径下的文件。另外脚本中还使用到 for 循环，对 inputs 获取的一组元素进行循环，在 python 语言中循环变量（input）可以不用事先声明直接使用。

```
find_elements_by_xx('xx')
```

find_elements 用于获取一组元素。

下面通过 css 方式来勾选一组元素，打印当所勾选元素的个数并对最后一个勾选的元素取消勾选。

```
#coding=utf-8

from selenium import webdriver

import os


driveriver = webdriver.Firefox()

file_path = 'file:///+' + os.path.abspath('checkbox.html')

driver.get(file_path)

# 选择所有的 type 为 checkbox 的元素并单击勾选

checkboxes = driver.find_elements_by_css_selector('input[type=checkbox]')

for checkbox in checkboxes:
    checkbox.click()
```

```
# 打印当前页面上 type 为 checkbox 的个数

print len(driver.find_elements_by_css_selector('input[type=checkbox]'))

# 把页面上最后1个 checkbox 的勾给去掉

driver.find_elements_by_css_selector('input[type=checkbox]').pop().click()

driver.quit()
```

len()

len 为 python 语言中的方法，用于返回一个对象的长度（或个数）。

pop()

pop 也为 python 语言中提供的方法，用于删除指定位置的元素，pop()为空默认选择最一个元素。

第九节 层级定位

在实际的项目测试中，经常会有这样的需求：页面上有很多个属性基本相同的元素，现在需要具体定位到其中的一个。由于属性基本相当，所以在定位的时候会有些麻烦，这时候就需要用到层级定位。先定位父元素，然后再通过父元素定位子孙元素。

level_locate.html

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
<title>Level Locate</title>
<script type="text/javascript" async=""
" src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
```

```
<link href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css"
      rel="stylesheet" />

</head>

<body>

    <h3>Level locate</h3>

    <div class="span3">

        <div class="well">

            <div class="dropdown">

                <a class="dropdown-toggle" data-toggle="dropdown" href="#">Link1</a>

                <ul class="dropdown-menu" role="menu" aria-labelledby="dLabel" id="dropdown1" >

                    <li><a tabindex="-1" href="#">Action</a></li>
                    <li><a tabindex="-1" href="#">Another action</a></li>
                    <li><a tabindex="-1" href="#">Something else here</a></li>
                    <li class="divider"></li>
                    <li><a tabindex="-1" href="#">Separated link</a></li>
                </ul>
            </div>
        </div>
    </div>

    <div class="span3">

        <div class="well">

            <div class="dropdown">

                <a class="dropdown-toggle" data-toggle="dropdown" href="#">Link2</a>

                <ul class="dropdown-menu" role="menu" aria-labelledby="dLabel" >

                    <li><a tabindex="-1" href="#">Action</a></li>
                    <li><a tabindex="-1" href="#">Another action</a></li>
                    <li><a tabindex="-1" href="#">Something else here</a></li>
                    <li class="divider"></li>
                    <li><a tabindex="-1" href="#">Separated link</a></li>
                </ul>
            </div>
        </div>
    </div>

```

```

        </div>

        </div>

        </div>

    </body>

<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>

</html>

```

将上面的代码保存为 level_locate.html，通过浏览器打开将看到以下页面。

Level locate



图 3.6

通过对上面代码的分析，发现两个下拉菜单中每个选项的 link text 都相同， href 也一样，所以在这里就需要使用层级定位了。

具体思路是：先点击显示出 1 个下拉菜单，然后再定位到该下拉菜单所在的 ul，再定位这个 ul 下的某个具体的 link 。在这里，我们定位第 1 个下拉菜单中的 Another action 这个选项。

```

#coding=utf-8

from selenium import webdriver

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.common.action_chains import ActionChains

import time

import os


driver = webdriver.Firefox()

file_path = 'file:/// + os.path.abspath('level_locate.html')

```

```

driver.get(file_path)

#点击 Link1 链接（弹出下拉列表）
driver.find_element_by_link_text('Link1').click()

#在父亲元件下找到 link 为 Action 的子元素
menu =
    driver.find_element_by_id('dropdown1').find_element_by_link_text('Another
action')

#鼠标移动到子元素上
ActionChains(driver).move_to_element(menu).perform()
time.sleep(5)
driver.quit()

driver.find_element_by_id('xx').find_element_by_link_text('xx').click()

```

这里用到了二次定位，通过对 Link1 的单击之后，出现下拉菜单，先定位到下拉菜单，再定位下拉菜单中的选项。当然，如果菜单选项需要单击，可通过二次定位后也直接跟 click() 操作。

ActionChains(driver)

driver: webdriver 实例执行用户操作。

ActionChains 用于生成用户的行为；所有的行为都存储在 actionchains 对象。通过 perform() 执行存储的行为。

move_to_element(menu)

move_to_element 方法模式鼠标移动到一个元素上，上面的例子中 menu 已经定义了他所指向的是哪一个元素。

perform()

执行所有 ActionChains 中存储的行为。

定位后的效果如下，鼠标点击 Link1 菜单，鼠标移动下 Another action 选项上，选项出现选中色。

Level locate

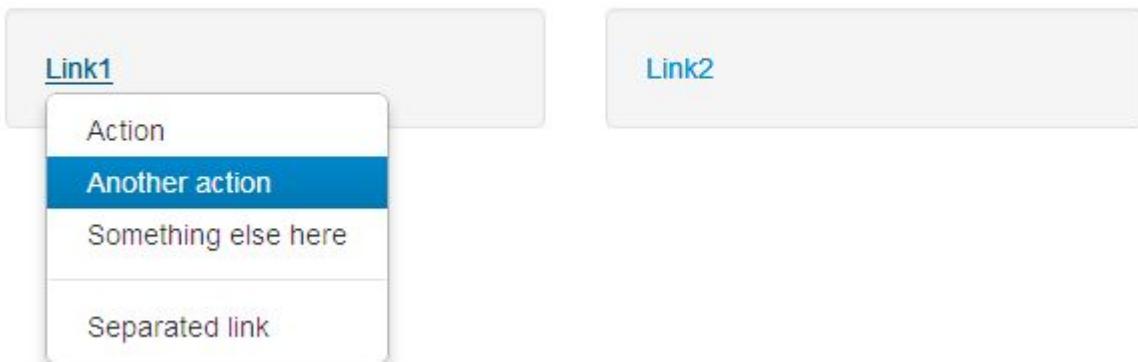


图 3.7

第十节 定位 frame 中的对象

在 web 应用中经常会出现 frame 嵌套的应用，假设页面上有 A、B 两个 frame，其中 B 在 A 内，那么定位 B 中的内容则需要先到 A，然后再到达 B。

`switch_to_frame` 方法可以把当前定位的主体切换到了 frame 里。怎么理解这句话呢？我们可以从 frame 的实质去理解。frame 中实际上是嵌入了另一个页面，而 webdriver 每次只能在一个页面识别，因此才需要用 `switch_to.frame` 方法去获取 frame 中嵌入的页面，对那个页面里的元素进行定位。

下面的代码中 `frame.html` 里有个 id 为 `f1` 的 frame，而 `f1` 中又嵌入了 id 为 `f2` 的 frame，该 frame 加载了百度的首页。

`frame.html`

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
    <title>frame</title>
<script type="text/javascript" async=""
"src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<link href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css"
rel="stylesheet" />
    <script type="text/javascript">$(document).ready(function() {
```

```

        });
    </script>
</head>
<body>
<div class="row-fluid">
    <div class="span10 well">
        <h3>frame</h3>
        <iframe id="f1" src="inner.html" width="800" height="600"></iframe>
    </div>
</div>
</body>
<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>
</html>

```

inner.html

```

<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>inner</title>
</head>
<body>
    <div class="row-fluid">
        <div class="span6 well">
            <h3>inner</h3>
            <iframe id="f2" src="http://www.baidu.com" width="700" height="400">
            </iframe>
        </div>
    </div>
</body>
</html>

```

frame.html 中嵌套 inner.html , 两个文件和我们的脚本文件放同一个目录下, 通过浏览器打开, 得到下列页面:





图 3.8

下面通过 `switch_to_frame` 方法来定位 frame 内的元素：

```
#coding=utf-8

from selenium import webdriver
import time
import os

driver = webdriver.Firefox()

file_path = 'file:/// + os.path.abspath('frame.html')

driver.get(file_path)

driver.implicitly_wait(30)

#先找到到 ifrome1 (id = f1)

driver.switch_to_frame("f1")

#再找到其下面的 ifrome2 (id =f2)
```

```

driver.switch_to_frame("f2")

#下面就可以正常操作元素了

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

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

time.sleep(3)

driver.quit()

```

switch_to_frame 的参数问题。官方说 name 是可以的，但是经过实验发现 id 也可以。所以只要 frame 中 id 和 name，那么处理起来是比较容易的。如果 frame 没有这两个属性的话，你可以直接手动添加。

第十一节 对话框处理

页面上弹出的对话框是自动化测试经常会遇到的一个问题；很多情况下对话框是一个 iframe，如上一节中介绍的例子，处理起来稍微有点麻烦；但现在很多前端框架的对话框是 div 形式的，这就让我们的处理变得十分简单。



图 3.9

图 3.9 为百度首页的登录对话框，下面通过脚本对百度进行登录操作：

```
#coding=utf-8
from selenium import webdriver

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

#点击登录链接
driver.find_element_by_name("tj_login").click()

#通过二次定位找到用户名输入框
div=driver.find_element_by_class_name("tang-content").find_element_by_name("userNamediv.send_keys("username")"

#输入登录密码
driver.find_element_by_name("password").send_keys("password")

#点击登录
driver.find_element_by_id("TANGRAM__PSP_10__submit").click()

driver.quit()
```

本例中并没有用到新方法，唯一的技巧是用到了二次定位，这个技巧在层级定位中已经有过使用。

```
driver.find_element_by_class_name("tang-content").find_element_by_name("userNamediv.send_keys("username")
```

第一次定位找到弹出的登录框，在登录框上再次进行定位找到了用户名输入框。

第十二节 浏览器多窗口处理

有时候我们在测试一个 web 应用时会出现多个浏览器窗口的情况，在 selenium1.0 中这个问题比较难处理。webdriver 提供了相关方法可以很轻松的在多个窗口之间切换并操作不同窗口上的元素。





图 3.10

要想在多个窗口之间切换，首先要获得每一个窗口的唯一标识符号（句柄）。通过获得的句柄来区别分不同的窗口，从而对不同窗口上的元素进行操作。

```
#coding=utf-8

from selenium import webdriver
import time

driver = webdriver.Firefox()

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

#获得当前窗口

nowhandle=driver.current_window_handle

#打开注册新窗口

driver.find_element_by_name("tj_reg").click()

#获得所有窗口
```

```

allhandles=driver.window_handles

#循环判断窗口是否为当前窗口

for handle in allhandles:

    if handle != nowhandle:

        driver.switch_to_window(handle)

        print 'now register window!'

    #切换到邮箱注册标签

        driver.find_element_by_id("mailRegTab").click()

        time.sleep(5)

    driver.close()

#回到原先的窗口

driver.switch_to_window(nowhandle)

driver.find_element_by_id("kw").send_keys(u"注册成功!")

time.sleep(3)

driver.quit()

```

处理过程：

这个处理过程相比我们前面的元素操作来说稍微复杂一些，执行过程为：首选通过 nowhandle 获得当前窗口（百度首页）的句柄；然后，打开注册窗口（注册页）；通过 allhandles 获得所有窗口的句柄；对所有句柄进行循环遍历；判断窗口是否为 nowhandle（百度首页），如果不是则获得当前窗口（注册页）的句柄；然后，对注册页上的元素进行操作。最后，回返到首页。

为了使执行过程更多更容易理解，在切换到注册页时，打印了'now register window!'一条信息；切换回百度首页时，我们在输入框输入了“注册成功！”。注意，我们在切换到注册页时，只是切换了一下邮箱注册标签，如果要直执行注册过程还需要添加更多的操作步骤。

在本例中所有用到的新方法：

[current_window_handle](#)

获得当前窗口句柄

window_handles

返回的所有窗口的句柄到当前会话

switch_to_window()

用于处理多窗口操作的方法，与我们前面学过的 switch_to_frame() 是类似， switch_to_window()用于处理多窗口之前切换， switch_to_frame() 用于处理多框架的切换。

close()

如果你足够细心会发现我们在关闭“注册页”时用的是 close()方法，而非 quit(); close()用于关闭当前窗口， quit()用于退出驱动程序并关闭所有相关窗口。

第十二节 alert/confirm/prompt 处理



webdriver 中处理 JavaScript 所生成的 alert、confirm 以及 prompt 是很简单的。具体思路是使用 switch_to.alert()方法定位到 alert/confirm/prompt。然后使用 text/accept/dismiss/send_keys 按需进行操作。

- text 返回 alert/confirm/prompt 中的文字信息。
- accept 点击确认按钮。
- dismiss 点击取消按钮，如果有的话。
- send_keys 输入值，这个 alert\confirm 没有对话框就不能用了，不然会报错。

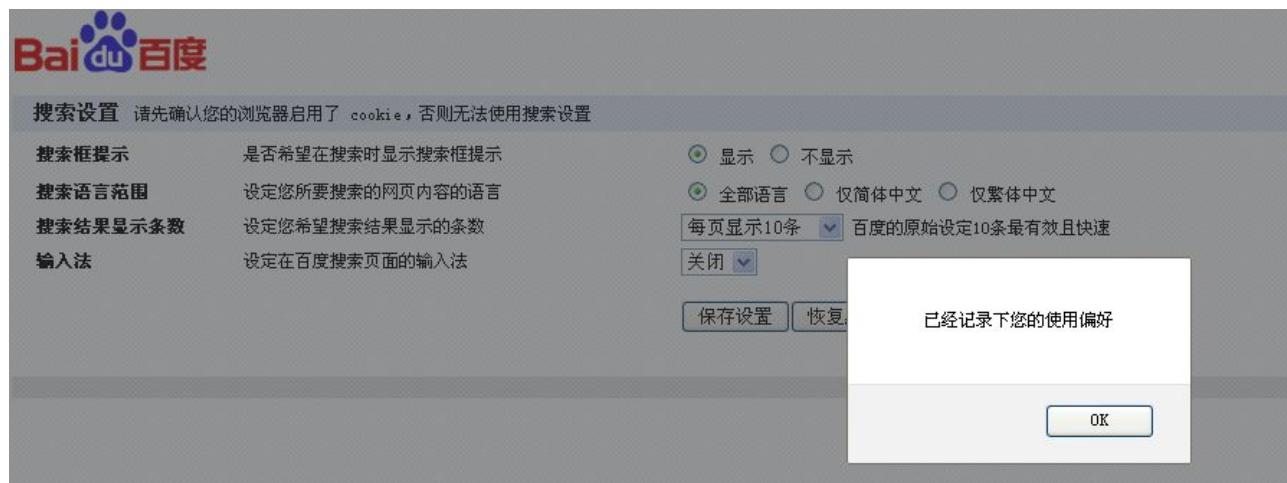


图 3.11

图 3.11 所给出的是百度设置页面，在设置完成后点击“保存设置”所弹的提示框。下面通过脚本来处理这个弹窗。

```
#coding=utf-8

from selenium import webdriver
import time

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

#点击打开搜索设置
driver.find_element_by_name("tj_setting").click()

driver.find_element_by_id("SL_1").click()

#点击保存设置
driver.find_element_by_xpath("//div[@id='gxszButton']/input").click()

#获取网页上的警告信息
alert=driver.switch_to_alert()

#接收警告信息
alert.accept()

dirver.quit()
```

switch_to_alert()

用于获取网页上的警告信息。我们可以对警告信息做以下操作：

```
#接受警告信息
alert = driver.switch_to_alert()
alert.accept()

#得到文本信息并打印
alert = driver.switch_to_alert()
print alert.text()
```

```
#取消对话框（如果有的话）

alert = driver.switch_to_alert()

alert.dismiss()

#输入值（如果有的话）

alert = driver.switch_to_alert()

alert.send_keys("xxx")
```

第十三节 下拉框处理



下拉框也是 web 页面上非常常见的功能，webdriver 对于一般的下拉框处理起来也相当简单，要想定位下拉框中的内容，首先需要定位到下拉框；这样的二次定位，我们在前面的例子中已经有过使用，下面通过一个具体的例子来说明具体定位方法。

drop_down.html

```
<html>
  <body>
    <select      id="ShippingMethod"      onchange="updateShipping(options[selectedIndex]);"
      name="ShippingMethod">

      <option value="12.51">UPS Next Day Air ==> $12.51</option>
      <option value="11.61">UPS Next Day Air Saver ==> $11.61</option>
      <option value="10.69">UPS 3 Day Select ==> $10.69</option>
      <option value="9.03">UPS 2nd Day Air ==> $9.03</option>
      <option value="8.34">UPS Ground ==> $8.34</option>
      <option value="9.25">USPS Priority Mail Insured ==> $9.25</option>
      <option value="7.45">USPS Priority Mail ==> $7.45</option>
      <option value="3.20" selected="">USPS First Class ==> $3.20</option>

    </select>
  </body>
</html>
```

保存并通过浏览器打开，如下：



图3.12

图 3.12 是最常现在我们来通过脚本选择下拉列表里的\$10.69

```
#-*-coding=utf-8
from selenium import webdriver
import os,time

driver=webdriver.Firefox()
file_path = 'file:/// + os.path.abspath('drop_down.html')
driver.get(file_path)
time.sleep(2)

#先定位到下拉框
m=driver.find_element_by_id("ShippingMethod")

#再点击下拉框下的选项
m.find_element_by_xpath("//option[@value='10.69']").click()
time.sleep(3)

driver.quit()
```

需要说明的是在实际的 web 测试时，会发现各种类型的下拉框，并非我们上面所介绍的传统的下拉框。如图 3.x，对这种类型的下拉框一般的处理是两次点击，第一点击弹出下拉框，第二次点击操作元素。当然，也有些下拉框是鼠标移上去直接弹出的，那么我们可以使用 move_to_element() 进行操作。



图 3.13

第十四节 分页处理

对于 web 页面上的分页功能，我们一般做以下操作：

- 获取总页数
- 翻页操作（上一页，下一页）

对于有些分页功能提供上一页，下一页按钮，以及可以输入具体页面数跳转功能不在本例的讨论范围。

```
....  
  
<select id="pageElm_a74e_ce2c" class="yem" action="page" data-page="5">  
  
    <option value="1">1/5</option>  
  
    <option value="2">2/5</option>  
  
    <option value="3">3/5</option>  
  
    <option value="4">4/5</option>  
  
    <option value="5">5/5</option>  
  
</select>  
  
....
```

上面代码为分页功能的代码片断，显示效果如下：



图 3.13

```
#coding=utf-8
```

```
from selenium import webdriver
from time import sleep

driver = webdriver.Firefox()

driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F%2Fvod.kuaibo.com%2F%3Fly%3Ddefault")

#登录系统

driver.find_element_by_id("user_name").clear()
driver.find_element_by_id("user_name").send_keys("username")
driver.find_element_by_id("user_pwd").clear()
driver.find_element_by_id("user_pwd").send_keys("password")
driver.find_element_by_id("dl_an_submit").click()
sleep(2)

#获取所有分页的数量，并打印

total_pages=len(driver.find_element_by_tag_name("select").find_elements_by_tag_name("option"))

print "total page is %s" %(total_pages)
sleep(3)

#再次获取所分页，并执行循环翻页操作

pages=driver.find_element_by_tag_name("select").find_elements_by_tag_name("option")

for page in pages:
    page.click()
    sleep(2)

sleep(3)
```

```
driver.quit()
```

`len()`方法在定位一组对象有时已经用过，用于获取对象的个数。

这里同样用到了二次定位，只是第二次定位用的是 `find_elements` 方法，获取的是一组元素。通过上面的脚本可以看到，我们第一次获取到一组元素后，打印了所有分页的个数。第二次获取所有分页后，通过 `for` 循环来翻阅每一页，每翻一页休眠 2 秒，当然，我们也可以在翻页后对列表的文件做更多操作。

第十五节 上传文件

文件上传操作也比较常见功能之一，上传功能操作 `webdriver` 并没有提供对应的方法，关键上传文件的思路。

上传过程一般要打开一个系统的 `window` 窗口，从窗口选择本地文件添加。所以，一般会卡在如何操作本地 `window` 窗口。其实，上传本地文件没我们想的那么复杂；只要定位上传按钮，通过 `send_keys` 添加本地文件路径就可以了。绝对路径和相对路径都可以，关键是上传的文件存在。下面通过例子演示操作过程。

`upload_file.html`

```
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
<title>upload_file</title>
<script type="text/javascript" async="" src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<link href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet" />
<script type="text/javascript">
</script>
</head>
<body>
    <div class="row-fluid">
        <div class="span6 well">
            <h3>upload_file</h3>
            <input type="file" name="file" />
        </div>
    </div>
</body>
```

```
<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>
</html>
```

通过浏览器打开，得到下列页面：

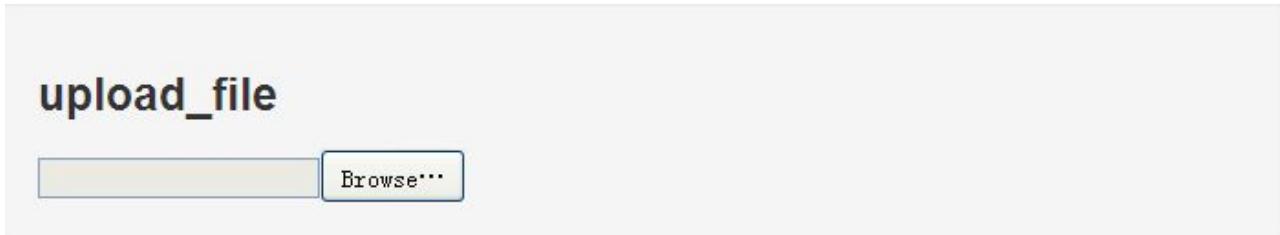


图 3.14

操作上传脚本：

```
#coding=utf-8

from selenium import webdriver
import os,time

driver = webdriver.Firefox()

#打开上传文件页面
file_path = 'file:///+' + os.path.abspath('upload_file.html')
driver.get(file_path)

#定位上传按钮，添加本地文件
driver.find_element_by_name("file").send_keys('D:\\selenium_use_case\\upload_file.txt')
time.sleep(2)

driver.quit()
```

从上面例子可以看到，`send_keys()`方法除可以输入内容外，也可以跟一个本地的文件路径。从而达到上传文件的目的。

文件上传成功的效果如图3.15：

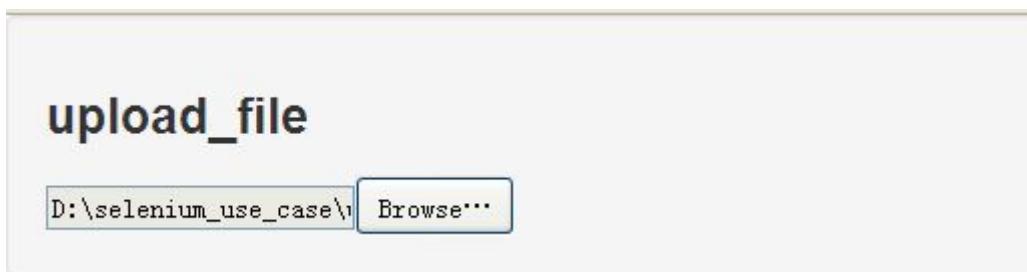


图 3.15

第十六节 下载文件

`webdriver` 允许我们设置默认的文件下载路径。也就是说文件会自动下载并且存在设置的那个目录中。

要想下载文件，首选要先确定你所要下载的文件的类型。要识别自动文件的下载类型可以使用 `curl`，如图3.16：

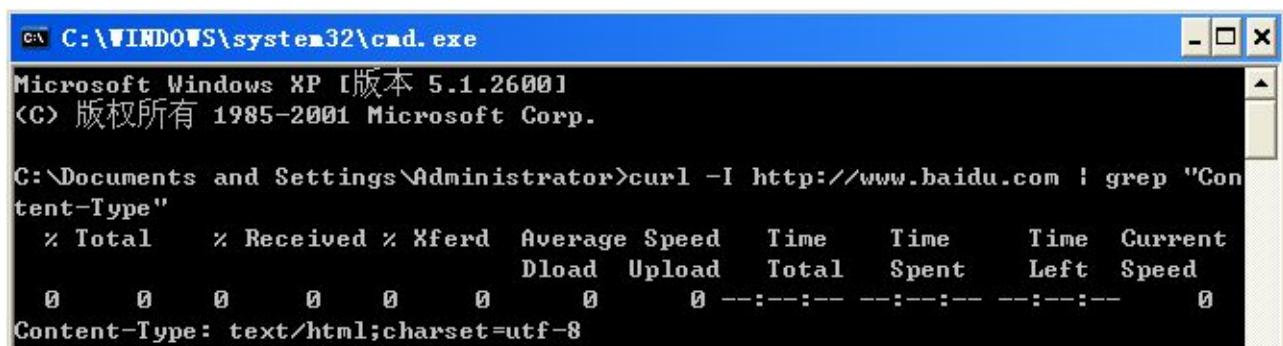


图 3.16

`curl` 是利用 URL 语法在命令行方式下工作的开源文件传输工具。

`Content-Type`，内容类型，一般是指网页中存在的 `Content-Type`，用于定义网络文件的类型和网页的编码，决定浏览器将以什么形式、什么编码读取这个文件。

另一种方法是使用 `requests` 模块来查找内容类型。`Requests` 是一个 Python 的 HTTP 客户端库，默认下载的 `python` 环境包不包含这个类库，需要另外安装。使用方法如下：

```
import requests

print requests.head('http://www.python.org').headers['content-type']
```

一旦确定了内容的类型，就可以用它来设置 Firefox 的默认配置文件，具体实例如下：

```
#coding=utf-8

import os

from selenium import webdriver


fp = webdriver.FirefoxProfile()

fp.set_preference("browser.download.folderList",2)

fp.set_preference("browser.download.manager.showWhenStarting",False)

fp.set_preference("browser.download.dir", os.getcwd())

fp.set_preference("browser.helperApps.neverAsk.saveToDisk",

"application/octet-stream")



browser = webdriver.Firefox(firefox_profile=fp)

browser.get("http://pypi.python.org/pypi/selenium")

browser.find_element_by_partial_link_text("selenium-2").click()
```

browser.download.dir 用于指定你所下载文件的目录。

os.getcwd() 该函数不需要传递参数，用于返回当前的目录。

application/octet-stream 为内容的类型。

第十七节 调用 JavaScript

当 `webdriver` 遇到没法完成的操作时，笔者可以考虑借用 `JavaScript` 来完成，比下下面的例子，通过 `JavaScript` 来隐藏页面上的元素。除了完成 `webdriver` 无法完成的操作，如果你熟悉 `JavaScript` 的话，那么使用 `webdriver` 执行 `JavaScript` 是一件非常高效的事情。

`webdriver` 提供了 `execute_script()` 接口用来调用 `js` 代码。

`js.html`

<http://fnng.cnblogs.com>

```

<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <title>js</title>
        <script type="text/javascript" async="" src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
        <link href="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/css/bootstrap-combined.min.css" rel="stylesheet" />
        <script type="text/javascript">
            $(document).ready(function(){
                $('#tooltip').tooltip({ "placement": "right" });
            });
        </script>
    </head>

    <body>
        <h3>js</h3>
        <div class="row-fluid">
            <div class="span6 well">
                <a id="tooltip" href="#" data-toggle="tooltip" title=" selenium.webdriver(python)">hover to see tooltip</a>
                <a class="btn">Button</a>
            </div>
        </div>
    </body>
<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.2/js/bootstrap.min.js"></script>
</html>

```

保存 **js.html** 文件并通过浏览器打开，效果如图3.17：



图3.17

执行 **js** 一般有两种场景：

- 一种是在页面上直接执行 **JS**
- 另一种是在某个已经定位的元素上执行 **JS**

```
#coding=utf-8

from selenium import webdriver
import time,os

driver = webdriver.Firefox()

file_path = 'file:///+' + os.path.abspath('js.html')
driver.get(file_path)
```

#####通过 JS 隐藏选中的元素#####第一种方法：

#隐藏文字信息

```
driver.execute_script('$( "#tooltip" ).fadeOut();')
time.sleep(5)
```

#隐藏按钮：

```
button = driver.find_element_by_class_name('btn')

driver.execute_script('$( arguments[0] ).fadeOut()',button)
time.sleep(5)

driver.quit()
```

execute_script(script, *args)

在当前窗口/框架 同步执行 JavaScript

script: JavaScript 的执行。

***args:** 适用任何 JavaScript 脚本。

关于 JavaScript 代码的解析不在本书的范围之内,请读者通过其它资料学习理解 JavaScript 的使用。

隐藏之后的效果如图3.18

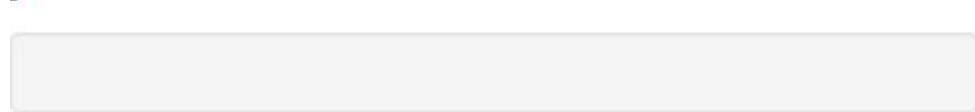


图 3.18

第十八节、控制浏览器滚动条

有时候 web 页面上的元素并非直接可见的，就算把浏览器最大化，我们依然需要拖动滚动条才能看到想要操作的元素，这个时候就要控制页面滚动条的拖动，但滚动条并非页面上的元素，可以借助 JavaScript 来完成操作。

一般用到操作滚动条的会两个场景：

- 注册时的法律条文的阅读，判断用户是否阅读完成的标准是：滚动条是否拉到最下方。
- 要操作的页面元素不在视觉范围，无法进行操作，需要拖动滚动条

用于标识滚动条位置的代码

```
<body    onload= "document.body.scrollTop=0 ">
<body    onload= "document.body.scrollTop=100000 ">
```

如果滚动条在最上方的话，scrollTop=0，那么要想使用滚动条在最下方，可以 scrollTop=100000，这样就可以使滚动条在最下方。图 3.19

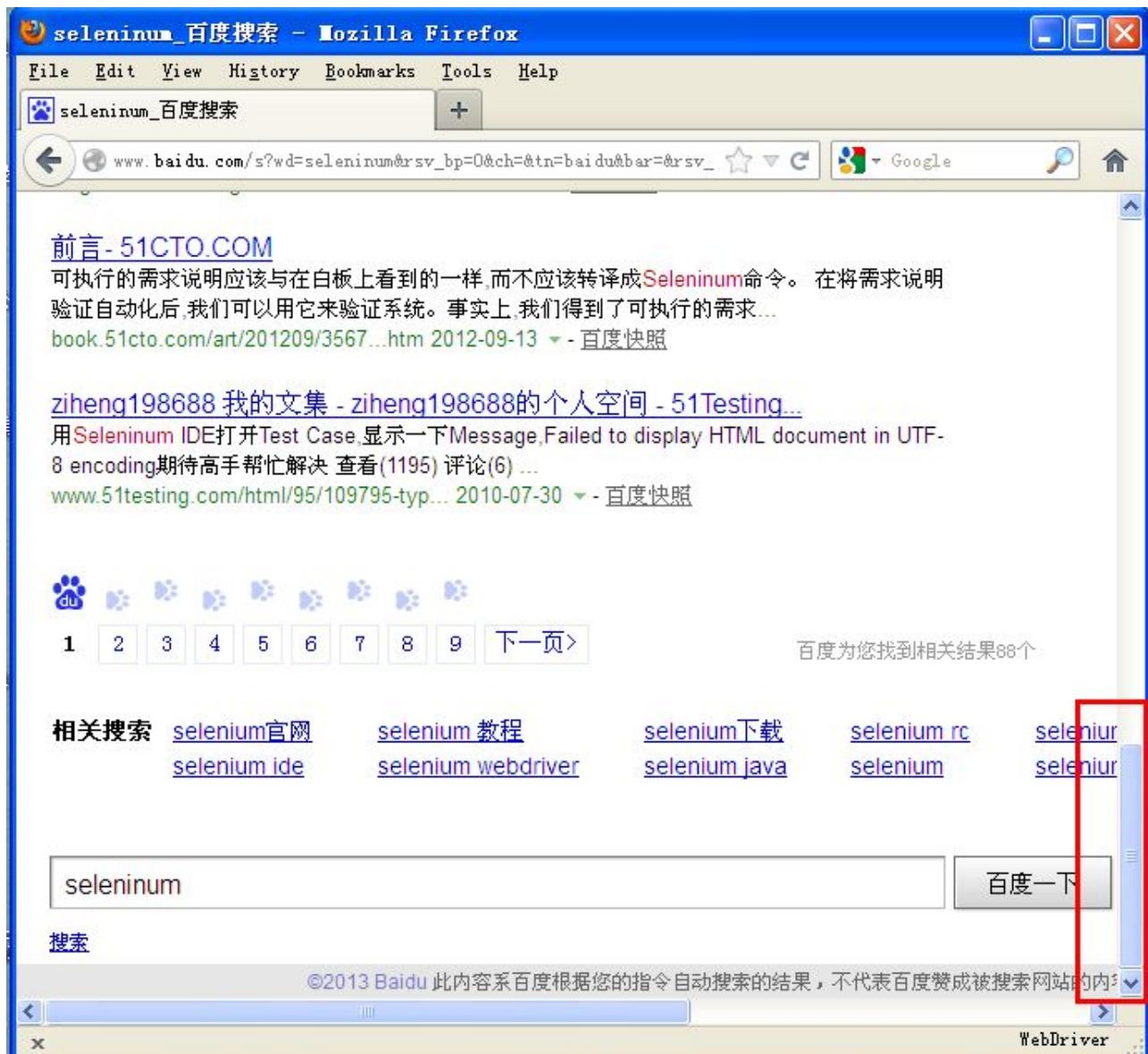


图 3.19

图 3.19 百度搜索结果页且滚动条在页面底，下面通过脚本实现：

```
#coding=utf-8

from selenium import webdriver

import time

#访问百度

driver=webdriver.Firefox()

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

#搜索

```
driver.find_element_by_id("kw").send_keys("selenium")
driver.find_element_by_id("su").click()
time.sleep(3)
```

#将页面滚动条拖到底部

```
js="var q=document.documentElement.scrollTop=10000"
driver.execute_script(js)
time.sleep(3)
```

#将滚动条移动到页面的顶部

```
js_="var q=document.documentElement.scrollTop=0"
driver.execute_script(js_)
time.sleep(3)
```

driver.quit()

第十九节 cookie 处理

有时候我们需要验证浏览器中是否存在某个 **cookie**, 因为基于真实的 **cookie** 的测试是无法通过白盒和集成测试完成的。**webdriver** 可以读取、添加和删除 **cookie** 信息。

webdriver 操作 **cookie** 的方法有:

- `get_cookies()` 获得所有 **cookie** 信息
- `get_cookie(name)` 返回特定 name 有 **cookie** 信息
- `add_cookie(cookie_dict)` 添加 **cookie**, 必须有 name 和 value 值
- `delete_cookie(name)` 删除特定(部分)的 **cookie** 信息

- delete_all_cookies() 删除所有 cookie 信息

通过 webdriver 操作 cookie 是一件非常有意思的事儿，有时候我们需要了解浏览器中是否存在了某个 cookie 信息，webdriver 可以帮助我们读取、添加，删除 cookie 信息。

3.19.1 打印 cookie 信息

```
#coding=utf-8

from selenium import webdriver
import time

driver = webdriver.Chrome()
driver.get("http://www.youdao.com")

# 获得 cookie 信息
cookie= driver.get_cookies()

#将获得 cookie 的信息打印
print cookie

driver.quit()
```

运行打印信息：

```
[{u'domain': u'.youdao.com', u'secure': False, u'value': u'aGFzbG9nZ2VkPXydwU=', u'expiry': 1408430390.991375, u'path': u'/', u'name': u'_PREF_ANONYUSER_MYTH'}, {u'domain': u'.youdao.com', u'secure': False, u'value': u'1777851312@218.17.158.115', u'expiry': 2322974390.991376, u'path': u'/', u'name': u'OUTFOX_SEARCH_USER_ID'}, {u'path': u'/', u'domain': u'www.youdao.com', u'name': u'JSESSIONID', u'value': u'abcUX9zdw0minadIhtvcu', u'secure': False}]
```

3.19.2、对 cookie 操作

上面的方式打印了所有 cookie 信息，太多太乱，我们只想有真对性的打印自己想要的信息，看下面的例子：

```
#coding=utf-8

from selenium import webdriver
import time

driver = webdriver.Firefox()
driver.get("http://www.youdao.com")

#向 cookie 的 name 和 value 添加会话信息。
driver.add_cookie({'name':'key-aaaaaaaa', 'value':'value-bbbb'})

#遍历 cookies 中的 name 和 value 信息打印，当然还有上面添加的信息
for cookie in driver.get_cookies():
    print "%s -> %s" % (cookie['name'], cookie['value'])

##### 下面可以通过两种方式删除 cookie #####
# 删除一个特定的 cookie
driver.delete_cookie("CookieName")

# 删除所有 cookie
driver.delete_all_cookies()

time.sleep(2)

driver.close()
```

运行打印信息：

```
YOUTAO_MOBILE_ACCESS_TYPE -> 1
_PREF_ANONYUSER__MYTH -> aGFzbG9nZ2VkPXRydWU=
OUTFOX_SEARCH_USER_ID -> -1046383847@218.17.158.115
JSESSIONID -> abc7qSE_SBGsVgnVLBvcu
key-aaaaaaaa -> value-bbbb # 这一条是我们自己添加的
```

第二十节 获取对象的属性

获取测试对象的属性能够帮我们更好的进行对象的定位。比如页面上有很多标签为 input 元素，而我们需要定位其中 1 个有具有 data-node 属性不一样的元素。由于 webdriver 是不支持直接使用 data-node 来定位对象的，所以我们只能先把所有标签为 input 都找到，然后遍历这些 input，获取想要的元素。

例如，有下面一组元素：

```
<input type="checkbox" data-node="594434499" data-convert="1" data-type="file">
<input type="checkbox" data-node="594434498" data-convert="1" data-type="file">
<input type="checkbox" data-node="594434493" data-convert="1" data-type="file">
<input type="checkbox" data-node="594434497" data-convert="1" data-type="file">
```

通过 find_elements 获得一组元素，通过循环遍历打到想要的元素：

```
# 选择页面上所有的 tag name 为 input 的元素
inputs = driver.find_elements_by_tag_name('input')

#然后循环遍历出 data-node 为594434493的元素，单击勾选

for input in inputs:
    if input.get_attribute('data-node') == '594434493':
        input.click()
....
```

如果读者细心会发现，我们在前面定位一组对象时已经用到了这个方法，当时判断是具有一组相同属性的元素，对其进行操作。这里判断是属性值不同的元素对其进行操作。灵活的运用这个技巧，才会让我们面对各种对象和需求时变得游刃有余。

第二十一节 验证码问题

对于 web 应用来说，大部分的系统在用户登录时都要求用户输入验证码，验证码的类型的很多，有字母数字的，有汉字的，甚至还要用户输入一条算术题的答案的，对于系统来说使用验证码可以有效防止采用机器猜测方法对口令的刺探，在一定程度上增加了安全性。但对于测试人员来说，不管是进行性能测试还是自动化测试都是一个棘手的问题。



图 3.20

下面笔者根据自己的经验来谈一下处理验证码的几种方法。

去掉验证码

这是最简单的方法，对于开发人员来说，只是把验证码的相关代码注释掉即可，如果是在测试环境，这样做可省去了测试人员不少麻烦，如果自动化脚本是要在正式环境跑，这样就给系统带来了一定的风险。

设置万能码

去掉验证码的主要是安全问题，为了应对在线系统的安全性威胁，可以在修改程序时不取消验证码，而是程序中留一个“后门”---设置一个“万能验证码”，只要用户输入这个“万能验证码”，程序就认为验证通过，否则按照原先的验证方式进行验证。

验证码识别技术

例如可以通过 Python-tesseract 来识别图片验证码，Python-tesseract 是光学字符识别 Tesseract OCR 引

擎的 Python 封装类。能够读取任何常规的图片文件(JPG, GIF ,PNG ,TIFF 等)。不过，目前市面上的验证码形式繁多，目前任何一种验证码识别技术，识别率都不是 100% 。

记录 cookie

通过向浏览器中添加 cookie 可以绕过登录的验证码，这是比较有意思的一种解决方案。我们可以在用户登录之前，通过 add_cookie() 方法将用户名密码写入浏览器 cookie ，再次访问系统登录链接将自动登录。例如下面的方式：

```
....  
#访问 xxxx 网站  
  
driver.get("http://www.xxxx.cn/")  
  
#将用户名密码写入浏览器 cookie  
  
driver.add_cookie({'name':'Login_UserName', 'value':'username'})  
  
driver.add_cookie({'name':'Login_Passwd', 'value':'password'})  
  
#再次访问 xxxx 网站，将会自动登录  
  
driver.get("http://www.xxxx.cn/")  
  
time.sleep(3)  
  
....  
  
driver.quit()
```

使用 cookie 进行登录最大的难点是如何获得用户名密码的 name ，如果找不到不到 name 的名字，就没办法向 value 中输用户名、密码信息。

笔者的建议是可以通过 get_cookies() 方法来获取登录的所有的 cookie 信息，从而进行找到用户名、密码的 name 对象的名字；当然，最简单的方法还是询问前端开发人员。

第二十二节 wedriver 原理

webdriver 原理：

1. WebDriver 启动目标浏览器，并绑定到指定端口。该启动的浏览器实例，做为 web driver 的 remote server。

2. Client 端通过 CommandExecutor 发送 HTTPRequest 给 remote server 的侦听端口（通信协议： the webriver wire protocol）

3. Remote server 需要依赖原生的浏览器组件（如： IEDriverServer.exe、 chromedriver.exe），来转化转化浏览器的 native 调用。

总结：

通过本章的学习，我们比较全面的掌握了如何使用 webdriver 所提供的方法对页面上各种元素进行操作。不过在实际的自动化测试过程中，读者会遇到各种各样的问题，笔者建议读者从以下几个方面进行提高：

- 1、熟练掌握 xpath\CSS 定位的使用，这样在遇到各种难以定位的属性时才不会变得束手无策。
- 2、准备一份 python 版本的 webdriver API ，遇到不理解地方，及时查到 API 的使用
- 3、学习掌握 JavaScript 语言，掌握 JavaScript 好处前面已经有过阐述，可以让我们的自动化测试工作更加游刃有余。
- 4、自动化测试归根结底是与前端打交道，多多熟悉前端技术，如 http 请求， HTML 语言， cookie /session 机制等。

第四章 自动化测试模型

一个自动化测试框架就是一个集成体系，在这一体系中包含测试功能的函数库、测试数据源、测试对象识别标准，以及种可重用的模块。自动化测试框架在发展的过程中经历了几个阶段，模块驱动测试、数据驱动测试、对象驱动测试。本章就带领读者了解这几种测试模型。

第一节、自动化测试模型介绍

自动化测试模型是自动化测试架构的基础，自动化测试的发展也经历的不同的阶段，不断有新的模型（概念）被提出，了解和使用这些自动化模型将帮助我们构建一个灵活可维护性的自动化架构。

4.1.1 线性测试

通过录制或编写脚本，一个脚本完成一个场景（一组完整功能操作），通过对脚本的回放来进行自动化测试。这是早期进行自动化测试的一种形式；我们在上一章中练习使用 webdriver API 所编写的脚本也是这种形式。

测试脚本一

```
from selenium import webdriver
import time

driver = webdriver.Firefox()
driver.get("http://www.aaa.com")

driver.find_element_by_id("tbUserName").send_keys("username")
driver.find_element_by_id("tbPassword").send_keys("123456")
driver.find_element_by_id("btnLogin").click()

#执行具体用例操作
.....
driver.quit()
```

测试脚本二

```
from selenium import webdriver
import time
```

```

driver = webdriver.Firefox()

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

driver.find_element_by_id("tbUserName").send_keys("username")
driver.find_element_by_id("tbPassword").send_keys("123456")
driver.find_element_by_id("btnLogin").click()

#执行具体用例操作
.....
driver.quit()

```

通过上面的两个脚本，我们发现它优势就是每一个脚本都是独立的，任何一个脚本文件拿出来就能单独运行；当然，缺点也很明显，用例的开发与维护成本很高：

一个用例对应一个脚本，假如登陆发生变化，用户名的属性发生改变，不得不需要对每一个脚本进行修改，测试用例形成一种规模，我们可能将大量的工作用于脚本的维护，从而失去自动化的意义。

这种模式下数据和脚本是混在一起的，如果数据发生变也需要对脚本进行修改。这种模式下脚本的没有可重复使用的概念。

4.1.2 模块化与类库

我们会清晰的发现在上面的脚本中，其实有不少内容是重复的；于是我们就考虑能不能把重复的部分写成一个公共的模块，需要的时候进行调用，这样就大大提高了我们编写脚本的效率。

login.py

```

#登录模块

def login():

    driver.find_element_by_id("tbUserName").send_keys("username")

    driver.find_element_by_id("tbPassword").send_keys("456123")

    driver.find_element_by_id("btnLogin").click()

```

quit.py

```

#退出模块

def quit_():

    .....

```

测试用例：

```
#coding=utf-8

from selenium import webdriver

import login, quit_ #调用登录、退出模块

driver = webdriver.Firefox()

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

#调用登录模块

login.login()

#其它个性化操作

.....
#调用退出模块

quit_.quit()
```

注意，上面用例非完整代码。

通过阅读上面的代码发现，我们可以把脚本中相同的部分代码独立出来，形成模块或库；这样做有两方面的优点：

一方面提高了开发效率，不用重复的编写相同的脚本；假如，我已经写好一个登录模块，我后续需要做的就是在需要的地方调用，不同重复造轮子。

另一方面方便了代码的维护，假如登录模块发生了变化，我只用修改 login.py 文件中登录模块的代码即可，那么所有调用登录模块的脚本不用做任何修改。

4.1.3 数据驱动

数据驱动应该是自动化的一个进步；从它的本意来讲，数据的改变（更新）驱动自动化的执行，从而引起测试结果的改变。这显然是一個非常高级的概念和想法。其实，我们可直白的理解成参数化，输入数据的不同从而引起输出结果的变化。

```
#coding=utf-8

from selenium import webdriver

import time

values=['selenium','webdriver',u'虫师']

# 执行循环
```

```
for serch in values:  
  
    driver = webdriver.Firefox()  
  
    driver.get("http://www.xxxx.com")  
  
    driver.find_element_by_id("kw").send_keys(serch)  
  
    time.sleep(3)  
  
    ....
```

不管我们读取的是数组，还是字典、函数，又或者是 csv、txt 文件。我们实现了数据与脚本的分离，换句话说，我们实现了参数化。我们传一千条数据，通过脚本的执行，可以返回一千条结果出来。

同样的脚本执行不同的数据从而得到了不同的结果，是不是增强的脚本的复用性呢！？

其实，模块化与参数化这对开发来说是完全没有什么技术含量的；对于当初 QTP 自动化工具来说地确是一个卖点，因为它面对的大多是不懂开发的测试，当然，随着时代的发展，懂开发的测试人员越来越多。

4.1.4 关键字驱动

理解了数据驱动，无非是把“数据”换成“关键字”，通过关键字的改变引起测试结果的改变。

关键字驱动用编程方式就不太容易表现了。QTP、robot framework 等都是以关键字驱动为主的自动化工具，因为这类工具主打的易用性，“填表格”式的关键字驱动帮我们封装了很多底层的东西，我们只要考虑三个问题就可以了：我要做什么？ 对谁做？ 怎么做？。

我们可以把 selenium IDE 看做是一种关键字驱动的自动化工具。

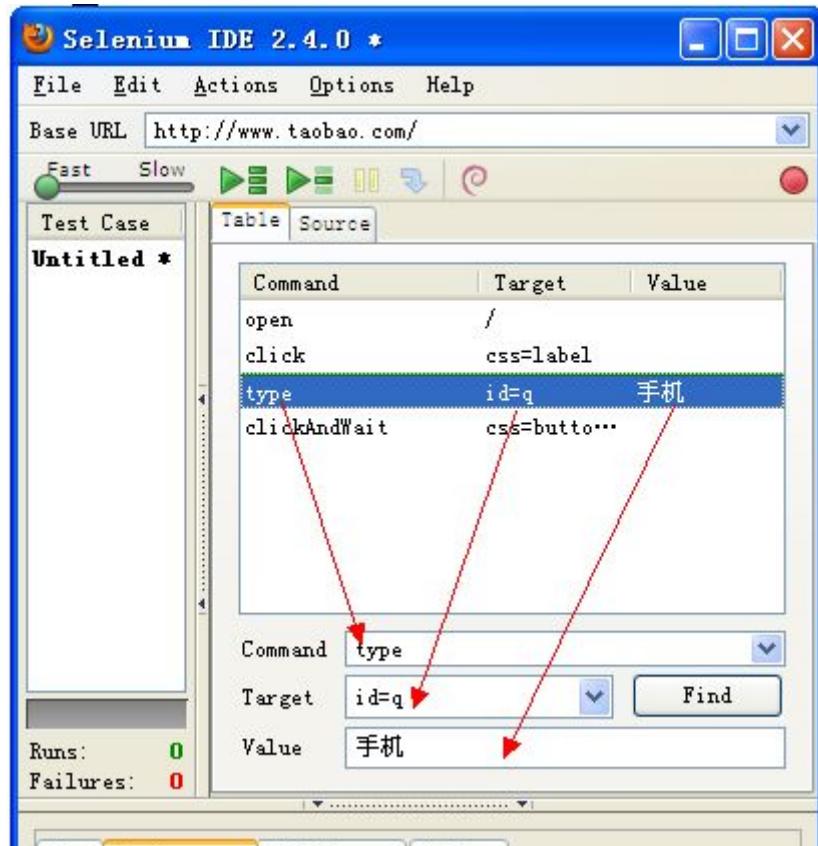


图4.1

自动化脚本转化成表格是这样的：

http://www.taobao.com/		
Command	Target	Value
open	/	
type	Id=q	手机
clickAndWait	Css=button.tsearch-submit	
verifyTextPresent	手机	

图4.2

Selenium IDE 脚本分：命令（command）、对象（target）、值（value）

通过这样的格式去描述不同的对象，从而引起最终结果的改变。也就是说一切以对象为出发点。当然，这样的脚本，显然对于不懂代码的同学非常直观！我要做什么（命令）？对谁做（对象）？怎么做（值）？

更高级的关键字驱动，可以自己定义 keyword 然后“注册”到框架；从而实现更强大的功能和扩展性。

小结：

这里简单介绍了自动化测试的几种不同的模型，虽然简单阐述了他们的优缺点，但他们并非后者淘汰前者的关系，在实施自动化更多的是以需求为出发点，混合的来使用以上模型去解决问题；使我们的脚本更易于开发与维护。

第二节、登录模块化

通过上一节对测试模型的学习可以发现，在我们的目前的脚本中有很多代码是可以模块化的，比如登录模块。我们的每一个用例的执行都需要登录脚本，那可我们是否可以将登录脚本独立到单独的文件调用。

下面以快播私有云的登录退出测试用例为例：

webcloud.py

```
#coding=utf-8

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time

class Login(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://passport.kuaibo.com"
        selfverificationErrors = []
        self.accept_next_alert = True

    #私有云登录用例
    def test_login(self):
```

```

driver = self.driver
driver.get(self.base_url
"/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")

driver.maximize_window()
#登陆
driver.find_element_by_id("user_name").clear()
driver.find_element_by_id("user_name").send_keys("username")
driver.find_element_by_id("user_pwd").clear()
driver.find_element_by_id("user_pwd").send_keys("123456")
driver.find_element_by_id("dl_an_submit").click()
time.sleep(3)

#新功能引导
driver.find_element_by_class_name("guide-ok-btn").click()
time.sleep(3)

#退出
driver.find_element_by_class_name("Usertool").click()
time.sleep(2)
driver.find_element_by_link_text("退出").click()
time.sleep(2)

def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
    unittest.main()

```

注明：本用例暂不关注 unittest 内容，第六章会详细的讲解 unittest 单元测试框架。

从业务流程及用例分析，每一个自动化测试用例的执行过程为：先执行登录操作，然后执行具体的操作（如文件/文件夹的创建、删除、移动、重命名等操作），最后执行退出操作。如上面的测试用例，登录与退出操作是相对固定的，那么我们可以把登录与退出操作进行模块化，然后调用，一方面不用写重复代码，另一方面可以使测试用例更关注具体的用例代码。

在与 webcloud.py 相同的目录下创建 login.py 文件，脚本内容如下：

```
#coding=utf-8

from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
import unittest, time

#登陆模块(函数)

def login(self):
    driver = self.driver
    driver.maximize_window()
    driver.find_element_by_id("user_name").clear()
    driver.find_element_by_id("user_name").send_keys("username")
    driver.find_element_by_id("user_pwd").clear()
    driver.find_element_by_id("user_pwd").send_keys("123456")
    driver.find_element_by_id("dl_an_submit").click()
    time.sleep(3)
```

webcloud.py

```
#coding=utf-8

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time
import login #导入登录文件

class Login(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://passport.kuaibo.com"
        selfverificationErrors = []
        self.accept_next_alert = True

    #私有云登录用例
```

```

def test_login(self):
    driver = self.driver
    driver.get(self.base_url
    "/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")
    +
#调用登录模块
    login.login(self)

#新功能引导
    driver.find_element_by_class_name("guide-ok-btn").click()
    time.sleep(3)

#退出
    driver.find_element_by_class_name("UserTool").click()
    time.sleep(2)
    driver.find_element_by_link_text("退出").click()
    time.sleep(2)

def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
    unittest.main()

```

python 基础知识补充:

进行到这里，我们有必要补充一下 python 语言中函数、类、方法的使用，这将有助于我们自动化测试脚本的开发。下面打开 python IDLE：

函数的基本使用：

```

#例1
>>> def add(a,b):
    c=a+b
    print c

```

```
>>> add(1, 3)
4

#例2

>>> def add2(a=1,b=3):
    c=a+b
    return c

>>> d=add2()
>>> print d
4
```

通过 def 关键字可创建函数，在例1中我们创建了 add() 函数， 默认接收两个参数化 a、b， 把 a、b 相加结果给 c，并将结果函数内打印。

例2中创建了 add2() 函数，这一次对 a、b 设置了默认值，同样对 a、b 做加法，并将结果用 return 返回；d 在接收 add2() 时用的是默认值，将后将 d 接收的结果打印。

类与方法的基本使用：

```
>>> class Counter:
    def add(self,a,b):
        c=a+b
        print c
    def subtract(self,a,b):
        c=a-b
        print c

>>> d=Counter()
>>> d.add(5,3)
8
>>> d.subtract(5,3)
2
```

通过 class 关键字我们创建了一个 Counter 类， 定义了 add() 和 subtract() 两个方法分别来完成加法和减法运算，并将计算结果打印。

通过上面的例子我们明显的发现类的方法与函数有一个明显的区别，在类的方法中必须有个额外的第一个参数（self），但在调用类的方法时却不必为这个参数赋值。self 参数所指的是对象本身，所以习惯性地命名为 self。

为何 Python 给 self 赋值而你不必给 self 赋值？

创建了一个类 MyClass，实例化 MyClass 得到了 MyObject 这个对象，然后调用这个对象的方法 MyObject.method(a, b)，在这个过程中，Python 会自动转为 MyClass.method(MyObject, a, b)，这就是 Python 的 self 的原理。即使你的类的方法不需要任何参数，但还是得给这个方法定义一个 self 参数，虽然我们在实例化调用的时候不用理会这个参数。

下面回到用例本身来讨论如何模块化和调用的，在 login.py 文件中：

```
def login(self):
    driver = self.driver
```

这里用到的是方法，(driver = self.driver) driver 为对象身的 driver，这一句很重要，否则我们无法在 login() 方法中使用 driver 操作浏览器。

在 webdriver.py 文件中：

```
#导入登录文件
import login
.....
#调用登录模块
login.login(self)
.....
```

首先导入 login 文件，然后对文件中的 login() 方法进行调用。

小作业：

下面笔者自己动手把退出的相关操作也进行模块化吧！（提示，可以单独创建一个退出文件写退出函数，也可以在 login.py 中写退出函数；当然也可以写一个登陆类，分别写登陆和退出方法。）

第三节、数据驱动（参数化）

在测试模型一节的数据驱动中我们已经介绍了如何通过 python 定于的数组对百度输入数据进行参数化设置，将其它循环的读取 values 数组中每一个数据。这里我们将通过读取 txt 文件中的数据来实现参数化。

创建 data.txt 文件，向文件内写放三行数据，如图4.3

d:\abc\data.txt

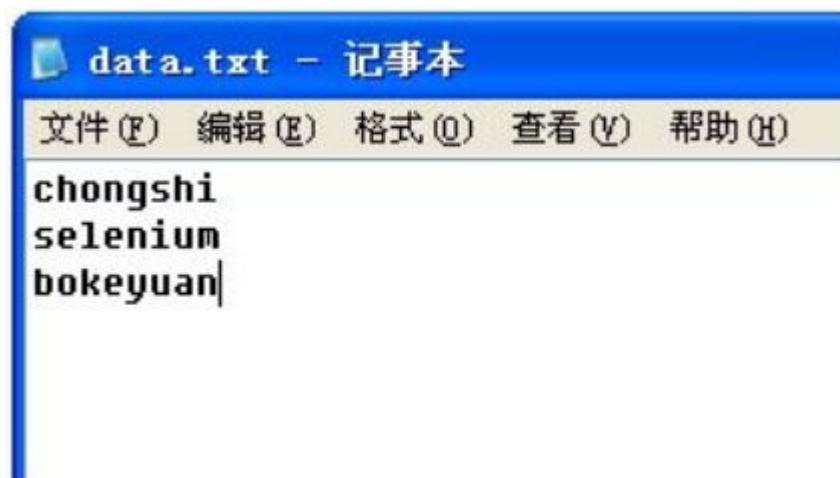


图4.3

baidu_read_data.py

```
#coding=utf-8

from selenium import webdriver

import os,time


source = open("D:\\abc\\data.txt", "r")

values = source.readlines()

source.close()
```

```
# 执行循环
```

```
for serch in values:
```

```
    browser = webdriver.Firefox()
    browser.get("http://www.baidu.com")
    browser.find_element_by_id("kw").send_keys(serch)
    browser.find_element_by_id("su").click()
    browser.quit()
```

data.txt 文件内容参考图5.x

open 方法以只读方式 (r) 打开本地的 data.txt 文件, readlines 方法是逐行的读取文件内容。

通过 for 循环, serch 可以每次获取到文件中的一行数据, 在定位到百度的输入框后, 将数据传入 send_keys(serch)。这样通过循环调用, 直到文件中的所有内容全被读取。

登录参数化（读取 txt 文件）

现在按照上面的思路, 对自动化脚本中用户名、密码进行参数化, 通过 python 文档我们发现 python 读取文件的方式有: 整个文件读取、逐行读取、固定字节读取。并没有找到一次读取两条数据的好方法。

创建两个文件, 分别存放用户名密码, 如图4.4:



图 4.4

打开之前编写的 login.py 文件, 做如下修改:

```
#coding=utf-8
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
```

```

import unittest, time, os

source = open("D:\\selenium_python\\data\\username.txt", "r") #用户名文件
un = source.read() #读取用户名
source.close()

source2 = open("D:\\selenium_python\\data\\password.txt", "r") #密码文件
pw = source2.read() #读取密码
source2.close()

def login(self):
    driver = self.driver
    driver.maximize_window()
    driver.find_element_by_id("user_name").clear()
    driver.find_element_by_id("user_name").send_keys(un)
    driver.find_element_by_id("user_pwd").clear()
    driver.find_element_by_id("user_pwd").send_keys(pw)
    driver.find_element_by_id("dl_an_submit").click()
    time.sleep(3)

```

分别打开两个 txt 文件，通过 un 和 pw 来接收用户名和密码信息，将接收的数据通过 send_key(xx) 转入到执行程序中。

运行我们前面创建的 webcloud.py 文件，程序可以正常的执行。虽然这样做比较丑，但是确实达到了数据与脚本分离的目的。

缺点：

虽然目的达到了这，但这样的实现有很多问题：

1、用户名密码分别在不同的文件里，修改用户名和密码比较麻烦。

2、username.txt 和 password.txt 文件中只能保存一个用户密码，无能很好的循环读取。

登录参数化（函数）

函数是我们前面刚介绍的 python 知识，函数可以预先给参数化赋值，借助这个特性，我们可以通过调用函数的方式对用户名密码进行参数化

userinfo.py

```
def fun(un='testing', pw=123456):
    print "success reader username and password!!"
    return un, pw
```

我们为两个参数 un 和 pw 赋了初值，赋值内容如果是字符串需要加引号，如果是数字可以不需要引号。再次打开 login.py 文件，做如下修改：

```
#coding=utf-8

from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException
import unittest, time
import userinfo #导入函数

#通过两个变量，来接收调用函数获得用户名&密码
us,pw = userinfo.fun()

#打印两个变量
print us,pw

def login(self):
    driver = self.driver
    driver.maximize_window()
    driver.find_element_by_id("user_name").clear()
    driver.find_element_by_id("user_name").send_keys(un)
    driver.find_element_by_id("user_pwd").clear()
    driver.find_element_by_id("user_pwd").send_keys(pw)
    driver.find_element_by_id("dl_an_submit").click()
    time.sleep(3)
```

单独运行 login.py 文件：

```
>>> ===== RESTART =====
>>>
success reader username and password!!
testing 123456
```

说明我们对函数的参数调用是成功的，将 un、pw 两个变量的值传入用户名密码的 send_keys() 方法中即可。

登录参数化（读取字典）

既然我们是固定的读取用户名和密码两个值，那么可以借助 python 字典的方式来完成这个需求。字典由大括号内的多键值对组成；下面继续在 python IDLE 交互模式下演示字典的创建与使用。

```
>>> data = {'abc':'123','def':'456'}
>>> print data
{'abc': '123', 'def': '456'}
>>> data.keys()
['abc', 'def']
>>> data.values()
['123', '456']
>>> data.items()
[('abc', '123'), ('def', '456')]
```

创建字典用大括号，数据由 key/value 键值对组成，keys() 方法返回字典中的键列表。values() 返回字典中的值列表，items() 返回 (key, value) 元组。

下面创建一个存放字典的函数 文件 userinfo.py :

```
def zidian():
    data={'username':'123456','testing360':'123123'}
```

```

print "success reader username and password!!"

return data

```

字典的可以方便的存放 k, v 键值对，一个键对应一个值；注意，如果密码中有非数字，需要加引号。

需要说明的是我们的需求并不适合循环的读取用户名密码，不过我们可以写一小程序单独验证这种循环读取的方式：

loop_reader.py

```

#coding=utf-8

import userinfo #导入函数

#获取字典数据
info = userinfo.zidian()

#通过 items() 循环读取元组（键/值对）
for us,pw in info.items():
    print us
    print pw

```

运行结果如下：

```

>>> ===== RESTART =====
>>>
success reader username and password!!
username
123456
testing360
123456

```

在实际使用中，可以将 un、pw 两个变量循环获得的值传入相应的 send_keys()方法中。

目前方式解决了两个问题：一次传入两个值，以及循环读取。

表单参数化（csv）

假如我有自动化脚本中要参数化一张表单，表单需要填写用户名、邮箱，年龄，性别等信息，使用上面的方法就很难来解决这个问题，下面通过读取 csv 文件的方法来解决这个问题。

创建 userinfo.csv 文件，如图 5.x

	A	B	C	D	E	F
1	testing	123456@126.com	23	男		
2	testing2	123123@qq.com	18	女		
3	testing3	456123@gmail.com	29	女		
4						
5						
6						

图 5.x

通过 WPS 或 excel 创建表格，文件另存为选择 CSV 格式，下面修改 loop_reader.py 文件进行循环读取：

```
#coding=utf-8

import csv #导入 csv 包

#读取本地 csv 文件

my_file='D:\\selenium_python\\data\\userinfo.csv'

data=csv.reader(file(my_file,'rb'))


#循环输出每一行信息

for user in data:

    print user[0]

    print user[1]

    print user[2]

    print user[3]
```

运行结果：

```
>>> ===== RESTART =====
```

```
>>>  
testing  
123456@126.com  
23  
男  
testing2  
123123@qq.com  
18  
女  
testing3  
456123@gmail.com  
29  
女
```

csv.reader()用于读取 CSV 文件， user[0] 表示表格中第一列的数据（用户名）， user[0] 表示表格中第二列的数据（邮箱），后面类推。

通过 CSV 读取文件比较灵活，可以循环读取每一条数据，从而又不局限每次所读取数据的个数。

我们这里举例了多种方式来进行参数化，虽然最终看来 CSV 的方式最灵活，这里更多的是想告诉读者解决问题的方法是多样的，我们可以用 python 的各种技巧来选择最简单的方法来解决问题。从这个过程中我们也学到了不少 python 编程技术。

我这里可以说是用 python 最基础的知识点解决了问题，这里只算提供一个思路，温习一下 python ，你一定可以找到更完美优雅的方法；解决问题的方法是多样的，使用最贴合需求的方法，简单解决问题。这一节写的比较多，对构建自动化框架来说，参数化是非常重要的一个知识点。

脚本的模块化与参数化是我们在自动化脚本开发中用到最多的两个技巧，希望读者能认真体会，灵活的运行到具体的项目中。

第五章 自动化测试用例设计

本章将简单探讨自动化测试用例的设计，笔者认为不管是手工测试，自动化测试，还是性能测试都是以测试用例为前提的。那么测试用例是测试人员综合自己人经验从需求中挖掘和提炼而来。所以不管什么类型的测试工作，我们都不能盲目开展。任何测试工作都应该以需求为基础，以测试用例为导向进行实施。

第一节、手工测试用例与自动化测试用例

手工测试用例是针对手工测试人员，自动化测试用例是针对自动化测试框架，前者是手工测试用例人员应用手工方式进行用例解析，后者是应用脚本技术进行用例解析，两者最大的各自特点在于，前者具有较好的异常处理能力，而且能够基于测试用例，制造各种不同的逻辑判断，而且人工测试步步跟踪，能够细致的定位问题。而后者是完全按照测试用例的方式测试，而且异常处理能力不强，往往一个自动化测试用例运行完毕后，报一堆错误，对于测试人员来定位错误是一个难点，这样往往发现的问题很少。

手工测试用例与自动化测试用例对比：

手工测试用例

- 较好的异常处理能力，能通过人为的逻辑判断校验当前步骤的功能实现正确与否。
- 人工执行用例具有一定的步骤跳跃性。
- 人工测试步步跟踪，能够细致的定位问题。
- 主要用来发现功能缺陷

自动化测试用例

- 执行对象是脚本，任何一个判断都需要编码定义。
- 用例步骤之间关联性强。
- 主要用来保证产品主体功能正确完整和让测试人员从繁琐重复的工作中解脱出来。

-
- 目前自动化测试阶段定位在冒烟测试和回归测试。

通过对比我们可以看到，手工测试用例与自动化测试用例之间是存在差异的。所以，直接拿手工测试用例来直接转化成自动化测试脚本。

在此笔者需要强调：自动化测试替代不了手工测试，目的仅仅在于让测试人员从繁琐重复的机械式测试过程解脱出来，把时间和精力投入到更有价值的地方，从而挖掘更多的产品缺陷。目前自动化测试更多的时候是定位在冒烟测试和回归测试；冒烟测试执行的是主体功能点的用例。回归测试执行全部或部分的测试用例。

怎么编写自动化测试用例，如何将自动化测试用例和手工测试用例相辅相成。

用例选型注意事项：

- 1、 不是所有的手工用例都要转为自动化测试用例。
- 2、 考虑到脚本开发的成本，不要选择流程太复杂的用例。如果有必要，可以考虑把流程拆分多个用例来实现脚本。
- 3、 选择的用例最好可以构建成场景。例如一个功能模块，分 n 个用例，这 n 个用例使用同一个场景。这样的好处在于方便构建关键字测试模型。
- 4、 选择的用例可以带有目的性，例如这部分用例是用例做冒烟测试，那部分是回归测试等，当然，会存在重叠的关系。如果当前用例不能满足需求，那么唯有修改用例来适应脚本和需求。
- 5、 选取的用例可以是你认为是重复执行，很繁琐的部分，例如字段验证，提示信息验证这类。这部分适用回归测试。
- 6、 选取的用例可以是主体流程，这部分适用冒烟测试。
- 7、 自动化测试也可以用来做配置检查，数据库检查。这些可能超越了手工用例，但是也算用例拓展的一部分。项目负责人可以有选择地增加。
- 8、 如果平时在手工测试时，需要构造一些复杂数据，或重复一些简单机械式动作，告诉自动化脚本，让他来帮你。或许你的效率因此又提高了。

第二节、测试类型

测试静态内容

静态内容测试是最简单的测试,用于验证静态的、不变化的 UI 元素的存在性。例如:

- 每个页面都有其预期的页面标题?这可以用来验证链接指向一个预期的页面。
- 应用程序的主页包含一个应该在页面顶部的图片吗?
- 网站的每一个页面是否都包含一个页脚区域来显示公司的联系方式,隐私政策,以及商标信息?
- 每一页的标题文本都使用的<h1>标签吗?每个页面有正确的头部文本内吗?

您可能需要或也可能不需要对页面内容进行自动化测试。如果您的网页内容是不易受到影响手工对内容进行测试就足够了。如果,例如您的应用文件的位置被移动,内容测试就非常有价值。

测试链接

Web 站点的一个常见错误为的失效的链接或链接指向无效页。链接测试涉及点各个链接和验证预期的页面是否存在。如果静态链接不经常更改,手动测试就足够。但是,如果你的网页设计师经常改变链接,或者文件不时被重定向,链接测试应该实现自动化。

功能测试

在您的应用程序中,需要测试应用的特定功能,需要一些类型的用户输入,并返回某种类型的结果。通常一个功能测试将涉及多个页面,一个基于表单的输入页面,其中包含若干输入字段、提交“和”取消“操作,以及一个或多个响应页面。用户输入可以通过文本输入域,复选框,下拉列表,或任何其他的浏览器所支持的输入。

功能测试通常是需要自动化测试的最复杂的测试类型,但也通常是最重要的。典型的测试是登录,注册网站账户,用户帐户操作,帐户设置变化,复杂的数据检索操作等等。功能测试通常对应着您的应用程序的描述应用特性或设计的使用场景。

测试动态元素

通常一个网页元素都有一个唯一的标识符,用于唯一地定位该网页中的元素。通常情况下,唯一标识符用 HTML 标记的' id' 属性或' name' 属性来实现。这些标识符可以是一个静态的,即不变的、字符串常量。

它们也可以是动态生产值,在每个页面实例上都是变化的。例如,有些 Web 服务器可能在一个页面实例上命名所显示的文件为 doc3861,并在其他页面实例上显示为 doc6148,这取决于用户在检索的‘文档’。验证文件是否存在测试脚本,可能无法找到不变的识别码来定位该文件。通常情况下,具有变化的标识符的动态元素存在于基于用户操作的结果页面上,然而,显然这取决于 Web 应用程序。

下面是一个例子。

```
<input id="addForm:_ID74:_ID75:0:_ID79:0:checkBox" type="checkbox" value="true" />
```

这是一个 HTML 标记的复选框,其 ID (addForm:_ID74:_ID75:0:_ID79:0:checkBox) 是一个动态生成的值。这个页面下次被打开时入框的 ID 将可能是一个不同的值。

Ajax 的测试

Ajax 是一种支持动态改变用户界面元素的技术。页面元素可以动态更改,但不需要浏览器重新载入页面,如动画,RSS 源,其他实时数据更新等等。**Ajax** 有不计其数的更新网页上的元素的方法。但是了解 **AJAX** 的最简单的方式,可以这样想,在 **Ajax** 驱动的应用程序中,数据可以从应用服务器检索,然后显示在页面上,而不需重新加载整个页面。只有一小部分的页面,或者只有元素本身被重新加载。

断言 assert 与验证 verify

什么时候使用断言命令,什么时候使用验证命令?这取决于你。差别在于在检查失败时,你想让测试程序做什么。你想让测试终止,还是想继续而只简单地记录检查失败?

这需要权衡。如果您使用的断言,测试将在检查失败时停止,并不运行任何后续的检查。有时候,也许是经常的,这是你想要的。如果测试失败,你会立刻知道测试没有通过。**TestNG** 和 **JUnit** 等测试引擎提供在开发测试脚本时常用的插件,可以方便地标记那些测试为失败的测试。优点:你可以直截了当地看到检查是否通过。缺点:当检查失败,后续的检查不会被执行,无法收集那些检查的结果状态。

相比之下,验证命令将不会终止测试。如果您的测试只使用验证,可以得到保证是一假设没有意外的异常—测试会被执行完毕,而不管是否发现缺陷。缺点:你必须做更多的工作,以检查您的测试结果。也就是说,你不会从 **TestNG** 和 **JUnit** 得到反馈。您将需要在打印输出控制台或日志文件中查看结果。每次运行测试,你都需要花时间去查看结果输出。如果您运行的是数以百计的测试,每个都有它自己的日志,这将耗费时间。及时得到反馈会更合适,因此断言通常比验证更常使用。

第三节、python 异常断言

在实际的脚本开发中，我们需要用到 python 的异常处理来捕捉异常和抛异常，所以我们有需要学习和使用 python 的异常处理。

```
>>> open('abc.txt','r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'abc.txt'
```

打开一个不存在的文件 `abc.txt` 文件，当系统找不到 `abc.txt` 文件时，就会抛出给我们一个 `IOError` 类型的错误，`No such file or directory: abc.txt`（没有 `abc.txt` 这样的文件或目录）

Try...except...

假如，我们已经知道这种类型的错误，那么就可以通过一个异常捕捉来捕捉这个错误。我们可以通过 `try...` 来接收这个错误。打开文件写入：

```
try:
    open("abc.txt",'r')
except IOError:
    pass
```

再来运行程序就会看不到任何错误，因为我们用 `except` 接收了这个 `IOError` 错误。`pass` 表示实现了相应的实现，但什么也不做。

假如我不是打开一个文件，而是输出一个没有定义的变量呢？

```
try:
    print aa
except IOError:
    pass
```

显然，在上面的代码中我并没有对 `aa` 赋值，运行结果：

```
Traceback (most recent call last):
  File "/home/fnngj/py_se/tryy.py", line 3, in <module>
    print aa
NameError: name 'aa' is not defined
```

我们已经用 `except` 接收错误了，为什么错误还是还是抛出来了。如果你细心会发现这一次的错误类型是 `NameError`，而我接收类型是 `IOError`，所以要想接收这个 `print` 的错误，那么需要修改接收错误的类型为 `NameError`

虽然，我知道 `print` 语句是可能会抛一个 `NameError` 类型的错误，虽然接收了这个类型错误，但我不知道具体的错误提示信息是什么。那么，我能不能把错误信息打印出来呢？当然可以：

```
try:
    print aa
except NameError, msg:
    print msg
```

我们在接收错误类型的后面定义一个变量 `msg` 用于接收具体错误信息，然后将 `msg` 接收的错误信息打印。再来运行程序：

name 'aa' is not defined

现在只有一行具体错误信息。

异常的抛出机制：

- 1、如果在运行时发生异常，解释器会查找相应的处理语句（称为 `handler`）。
- 2、要是在当前函数里没有找到的话，它会将异常传递给上层的调用函数，看看那里能不能处理。
- 3、如果在最外层（全局“main”）还是没有找到的话，解释器就会退出，同时打印出 `traceback` 以便让用户找到错误产生的原因。

注意： 虽然大多数错误会导致异常，但一个异常不一定代表错误，有时候它们只是一个警告，有时候它们可能是一个终止信号，比如退出循环等。

try...finally...

Try...finally...子句用来表达这样的情况：

我们不管线捕捉到的是什么错误，无论错误是不是发生，这些代码“必须”运行，比如文件关闭，释放锁，把数据库连接返还给连接池等。

创建文件 poem.txt

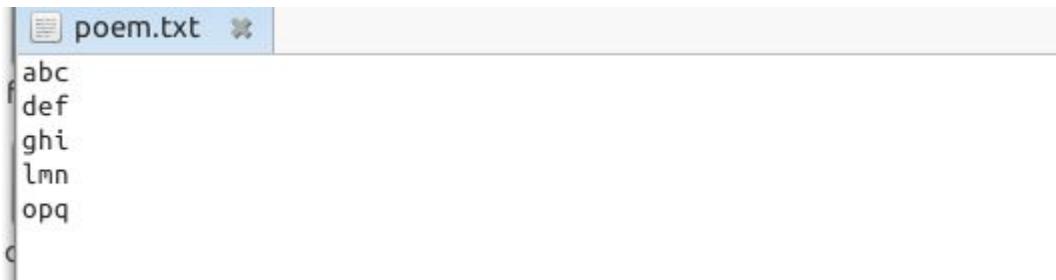


图 5.1

tryf.py

```
import time

try:
    f = file('poem.txt')
    while True: # our usual file-reading idiom
        line = f.readline()
        if len(line) == 0:
            break
        time.sleep(2)
        print line,
finally:
    f.close()
    print 'Cleaning up...closed the file'
```

运行程序(在 windows 命令提示符或 linux 终端下运行):

```
...$ python tryf.py
abc
efg
^CCleaning up...closed the file
Traceback (most recent call last):
  File "tryf.py", line 18, in <module>
    time.sleep(2)
KeyboardInterrupt
```

程序读 poem.txt 文件中每一行数据打印，但是我有意在每打印一行之前用 time.sleep 方法暂停2秒钟。这样做的原因是让程序运行得慢一些。在程序运行的时候，按 Ctrl-c 中断/取消程序。

我们可以观察到 KeyboardInterrupt 异常被触发，程序退出。但是在程序退出之前，finally 从句仍然被执行，把文件关闭。

到目前为止，我们只讨论了如何捕捉异常，那么如何抛出异常呢？

Raise 抛出异常

使用 raise 来抛出一个异常：

tryr.py

```
#coding=utf-8
filename = raw_input('please input file name:')

if filename=='hello':
    raise NameError('input file name error !')
```

程序要求用户输入一个文件名，如果用户输入的文件名是 hello，那么抛出一个 NameError 的异常，用户输入 hello 和 NameError 异常之间没有任何必然联系，我只是人为的通过 raise 来这样定义，我当然也可以定义称 TypeError，但我定义的异常类型必须是 python 提供的。

异常	描述
AssertionError	assert语句失败
AttributeError	试图访问一个对象没有的属性
IOError	输入输出异常，基本是无法打开文件
ImportError	无法引入模块或者包，基本是路径问题
IndentationError	语法错误，代码没有正确的对齐
IndexError	下标索引超出序列边界
KeyError	试图访问你字典里不存载的键
KeyboardInterrupt	Ctrl+C被按下
NameError	使用一个还未赋予对象的变量
SyntaxError	Python代码逻辑语法出错，不能执行
TypeError	传入的对象类型与要求不符
UnboundLocalError	试图访问一个还未设置的全局变量，基本上是由于另有一个同名的全局变量，导致你以为在访问
ValueError	传入一个不被期望的值，即使类型正确

图 5.2

第四节、webdriver 错误截图

Webdriver 提供错误截图函数 `get_screenshot_as_file()`, 可以帮助我们跟踪 bug, 在脚本无法继续执行时候, `get_screenshot_as_file()` 函数将截取当前页面的截图保存到指定的位置, 这是一个非常棒的功能, 下面实例展示 `get_screenshot_as_file()` 函数的使用。

```
#coding=utf-8

from selenium import webdriver

browser = webdriver.Firefox()

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

#捕捉百度输入框异常

try:

    browser.find_element_by_id("kwsss").send_keys("selenium")

    browser.find_element_by_id("su").click()

except:

    browser.get_screenshot_as_file("/home/fnngj/python/error_png.png")

browser.quit()
```

显然, 我们对百度输入框的 `id` 定位动了手脚, 并没有 `id=kwsss` 元素, 所以脚本运行到此处后就无法继续执行了, 我们通过 `try` 对捕捉了这个异常, 在 `except` 中, 我们通过 `get_screenshot_as_file()` 函数截图当前页面并保存到指定的路径下面。

打开`/home/fnngj/python/` 我们将看到生成的 `error_png.png` 文件:



图5.3

第五节、自动化测试用例设计实例

上一节我们简单讨论了手工测试用例与自动化测试用之间的差异，以及自动化测试用例设计时的注意事项，这一节就通过实例向读者介绍如何编写具体的自动化测试用。

笔者以快播私有云产品为例：

<http://webcloud.kuaibo.com/>

快播私有云是快播社区的产品之一，为用户提供免费的在线空间，读者进入空间后可以收藏用户分享的影片，同时可以将自己的影片分享给其他用户。对于私有云本身具有创建文件夹，文件/文件夹重命名，删除到回收，文件/文件夹的移动，去除重复影片，影片播放等功能。

在编写用例之间，笔者再次强调几点编写自动化测试用例的原则：



1、一个脚本是一个完整的场景，从用户登陆操作到用户退出系统关闭浏览器。

2、一个脚本只验证一个功能点，不要试图用户登陆系统后把所有的功能都进行验证再退出系统

3、尽量只做功能中正向逻辑的验证，不要考虑太多逆向逻辑的验证，逆向逻辑的情况很多（例如手机号输错有很多种情况），验证一方面比较复杂，需要编写大量的脚本，另一方面自动化脚本本身比较脆弱，很多非正常的逻辑的验证能力不强。（我们尽量遵循用户正常使用原则编写脚本即可）

4、脚本之间不要产生关联性，也就是说编写的每一个脚本都是独立的，不能依赖或影响其他脚本。

5、如果对数据进行了修改，需要对数据进行还原。

6、在整个脚本中只对验证点进行验证，不要对整个脚本每一步都做验证。

5.5.1 登陆用例实例



图5.4

笔者建议通过 excle 表格来编写自动化测试用例。

用例001：

用例 id	login	用户登录	
步骤:	动作	数据	验证点
1	打开打开登陆页	http://webcloud.kuaibo.com	
2	用户登陆	username 123456	匹配用户昵称“虫师”
3	用户退出		

备注：通过匹配用户登录之后的昵称来判断用户是否登录成功。

用例脚本 (login.py):

```
#coding=utf-8

from selenium import webdriver

from selenium.webdriver.common.action_chains import ActionChains

import time
```

```
driver = webdriver.Firefox()

driver.get("http://passport.kuaibo.com/login/?referrer=http%3A%2F%2Fwebcloud
.kuaibo.com%2F")

driver.maximize_window() #浏览器最大化

#登陆快播私有云

driver.find_element_by_id("user_name").send_keys("testing360")

driver.find_element_by_id("user_pwd").send_keys("198876")

driver.find_element_by_id("dl_an_submit").click()

time.sleep(3)

#获取用户名

now_user=driver.find_element_by_xpath("//div[@id='Nav']/ul/li[4]/a[1]/span")
.text

#用户名是否等于虫师，不等于将抛出异常

if now_user==u'虫师':
    print '登陆成功'
else:
    raise NameError('user name error!')


#退出

driver.find_element_by_class_name("Usertool").click()

time.sleep(2)

driver.find_element_by_link_text("退出").click()

time.sleep(2)

driver.close()
```

5.5.2 添加文件用例实例



图5.5

用例002：

用例 id	collect	添加用户分享文件	
步骤:	动作	数据	验证点
1	打开打开登陆页	http://webcloud.kuaibo.com	
2	用户登陆	username 123456	
3	收藏用户分享文件		判断列表文件总数加1
4	用户退出		

备注：通过计算用户列表中的文件的数量来判断文件是否添加成功。

用例脚本 (collect.py):

注：用例登陆与退出参考用例001，本用例只关注收藏用户分享的逻辑代码。

```
#判断当前文件个数

inputs=driver.find_elements_by_tag_name('input')

n=0

for i in inputs:

    if i.get_attribute('type')=="checkbox":

        n=n+1

print u"当前列表文件为 %d" %n
```

```
#收藏用户分享文件

driver.find_element_by_class_name("collect").click()

time.sleep(3)

#再次获取当前文件的个数

inputs=driver.find_elements_by_tag_name('input')

ns=0

for ii in inputs:

    if ii.get_attribute('type')=="checkbox":

        ns=ns+1

print u"当前列表文件为 %d" %ns

#判断执行收藏文件之后比收藏之间文件加1，否则抛异常

if ns==n+1:

    print "ok!"

else:

    raise NameError('添加文件失败！！')
```

5.5.3 删除文件用例实例



图5.6

用例003：

用例 id	del_one_file	删除单个文件	
步骤:	动作	数据	验证点
1	打开打开登陆页	http://webcloud.kuaibo.com	
2	用户登陆	username 123456	
3	删除单个文件		判断列表文件总数减1
4	添加文件1个文件		
5	用户退出		

备注：因为删除了一个文件对文件的数据发生的改变，如果多次执行脚本，列表中的文件被删除完了就会引起，所以在删除一个文件后，需要再添加一文件，但添加文件操作不做验证。

用例脚本 (del_one_file.py):

```
#判断当前文件个数

inputs=driver.find_elements_by_tag_name('input')

n=0

for i in inputs:

    if i.get_attribute('type')=="checkbox":

        n=n+1

print u"当前列表文件为 %d" %n
```

```
#删除操作
```

```
driver.find_element_by_xpath("//html/body/div/div[2]/div[2]/div/div[4]/table/tbody/tr/td/input").click()

driver.find_element_by_class_name("delete").click()

driver.find_element_by_xpath("//html/body/div[2]/div[2]/div[2]/div").click()

time.sleep(4)
```

```
#再次获取当前文件的个数
```

```
inputs=driver.find_elements_by_tag_name('input')

ns=0

for ii in inputs:

    if ii.get_attribute('type')=="checkbox":

        ns=ns+1

print u"当前列表文件为 %d" %ns
```

```
#判断执行删除单个文件之后比删除之后文件减1，否则抛异常
```

```
f ns==n-1:

    print "ok!"

else:

    raise NameError('删除文件失败！！')
```

```
#收藏用户分享单个文件
```

```
driver.find_element_by_class_name("collect").click()

time.sleep(3)
```

5.5.4 重命名文件用例实例



图5.7

用例004：

用例 id	renaming	文件重命名	
步骤:	动作	数据	验证点
1	打开打开登陆页	http://webcloud.kuaibo.com	
2	用户登陆	username 123456	
3	文件重命名	新文件名	
4	用户退出		

备注：文件的重命名其实我们很难找到证据（验证点）证明重命名成功，那么脚本整个运行没有报错，我们也可模糊的判断功能测试是 OK 的。

用例脚本（renaming.py）

```
#勾选重命名的文件

driver.find_element_by_xpath("/html/body/div/div[2]/div[2]/div/div[4]/table/tbody[5]/tr/td/input").click()

time.sleep(3)

#鼠标移动到“更多”按钮弹下拉框
```

```

element=driver.find_element_by_class_name("more-fe")

ActionChains(driver).move_to_element(element).perform()

time.sleep(2)

#在 li 标签（更多 下拉框）中筛选到 data-action==rename（重命名）选项点击

lis=driver.find_elements_by_tag_name('li')

for li in lis:

    if li.get_attribute('data-action') == 'rename':

        li.click()

    time.sleep(2)

```

在 input 标签中筛选 type==text 的重命名输入框

```

inputs=driver.find_elements_by_tag_name('input')

for input in inputs:

    if input.get_attribute('type') == 'text':

        input.send_keys(u"新文件名") #进行重名操作

        input.send_keys(Keys.ENTER) #回车确认重命名

    time.sleep(2)

```

总结：

在本章中，简单对比了手工测试用户与自动化测试用例的区别，自动化测试用例编写的原则，如何通过 python 捕捉异常和抛出异常，以及 webdriver 提供的 get_screenshot_as_file() 函数，以及如何编写自动化用例与脚本等。

不过笔者先不要急于开始实施自动化测试，虽然我们可以编写单个的测试用例，并通过异常捕捉判断用例是否运行成功。但只有与通过测试框架的整合，我们才能真正有效可行的运用自动化测试技术。

第六章 引入 unittest 单元测试框架

第一节、selenium IDE 介绍

selenium IDE 是 selenium 家族的一员，它是嵌入到 firefox 浏览器的一个插件，这里之所以要介绍 selenium IDE 是因为我们可以将 selenium IDE 录制的脚本转换成不同语言脚本，有助于帮助我们尽快熟悉脚本语言以及测试框架。

6.1.1 selenium IDE 安装

通过 firefox 浏览器访问 selenium 下载页面：<http://docs.seleniumhq.org/download/>

在 selenium IED 下载介绍部分，点击版本号链接，如图 6.1：

Selenium IDE

Selenium IDE is a Firefox plugin which records and plays back user interactions with the browser. Use this to either create simple scripts or assist in exploratory testing. It can also export Remote Control or WebDriver scripts, though they tend to be somewhat brittle and should be overhauled into some sort of Page Object-y structure for any kind of resiliency.

Download latest released version [2.4.0](#) released on 16/Sep/2013 or view the [Release Notes](#) and then [install some plugins](#).

Download version under development [unreleased](#) (currently disabled)

图 6.1

firefox 浏览器将自动识别需要下载的 selenium IED 插件，如图 6.2，点击 Install Now 按钮，安装 selenium IED 插件。



图 6.2

安装完成后重启 firefox 浏览器，通过菜单栏“Tools(工具)” ---> selenium IDE 可以打开，或通过 Ctrl+Alt+S 快捷键打开。

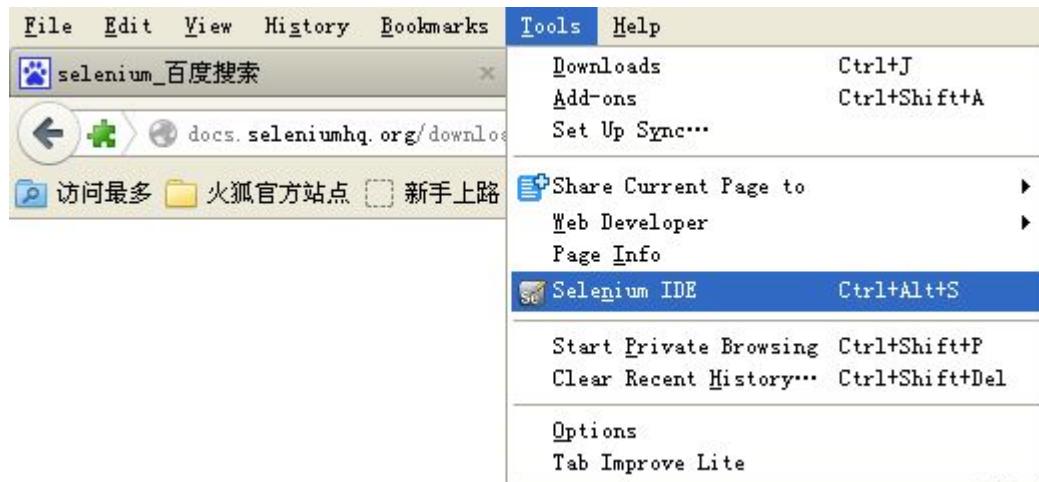


图 6.3

6.1.2 selenium IDE 界面介绍

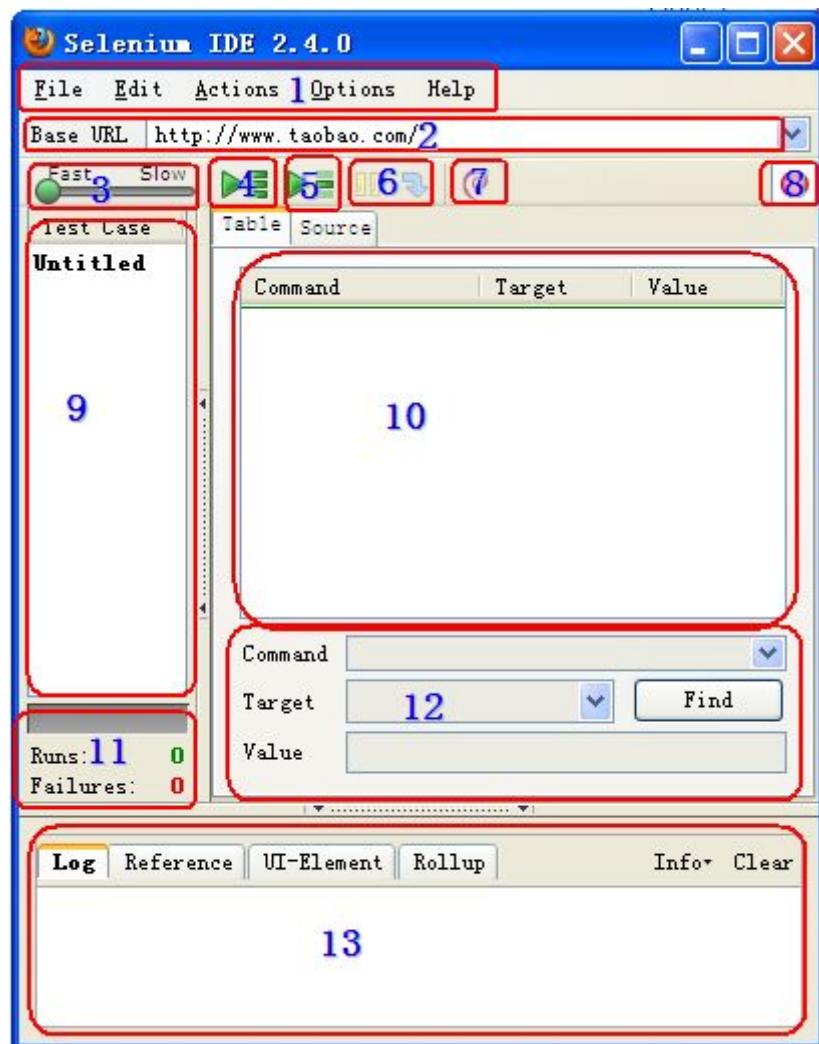


图 6.3

为了方便简洁，我们就按照上图的数字标记介绍 selenium IDE 界面各个部分人作用：

1---文件(File): 创建、打开和保存测试案例和测试案例集。

编辑(Edit): 复制、粘贴、删除、撤销和选择测试案例中的所有命令。

Options (设置): 用于设置 selenium IDE。

2---用来填写被测网站的地址。

3---速度控制：控制案例的运行速度。

4---运行所有：运行一个测试案例集中的所有案例。

5---运行：运行当前选定的测试案例。

6---暂停/恢复：暂停和恢复测试案例执行。

7---|单步：可以运行一个案例中的一行命令。

8---录制：点击之后，开始记录你对浏览器的操作。

9---案例集列表。

10---测试脚本；**table** 标签：用表格形式展现命令及参数。**source** 标签：用原始方式展现，默认是 **HTML** 语言格式，也可以用其他语言展示。

11---查看脚本运行通过/失败的个数。

12---当选中前命令对应参数。

13---日志/参考/UI 元素/Rollup

日志：当你运行测试时，错误和信息将会自定显示。

参考：当在表格中输入和编辑 **selenese** 命令时，面板中会显示对应的参考文档。

UI 元素/Rollup：参考帮助菜单中的，**UI-Element Documentation**。

6.1.3 selenium IDE 录制脚本

打开 selenium IDE 录制按钮默认为启动状态，在地址栏中输入要录制的 URL（如，<http://www.baidu.com>），脚本录制完成，关闭录制按钮，如图 6.4



图 6.4

6.1.4 selenium IDE 编辑脚本

selenium IDE 为我们录制的脚本不是100%符合我们的需求的，所以，编辑录制的脚本是必不可少的工作。

1. 编辑一行命令或注释。

在 Table 标签下选中某一行命令，命令由 command、Target、value 三部分组成。可以对这三部分内容那进行编辑。

Command	Target	Value
open	about:home	
type	id=searchT...	selenium rc...
clickAndWait	id=searchS...	
click	//ol[@id='r...	

Command type
Target id=searchTe... ▾ Find
Value selenium rc 环境配置

图 6.5

2. 插入命令

在某一条命令上右击，选择“insert new command”命令，就可以插入一个空白，然后对空白行进程编辑。

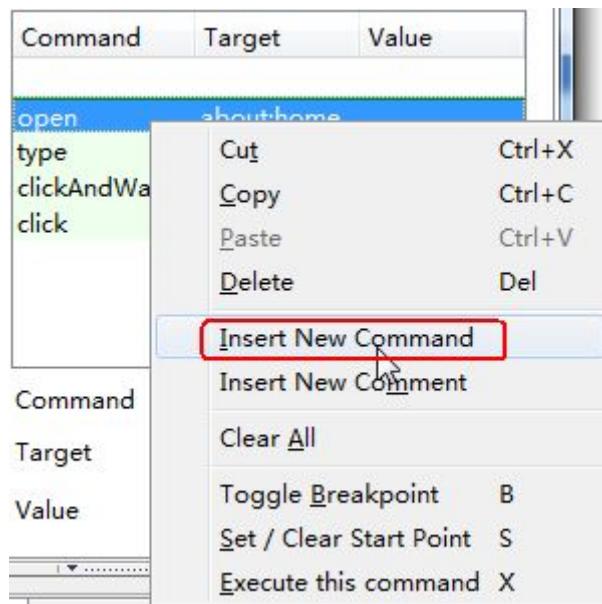


图 6.6

3. 插入注解

以上面同样的方式右击选择“insert new comment”命令插入注解空白行，本行内容不被执行，可以帮助我们更好的理解脚本，插入的内容以紫色字体显示。

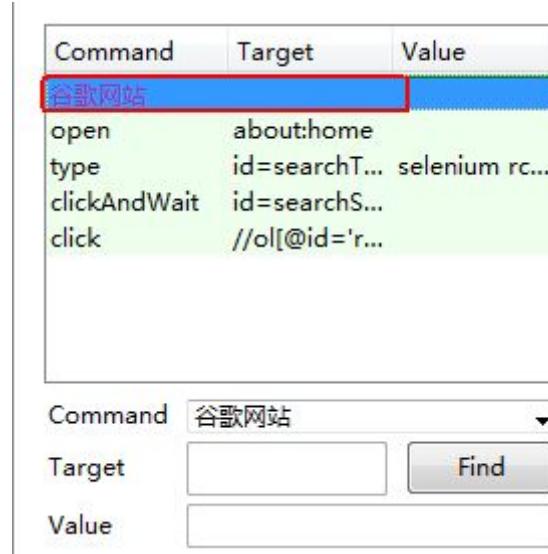


图 6.7

4. 移动命令或注解

有时我们需要移动某行命令的顺序，我们只需要左击鼠标拖动到相应的位置即可。

Command	Target	Value
open	about:home	
type	id=searchT...	selenium rc...
clickAndWait	id=searchS...	
click	//ol[@id='r...	

图6.8

5. 定位辅助

当 selenium IDE 录制脚本时，它会存储额外的信息，支持用户挑选其他格式的定位器来代替默认格式的定位器，这种特殊性对于学习定位器很有用。

The screenshot shows the Selenium IDE interface with a dropdown menu open under the 'Value' column of a table. The table contains the following commands:

Command	Target	Value
open	/	
type	id=lst-ib	selenium
click	name=btnG	value="Google 搜索"

The dropdown menu lists several locator strategies:

- name=btnG (highlighted with a red box)
- css=input[name...]
- //input[@name...]
- //span[@id='b...']
- //input[51]
- xpath:attributes
- xpath:idRelative
- xpath:position

图6.9

我们可以选择其他的命令来代替“name=btnG”命令，当然，脚本依然可以运行的。

关于 selenium IDE 的更多使用技巧不在不是本书的讨论重点，请读者参考其资料进行学习。

第二节、引入 unittest 框架

通过 selenium IDE 完成脚本的录制之后，可以将其导出为加了 python unittest 单元测试框架的相应脚本。如图 6.10。

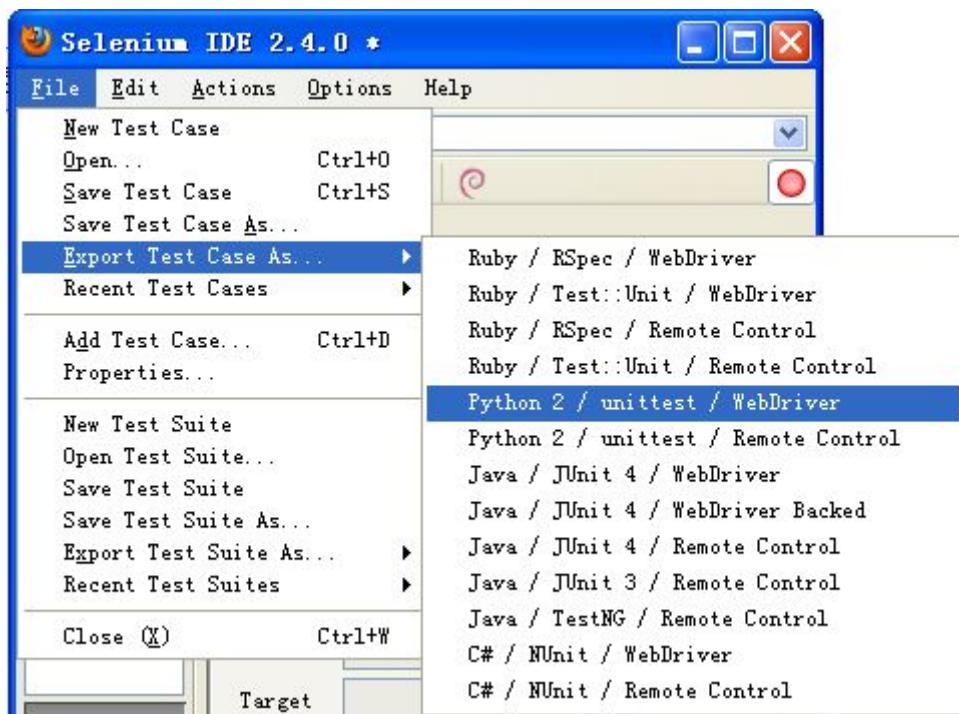


图 6.10

将脚本导出，保存为 baidu.py，通过 python IDLE 编辑器打开。如下：

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re

class Baidu(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"          self.verificationErrors = []
        self.accept_next_alert = True
```

```

def test_baidu(self):
    driver = self.driver
    driver.get(self.base_url + "/")
    driver.find_element_by_id("kw").send_keys("selenium webdriver")
    driver.find_element_by_id("su").click()
    driver.close()

def is_element_present(self, how, what):
    try: self.driver.find_element(by=how, value=what)
    except NoSuchElementException, e: return False
    return True

def is_alert_present(self):
    try: self.driver.switch_to_alert()
    except NoAlertPresentException, e: return False
    return True

def close_alert_and_get_its_text(self):
    try:
        alert = self.driver.switch_to_alert()
        alert_text = alert.text
        if self.accept_next_alert:
            alert.accept()
        else:
            alert.dismiss()
        return alert_text
    finally: self.accept_next_alert = True

def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
    unittest.main()

```

加入 unittest 框架后，看上去比我们之前见的脚本复杂了很多，除了中间操作浏览器的几行是我们所熟悉的脚本，其它的方法都是做什么用的？我们来简单分析一下！

```
import unittest
```

首先要引入 unittest 框架包。

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

Baidu 类继承 unittest.TestCase 类，从 TestCase 类继承是告诉 unittest 模块的方式，这是一个测试案例。

```
def setUp(self):
    self.driver = webdriver.Firefox()
    self.base_url = "http://www.baidu.com/"
```

setUp 用于设置初始化的部分，在测试用例执行前，这个方法中的函数将先被调用。这里将浏览器的调用和 URL 的访问放到初始化部分。

```
self.verificationErrors = []
```

脚本运行时，错误的信息将被打印到这个列表中。

```
self.accept_next_alert = True
```

是否继续接受下一个警告。

```
def test_baidu(self):
    driver = self.driver
    driver.get(self.base_url + "/")
    driver.find_element_by_id("kw").send_keys("selenium webdriver")
    driver.find_element_by_id("su").click()
```

test_baidu 中放置的就是我们的测试脚本了，这部分我们并不陌生；因为我们执行的脚本就在这里。

```
def is_element_present(self, how, what):
    try: self.driver.find_element(by=how, value=what)
    except NoSuchElementException, e: return False
    return True
```

is_element_present 函数用来查找页面元素是否存在，try...except... 为 python 语言的异常捕捉。is_element_present 函数在这里用处不大，通常删除，因为判断页面元素是否存在一般都加在 testcase 中。

```
def is_alert_present(self):
    try: self.driver.switch_to_alert()
    except NoAlertPresentException, e: return False
    return True
```

对弹窗异常的处理



```
def close_alert_and_get_its_text(self):
    try:
        alert = self.driver.switch_to_alert()
        alert_text = alert.text
        if self.accept_next_alert:
            alert.accept()
    else:
        alert.dismiss()
    return alert_text
finally: self.accept_next_alert = True
```

关闭警告以及对得到文本框的处理，if 判断语句前面已经多次使用，并不陌生；try....finally...为 python 的异常处理。

```
def tearDown(self):
    self.driver.quit()
    self.assertEqual([], self.verificationErrors)
```

`tearDown` 方法在每个测试方法执行后调用，这个地方做所有测试用例执行完成的清理工作，如退出浏览器等。

```
self.assertEqual([], self.verificationErrors)
```

这个是难点，对前面 `verificationErrors` 方法获得的列表进行比较；如查 `verificationErrors` 的列表不为空，输出列表中的报错信息。

```
if __name__ == "__main__":
    unittest.main()
```

`unittest.main()` 函数用来测试 类中以 `test` 开头的测试用例

这样一一分析下来，我们对 `unittest` 框架有了初步的了解。运行脚本，因为引入了 `unittest` 框架，所以控制台输出了用例的执行个数、时间以及是否 `OK` 等信息。

```
>>> ===== RESTART =====
>>>
.
-----
Ran 1 test in 10.656s
OK
```

>>>

本节只是初步对 `unittest` 的框架进行了分析，还有许多细节将放在后面章节进行讨论，例如 `python` 的异常处理机制等。

第三节、`unittest` 单元测试框架解析

通过上一节的学习我们对 `unittest` 框架有了初步的了解。这一节将详细介绍 `unittest` 框架是如何帮助我们完成单元测试的。因为 `unittest` 是我们后续开展自动化测试的基础，所以，请读者认真学习 `unittest` 框架的使用。

单元测试负责对最小的软件设计单元（模块）进行验证，它使用软件设计文档中对模块的描述作为指南，对重要的程序分支进行测试以发现模块中的错误。`unittest` 框架（又名 PyUnit 框架）为 `python` 语言的单元测试框架，从 Python 2.1 及其以后的版本都将 PyUnit 作为一个标准模块放入 `python` 开发包中。

下面通过具体例子介绍 `unittest` 的使用：

`widget.py`（被测试类）

```
#coding= utf-8

# 将要被测试的类
class Widget:

    def __init__(self, size = (40, 40)):
        self._size = size

    def getSize(self):
        return self._size

    def resize(self, width, height):
        if width < 0 or height < 0:
            raise ValueError, "illegal size"
        self._size = (width, height)

    def dispose(self):
        pass
```

python 基础知识补充：

__init__()

__init__()方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的初始化。

getSize()

在面向对象的编程语言中都会有类的概念，类具有封装性；在 C++、java 等语言中通过 private（私有）、protected（保护）、public（公有）等修饰符来限定访问权限。在 Python 中没有显式的 private 和 public 限定符，如果要将一个方法声明为 private 的，只要在方法名前面加上“__”即可。

所以，我们前面定义的__init__()方法是一个私有的方法，不能直接被外部使用。那么如何才能使用类中私有的成员函数着，就通过 getXX 和 setXX 方法来访问。一个赋值函数（getXX），一个取值函数（setXX）。

从上面的例子中看到，__init__()方法中默认参数是 size=(40, 40) 在函数体中定义 self._size = size。通过变量传递，self._size=(40, 40)，但 self._size 是私有的，不可被类以外的方法和函数调用。（self 表示类本身，后面章节中会进一步解释。）

所以，在 getSize()方法中定义返回 self._size，那么就可以调用 getSize()方法来使用__init__()方法中 self._size。

auto.py（测试类）

```
#coding= utf-8

from widget import Widget
import unittest

# 执行测试的类
class WidgetTestCase(unittest.TestCase):

    def setUp(self):
        self.widget = Widget()

    def testSize(self):
        self.assertEqual(self.widget.getSize(), (40, 40))

    def tearDown(self):
        self.widget = None

# 构造测试集
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testSize"))
    return suite
```

```
# 测试
if __name__ == "__main__":
    unittest.main(defaultTest = 'suite')
```

- 用 import 语句引入 unittest 模块
- 让所有执行测试的类都继承于 TestCase 类，可以将 TestCase 看成是对特定类进行测试的方法的集合
- setUp() 方法中进行测试前的初始化工作，tearDown() 方法中执行测试后的清除工作。setUp() 和 tearDown() 都是 TestCase 类中定义的方法
- 在 testSize() 中调用 assertEquals() 方法，对 Widget 类中 getSize() 方法的返回值和预期值进行比较，确保两者是相等的，assertEquals() 也是 TestCase 类中定义的方法。
- 提供名为 suite() 的全局方法，PyUnit 在执行测试的过程调用 suit() 方法来确定有多少个测试用例需要被执行，可以将 TestSuite 看成是包含所有测试用例的一个容器。



框架分析

软件测试中最基本的组成是单元测试用例（test case），我们在实际测试过程中，不可能真对一个功能（类）只写一个用例。TestCase 在 PyUnit 测试框架中被视为测试单元的运行实体，Python 程序员可以通过它派生自定义的测试过程与方法（测试单元），利用 Command 和 Composite 设计模式，多个 TestCase 还可以组合成测试用例集合。

编写测试用例

采用 PyUnit 提供的动态方法，只编写一个测试类来完成对整个软件模块的测试，这样对象的初始化工作可以在 setUp() 方法中完成，而资源的释放则可以在 tearDown() 方法中完成。

对的 widget.py 被测试类的多方法进行测试。

```
# 执行测试的类
class WidgetTestCase(unittest.TestCase):

    def setUp(self):
        self.widget = Widget()

    # 测试 getSize() 方法的测试用例
    def testSize(self):
```

```

self.assertEqual(self.widget.getSize(), (40, 40))

# 测试 resize()方法的测试用例
def testResize(self):
    self.widget.resize(100, 100)
    self.assertEqual(self.widget.getSize(), (100, 100))

def tearDown(self):
    self.widget.dispose()
    self.widget = None

```

我们可以在一个测试类中，写多个测试用例对被测试类的方法进行测试。

组织用例集

完整的单元测试很少只执行一个测试用例，开发人员通常都需要编写多个测试用例才能对某一软件功能进行比较完整的测试，这些相关的测试用例称为一个测试用例集，在 PyUnit 中是用 TestSuite 类来表示的。

可以在单元测试代码中定义一个名为 suite() 的全局函数，并将其作为整个单元测试的入口，PyUnit 通过调用它来完成整个测试过程。

```

def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testSize"))
    suite.addTest(WidgetTestCase("testResize"))
    return suite

```

如果用于测试的类中所有的测试方法都以 test 开头，Python 程序员甚至可以用 PyUnit 模块提供的 makeSuite() 方法来构造一个。

```

def suite():
    return unittest.makeSuite(WidgetTestCase, "test")

```

TestSuite 类可以看成是 TestCase 类的一个容器，用来对多个测试用例进行组织，这样多个测试用例可以自动在一次测试中全部完成。



运行测试集

PyUnit 使用 TestRunner 类作为测试用例的基本执行环境，来驱动整个单元测试过程。Python 开发人员

在进行单元测试时一般不直接使用 TestRunner 类，而是使用其子类 TextTestRunner 来完成测试，并将测试结果以文本方式显示出来：

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

对 widget.py 被测试类，下面通过 PyUnit 编写完整的单元测试用例：

text_runner.py

```
#coding=utf-8
from widget import Widget
import unittest
# 执行测试的类
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget()

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testSize(self):
        self.assertEqual(self.widget.getSize(), (40, 40))

    def testResize(self):
        self.widget.resize(100, 100)
        self.assertEqual(self.widget.getSize(), (100, 100))

# 测试
if __name__ == "__main__":
    # 构造测试集
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase("testSize"))
    suite.addTest(WidgetTestCase("testResize"))

    # 执行测试
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

PyUnit 模块中定义了一个名为 main 的全局方法，使用它可以很方便地将一个单元测试模块变成可以直接运行的测试脚本，main() 方法使用 TestLoader 类来搜索所有包含在该模块中的测试方法，并自动执行它们。如果 Python 程序员能够按照约定（以 test 开头）来命名所有的测试方法，那就只需要在测试模块的最后加入如下几行代码即可：

```
#coding=utf-8
```

```

from widget import Widget
import unittest
# 执行测试的类
class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget()

    def tearDown(self):
        self.widget.dispose()
        self.widget = None

    def testSize(self):
        self.assertEqual(self.widget.getSize(), (40, 40))

    def testResize(self):
        self.widget.resize(100, 100)
        self.assertEqual(self.widget.getSize(), (100, 100))

# 测试
if __name__ == "__main__":
    unittest.main()

```

python 基础知识补充:

`if __name__ == “__main__”:` 语句说明

后面我们会经常用到这个语句，在解释之前先补充点 python 知识：

1、python 文件的后缀为.py；

2、.py 文件既可以用来直接执行，就像一个小程序一样，也可以用来作为模块被导入

3、在 python 中导入模块一般使用的是 import

顾名思义，if 就是如果的意思，在句子开始处加上 if，就说明，这个句子是一个条件语句。接着是`__name__`，`__name__`作为模块的内置属性，简单点说呢，就是.py 文件的调用方式。最后是`__main__`，刚才我也提过，.py 文件有两种使用方式：作为模块被调用和直接使用。如果它等于"`__main__`"就表示是直接执行。

我们在实际的自动化测试用例开发过程中，首先要保证开发的单个用例文件 (.py) 是运行通过的，如何跑单个文件上的用例，那么就可以在 `if __name__ == “__main__”:` 后面编写执行用的语句，如上面介绍的 `TextTestRunner()`方法来构造测试集，或直接使用 `unittest.main()` 来运行所有用例。

那么一旦这个用例文件 (.py) 稳定之后，就需要将这个用例文件添加到用例集中，这个用例文件就被做一个模块被调用；这个时候 `if __name__ == “__main__”:` 后面的内容将不会被执行。

第四节、批量执行测试用例

通过对前面对 `unittest` 框架的学习我们了解到，可以在一个`.py` 文件里编写多个测试用例，然后执行文件里的所有用例，这显然是一个不错的做法，我们可以将一些相关的用例放到一个文件里，`unittest` 支持这么做，但假如我们成百上千的用例呢，放一个`.py` 文件显然有些不太合理。

比较合理的做法是把相关的几条用例放到一个`.py` 文件里，把所有`.py` 文件放到一个文件夹下，然后通过一个程序执行文件夹下面的所有用例。



图 5.x

如图 5.x 显然是一个比较理想的效果，下面来组织这样一个结构。

编写 `baidu.py` 文件放入 `test_case` 文件夹下：

```
#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re

class Baidu(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"
        selfverificationErrors = []
        self.accept_next_alert = True

    #百度搜索用例
    def test_baidu_search(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").send_keys("selenium webdriver")
```

```

driver.find_element_by_id("su").click()
time.sleep(2)
driver.close()

#百度设置用例
def test_baidu_set(self):
    driver = self.driver

    #进入搜索设置页
    driver.get(self.base_url + "/gaoji/preferences.html")

    #设置每页搜索结果为 100 条
    m=driver.find_element_by_name("NR")
    m.find_element_by_xpath("//option[@value='100']").click()
    time.sleep(2)

    #保存设置的信息
    driver.find_element_by_xpath("//input[@value='保存设置']").click()
    time.sleep(2)
    driver.switch_to_alert().accept()

def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
    unittest.main()

```

下面编写 test_case.py 点文件来读取 test_case 文件夹下的文件：

```

-*-coding=utf-8 -*-
import os
#列出某个文件夹下的所有 case,这里用的是 python,
#所在 py 文件运行一次后会生成一个 pyc 的副本
caselist=os.listdir('D:\\selenium_use_case\\test_case')
for a in caselist:
    s=a.split('.')[1] #选取后缀名为 py 的文件
    if s=='py':
        #此处执行 dos 命令并将结果保存到 log.txt
        os.system('D:\\selenium_use_case\\test_case\\%s 1>>log.txt 2>&1%a')

```

python 知识补充：

python 的 os 模块可以用来操作本地文件，通过 os.listdir() 函数获得指定目录中的内容；下面通过小例子单独理解这段程序的匹配用法。打开 python IDLE 的交互模式输入以下代码。

```
>>> x = 'testing.py'
>>> s = x.split('.')[1]
>>> if s=='py':
    print s

py
```

split()用于字符串分割，本例中以文件名的点(.)作为分割。被分割之的字符串‘testing.py’会变成[‘testing’, ‘py’]，[n]表示数组，因为数组是从0开始计算的，所以[0]取的是‘testing’，那么[1]取的就是‘py’。

把取到的结果赋值给s，if判断s等于(==)‘py’则执行后面的语句。

os.system('D:\\selenium_use_case\\test_case\\%s 1>>log.txt 2>&1' %a)语句根据上面的判断条件选取D:\\selenium_use_case\\test_case\\目录下的文件执行，并将执行结果保存到log.txt文件中。

查看log.txt文件：

```
..
-----
Ran 2 tests in 32.469s

OK
```

到目前为止，一个极其简陋的自动化测试环境完成，我们要做的就是编写测试用例放入test_case文件夹下，通过test_case.py程序执行所以测试用例。然后通过log.txt查看脚本的执行情况。

如果脚本运行失败，log.txt将显示出错信息。如下：

```
=====
ERROR: test_baidu_search (__main__.Baidu)
-----
Traceback (most recent call last):
  File "D:\\selenium_python\\test_case\\baidu.py", line 22, in test_baidu_search
    driver.find_element_by_id("kwss").send_keys("selenium webdriver")
  File "C:\\Python27\\lib\\site-packages\\selenium\\webdriver\\remote\\webdriver.py", line 198, in
find_element_by_id
    return self.find_element(by=By.ID, value=id_)
  File "C:\\Python27\\lib\\site-packages\\selenium\\webdriver\\remote\\webdriver.py", line 680, in
find_element
    {'using': by, 'value': value})['value']
  File "C:\\Python27\\lib\\site-packages\\selenium\\webdriver\\remote\\webdriver.py", line 165, in
execute
    self.error_handler.check_response(response)
  File "C:\\Python27\\lib\\site-packages\\selenium\\webdriver\\remote\\errorhandler.py", line 158,
in check_response
```

```
raise exception_class(message, screen, stacktrace)
NoSuchElementException:      Message:      u'Unable      to      locate      element:
{"method":"id","selector":"kwss"}' ; Stacktrace:
at                                     FirefoxDriver.findElementInternal_
(file:///c:/docume^1/admini^1/locals^1/temp/tmptamufm/extensions/fxdriver@googlecode.com/co
mponents/driver_component.js:8444)
at                                     fxdriver.Timer.setTimeout/.notify
(file:///c:/docume^1/admini^1/locals^1/temp/tmptamufm/extensions/fxdriver@googlecode.com/co
mponents/driver_component.js:386)

-----
Ran 2 tests in 44.360s

FAILED (errors=1)
```

通过上面的错误信息，我们可以比较清晰的判断脚本执行错误的发生为止。如上面的错误信息，我们修改百度输入框的定位 `find_element_by_id("kwss")`，导致脚本无法定位到输入框。

由于引入 `unittest` 框架的作用，一条用例失败后不会影响下一条用例的执行。因此，从运行结果我们可以看到，跑了两条用例，失败为(`errors=1`)。

总结:

通过本章的学习，我们掌握了 `selenium IDE` 安装与使用，以及如何将 `selenium IDE` 录制的脚本导出为 `python unittest` 单元测试框架相应的脚本；初步学习了解了 `unittest` 单元测试框架的使用。如何 `python` 语言批量的执行文件夹的用例文件。

第七章 引入测试报告与结构优化

我想在开始本章的学习之前先来回顾一下目前的测试结构，因为本章节会对结构做一个改进，以帮助我们更好的实施自动化测试。

/selenium_python/test_case/baidu.py	-----测试用例
/test_case/webcloud.py	-----测试用例
/test_case/login.py	-----登录模块
/test_case/quit.py	-----退出模块
/data/userinfo.csv	-----用户数据参数化文件
/test_case.py	----执行所有用例
/log.txt	----用例执行结果文件

本章的重点就是抛弃 log.txt 这种丑陋日志文件，引入漂亮的 HTMLTestRunner 测试报告，并且对目前的结构做一些调整，其实能适合 HTMLTestRunner 报告的生成，另一方面使用例更容易编写和扩展。

第一节、生成 HTMLTestRunner 测试报告

在脚本运行完成之后，除了在 log.txt 文件看到运行日志外，我们更希望能生一张漂亮的测试报告来展示用例执行的结果。

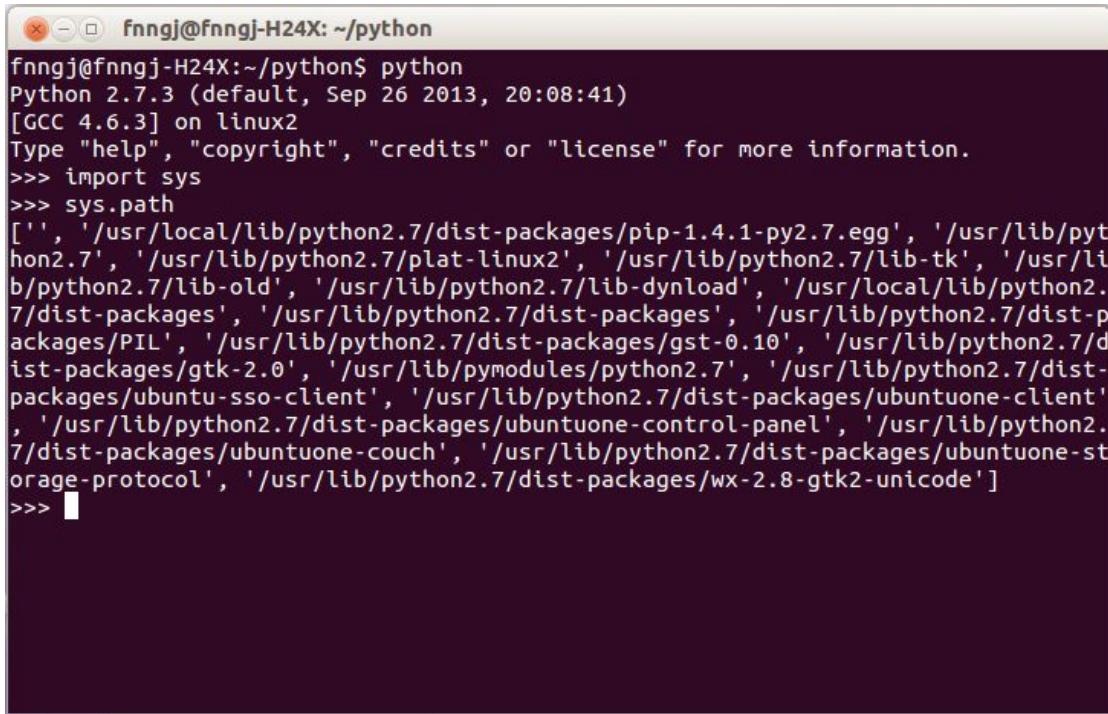
HTMLTestRunner 是 Python 标准库的 unittest 模块的一个扩展。它生成易于使用的 HTML 测试报告。HTMLTestRunner 是在 BSD 许可证下发布。

首先要下 HTMLTestRunner.py 文件，下载地址：

<http://tungwaiyip.info/software/HTMLTestRunner.html>

Windows：将下载的文件放入...\\Python27\\Lib 目录下。

Linux (ubuntu)：下面需要先确定 python 的安装目录，打开终端，输入 python 命令进入 python 交互模式，通过 sys.path 可以查看本机 python 文件目录，如图 7.1



```
fnngj@fnngj-H24X:~/python$ python
Python 2.7.3 (default, Sep 26 2013, 20:08:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/pip-1.4.1-py2.7.egg', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-linux2', '/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload', '/usr/local/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages/PIL', '/usr/lib/python2.7/dist-packages/gst-0.10', '/usr/lib/python2.7/dist-packages/gtk-2.0', '/usr/lib/pymodules/python2.7', '/usr/lib/python2.7/dist-packages/ubuntu-sso-client', '/usr/lib/python2.7/dist-packages/ubuntuone-client', '/usr/lib/python2.7/dist-packages/ubuntuone-control-panel', '/usr/lib/python2.7/dist-packages/ubuntuone-couch', '/usr/lib/python2.7/dist-packages/ubuntuone-storage-protocol', '/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode']
```

图 7.1

以管理员身份将 HTMLTestRunner.py 文件考本到 /usr/lib/python2.7/dist-packages/ 目录下：

root@user-H24X:/home/user/python# cp HTMLTestRunner.py /usr/lib/python2.7/dist-packages/

(提示，切换到 root 用户切换才能拷贝文件到系统目录)

在 python 交互模式引入 HTMLTestRunner 包，如果没有报错，则说明添加成功。

```
>>> import HTMLTestRunner
>>>
```

下面继续以 baidu.py 文件为例生成 HTMLTestRunner 测试报告：

```
#coding=utf-8

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
```

```
import unittest, time, re
import HTMLTestRunner #引入 HTMLTestRunner 包

class Baidu(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.baidu.com/"
        selfverificationErrors = []
        self.accept_next_alert = True

    #百度搜索用例
    def test_baidu_search(self):
        driver = self.driver
        driver.get(self.base_url + "/")
        driver.find_element_by_id("kw").send_keys("selenium webdriver")
        driver.find_element_by_id("su").click()
        time.sleep(2)
        driver.close()

    #百度设置用例
    def test_baidu_set(self):
        driver = self.driver

        #进入搜索设置页
        driver.get(self.base_url + "/gaoji/preferences.html")

        #设置每页搜索结果为 100 条
        m=driver.find_element_by_name("NR")
        m.find_element_by_xpath("//option[@value='100']").click()
        time.sleep(2)

        #保存设置的信息
```

```

driver.find_element_by_xpath("/html/body/form/div/input").click()
time.sleep(2)
driver.switch_to_alert().accept()

def tearDown(self):
    self.driver.quit()
    self.assertEqual([], selfverificationErrors)

if __name__ == "__main__":
    #定义一个单元测试容器
    testunit=unittest.TestSuite()

    #将测试用例加入到测试容器中
    testunit.addTest(Baidu("test_baidu_search"))
    testunit.addTest(Baidu("test_baidu_set"))

    #定义个报告存放路径，支持相对路径
    filename = 'D:\\selenium_python\\report\\result.html'
    fp = file(filename, 'wb')
    #定义测试报告
    runner =HTMLTestRunner.HTMLTestRunner(
        stream=fp,
        title=u'百度搜索测试报告',
        description=u'用例执行情况：')

    #运行测试用例
    runner.run(testunit)

```

代码分析:

```
import HTMLTestRunner
```

要使用 HTMLTestRunner 首先要导入模块

```
TestSuite()
```

在第六章 unittest 单元测试框架解析中有过讲解, TestSuite() 可以被看作一个容器(测试套件), 通过 addTest 方法我们可罗列具体所要执行的测试用例。当然了, 如果使用 unittest.main() 的话默认会执行所有以 test 开头的测试用例。

```
filename = 'D:\\selenium_python\\report\\result.html'
fp = file(filename, 'wb')
```

定位报告文件的存放路径。

```
runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
    description=u'用例执行情况: ')
```

定义 HTMLTestRunner 测试报告, stream 定义报告所写入的文件; title 为报告的标题; description 为报告的说明与描述。

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

运行测试容器中的用例, 并将结果写入的报告中。

脚本运行结束, 生成测试报告, 如图 7.2

百度搜索测试报告

Start Time: 2013-11-27 11:10:14
Duration: 0:00:18.375000
Status: Pass 2

用例执行情况:

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

Test Group/Test case	Count	Pass	Fail	Error	View
Baidu	2	2	0	0	Detail
test_baidu_search			pass		
test_baidu_set			pass		
Total	2	2	0	0	

图 7.2

问题思考:

这个报告是根据一个.py 文件生成的, 这样就迫使我们把所有用例都写在一个.py 文件里, 如果我们每一个用例都写在不同的.py 文件里将生成很多个报告, 这样做不利于阅读; 但写在一个.py 文件里, 如果用例非常多的话, 同样不利于维护。下面将进一步学习 TestSuite 的使用以帮助我们解决这个问题。

第二节、测试套件

现在我们来进一步学习 TestSuite (测试套件) 的使用，从来帮助我们解决上一节中的遗留问题。我们之前使用 TestSuite 只是在一个.py 文件里添加多个测试用例，那么可不可以法把多个.py 文件中的用例通过测试套件来组织？答案是肯定。这一节就来解决这个问题，我们最终要达到的目的如图 7.3。

通过测试套件组织的用例结构：

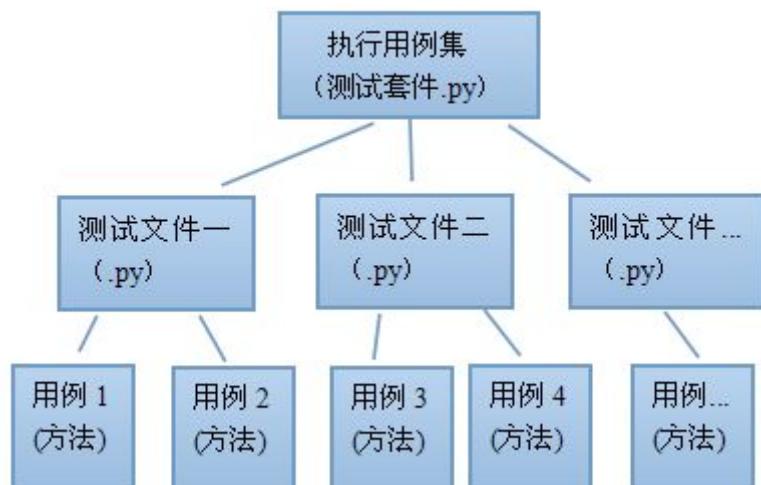


图 7.3

测试套件的问题解决了，所有用例生成一张测试报告的问题自然也可以解决了。

7.2.1、测试套件实例

下面通过一个完整例子来说明如何通过套件组建我们的测试用例。

baidu.py

```

#coding=utf-8

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re

class Baidu(unittest.TestCase):
  
```

```

def setUp(self):
    self.driver = webdriver.Firefox()
    self.driver.implicitly_wait(30)
    self.base_url = "http://www.baidu.com/"
    selfverificationErrors = []
    self.accept_next_alert = True

#百度搜索用例
def test_baidu_search(self):
    driver = self.driver
    driver.get(self.base_url + "/")
    driver.find_element_by_id("kw").send_keys("selenium webdriver")
    driver.find_element_by_id("su").click()
    time.sleep(2)
    driver.close()

#百度设置用例
def test_baidu_set(self):
    driver = self.driver

    #进入搜索设置页
    driver.get(self.base_url + "/gaoji/preferences.html")

    #设置每页搜索结果为 100 条
    m=driver.find_element_by_name("NR")
    m.find_element_by_xpath("//option[@value='100']").click()
    time.sleep(2)

    #保存设置的信息
    driver.find_element_by_xpath("//html/body/form/div/input").click()
    time.sleep(2)
    driver.switch_to_alert().accept()

def tearDown(self):

```

```

        self.driver.quit()

        self.assertEqual([], self.verifyErrors)

if __name__ == "__main__":
    unittest.main()

```

youdao.py

```

#coding=utf-8

from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re

class Youdao(unittest.TestCase):

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://www.youdao.com"
        self.verifyErrors = []
        self.accept_next_alert = True

    #有道搜索用例
    def test_youdao_search(self):
        driver = self.driver
        driver.get(self.base_url + "/")

        driver.find_element_by_id("query").send_keys(u"虫师")
        driver.find_element_by_id("qb").click()
        time.sleep(2)

    def tearDown(self):

```

```

        self.driver.quit()

        self.assertEqual([], self.verifyErrors)

if __name__ == "__main__":
    unittest.main()

```

上面的两个测试用例文件分析，baidu.py 文件编写了两种用例，youdao.py 文件中编写一条用例，分别运行两个用例文件都可以正常的进行测试。

下面我们创建 all_tests.py 文件，通过套件的方式来执行这两个文件里的三条用例(注意，all_tests.py 文件要与 baidu.py 和 youdao.py 在同一目录下)：

```

#coding=utf-8

import unittest

#这里需要导入测试文件

import baidu,youdao


testunit=unittest.TestSuite()

#将测试用例加入到测试容器(套件)中

testunit.addTest(unittest.makeSuite(baidu.Baidu))
testunit.addTest(unittest.makeSuite(youdao.Youdao))

#执行测试套件

runner = unittest.TextTestRunner()

runner.run(testunit)

```

运行结果：

```

>>> ===== RESTART =====
>>>
...
Time Elapsed: 0:00:23.922000

```

这里与前面稍有不同的是用到了 unittest 的 `makeSuite`：

`makeSuite` 用于生产 `testsuite` 对象的实例，把所有的测试用例组装成 `TestSuite`，最后把 `TestSuite` 传给 `TestRunner` 进行执行。

baidu.Baidu

baidu 为导入的.py 文件，Baidu 为类名，调用类名，默认类下面的所有以 test 开头的方法（测试用例）会被执行。

7.2.2、整合 HTMLTestRunner 测试报告

通过测试套件解决执行多个测试文件的问题，那么生成多个测试文件的 HTMLTestRunner 报告的问题自然也解决了。打开 all_tests.py 文件，将生成 HTMTestRunner 报告的相关代码添加进来：

```
#coding=utf-8

import unittest

#这里需要导入测试文件
import baidu,youdao
import HTMLTestRunner

testunit=unittest.TestSuite()

#将测试用例加入到测试容器(套件)中
testunit.addTest(unittest.makeSuite(baidu.Baidu))
testunit.addTest(unittest.makeSuite(youdao.Youdao))

#执行测试套件
#runner = unittest.TextTestRunner()
#runner.run(testunit)

#定义个报告存放路径，支持相对路径。
filename = 'D:\\selenium_python\\report\\result2.html'
fp = file(filename, 'wb')

runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
```

```
description=u'用例执行情况: ')
#执行测试用例
runner.run(testunit)
```

本例中并没有引入新方法，希望读者能将代码敲写一遍，认真体会代码的含义。

下面查看 [D:\selenium_python\report](#) 目录下所生成有报告：

百度搜索测试报告

Start Time: 2013-11-27 15:29:51
Duration: 0:00:24.235000
Status: Pass 3

用例执行情况：

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

Test Group/Test case	Count	Pass	Fail	Error	View
baidu.Baidu	2	2	0	0	Detail
test_baidu_search			pass		
test_baidu_set			pass		
youdao.Youdao	1	1	0	0	Detail
test_youdao_search			pass		
Total	3	3	0	0	

图 7.4

多个测试用例 (.py) 文件被整合到一个报告中的，这样在编写新的测试用例后，在 all_tests.py 文件中添加到测试套件中就可以了。

7.2.3、易读的测试报告

虽然在我们在测试用例开发时为每个用例添加了注释，但测试报告一般是给非测试人员阅读的，如果能在报告中为每一个测试用例添加说明，那么将会使报告更加易于阅读。

打开我们的测试用例文件，为每一个测试用例（方法）下面添加注释，如下：

```
.....
#百度搜索用例
def test_baidu_search(self):
    """百度搜索"""
    driver = self.driver
    driver.get(self.base_url + "/")
.....
```

再次打开生成的测试报告：

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

Test Group/Test case	Count
baidu.Baidu	2
test_baidu_search: 百度搜索	
test_baidu_set: 百度设置	
youdao.Youdao	1
test_youdao_search: 有道搜索	
Total	3

图 7.5

小 u 是避免中文引起的乱码问题。有读者可能会对三引号的作用比好奇。

这里再来看看一下引号的使用：

单双引号的使用：

```
#输出字符串
>>> print 'abc'
abc
>>> print "abc"
abc

输出带引号的字符串
>>> print "ab'c'de"
ab'c'de
>>> print 'ab"c"d'
ab"c"d
>>> print "ab"c"d"
SyntaxError: invalid syntax
```

在 python 中是不区分单双引号的，但必须要成对出现。而且引号之间左侧开始找最近一个引号默认和自己是一对，如： "ab"c"d" ，被拆分为"ab"、 c、 "d" ，因为 c 不是字符串，所以，就会提示语法错误。

三引号的使用：

```
>>> print """
我可以写
任意多行
的注释
"""
```

我可以写
任意多行
的注释

```
>>> a= '''
```

任意多行的
注释

```
'''
```

```
>>> print a
```

任意多行的
注释

三引号用于表示多行注释，同样三引号也不分单双，而且我们可以把一个多行注释做为一个字符串赋值给一个变量。

再来看看函数下面的三引号使用：

```
>>> def fun(a=3,b=5):
    """加法运算的函数"""
    c= a+b
    print c

>>> fun()
8
>>> help(fun)

Help on function fun in module __main__:

fun(a=3, b=5)
    加法运算的函数
```

定一个 fun() 函数并在函数下面加注释，调用函数时并没有发现注释信息，但是使用 help() 查看 fun 函数，然后注释信息就显示出来了。我们编写的类下面的测试用例方法也是一样的原理。

7.2.3、报告文件名取当前时间

每次运行测试之前都要手动的去修改报告的名称，如果有修改就会把之前的报告覆盖，这样做就会显示得很麻烦，那么有没有办法使每次生成的报告名称都不一样，为了更好的区分报告可以在报告中添加当前的时间，这样我们要想查找某天某时所生成的报告就会变得非常容易。

对于 time 模块来说我们并不陌生，一直在使用 sleep() 给测试用例程序添加休眠时间，那么有没有取当前时间的方法？下面打开 python IDLE 学习 time 下面的几个新方法。

```
>>> time.time()
1385701893.75

>>> time.ctime()
'Fri Nov 29 13:14:41 2013'

>>> time.localtime()
time.struct_time(tm_year=2013, tm_mon=11, tm_mday=29, tm_hour=13, tm_min=11,
tm_sec=22, tm_wday=4, tm_yday=333, tm_isdst=0)

>>> time.strftime("%Y%m%d%H%M%S", time.localtime())
'20131129131356'

>>> time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
'2013-11-29 13:14:06'
```

[time.time\(\)](#) 获取当前时间戳

[time.localtime\(\)](#) 当前时间的 struct_time 形式

[time.ctime\(\)](#) 当前时间的字符串形式

[time.strftime\("%Y-%m-%d %H:%M:%S", time.localtime\(\)\)](#)

如：2013-11-29 13:14:06

我们已经知道了 time 下面的这个方面的可以取到前面实践，我们要做的就是把取到的当前时间加到测试报告的文件名中。

打开 all_test.py 做如下修改：

```

.....
#取前面时间

now = time.strftime ("%Y-%m-%M-%H_%M_%S",time.localtime(time.time()))

#把当前时间加到报告中

filename = "D:\\selenium_python\\report\\\"+now+'result.html'
fp = file(filename, 'wb')

runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
description=u'用例执行情况: ')
.....

```

重新运行所有测试用例，查看生成的测试报告的文件名，如图 6.x 中最上面文件，文件名以前面运行实践命名，所以很容易地找到想要的报告，而且在每次运行测试前不用修改手动修改报告的名字。

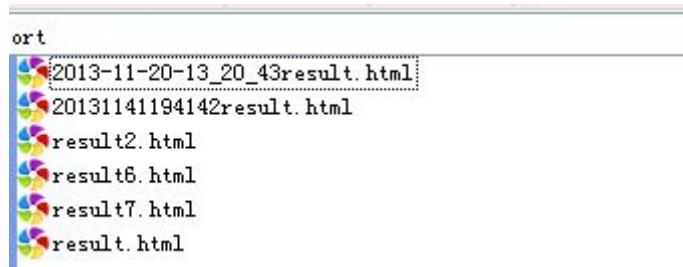


图 7.6

第三节、结构改进

到目前为止，我们已经成功的解决了通过测试套件执行多个测试用例文件的问题，并且将输出结果整合在一个测试报告里，目的达到了。但目前的测试结构并不合理，我们需要进一步对结构进行调整，以便于测试的维护是管理。

7.3.1、all_tests.py 文件移出来

我们首先要调整的是 all_tests.py 文件的位置， all_tests.py 是执行所有用例的程序，而并非测试用例本

身，所以应该把它移出来。结构上会更为合理。

/selenium_python/test_case/baidu.py	-----测试用例
/test_case/youdao.py	-----测试用例
/test_case/webcloud.py	-----私有云用例
/test_case/login.py	-----登录模块
/test_case/quit.py	-----退出模块
/test_case/__init__.py	
/data/userinfo.csv	-----用户数据参数化文件
/all_case.py	----执行所有用例

目录结构应该是这样的， test_case 目录下存放具体的执行用例， all_tests.py 应该从 test_case 目录分离出来。那么直接把 all_tests.py 文件移出来后运行一下：

```
>>> ===== RESTART =====
>>>

Traceback (most recent call last):
  File "D:\selenium_python\all_tests.py", line 5, in <module>
    import baidu,youdao
ImportError: No module named baidu
```

提示我们找不到 baidu 和 youdao 两个文件；修改 all_tests.py 文件，另外，需要在 test_case 目录下创建一个 __init__.py 文件，文件内容可以为空。将 D:\selenium_python\test_case 目录添加到 sys.path 下：

```
#coding=utf-8

import unittest

#把 test_case 目录添加到 path 下，这里用的相对路径
import sys
sys.path.append("\test_case")

from test_case import youdao
from test_case import baidu

#这里需要导入测试文件
import baidu,youdao
....
```

现在可以正常运行 all_tests.py 文件了， __init__.py 文件的作用是什么？为什么需要将测试目录添加到系统的 path 才能调用？下一小节解析 __init__.py 文件的作用。

7.3.2、`__init__.py`文件解析

关于`__init__.py`文件的作用，老王python的博客中清晰的说明，这里引用截取博客中的部分作内容，帮助理解。

还记得上面提到的`__init__.py`文件吧，这文件是干嘛的，为什么要在引用的目录下加这个文件？

要弄明白这个问题，首先要知道，python在执行`import`语句时，到底进行了什么操作，按照python的文档，它执行了如下操作：

第1步，创建一个新的，空的`module`对象（它可能包含多个`module`）；

第2步，把这个`module`对象插入`sys.module`中

第3步，装载`module`的代码（如果需要，首先必须编译）

第4步，执行新的`module`中对应的代码。

在执行第3步时，首先要找到`module`程序所在的位置，搜索的顺序是：

当前路径（以及从当前目录指定的`sys.path`），然后是`PYTHONPATH`，然后是python的安装设置相关的默认路径。正因为存在这样的顺序，如果当前路径或`PYTHONPATH`中存在与标准`module`同样的`module`，则会覆盖标准`module`。也就是说，如果当前目录下存在`xml.py`，那么执行`import xml`时，导入的是当前目录下的`module`，而不是系统标准的`xml`。

了解了这些，我们就可以先构建一个`package`，以普通`module`的方式导入，就可以直接访问此`package`中的各个`module`了。python中的`package`必须包含一个`__init__.py`的文件。

-----以上引用“老王python”

理解了上面这段话也就明白了为什么，`all_tests.py`和测试用例（.py）文件在一个目录下时可以正常调用，移出来之后就提示找不到模块了；python查找模块是先从当前目录下查找，如果找不到再到python的安装设置的相关路径下查找。这也是为什么我们把`D:\selenium_python\test_cast`目录添加到系统path下，就可以正常的调用了。为了标识一下目录是可引用的包，那么就需要在目录下创建一个`__init__.py`文件。

其实`__init__.py`文件中可以有内容；我们在导入一个包时，实际上导入了它的`__init__.py`文件。在`__init__.py`文件中添加导入包。

```
import baidu
import youdao
```

然后，all_tests.py 文件可是这样修改：

```
#coding=utf-8
import unittest
import sys
sys.path.append("\test_case")
from test_case import *
```

“*” 星号表示导入 test_case 目录下的所有文件；在 test_case 目录下创建新的测试用例文件，只用在__init__.py 文件下添加就可以了。而对于 all_tests.py 文件来说不需要做任何修改。

7.3.3、把公共模块文件移进去

理解 python 调用包的机制，那么就可以随意的调整我们目录结构，使其更便于管理。在第五章中，我们 webcloud.py 测试用例的登录和退出进行了模块化，分别创建了 login.py 和 quit_login.py 两个文件，他们属于公共模块，并非完整的用例本身，所以我们可以单独创建一个文件夹，将他们移进去。

在 test_case 目录下创建 public 目录，把 login.py 和 quit_login.py 文件移进去。同样需要在 public 目录下创建空的__init__.py 文件；同时修改 test_case 目录下的 webcloud.py 文件：

```
#把 public 目录添加到 path 下，这里用的相对路径
import sys
sys.path.append("\public")
#导入登录、退出模块
from public import login
from public import quit_login
.....
#私有云登录用例
def test_login(self):
    driver = self.driver
    driver.get(self.base_url
               +
               "/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")
#调用登录模块
login.login(self)
```

```
#调用退出模块
quit_login.quit_(self)

....
```

关于模块的创建与调用请参考本书第四章内容。

第四节、用例的读取

7.3.1、改进用例的读取

打开 all_tests.py 文件，虽然导入包的部分我们用“from test_case import *”方便的替换具体导入每个文件的做法，但在测试套件部分，我们会发现每创建一条用例（.py 文件）都需要在测试套件中添加，随着用例的增加，测试套件可能要罗列几百上千条用例，非常不便于管理。

```
....
testunit=unittest.TestSuite()
#将测试用例加入到测试容器(套件)中
testunit.addTest(unittest.makeSuite(baidu.Baidu))
testunit.addTest(unittest.makeSuite(youdao.Youdao))
testunit.addTest(unittest.makeSuite(webcloud.Login))
....
```

那么我们可不可以通过一个 for 循环来解决这个问题，首先需要把用例文件组装一数组，通过循环读取的方法来读取测试套件中的每一条用例；

先来熟悉一下 python 中数组的使用，继续在 python 交互模式下练习。

```
>>> array = [1,2,3]
>>> print array
[1, 2, 3]
>>> array2=['a','b','c']
>>> print array2
['a', 'b', 'c']
>>> array3=['abc',123,"中国人"]
```

```
>>> for data in array3:
    print data

abc
123
中国人
```

数组中括号表示，里面元素之间用逗号分割；可以数字或字符串，我们通过 for 语句可以循环读取数组的元素。

下面和我一起动手来实现这个功能吧！修改后 all_tests.py 文件如下：

```
#coding=utf-8

import unittest
#把 test_case 目录添加到 path 下，这里用的相对路径
import sys
sys.path.append("\test_case")

#导入 test_case 目录下的所有文件
from test_case import *
import HTMLTestRunner

#将用例组装数组
alltestnames = [
    baidu.Baidu,
    youdao.Youdao,
    webcloud.Login,
]

#创建测试套件
testunit=unittest.TestSuite()

#循环读取数组中的用例
for test in alltestnames:
    testunit.addTest(unittest.makeSuite(test))

#定义个报告存放路径，支持相对路径
filename = 'D:\\selenium_python\\report\\result.html'
```

```

fp = file(filename, 'wb')

runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
    description=u'用例执行情况：')

#执行测试用例
runner.run(testunit)

```

为了使 all_tests.py 文件在用例增加或删除时不需要做任何修改，我们可把 alltestnames 数组放到一个单独的文件中，创建 allcase_list.py 文件，与 all_tests.py 保持同级目录：

```

#coding=utf-8
#把 test_case 目录添加到 path 下，这里用的相对路径
import sys
sys.path.append("\test_case")
from test_case import *

#用例文件列表
def caselist():
    alltestnames = [
        baidu.Baidu,
        youdao.Youdao,
        webcloud.Login,
    ]
    print "success  read case list!!"

    return alltestnames

```

因为在 allcase_list.py 文件调用用例，所以，我们也需要把导入用例文件的相关操作移动过来。这样 all_tests.py 文件将会显得更加清爽，下面在 all_tests.py 文件中调用 caselist 函数：

```

.....
import allcase_list #调用数组文件

#获取数组方法
alltestnames = allcase_list.caselist()
.....

```

再次运行 all_tests.py 文件。发现可以正常的实现用例的读取。

7.3.2、discover 解决用例的读取

回顾一下，到目前位置，我们的结构复杂了很多，处理问题的过程也变得复杂的很多，假如我们创建了一条用例 aaa.py，需要在用例当前目录下打开 __init__.py 文件，添加 “import aaa”；还需要打开 allcase_list.py 文件，在 alltestnames 数组中添加 aaa.calss_name。然后，用例才能添加到测试套件中执行。

在第三章中我们通过编写一个 python 小程序可以轻松地解决用例文件的读取，这所以会变得复杂，一方面是我们模块化一部分代码，最主要的原因是引入了 HTMLTestRunner 测试报告，接着为了整合报告不得不利用测试套件来读取用例。

那么 unittest 有没有提供一种简单的方法，可以通过文件的名称来判断是否为测试用例文件，如何为用例文件则自动添加到测试套件中。有的，通过查询文档发现 unittest 的 TestLoader 成员下面提供了 discover() 方法可解决这个问题。

TestLoader：测试用例加载器，其包括多个加载测试用例的方法。返回一个测试套件。

```
discover(start_dir, pattern='test*.py', top_level_dir=None)
```

找到指定目录下所有测试模块，并可递归查到子目录下的测试模块，只有匹配到文件名才能被加载。如果启动的不是顶层目录，那么顶层目录必须要单独指定。

start_dir：要测试的模块名或测试用例目录。

pattern='test*.py'：表示用例文件名的匹配原则。星号“*”表示任意多个字符。

这里需要说明一个测试用例的创建规则：我们在实际的测试用开发中用例的创建也应该分两个阶段，用例刚在目录下被创建，可命名为 aa.py，当用例创建完成并且运行稳定后再添加到测试套件中。那么可以将 aa.py 重新命名为 start_aa.py，那么测试套件在运行时只识别并运行 start 开头的.py 文件。

理解了这个规则，我们就可以按照一个约定来命名自己用例文件名。这里我们将 baidu.py 、youdao.py 和 webcoud.py 三个文件重命名为以 start_开头。

top_level_dir=None：测试模块的顶层目录。如果没顶层目录（也就是说测试用例不是放在多级目录中），默认为 None。

理解了 discover() 方法的结构，下面看如何应用到实际项目中，打开 all_tests.py 文件：

```
#coding=utf-8
import unittest
import HTMLTestRunner
import os ,time

listaa='D:\\selenium_python\\test_case'
def creatsuite():
    testunit=unittest.TestSuite()
```

```
#discover 方法定义
discover=unittest.defaultTestLoader.discover(listaa,
                                              pattern ='start_*.py',
                                              top_level_dir=None)

#discover 方法筛选出来的用例，循环添加到测试套件中
for test_suite in discover:
    for test_case in test_suite:
        testunit.addTests(test_case)
        print testunit
return testunit

alltestnames = createsuite()

now = time.strftime('%Y-%m-%M-%H_%M_%S',time.localtime(time.time()))
filename = 'D:\\selenium_python\\report\\'+now+'result.html'
fp = file(filename, 'wb')

runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
    description=u'用例执行情况：')

#执行测试用例
runner.run(alltestnames)
```

查看生成的 HTML 测试报告：

百度搜索测试报告

Start Time: 2013-11-28 11:06:50

Duration: 0:00:54.735000

Status: Pass 4

用例执行情况:

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

Test Group/Test case	Count
test_case.baidu.Baidu	2
test_baidu_search: 百度搜索	
test_baidu_set: 百度设置	
test_case.youdao.Youdao	1
test_youdao_search: 有道搜索	
test_case.webcloud.Login	1
test_login: 私有云登录	
Total	4

图7.7

如图7.7 虽然生成的报告中，最后一条用例显得和前面的用例不太协调，但是我们的所面临的问题得到了解决，成功的引入并整合和了HTMLTestRunner 测试报告。通过discover() 解决了用例读取的麻烦。

回顾当前目录结构:

/selenium_python/test_case/baidu.py	-----测试用例
/test_case/youdao.py	-----测试用例
/test_case/webcloud.py	-----私有云用例
/test_case/__init__.py	
/test_case/public/login.py	-----登录模块
/test_case/public/quit.py	-----退出模块
/test_case/public/__init__.py	
/data/userinfo.csv	-----用户数据参数化文件
/report/now_time_result.html	-----HTML 测试报告
/all_tests.py	----执行用例文件

test_case 目录用于存具体的用例，/test_case/public 目录存入测试用例所调用的公共模块；data 目录用于存放参数化的数据；report 目录用于存放测试报告，，all_case.py 文件执行 test_case 目录中的测试用例。

总结：

本节对目录结果结构做了很多的调整，起因是为了使用 HTMLTestRunner 测试报告，为了使所有用例文件整合在一个报告中，我们又引入了测试套件，为了使各种文件不至于混乱的放在一个目录下，我们又做文件做了分目录存放。在这个过程中，我们绕了不少弯路，笔者认为每个人的学习过程是这样的，不走弯路永远不知道哪条路最好，而且在这个过程中我们学到了更多 python 编程技术，开阔了解决问题的方法和思路。

第八章 自动化测试高级应用

既然要做自动，就得对得起自动化的这个名字。这一章我们将进一步的增加自动化测试的实用，增加自动发邮件功能、多线程 和定时任务，让我们的自动化工作真正变得高效而又强大起来。

第一节、自动发邮件功能

我们自动化脚本运行完成之后生成了测试报告，如果能将结果自动的发到邮箱就不用每次打开阅读，而且随着脚本的不段运行，生成的报告会越来越多，找到最近的报告也是一个比较麻烦的事件；如果能自动的将结果发到 boss 邮箱，也是个不错的选择。

python 的 `smtplib` 模块提供了一种很方便的途径发送电子邮件。它对 `smtp` 协议进行了简单的封装。

`smtp` 协议的基本命令包括：

HELO 向服务器标识用户身份

MAIL 初始化邮件传输 mail from:

RCPT 标识单个的邮件接收人；常在 MAIL 命令后面,可有多个 rcpt to:

DATA 在单个或多个 RCPT 命令后,表示所有的邮件接收人已标识,并初始化数据传输,以.结束

VRFY 用于验证指定的用户/邮箱是否存在；由于安全方面的原因,服务器常禁止此命令

EXPN 验证给定的邮箱列表是否存在,扩充邮箱列表,也常被禁用

HELP 查询服务器支持什么命令

NOOP 无操作,服务器应响应 OK

QUIT 结束会话

RSET 重置会话,当前传输被取消

MAIL FROM 指定发送者地址

RCPT TO 指明的接收者地址

一般 `smtp` 会话有两种方式，一种是邮件直接投递，就是说，比如你要发邮件給 `zzz@163.com`，那

就直接连接 `163.com` 的邮件服务器，把信投給 `zzz@163.com`; 另一种是验证过后的发信，它的过程是，比如你要发邮件給 `zzz@163.com`, 你不是直接投到 `163.com`, 而是通过自己在 `sina.com` 的另一个邮箱来发。这样就要先连接 `sina.com` 的 `smtp` 服务器，然后认证，之后在把要发到 `163.com` 的信件投到 `sina.com` 上，`sina.com` 会帮你把信投递到 `163.com`。

下面解析几种发邮件的实例，让我们深入理解发邮件的实现。

8.1.1、文件形式的邮件

```
#coding=utf-8
import smtplib
from email.mime.text import MIMEText
from email.header import Header

#发送邮箱
sender = 'abc@126.com'
#接收邮箱
receiver = '123456@qq.com'
#发送邮件主题
subject = 'python email test'
#发送邮箱服务器
smtpserver = 'smtp.126.com'
#发送邮箱用户/密码
username = 'abc@126.com'
password = '123456'

#中文需参数‘utf-8’，单字节字符不需要
msg = MIMEText('你好!', 'text', 'utf-8')
msg['Subject'] = Header(subject, 'utf-8')

smtp = smtplib.SMTP()
smtp.connect('smtp.126.com')
smtp.login(username, password)
smtp.sendmail(sender, receiver, msg.as_string())
smtp.quit()

import smtplib
```

导入 `smtplib` 发邮件模块，从面的脚本，邮件的发送、接收等相关服务，全部由 `smtplib.SMTP` 方法来完成。

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

导入 `email` 模块，`MIMEText` 和 `Header` 主要用来完邮件内容与邮件标题的定义。

`smtp.connect()`

用于链接邮件服务器

`smtp.login()`

配置发送邮箱的用户名密码

`smtp.sendmail()`

配置发送邮箱，接收邮箱，以及发送内容

`smtp.quit()`

关闭发邮件服务

运行程序，登录接收邮件的邮箱，会看一封发来的邮件。如图 7.x：

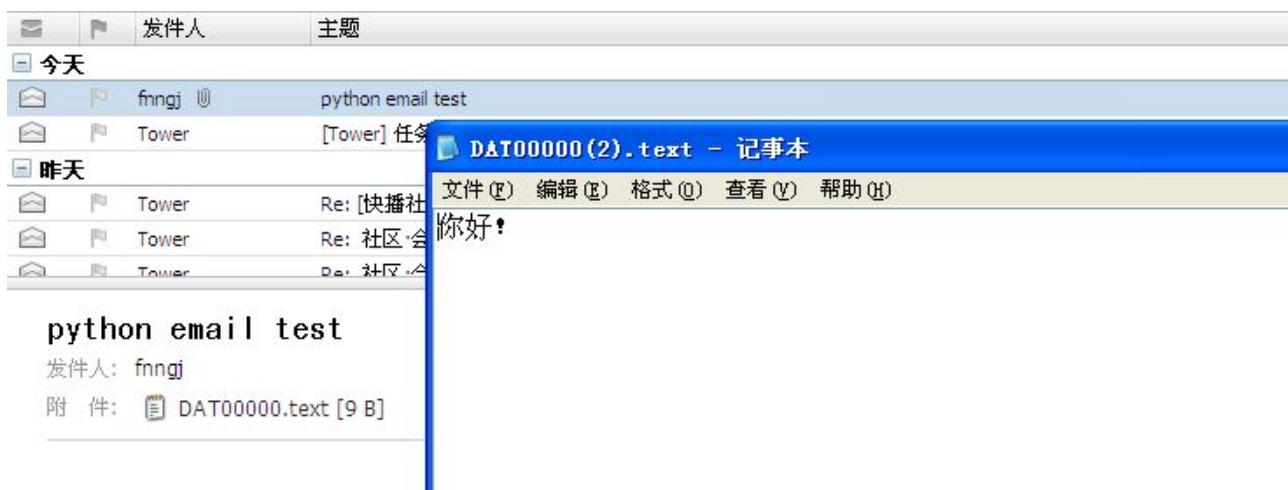


图 8.1

文件标题为“`python email test`”，邮件内容在生成的一个 `DAT000000.text` 文件里，打开文件可阅读文件内容。注意：不同邮件服务器对邮件文件的解析会有差异。

8.1.2、HTML 形式的邮件

```
#coding=utf-8
import smtplib
from email.mime.text import MIMEText
from email.header import Header

#邮件信息配置
sender = 'abc@126.com'
receiver = '123456@qq.com'
subject = 'python email test'
```

```

smtpserver = 'smtp.126.com'
username = 'abc@126.com'
password = '123456'

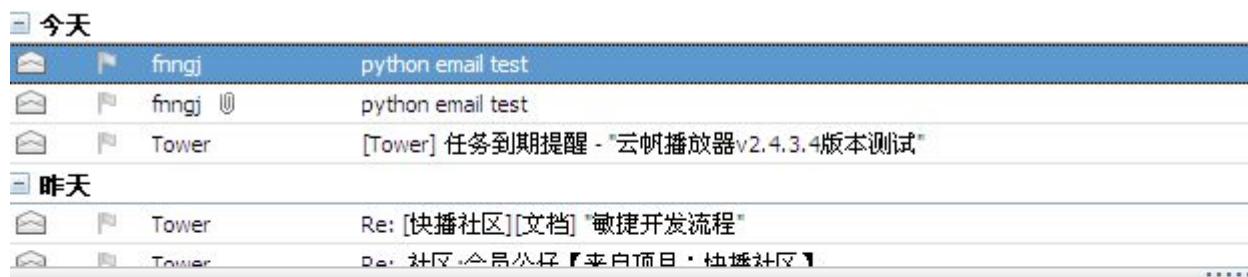
#HTML 形式的文件内容
msg = MIMEText ('<html><h1>你好! </h1></html>', 'html', 'utf-8')
msg['Subject'] = subject

smtp = smtplib.SMTP()
smtp.connect('smtp.126.com')
smtp.login(username, password)
smtp.sendmail(sender, receiver, msg.as_string())
smtp.quit()

```

相比上面的例子，我们唯一修改的就是文件内容的形式。这次的文件主题直接显示在邮件的正文，不需要通过附件打开进行阅读。

运行程序，登录接收邮件的邮箱查看，如图 8.2。



python email test

发件人: fnngj

你好!

图 8.2

8.1.3、获取测试报告

我们已经可以通过 `python` 编写发邮件程序了，现在要解决的问题如何在 `report` 目录下找到最新生成的报告，只有找到了才能把发邮件功能集成到我们的自动化应用中。

`newfile.py`

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

```

import os,datetime,time

result_dir = 'D:\\selenium_python\\report'

lists=os.listdir(result_dir)
lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn) if not os.path.isdir(result_dir+"\\"+fn) else 0)

print ('最新的文件为:  '+lists[-1])
file = os.path.join(result_dir,lists[-1])
print file

```

os.listdir()

用于获取目录下的所有文件列表

```

lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn) if not os.path.isdir(result_dir+"\\"+fn)
else 0)

```

在这段代码中，这一条语句是比较复杂难懂的，我们将其拆开来分析。

lists.sort()

Python 列表有一个内置的列表。**sort()**方法用于改变列表中元素的位置。还有一个 **sorted()**内置函数，建立了一种新的迭代排序列表。

key=lambda fn:

key 是带一个参数的函数，用来为每个元素提取比较值。默认为 **None**，即直接比较每个元素

lambda 提供了一个运行时动态创建函数的方法。我这里创建了 **fn** 函数。

下面一个小例子来演示通过 **sort()**方法对一数组进排序：

```

#定位一个数组
>>> lists=['c.txt','b.txt','d.txt','a.txt']
>>> lists
['c.txt', 'b.txt', 'd.txt', 'a.txt']
#取数组中的 key 做排序
>>> lists.sort(key=lambda lists:lists[0])
>>> print lists
['a.txt', 'b.txt', 'c.txt', 'd.txt']
>>> lists.sort(key=lambda lists:lists[1])
>>> lists
['c.txt', 'b.txt', 'd.txt', 'a.txt']

```

lists:lists[0] 表示取的是每个元组中的前半部分，即为： **c、b、d、a**， 所以可进行排序。

lists:lists[1] 表示取的是每个元组中的后半部分，即为： **txt**， 不能有效的进行排序规律，所以按照数组的原样输出。

os.path.getmtime()

`getmtime()`返回文件列表中最新文件的时间（最新文件的时间最大，所以我们会得到一个最大时间）

os.path.isdir()

`isdir()`函数判断某一路径是否为目录。

经过 `sort()`复杂的运算，我们获得了最新的 `lists` 文件列表，这个文件列表是根据文件的创建时间进行排序的。

lists[-1]

-1 表示取文件列表中的最大值，也就是最新被创建的文件。

os.path.join()

`join()`方法用来连接字符串，通过路径与文件名的拼接，我们将得到目录下最新被创建的文件名的完整路径。运行 `newfile.py` 得到如下结果：

```
>>> ===== RESTART =====
>>>
最新的文件为： 2013-12-17-10_23_32result.html
D:\selenium_python\report\2013-12-17-10_23_32result.html
```

8.1.4、整合自动发邮件功能

前提基本功已经练得差不多了，我们已经学会了如何通过 `python` 现一个发邮件的脚本，学会了如何在一个目录下找到最新的文件。下面就将他们整合到 `all_tests.py` 文件中。

```
#coding=utf-8
import unittest
import HTMLTestRunner
import os ,time,datetime
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.image import MIMEImage

#定义发送邮件
def sentmail(file_new):
    #发信邮箱
    mail_from='fnngj@126.com'
```

```

#收信邮箱
mail_to='123456@qq.com'
#定义正文
f = open(file_new, 'rb')
mail_body = f.read()
f.close()
msg=MIMEText(mail_body,_subtype='html',_charset='utf-8')
#定义标题
msg['Subject']=u"私有云测试报告"
#定义发送时间（不定义的可能有的邮件客户端会不显示发送时间）
msg['date']=time.strftime('%a, %d %b %Y %H:%M:%S %z')
smtp=smtplib.SMTP()
#连接 SMTP 服务器，此处用的126的 SMTP 服务器
smtp.connect('smtp.126.com')
#用户名密码
smtp.login('fnngj@126.com','123456')
smtp.sendmail(mail_from,mail_to,msg.as_string())
smtp.quit()
print 'email has send out !'

#查找测试报告，调用发邮件功能
def sendreport():
    result_dir = 'D:\\selenium_python\\report'
    lists=os.listdir(result_dir)
    lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn) if not os.path.isdir(result_dir+"\\"+fn) else 0)
    print (u'最新测试生成的报告: '+lists[-1])
    #找到最新生成的文件
    file_new = os.path.join(result_dir,lists[-1])
    print file_new
    #调用发邮件模块
    sentmail(file_new)

.....
if __name__ == "__main__":
    #执行测试用例
    runner.run(alltestnames)
    #执行发邮件
    sendreport()

```

sentmail(file_new)

定义一个 `sentmail()`发邮件函数，接收一个参数 `file_new`，表示接收最新生成的测试报告文件。

`open(file_new, 'rb')`

以读写（rb）方式打开最新生成的测试报告文件。

```
mail_body = f.read()
```

读取文件内容，将内容传递给 mail_body。

```
MIMEText(mail_body,_subtype='html',_charset='utf-8')
```

文件内容写入到邮件正文中。html 格式，编码为 utf-8。

```
sendreport()
```

定义 sendreport() 用于找最新生成的测试报告文件 file_new。调用并将 file_new 传给 sentmail() 函数。

程序执行过程：

执行 all_tests.py 文件，通过执行 runner.run(alltestnames) 方法开始执行测试用例，所有用例执行完成执行 sendreport() 方法，用于找到最新生成的报告，并将报告内容发送到指定的邮箱。

问题：

指定的邮箱可以正常收到邮件，但所得到的邮件内容是空的，这是由于 HTMLTestRunner 报告文件的机制所引起的。在测试用例运行之前生成报告文件，在整个程序没有彻底运行结束前，程序并没有把运行的结果写入到文件中，所以，在用例运行完成后发邮件，造成邮件内容是空的。

所以，我们不能在整个程序未运行结束时发送当前的测试报告，我们可以选择上一次运行结果的报告进行发送。对代码做如下修改：

```
.....  
def sendreport():  
    result_dir = 'D:\\selenium_python\\report'  
    lists=os.listdir(result_dir)  
    lists.sort(key=lambda fn: os.path.getmtime(result_dir+"\\"+fn) if not  
os.path.isdir(result_dir+"\\"+fn) else 0)  
    print (u'上一次测试生成的报告: '+lists[-2])  
    #找到上一次测试生成的文件  
    file_new = os.path.join(result_dir,lists[-2])  
    print file_new  
    #调用发邮件模块  
    sentmail(file_new)  
.....
```

lists[-2]

对文件列表中取的值做修改，-2 表示前一次生成的测试结果。

再次运行程序，邮件就正常收到了测试报告的内容。如图 8.3

私有云测试报告

发件人 : fnngj <fnngj@126.com>

时间 : 2013年12月18日(星期三) 下午4:27 (UTC+0:00 伦敦、都柏林、里斯本时间)

提示 : 你不在收件人里, 可能这封邮件是密送给你的。

私有云测试报告**Start Time:** 2013-12-18 16:11:37**Duration:** 0:00:47.500000**Status:** Pass 4

用例执行情况:

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

Test Group/Test case	Count	Pass	Fail	Error	View
start_baidu.Baidu	2	2	0	0	Detail
start_webcloud.Webcloud	1	1	0	0	Detail
start_youdao.Youdao	1	1	0	0	Detail
Total	4	4	0	0	

图 8.3

第二节、python 多进程/线程基础



在使用多线程之前, 我们首先要理解什么是进程和线程。

什么是进程?

计算机程序只不过是磁盘中可执行的, 二进制 (或其它类型) 的数据。它们只有在被读取到内存中, 被操作系统调用的时候才开始它们的生命期。进程 (有时被称为重量级进程) 是程序的一次执行。每个进程都有自己的地址空间, 内存, 数据栈以及其它记录其运行轨迹的辅助数据。操作系统管理在其上运行的所有进程, 并为这些进程公平地分配时间。

什么是线程?

线程 (有时被称为轻量级进程) 跟进程有些相似, 不同的是, 所有的线程运行在同一个进程中, 共享相同的运行环境。我们可以想像成是在主进程或“主线程”中并行运行的“迷你进程”。

8.2.1、单线程

在单线程中顺序执行两个循环。一定要一个循环结束后，另一个才能开始。总时间是各个循环运行时间之和。

onetherad.py

```
from time import sleep, ctime

def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
    print 'loop 0 done at:', ctime()

def loop1():
    print 'start loop 1 at:', ctime()
    sleep(2)
    print 'loop 1 done at:', ctime()

def main():
    print 'start:', ctime()
    loop0()
    loop1()
    print 'all end:', ctime()

if __name__ == '__main__':
    main()
```

运行结果：

```
start loop 0 at: Mon Dec 23 09:59:44 2013
loop 0 done at: Mon Dec 23 09:59:48 2013
start loop 1 at: Mon Dec 23 09:59:48 2013
loop 1 done at: Mon Dec 23 09:59:50 2013
all end: Mon Dec 23 09:59:50 2013
```

Python 通过两个标准库 `thread` 和 `threading` 提供对线程的支持。`thread` 提供了低级别的、原始的线程以及一个简单的锁。`threading` 基于 Java 的线程模型设计。锁（Lock）和条件变量（Condition）在 Java 中是对象的基本行为（每一个对象都自带了锁和条件变量），而在 Python 中则是独立的对象。

8.2.2、thread 模块

mtsleep1.py

```
import thread
from time import sleep, ctime
loops = [4,2]
def loop0():
    print 'start loop 0 at:', ctime()
    sleep(4)
    print 'loop 0 done at:', ctime()

def loop1():
    print 'start loop 1 at:', ctime()
    sleep(2)
    print 'loop 1 done at:', ctime()

def main():
    print 'start:', ctime()
    thread.start_new_thread(loop0, ())
    thread.start_new_thread(loop1, ())
    sleep(6)
    print 'all end:', ctime()

if __name__ == '__main__':
    main()
```

`start_new_thread()`要求一定要有前两个参数。所以，就算我们想要运行的函数不要参数，我们也要传一个空的元组。

这个程序的输出与之前的输出大不相同，之前是运行了 6, 7 秒，而现在则是 4 秒，是最长的循环的运行时间与其它的代码的时间总和。

```
start: Mon Dec 23 10:05:09 2013
start loop 0 at: Mon Dec 23 10:05:09 2013
start loop 1 at: Mon Dec 23 10:05:09 2013
loop 1 done at: Mon Dec 23 10:05:11 2013
loop 0 done at: Mon Dec 23 10:05:13 2013
all end: Mon Dec 23 10:05:15 2013
```

睡眠 4 秒和 2 秒的代码现在是并发执行的。这样，就使得总的运行时间被缩短了。你可以看到，`loop1` 甚至在 `loop0` 前面就结束了。

程序的一大不同之处就是多了一个“`sleep(6)`”的函数调用。如果我们没有让主线程停下来，那主线程就会运行下一条语句，显示“`all end`”，然后就关闭运行着 `loop0()` 和 `loop1()` 的两个线程并退出了。

我们使用 **6** 秒是因为我们已经知道，两个线程（你知道，一个要 **4** 秒，一个要 **2** 秒）在主线程等待 **6** 秒后应该已经结束了。

你也许在想，应该有什么好的管理线程的方法，而不是在主线程里做一个额外的延时 **6** 秒的操作。因为这样一来，我们的总的运行时间并不比单线程的版本来得少。而且，像这样使用 `sleep()` 函数做线程的同步操作是不可靠的。如果我们的循环的执行时间不能事先确定的话，那怎么办呢？这可能造成主线程过早或过晚退出。这就是锁的用武之地了。

mtsleep2.py

```
#coding=utf-8
import thread
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec, lock):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()
    #解锁
    lock.release()

def main():
    print 'starting at:', ctime()
    locks = []
    #以 loops 数组创建列表，并赋值给 nloops
    nloops = range(len(loops))

    for i in nloops:
        lock = thread.allocate_lock()
        #锁定
        lock.acquire()
        #追加到 locks[] 数组中
        locks.append(lock)

    #执行多线程
    for i in nloops:
        thread.start_new_thread(loop, (i, loops[i], locks[i]))

    for i in nloops:
        while locks[i].locked():
            pass

    print 'all end:', ctime()
```

```
if __name__ == '__main__':
    main()

    thread.allocate_lock()
```

返回一个新的锁定对象。

[acquire\(\)](#) / [release\(\)](#)

一个原始的锁有两种状态，锁定与解锁，分别对应 `acquire()` 和 `release()` 方法。

[range\(\)](#)

`range()` 函数来创建列表包含算术级数。

`range(len(loops))` 函数理解：

```
>>> aa= "hello"

#长度计算
>>> len(aa)
5

#创建列表
>>> range(len(aa))
[0, 1, 2, 3, 4]

#循环输出列表元素
>>> for a in range(len(aa)):
    print a
```

```
0
1
2
3
4
```

我们先调用 `thread.allocate_lock()` 函数创建一个锁的列表，并分别调用各个锁的 `acquire()` 函数获得锁。获得锁表示“把锁锁上”。锁上后，我们就把锁放到锁列表 `locks` 中。

下一个循环创建线程，每个线程都用各自的循环号，睡眠时间和锁为参数去调用 `loop()` 函数。为什么我们不在创建锁的循环里创建线程呢？有以下几个原因：(1) 我们想到实现线程的同步，所以要让“所有的马同时冲出栅栏”。(2) 获取锁要花一些时间，如果你的线程退出得“太快”，可能会导致还没有获得锁，线程就已经结束了的情况。

在线程结束的时候，线程要自己去做解锁操作。最后一个循环只是坐在那一直等（达到暂停主线程的目的），直到两个锁都被解锁为止才继续运行。

mtsleep2.py 运行结果：

```
starting at: Mon Dec 23 20:57:26 2013
start loop start loop0 1at: at:Mon Dec 23 20:57:26 2013
Mon Dec 23 20:57:26 2013
loop 1 done at: Mon Dec 23 20:57:28 2013
loop 0 done at: Mon Dec 23 20:57:30 2013
all end: Mon Dec 23 20:57:30 2013
```

8.2.3、threading 模块

我们应该避免使用 `thread` 模块，原因是它不支持守护线程。当主线程退出时，所有的子线程不论它们是否还在工作，都会被强行退出。有时我们并不期望这种行为，这时就引入了守护线程的概念。`threading` 模块则支持守护线程。

mtsleep3.py

```
#coding=utf-8
import threading
from time import sleep, ctime

loops = [4,2]

def loop(nloop, nsec):
    print 'start loop', nloop, 'at:', ctime()
    sleep(nsec)
    print 'loop', nloop, 'done at:', ctime()

def main():
    print 'starting at:', ctime()
    threads = []
    nloops = range(len(loops))

    #创建线程
    for i in nloops:
        t = threading.Thread(target=loop, args=(i, loops[i]))
        threads.append(t)

    #开始线程
    for t in threads:
        t.start()

    for t in threads:
        t.join()

    print 'all done'
```

```

for i in nloops:
    threads[i].start()

#等待所有结束线程
for i in nloops:
    threads[i].join()

print 'all end:', ctime()

if __name__ == '__main__':
    main()

```

运行时长：

```

starting at: Mon Dec 23 22:58:55 2013
start loop 0 at: Mon Dec 23 22:58:55 2013
start loop 1 at: Mon Dec 23 22:58:55 2013
loop 1 done at: Mon Dec 23 22:58:57 2013
loop 0 done at: Mon Dec 23 22:58:59 2013
all end: Mon Dec 23 22:58:59 2013

```

[start\(\)](#)

开始线程活动

[join\(\)](#)

等待线程终止

所有的线程都创建了之后，再一起调用 `start()` 函数启动，而不是创建一个启动一个。而且，不用再管理一堆锁（分配锁，获得锁，释放锁，检查锁的状态等），只要简单地对每个线程调用 `join()` 函数就可以了。

`join()` 会等到线程结束，或者在给了 `timeout` 参数的时候，等到超时为止。`join()` 的另一个比较重要的方面是它可以完全不用调用。一旦线程启动后，就会一直运行，直到线程的函数结束，退出为止。

使用可调用的类

`mtsleep4.py`

```

#coding=utf-8
import threading
from time import sleep, ctime

```

```

loops = [4,2]

class ThreadFunc(object):

    def __init__(self,func,args,name=''):
        self.name=name
        self.func=func
        self.args=args

    def __call__(self):
        apply(self.func,self.args)

def loop(nloop,nsec):
    print "seart loop",nloop,'at:',ctime()
    sleep(nsec)
    print 'loop',nloop,'done at:',ctime()

def main():
    print 'starting at:',ctime()
    threads=[]
    nloops = range(len(loops))

    for i in nloops:
        #调用 ThreadFunc 实例化的对象，创建所有线程
        t = threading.Thread(
            target=ThreadFunc(loop,(i,loops[i]),loop.__name__))
        threads.append(t)

    #开始线程
    for i in nloops:
        threads[i].start()

    #等待所有结束线程
    for i in nloops:
        threads[i].join()

    print 'all end:', ctime()

if __name__ == '__main__':
    main()

```

运行结果：

```

starting at: Tue Dec 24 16:39:16 2013
seart loop 0 at: Tue Dec 24 16:39:16 2013

```

```
seart loop 1 at: Tue Dec 24 16:39:16 2013
loop 1 done at: Tue Dec 24 16:39:18 2013
loop 0 done at: Tue Dec 24 16:39:20 2013
all end: Tue Dec 24 16:39:20 2013
```

创建新线程的时候，`Thread` 对象会调用我们的 `ThreadFunc` 对象，这时会用到一个特殊函数`__call__()`。由于我们已经有了要用的参数，所以就不用再传到 `Thread()` 的构造函数中。由于我们有一个参数的元组，这时要在代码中使用 `apply()` 函数。

我们传了一个可调用的类(的实例)，而不是仅传一个函数。

`__init__()`

方法在类的一个对象被建立时运行。这个方法可以用来对你的对象做一些初始化。

`apply()`

`apply(func [, args [, kwargs]])` 函数用于当函数参数已经存在于一个元组或字典中时，间接地调用函数。`args` 是一个包含将要提供给函数的按位置传递的参数的元组。如果省略了 `args`，任何参数都不会被传递，`kwargs` 是一个包含关键字参数的字典。

`apply()` 用法：

#不带参数的方法

```
>>> def say():
    print 'say in'

>>> apply(say)
say in
```

#函数只带元组的参数

```
>>> def say(a,b):
    print a,b

>>> apply(say, ('hello','虫师'))
hello 虫师
```

#函数带关键字参数

```
>>> def say(a=1,b=2):
    print a,b

>>> def haha(**kw):
    apply(say, (), kw)
```

```
>>> haha(a='a',b='b')
a b
```

8.2.4、multiprocessing 模块

multiprocessing 使用类似于 threading 模块的 API , multiprocessing 提供了本地和远程的并发性，有效的通过全局解释锁(Global Interpreter Lock, GIL)来使用进程(而不是线程)。由于 GIL 的存在，在 CPU 密集型的程序当中，使用多线程并不能有效地利用多核 CPU 的优势，因为一个解释器在同一时刻只会有一个线程在执行。所以，multiprocessing 模块可以充分的利用硬件的多处理器来进行工作。它支持 Unix 和 Windows 系统上的运行。

process1.py

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

运行结果：

```
hello bob
```

与 threading.Thread 类似，它可以利用 multiprocessing.Process 对象来创建一个进程。Process 对象与 Thread 对象的用法相同，也有 start(), run(), join() 的方法。

`multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={})`

target 表示调用对象，args 表示调用对象的位置参数元组。kwargs 表示调用对象的字典。Name 为别名。Group 实质上不使用。

扩展理解：

在*nix 上面创建的新的进程使用的是 fork:

一个进程，包括代码、数据和分配给进程的资源。`fork()`函数通过系统调用创建一个与原来进程几乎完全相同的进程，也就是两个进程可以做完全相同的事，但如果初始参数或者传入的变量不同，两个进程也可以做不同的事。

这意味着子进程开始执行的时候具有与父进程相同的全部内容。请记住这点，这个将是下面我们将讨论

基于继承的对象共享的基础。所谓基于继承的对象共享，是说在创建子进程之前由父进程初始化的某些对象可以在子进程中直接访问到。在 Windows 平台上，因为没有 fork 语义的系统调用，基于继承的共享对象比*nix 有更多的限制，最主要就是体现在要求 Process 的 `__init__` 当中的参数必须可以 Pickle。

但是，并不是所有的对象都是可以通过继承来共享，只有 multiprocessing 库当中的某些对象才可以。例如 Queue，同步对象，共享变量，Manager 等等。

在一个 multiprocessing 库的典型使用场景下，所有的子进程都是由一个父进程启动起来的，这个父进程称为 master 进程。这个父进程非常重要，它会管理一系列的对象状态，一旦这个进程退出，子进程很可能会处于一个很不稳定的状态，因为它们共享的状态也许已经被损坏掉了。因此，这个进程最好尽可能做最少的事情，以便保持其稳定性。

获取进程 ID

process2.py

```
from multiprocessing import Process
import os

def info(title):
    print title
    print 'module name:', __name__
    if hasattr(os, 'getppid'):
        print 'parent process:', os.getppid()
    print 'process id:', os.getpid()

def f(name):
    info('function f')
    print 'hello', name

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

运行结果：

```
main line
module name: __main__
process id: 8972
function f
module name: __main__
```

```
process id: 10648
```

```
hello bob
```

`multiprocessing` 提供了 `threading` 包中没有的 IPC(比如 `Pipe` 和 `Queue`)，效率上更高。应优先考虑 `Pipe` 和 `Queue`，**避免使用 Lock/Event/Semaphore/Condition** 等同步方式 (因为它们占据的不是用户进程的资源)。

`hasattr(object, name)`

判断对象 `object` 是否包含名为 `name` 的特性 (`hasattr` 是通过调用 `getattr(object, name)` 是否抛出异常来实现的)。`hasattr(os, 'getppid')` 用于判断系统是否包含 `getppid`。

`getpid()` 得到本身进程 `id`, `getppid()` 得到父进程进程 `id`, 如果已经是父进程, 得到系统进程 `id`。

`Process.PID` 中保存有 `PID`, 如果进程还没有 `start()`, 则 `PID` 为 `None`。

`Pipe` 和 `Queue`

8.2.5、`Pipe` 和 `queue`

`multiprocessing` 包中有 `Pipe` 类和 `Queue` 类来分别支持这两种 IPC 机制。`Pipe` 和 `Queue` 可以用来传送常见的对象。

(1) `Pipe` 可以是单向 (half-duplex), 也可以是双向 (duplex)。我们通过 `multiprocessing.Pipe(duplex=False)` 创建单向管道 (默认为双向)。一个进程从 `PIPE` 一端输入对象, 然后被 `PIPE` 另一端的进程接收, 单向管道只允许管道一端的进程输入, 而双向管道则允许从两端输入。

`pipe.py`

```
#coding=utf-8
import multiprocessing

def proc1(pipe):
    pipe.send('hello')
    print('proc1 rec:', pipe.recv())

def proc2(pipe):
    print('proc2 rec:', pipe.recv())
    pipe.send('hello, too')

pipe = multiprocessing.Pipe()
```

```

p1 = multiprocessing.Process(target=proc1, args=(pipe[0],))
p2 = multiprocessing.Process(target=proc2, args=(pipe[1],))

p1.start()
p2.start()
p1.join()
p2.join()

```

注：本程序只能在 linux/Unix 上运行，运行结果：

```

'proc2 rec:', 'hello'
'proc2 rec:', 'hello,too'

```

这里的 Pipe 是双向的。Pipe 对象建立的时候，返回一个含有两个元素的表，每个元素代表 Pipe 的一端 (Connection 对象)。我们对 Pipe 的某一端调用 `send()` 方法来传送对象，在另一端使用 `recv()` 来接收。

(2) Queue 与 Pipe 相类似，都是先进先出的结构。但 Queue 允许多个进程放入，多个进程从队列取出对象。Queue 使用 `multiprocessing.Queue(maxsize)` 创建，`maxsize` 表示队列中可以存放对象的最大数量。

queue.py

```

#coding=utf-8
import os
import multiprocessing
import time

# input worker
def inputQ(queue):
    info = str(os.getpid()) + '(put):' + str(time.time())
    queue.put(info)

# output worker
def outputQ(queue, lock):
    info = queue.get()
    lock.acquire()
    print(str(os.getpid()) + '(get):' + info)
    lock.release()

# Main
record1 = [] # store input processes
record2 = [] # store output processes
lock = multiprocessing.Lock() # 加锁，为防止散乱的打印
queue = multiprocessing.Queue(3)

```

```

# input processes
for i in range(10):
    process = multiprocessing.Process(target=inputQ, args=(queue,))
    process.start()
    record1.append(process)

# output processes
for i in range(10):
    process = multiprocessing.Process(target=outputQ, args=(queue, lock))
    process.start()
    record2.append(process)

for p in record1:
    p.join()

queue.close() # 没有更多的对象进来，关闭 queue

for p in record2:
    p.join()

```

注：本程序只能在 linux/Unix 上运行，运行结果

```

2702(get):2689(put):1387947815.56
2704(get):2691(put):1387947815.58
2706(get):2690(put):1387947815.56
2707(get):2694(put):1387947815.59
2708(get):2692(put):1387947815.61
2709(get):2697(put):1387947815.6
2703(get):2698(put):1387947815.61
2713(get):2701(put):1387947815.65
2716(get):2699(put):1387947815.63
2717(get):2700(put):1387947815.62

```

第三节、多进程执行测试用例

因为多进程与多线程特性在 python 中当中也属于比较高级的应用，对于初学者来说比理解起来有一定
<http://fnng.cnblogs.com>

难度，所以在介绍 python 的多进程与多线程时我们通过相当的篇幅和实例来进行讲解，目的是为了让读者深入的理解 python 的多进程与多线程。

为实现多进程运行测试用例，我需要对文件结构进行调整：

```
/selenium_proces/thread1/start_baidu.py      -----测试用例
/thread1/__init__.py
/thread2/start_youdao.py      -----测试用例
/thread2/__init__.py
/all_tests_process.py
```

我们创建了 thread1 和 thread2 两个文件夹，分别放入了两个测试用例；下面我们编写 all_tests_process.py 文件来通过多进程来执行测试用例。

```
#coding=utf-8
import unittest, time, os, multiprocessing
import commands
from email.mime.text import MIMEText
import HTMLTestRunner
import sys
sys.path.append('selenium_proces')

def EEEcreatsuite1():
    casedir=[]
    lista=os.listdir('D:\\selenium_proces\\')
    for xx in lista:
        if "thread" in xx:
            casedir.append(xx)
    print casedir

    suite=[]
    for n in casedir:
        testunit=unittest.TestSuite()
        discover=unittest.defaultTestLoader.discover(str(n),pattern
='start_*.py',top_level_dir=r'E:\\')
        for test_suite in discover:
            for test_case in test_suite:
                testunit.addTests(test_case)
                #print testunit
        suite.append(testunit)
    return suite,casedir
```

```

def EEEEmultiRunCase(suite,casedir):
    now = time.strftime('%Y-%m-%d-%H_%M_%S',time.localtime(time.time()))
    filename = 'D:\\selenium_python\\report\\'+now+'result.html'
    fp = file(filename, 'wb')

    proclist=[]
    s=0
    for i in suite:
        runner = HTMLTestRunner.HTMLTestRunner(
            stream=fp,
            title=str(casedir[s])+u'测试报告',
            description=u'用例执行情况：'
        )

        proc = multiprocessing.Process(target=runner.run,args=(i,))
        proclist.append(proc)
        s=s+1
    for proc in proclist: proc.start()
    for proc in proclist: proc.join()
    fp.close()

if __name__ == "__main__":
    runtmp=EEEcreatesuite1()
    EEEEmultiRunCase(runtmp[0],runtmp[1])

```

all_tests_process.py 程序稍微复杂，我们分段来进行讲解。

在 EEEcreatesuite1() 函数中：

```

.....
casedir=[]
listaa=os.listdir('D:\\selenium_proces\\')
for xx in listaa:
    if "thread" in xx:
        casedir.append(xx)
print casedir
.....

```

定义 casedir 数组，读取 selenium_proces 目录下的文件/文件夹，找到文件/文件夹的名包含“thread”的文件/文件夹添加到 casedir 数组中（即 thread1 和 thread2 两个文件夹）。

```

.....
suite=[]
for n in casedir:

```

```

testunit=unittest.TestSuite()
discover=unittest.defaultTestLoader.discover(str(n),pattern
='start_*.py',top_level_dir=r'E:\\')
for test_suite in discover:
    for test_case in test_suite:
        testunit.addTests(test_case)
    #print testunit
suite.append(testunit)

return suite,casedir

```

定位 suite 数组，for 循环读取 casedir 数组中的数据（即 thread1 和 thread2 两个文件夹）。通过 discover 分别读取文件夹下匹配 start_*.py 规则的用例文件，将所有用例文件添加到 testunit 测试条件中，再将测试套件追加到定义的 suite 数组中。

在整个 EEEcreatsuite1() 函数中 返回 suite 和 casedir 两个数组的值。

在 EEEEmultiRunCase 函数中：

```

.....
proclist=[]
s=0
for i in suite:
    runner = HTMLTestRunner.HTMLTestRunner(
        stream=fp,
        title=str(casedir[s])+u'测试报告',
        description=u'用例执行情况：'
    )

    proc = multiprocessing.Process(target=runner.run,args=(i,))
    proclist.append(proc)
    s=s+1

    for proc in proclist: proc.start()

    for proc in proclist: proc.join()
.....

```

定义 proclist() 函数，for 循环把 suite 数组中的用例执行结果写入 HTMLTestRunner 测试报告。multiprocessing.Process 创建用例执行的多进程，把创建的多进程追加到 proclist 数组中，

for 循环 proc.start() 开始进程活动，proc.join() 等待线程终止。

小结：

笔者虽然花费了不少力气介绍 python 的多进程/多线程，以及如何用多进程执行测试用例，但笔者并不推荐在 window 以及低配置的电脑上使用，另外一台电脑上开的进程不要超过三个，并行执行用例时

相互之间会发生一定的干扰。

第四节、定时任务

为了更对得起“自动化测试”的名号，我们可以设置定时任务，使我们自动化脚本在某个时间点自动运行脚本。实现这个需求的方式很多。

8.4.1、程序控制时间执行

通过程序来控制用例在什么时候执行的执行最简单的方式了。创建 all_tests_time.py 文件：

```
#coding=utf-8
import unittest
import HTMLTestRunner
import os ,time

listaa='D:\\selenium_python\\test_case'
def creatsuitel():
    testunit=unittest.TestSuite()
    discover=unittest.defaultTestLoader.discover(listaa,
                                                    pattern ='start_*.py',
                                                    top_level_dir=None)

    for test_suite in discover:
        for test_case in test_suite:
            testunit.addTests(test_case)
            print testunit
    return testunit

alltestnames = creatsuitel()

now = time.strftime('%Y-%m-%M-%H_%M_%S',time.localtime(time.time()))
filename = 'D:\\selenium_python\\report\\'+now+'result.html'
fp = file(filename, 'wb')

runner =HTMLTestRunner.HTMLTestRunner(
    stream=fp,
    title=u'百度搜索测试报告',
```

```
description=u'用例执行情况：')
```

```
#####控制什么时间脚本执行#####
k=1
while k <2:
    timing=time.strftime('%H_%M',time.localtime(time.time()))
    if timing == '12_00':
        print u"开始运行脚本:"
        runner.run(alltestnames) #执行测试用例
        print u"运行完成退出"
        break
    else:
        time.sleep(5)
        print timing
```

本程序是在 `all_tests.py` 文件基础上做的修改，为了便于读者对完整程序的阅读，所以这里贴出了整个程序代码，下面和读者重点分析最后一段代码。

`k=1, while` 循环中判断 `k<2`，我们在整个循环体中并没有对 `k` 的值做任何修改，所以，单从这个循环条件来判断，是一个死循环。但是，我们可以循环的执行过程中通过 `break` 语句跳出循环。

在循环体中，我们通过 `timing` 来获取当前时间，只获取当前的小时和分钟，来判断是否为 12:00，如果当前时间为 12 点，开始执行测试脚本，并通过 `break` 语句跳出循环。

如果当前时间不等于 12:00 那么修改 5 秒后，输出当前时间。当然这个休眠时间可以自由控制，但不能大于 60 秒，不然，有可能匹配不到 12 点 00 分。

为了运行方便，我们在命令提示符下运行（linux 可以在终端下运行），运行结果如图 8.4

```
C:\> C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:
D:\>cd selenium_python

D:\selenium_python>python all_tests_time.py 加载所有测试用例
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>]>
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>, <start_youdao.Youdao testMethod=test_youdao_search>]>
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>, <start_youdao.Youdao testMethod=test_youdao_search>, <thread2.start_youdao.Youdao testMethod=test_youdao_search>]h>1>
11_59
11_59
11_59
11_59
11_59 获得当前时间，5秒打印一次
11_59
11_59
11_59
11_59
开始运行脚本:
...
Time Elapsed: 0:01:18.828000
运行完成退出 开始执行测试用例

D:\selenium_python>
```

图 8.4

注意：

假如，我们要运行的时间不是今天的12:00，而是某年某月某天的一个时间，那么在获取当前时间时要获取年月日进行比较。

8.4.2、windows 添加任务计划

假如我们会在（每天、每周、每月）一个固定的时间运行测试用例，那么可以通过 windows 创建任务计划，这种方式更加方便，我们不用一直打开一个程序来判断当前的时间。

下面以 windows XP 为例：

打开“控制面板”->“任务计划”->“添加任务计划” 双击打开任务计划向导

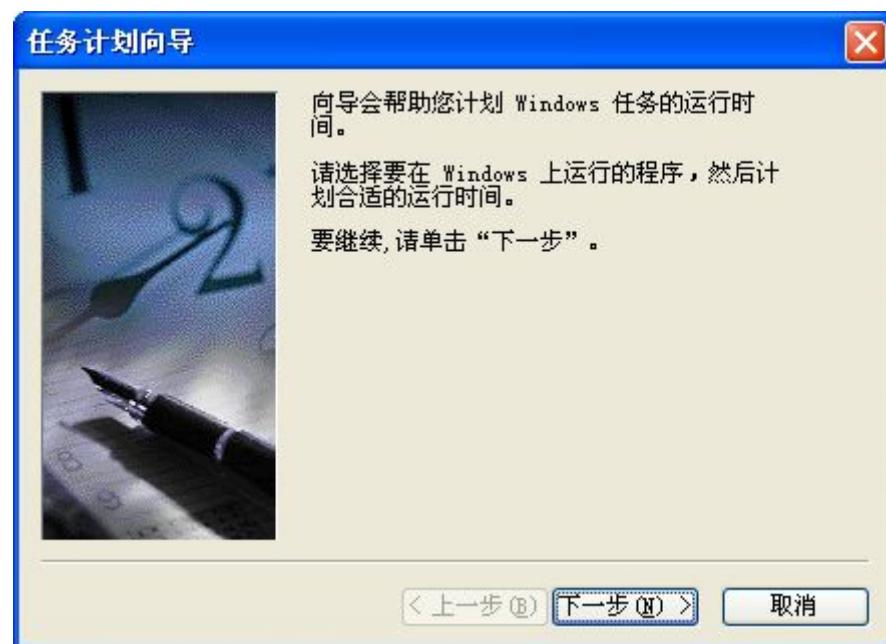


图8.5

点击“下一步”，可以在程序列表选择要执行的程序。

我这里要选择的是一个 python 脚本，点击“浏览”找到要运行的 python 脚本；再次点击“下一步”。

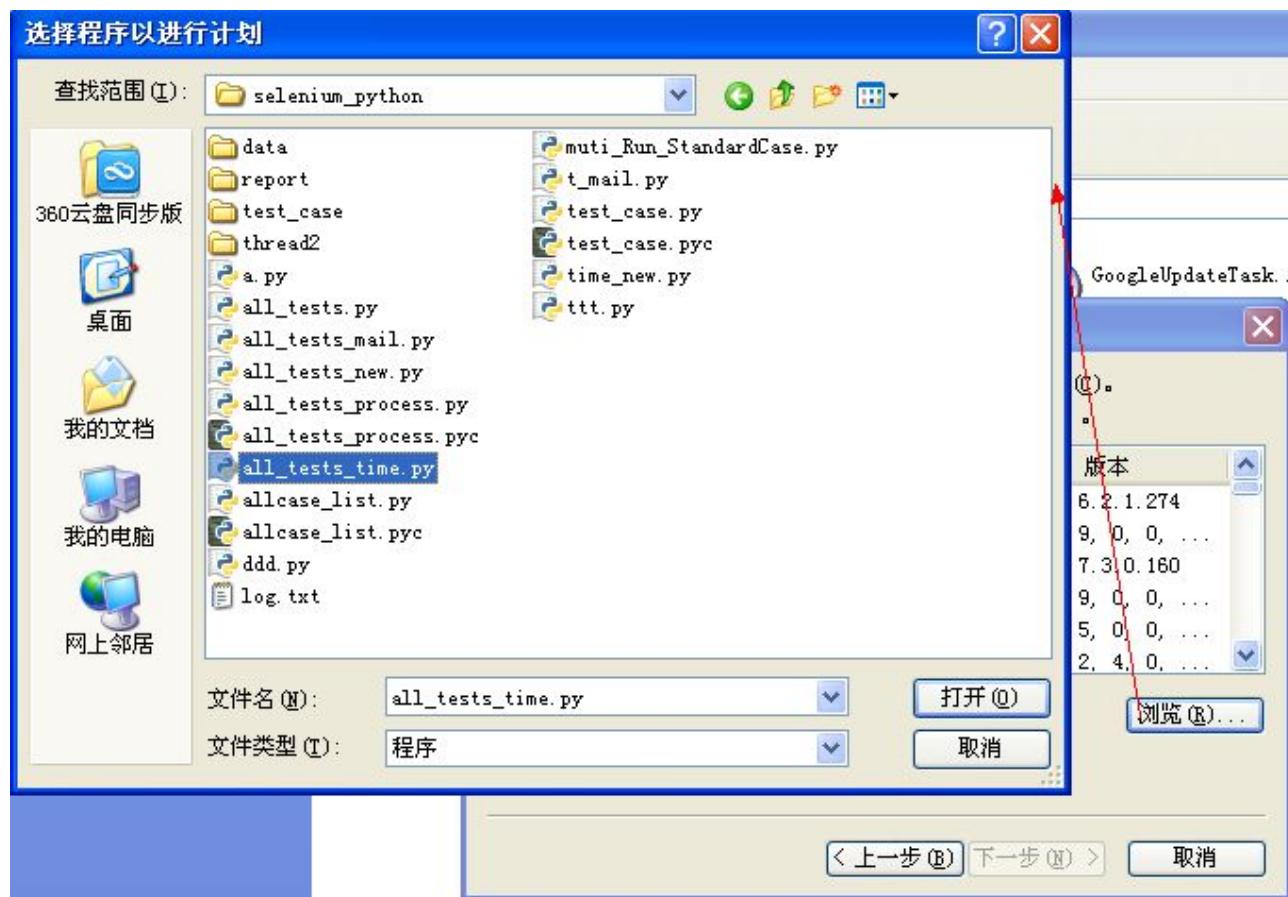


图8.6

对任务进行命名，选择执行脚本的频率（每天，每周，每月，一次性，计算机启动时，登录时），然后，点击“下一步”。



图8.7

选择任务执行起始时间和日期；点击“下一步”。

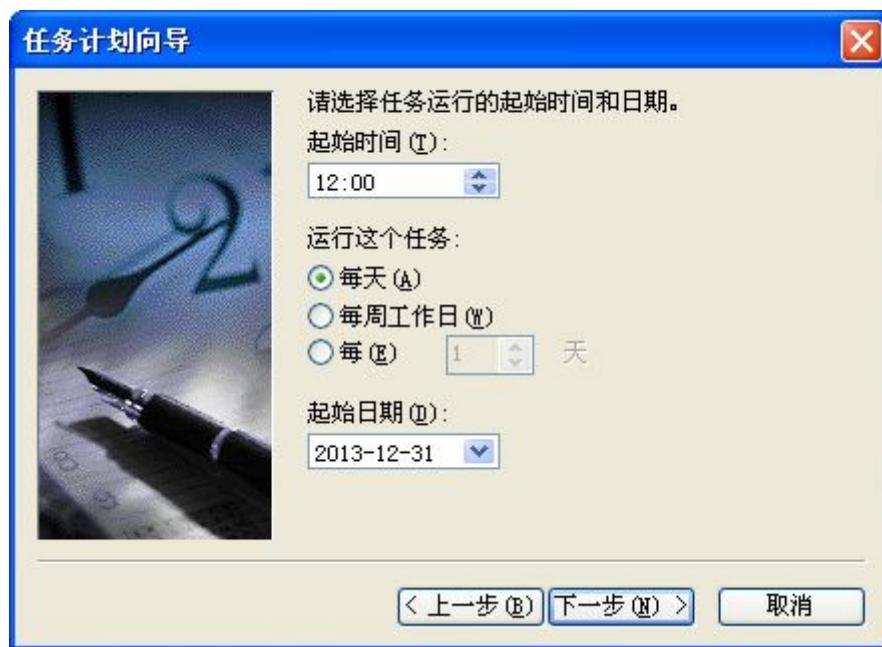


图8.8

输入管理员系统用户的密码。点击“下一步”。

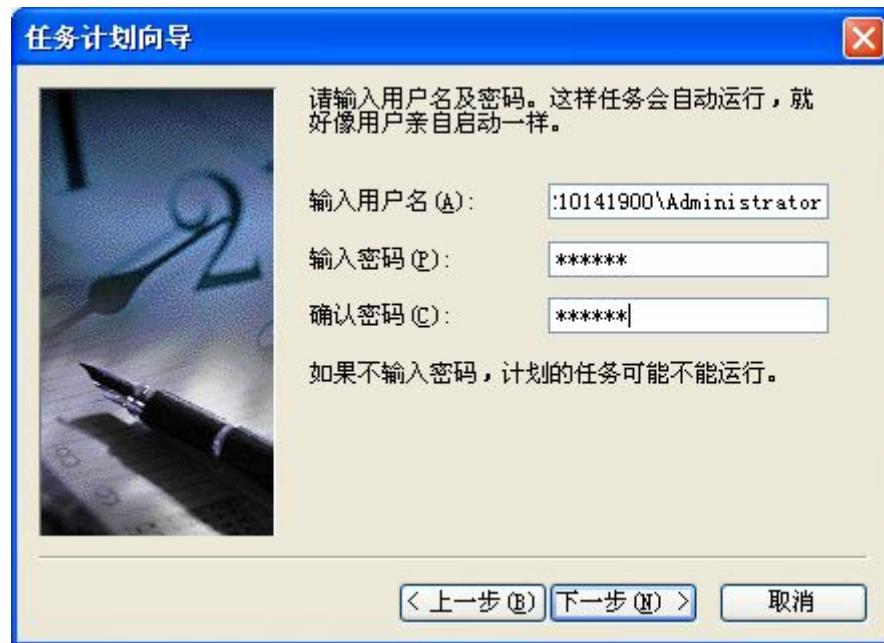


图8.9

确认任务的创建信息，点击“完成”任务计划创建成功。

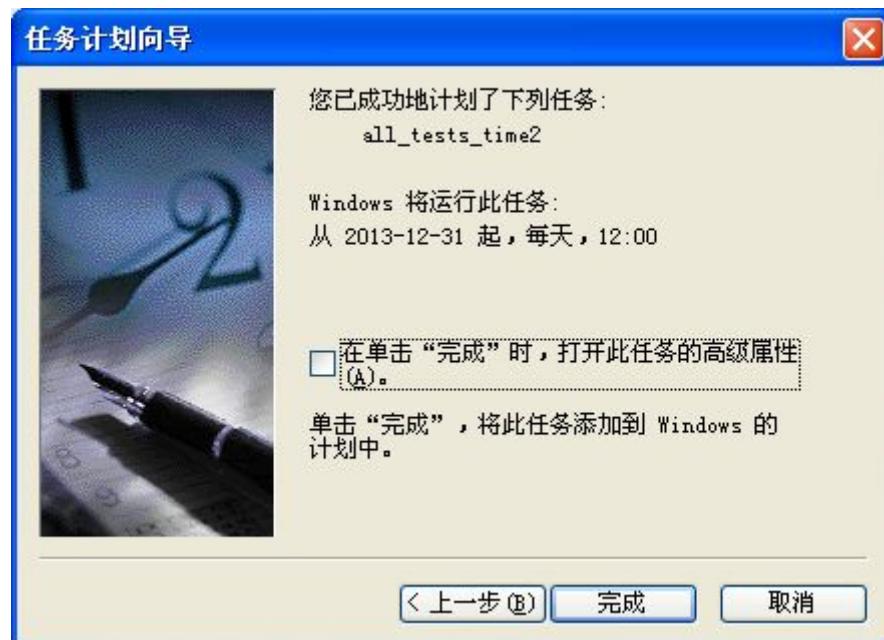


图8.10

任务创建完成，会在当前目录下显示创建完成的任务计划。右键单击创建的任务计划，选择“运行”将会立刻执行程序。



图8.11

运行效果如下：

```
C:\Python27\python.exe
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>]>
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>, <start_youdao.Youdao testMethod=test_youdao_search>]>
<unittest.suite.TestSuite tests=[<start_baidu.Baidu testMethod=test_baidu_search>, <start_baidu.Baidu testMethod=test_baidu_set>, <start_youdao.Youdao testMethod=test_youdao_search>, <thread2.start_youdao.Youdao testMethod=test_youdao_search>]>
...

```

图8.12

当然，我们也可以右键创建的任务计划，选择“属性”，对任务计划进行调整和修改。



图8.13

8.4.3、linux 实现定时任务

在 linux 下相对实现定时任务的方式比较灵活。我们可以通过 at 命令实现一次性计划任务，也可以通过 batch 实现周期性计划任务。

一、通过 at 命令创建任务

at 命令主要用于创建临时的任务，创建的任务只能被执行一次。

以下面以 ubuntu 为例演示 at 命令的使用：

/home/fnngj/test/ 目录下创建 file.py 文件

```
#!/usr/bin/python
#coding=utf-8
```

```
f=open('f.txt','w')
f.write('hello world!!!')

f.close()
```

以写 ('w') 操作打开当前目录下的 f.txt 文件 (没有此文件会自动创建), 向文件中写入 ‘hello world!’, 然后 close()关闭文件。

通过 at 执行创建的 file.py 程序。

```
fnngj@fnngj-VirtualBox:~/test$ at now+5 minutes
warning: commands will be executed using /bin/sh
at> python /home/fnngj/test/file.py
at> <EOT>      Ctrl+d 保存退出
job 17 at Wed Jan  8 17:56:00 2014
```

now+5 minutes 表示当前时间, 5分钟之后执行

python /home/fnngj/test/file.py 要执行的文件, python 命令, 文件要完整的路径

Ctrl+d 保存退出

查看创建的任务

```
fnngj@fnngj-VirtualBox:~/test$ at -l
18      Wed Jan  8 17:56:00 2014 a fnngj
fnngj@fnngj-VirtualBox:~/test$ atq
10      Wed Jan  8 12:57:00 2014 a fnngj
```

at -l / atq 两个命令查看 at 创建的任务。

删除已经设置的任务

```
fnngj@fnngj-VirtualBox:~/test$ atrm 10
fnngj@fnngj-VirtualBox:~/test$
```

10 是任务的“编号”, atrm 用于删除已经创建的任务

启动 atd 进程

linux 一般默认会启动 atd 进行

```
fnngj@fnngj-VirtualBox:~/test$ ps -ef | grep atd
daemon      806      1  0 16:33 ?        00:00:00 atd
fnngj     3300  1882  0 19:07 pts/1    00:00:00 grep --color=auto atd
```

上面表示 atd 进程已经启动

```
fnngj@fnngj-VirtualBox:~/test$ /etc/init.d/atd status   --启动 atd 进程
```

at 命令指定时间的方式

绝对计时方法:

- midnight noon teatime
- hh:mm [today]
- hh:mm tomorrow
- hh:mm 星期
- hh:mm MM/DD/YY

相对计时方法:

- now+n minutes
- now+n hours
- now+n days

用法:

指定在今天下午17: 30执行某命令（假设现在时间是下午14:30, 2014年1月11日）

命令格式:

- at 5:30pm
- at 17:30
- at 17:20 today
- at now+3 hours
- at now+180 minutes
- at 17:30 14.1.11
- at 17:30 1.11.14

查看 f.txt 文件内容

```
fnngj@fnngj-VirtualBox:~/test$ ls
f.txt~    f.txt    open.py  open.py~
fnngj@fnngj-VirtualBox:~/test$ cat f.txt
hello world!!
```

二、通过 crontab 命令创建任务

crontab 可以方便的用来创建周期性任务，也许你想每天某个时间执行 python 程序，或每周五的某个时间执行。crontab 像 windows 的计划任务一样方便，或者更加灵活。

file_time.py

```
#!/usr/bin/python
#coding=utf-8
import time

f=open('123.txt','a')
now = time.strftime('%Y-%m-%d-%H_%M_%S',time.localtime(time.time()))
f.write('file run time: '+now+'\n')

f.close()
```

这次，我们以追加的方式，获取当前时间写入到123.txt 文件中。也就是说程序每运行一次，获取一次当前时间追加（不是替换）写入到123.txt 文件中。

运行一次 file_time.py

```
fnngj@fnngj-VirtualBox:~/test$ python file_time.py
```

查看123.txt 文件内容

```
fnngj@fnngj-VirtualBox:~/test$ cat 123.txt
```

```
file run time: 2014-01-09-17_53_17
```

下面通过 crontab 来创建任务：

为更快的看到任务是否被多次执行的效果，我们要求 file_time.py 每小时过5分钟执行一次。

```
fnngj@fnngj-VirtualBox:~/test$ crontab -e
```

crontab: installing new crontab

输入 crontab - e 命令进入 crontab 文件:

```
# Edit this file to introduce tasks to be run by cron.  
#  
# Each task to run has to be defined through a single line  
# indicating with different fields when the task will be run  
# and what command to run for the task  
#  
# To define the time you can provide concrete values for  
# minute (m), hour (h), day of month (dom), month (mon),  
# and day of week (dow) or use '*' in these fields (for 'any').#  
# Notice that tasks will be started based on the cron's system  
# daemon's notion of time and timezones.  
#  
# Output of the crontab jobs (including errors) is sent through  
# email to the user the crontab file belongs to (unless redirected).  
#  
# For example, you can run a backup of all your user accounts  
# at 5 a.m every week with:  
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/  
#  
^G 求助      ^O 写入      ^R 读档      ^Y 上页      ^K 剪切文字      ^C 游标位置  
^X 离开      ^O 对齐      ^W 搜索      ^V 下页      ^U 还原剪切      ^T 拼写检查
```

图8.14

按键盘 i、o、a 任意一个键进入编辑状态，可以对文件进行修改。

```

fnngj@fnngj-VirtualBox: ~/test
GNU nano 2.2.6      文件: /tmp/crontab.K8vzDO/crontab

# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow   command
5 * * * *      python /home/fnngj/test/file_time.py

```

^G 求助 ^C 写入 ^R 读档 ^Y 上页 ^K 剪切文字 ^C 游标位置
 ^X 离开 ^J 对齐 ^W 搜索 ^V 下页 ^U 还原剪切 ^T 拼写检查

图8.15

分钟	小时	天	月	星期	命令/脚本
5	*	*	*	*	python /home/fnngj/test/file_time.py

按照上面的格式写入内容。

ctrl+x 离开，提示是否保存任务？按 y 保存任务退出。（[不同版本 linux 对 crontab 文件的编辑/退出会有差异。](#)）

完成 crontab 任务创建后，会有如下提示：

crontab: installing new crontab

启动 crontab 服务：

注意：在完成编辑以后，要重新启动 cron 进程，crontab 服务操作说明：

```

~# /etc/init.d/cron restart //重启服务
~# /etc/init.d/cron start //启动服务
~# /etc/init.d/cron stop //关闭服务
~# /etc/init.d/cron reload //重新载入配置

```

查看 crontab 任务计划:

```
root@fnngj-VirtualBox:~# cd /var/spool/crontabs/
root@fnngj-VirtualBox:/var/spool/cron/crontabs# ls
fnngj  root
root@fnngj-VirtualBox:/var/spool/cron/crontabs# cat fnngj  ---因为我们的计划是用 fnngj 用户创建
.....
# m h  dom mon dow    command
5    *   *   *   *      python /home/fnngj/test/file_time.py
```

查看123.txt 文件:

```
root@fnngj-VirtualBox:/home/fnngj# cat 123.txt
file run time: 2014-01-10-10_05_01
file run time: 2014-01-10-11_05_01
file run time: 2014-01-10-12_05_01
file run time: 2014-01-10-13_05_02
```

在创建完任务后，你可能需要等上一段时间才能看到文件中被写入的内容。

crontab 格式说明:

crontab 的命令格式

crontab {-l|-r|-e}

-l 显示当前的 crontab

-r 删除当前的 crontab

-e 使用编辑器编辑当前 crontab 文件

好多人都觉得周期计划任务设置起来比较麻烦，其实我们只要掌握规律就很好设置。



图8.16

在以上各个字段中，还可以使用以下特殊字符：

星号 (*): 代表所有可能的值，例如 month 字段如果是星号，则表示在满足其它字段的制约条件后每月都执行该命令操作。

逗号 (,): 可以用逗号隔开的值指定一个列表范围，例如，“1, 2, 5, 7, 8, 9”

中杠 (-): 可以用整数之间的中杠表示一个整数范围，例如“2-6”表示“2, 3, 4, 5, 6”

正斜线 (/): 可以用正斜线指定时间的间隔频率，例如“0-23/2”表示每两小时执行一次。同时正斜线可以和星号一起使用，例如*/10，如果用在 minute 字段，表示每十分钟执行一次。

实例：

规则： 把知道的具体的时间添上，不知道的都添加上*

分钟 小时 天 月 星期 命令/脚本

假如，我们每天早上4点要做一下操作，以下面方式表示：

分钟 小时 天 月 星期 命令/脚本
 * 4 * * * 【具体的操作】

假如，我们每周一和三下午的6点要做一下操作，以下面方式表示：

分钟	小时	天	月	星期	命令/脚本
*	18	*	*	1, 3	【具体的操作】

在上学的时候都有上机课，周一到周五，下午5点30上课结果。我们需要在5点30发一个通知，5点45自动关机。设定计划任务需要分两步完成，第一步提醒，第二步关机

分钟	小时	天	月	星期	命令/脚本
30	17	*	*	1-5	/usr/bin/wall < /hz/test/guanji.wall
45	17	*	*	1-5	/usr/bin/shutdown -h now

在 linux 下设置定时任务，本书默认读者是具备 linux 操作使用的能力的，如文件的基本操作，vi/vim 编辑器的基本使用。

第五节、WebDriver 方法二次封装

在自动化脚本越写越多的时候，发现 WebDriver 的 API 提供给我们的方法并不好用，例如 webdriver 所提供的元素定位方法又长又难写。这个时候我们就可以对这些方法做个简易的二次封装，使其更易于被使用。

这里我们抛弃之前的测试结构，从一个最简单的脚本来分析如何对 WebDriver 的方法做二次封装。这里依然使用百度搜索的例子。

```
# coding = utf-8
from selenium import webdriver

browser = webdriver.Firefox()
browser.get("http://www.baidu.com")

browser.find_element_by_id("kw").send_keys("selenium")
browser.find_element_by_id("su").click()

browser.quit()
```

在一个脚本中被最频繁使用的应该是 webdriver 的定位方法了，如 find_element_by_id() 定位方法又长又难写，我们就可以考虑把这样的方法再次封装到一个方法里调用。封装后的脚本如下：

```
# coding = utf-8
from selenium import webdriver

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

#封装 find_element_by_id() 方法
def findId(id):
    f = driver.find_element_by_id(id)
    return f

#调用 findId() 方法
findId("kw").send_keys("selenium")
findId("su").click()

driver.quit()
```

被二次封装后的 `findId()`方法最原始的调用还是 `find_element_by_id()` 方法，但 `findId()`的写法确实更加简便，从而并没有降低对方法的理解。

我们现在希望把封装的方法给所有的脚本所调用，所以，我们需要把这些方法封装到一个单独的文件中，在些之些我们先找出 `Element` 类中所提供的所有定位方法。

定位单个元素：

```
find_element_by_id()
find_element_by_name()
find_element_by_class_name()
find_element_by_tag_name()
find_element_by_link_text()
find_element_by_partial_link_text()
find_element_by_xpath()
find_element_by_css_selector()
```

定位一组元素：

```
find_elements_by_id()
find_elements_by_name()
find_elements_by_class_name()
find_elements_by_tag_name()
find_elements_by_link_text()
```

```
find_elements_by_partial_link_text()  
find_elements_by_xpath()  
find_elements_by_css_selector()
```

找出这些方法之后，创建 package 目录，在目录下创建 location.py 文件：

```
#coding=utf-8  
from selenium import webdriver  
  
'''  
本文件简易的封装定位单个元素和定位一组元素的方法  
'''  
  
'''定位单个元素封装'''  
def findId(driver,id):  
    f = driver.find_element_by_id(id)  
    return f  
  
def findName(driver,name):  
    f = driver.find_element_by_name(name)  
    return f  
  
def findClassName(driver,name):  
    f = driver.find_element_by_class_name(name)  
    return f  
  
def findTagName(driver,name):  
    f = driver.find_element_by_tag_name(name)  
    return f  
  
def findLinkText(driver,text):  
    f = driver.find_element_by_link_text(text)  
    return f  
  
def findPLinkText(driver,text):  
    f = driver.find_element_by_partial_link_text(text)  
    return f  
  
def findXpath(driver,xpath):  
    f = driver.find_element_by_xpath(xpath)  
    return f  
  
def findCss(driver,css):  
    f = driver.find_element_by_css_selector(css)
```

```

    return f

'''定位一组元素封装'''
def findsId(driver,id):
    f = driver.find_elements_by_id(id)
    return f

def findsName(driver,name):
    f = driver.find_elements_by_name(name)
    return f

def findsClassName(driver,name):
    f = driver.find_elements_by_class_name(name)
    return f

def findsTagName(driver,name):
    f = driver.find_elements_by_tag_name(name)
    return f

def findsLinkText(driver,text):
    f = driver.find_elements_by_link_text(text)
    return f

def findsPLinkText(driver,text):
    f = driver.find_elements_by_partial_link_text(text)
    return f

def findsXpath(driver,xpath):
    f = driver.find_elements_by_xpath(xpath)
    return f

def findsCss(driver,css):
    f = driver.find_elements_by_css_selector(css)
    return f

```

方法封装完成下面就可以直接在具体的测试脚本中调用了。

baidu.py

```

#coding=utf-8
from selenium import webdriver
import time
import sys
sys.path.append("\package")

```

```
from package import location

#调用 location.py 文件的定位方法
we = location

dr = webdriver.Chrome()
dr.get('http://www.baidu.com')

#调用封装的方法
we.findId(dr, "kw").send_keys('selenium')
time.sleep(2)
we.findId(dr, "su").click()
time.sleep(2)

dr.quit()
```

```
findId(dr,"kw")
```

这定位方法的传参与上面例子稍有差别，上面的例子中我们只传入了定位的元素属性 `findId("kw")`，而这里却转输入一个参数 `dr`（浏览器驱动），因为我们同样的脚本需要在不同的浏览器下运行，那么在封装的方法里就不能把驱动写死，所以，我们需要对调用的定位方法传入不同的浏览器驱动。

下面以本节的内容为基础，来封装你的 `webdriver api` 里的方法吧。

第九章 selenium grid2 分布式执行测试用例

我们在前面的章节中曾介绍过如何通过 `python` 的多线程来并行的在一台电脑上运行测试用例，**Selenium Grid** 允许用户将测试案例分布在几台机器上并行执行。用户可以在一个集中控制点控制不同的环境。在不同的浏览器 / 系统组合上面更为容易的运行测试案例。允许用户更多的利用虚拟资源减少了维护测试环境的成本。

Selenium-Grid 版本

`selenium-grid` 分为版本 1 和版本 2，其实它的 2 个版本并不是和 `selenium` 的版本 1 和 2 相对应发布的（即 `selenium-grid2` 的发布比 `selenium2` 要晚一点）。不过幸运的是现在的 `selenium-grid2` 基本能支持 `selenium2` 的所有功能了。

`selenium` 虽然分 1 和 2，但其实原理和基本工作方式都是一样的。只是版本 2 同时支持 `selenium1` 和 `selenium2` 两种协议，并且在一些小的功能和易用性上进行了优化。比如：指定测试平台的方式。

`selenium grid2` 已经集成到 `selenium server` 中了（即 `selenium-server-standalone-XXX.jar` 包中）所以，我们不用单独下载与安装 `selenium grid`。

第一节、selenium1 与 2 工作原理

在介绍 `selenium grid` 的工作原理之前，我们有必要先理解 `selenium 1` 与 `selenium 2` 的工作原理的差异，这将有助于帮助我们理解和使用 `selenium grid`。

Selenium 1 工作原理

`selenium1` 中除了使用 `selenium-core` 以外，进行自动化测试时都需要使用 `selenium-RC` 来作为代理（不管是本机还是远程），目的是为了解决同源问题；而造成同源问题的原因是因为 `selenium1` 中是使用 `Javascript` 来驱动测试执行的（浏览器由于安全问题不允许不同域之间的 JS 调用，即非同源不可调用；而 `selenium1` 中的工作方式就是在宿主页面注入 JS 并且通过调用 JS 来执行测试操作的，所以就涉及到同源问题）。所以为了达成目的，其解决方案就有 2 种：

1、使用 selenium-core:

selenium-core 是一组 js 库，用来驱动浏览器操作的所有库文件都在这里，整个 **selenium1** 可以认为核心组件就是这个 **selenium-core**; 而使用 **selenium-core** 的方式就是在被测试站点程序的源码里把 **selenium-core** 中的所有 js 库直接添加到页面里，这样页面正常加载的同时也会把 **selenium-core** 加载下来，并且天生就是同源的。

2、使用 selenium-RC:

RC 是一个 http 代理程序，用来注入到浏览器和被测 web 程序之间，这样浏览器所有的请求和接收的内容都会通过 **RC**; **RC** 会把浏览器的请求发送给真实的 **web** 程序，而在接收到 **web** 程序的响应内容时，并没有把内容原原本本的返回给浏览器客户端，而是把包含 **selenium-core** 的内容注入到响应内容中去，然后才发送响应内容给浏览器，这样就通过欺骗的方式让浏览器认为 **selenium1** 的驱动类库同样是同源的。

Selenium2 工作原理

selenium2 中因为使用的 **webdriver**，这个技术不是靠 **js** 驱动的，而是直接调用浏览器的原生态接口驱动的。所以就没有同源问题，也就不需要使用 **RC** 来执行本地脚本了（当然缺点就是并不是所有的浏览器都有提供很好的驱动支持，但 **JS** 却是所有浏览器都通用的）。所以 **selenium2** 中执行本地脚本的方式是：通过本地 **webdriver** 驱动直接调用本地浏览器接口就可以了。

Selenium 1 与 selenium 2 代码对比

我们可以通过 **selenium IDE** 将录制的脚本导入出成 **selenium 1** 和 **selenium 2** 两种格式做对比（**selenium IDE** 录制并导出脚本参考前面章节），按照惯例以录制百度人搜索为例：

Selenium 1 （RC） 代码:

```
from selenium import selenium

import unittest, time, re


class serc(unittest.TestCase):

    def setUp(self):

        self.verificationErrors = []

        self.selenium = selenium("localhost", 4444, "*chrome", "http://www.baidu.com/")

        self.selenium.start()
```

```

def test_serc(self):

    sel = self.selenium

    sel.open("/")
    sel.type("id=kw", "selenium grid")
    sel.click("id=su")
    sel.wait_for_page_to_load("30000")

def tearDown(self):
    self.selenium.stop()
    self.assertEqual([], self.verifications)

if __name__ == "__main__":
    unittest.main()

```

self.selenium = selenium("localhost", 4444, "*chrome", "http://www.baidu.com/")

明显的差异是在通过 selenium 操作之间我们要先指定代理的电脑，localhost 表示本机，4444 表示端口号，以及运行的浏览器"*chrome" 和访问的网址"http://www.baidu.com/"。

Selenium 2 (webdriver) 代码:

```

from selenium import webdriver

from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re

class Sewd(unittest.TestCase):

```

```

def setUp(self):

    self.driver = webdriver.Firefox()

    self.driver.implicitly_wait(30)

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

    selfverificationErrors = []

    self.accept_next_alert = True


def test_sewd(self):

    driver = self.driver

    driver.get(self.base_url + "/")

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

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

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


def tearDown(self):

    self.driver.quit()

    self.assertEqual([], self.verificationErrors)

if __name__ == "__main__":
    unittest.main()

```

Selenium 2 (webdriver) 的代码我们已经非常熟悉了，由于是直接调用浏览器的原生态接口驱动的，所以我们并不需要指定代理电脑及端口。这就是 selenium 1 与 selenium 2 最本质的区别。

selenium 2 调用远程环境

在 selenium 1 中调用远程环境是非常简单的，由于其本身就运行于 selenium server 之上，只用在代码中修改 主机地址及端口号即可。

在使用 selenium 2 由于其默认本地运行测试时是不需要 selenium server 的，但并不总是只执行本地测试的脚本，有时候可能需要在本地调用远程的环境来执行测试，（比如：因为测试环境覆盖原因）此时就需要一个类似 selenium1 中的 RC 来承担这个任务，也就是 selenium2 中的 selenium-server。

selenium-server 支持接收远程脚本的调用命令，然后操作其宿主机上的浏览器来到远程执行测试的任务。当然 **selenium-server** 为了兼容 **selenium1** 的脚本，它同样也支持 **seleniumRC** 所支持的功能（即能接收 **selenium1** 的调用命令）。在本地调用远程机器执行测试的代码是这样的：

```
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

driver = webdriver.Remote(
    command_executor=' http://127.0.0.1:4444/wd/hub' ,
    desired_capabilities=DesiredCapabilities.CHROME)

driver = webdriver.Remote(
    command_executor=' http://127.0.0.1:4444/wd/hub' ,
    desired_capabilities=DesiredCapabilities.OPERA)
```

第二节、selenium server 环境配置

由于 **selenium e RC** 是以代理的方式运行的，所以我们必须要借助于 **selenium server** 才能运行。**Selenium server** 的运行又依赖于 **java** 环境，所以我们必须安装 **java** 环境并启动 **selenium server** 运行 **selenium RC**。

要想在 **webdriver** 中运行远程环境就必须要安装 **selenium server**，要想运行 **selenium server** 同样也需要安装 **java** 环境，下面跟笔者一起配置 **selenium server** 环境。

第一步、下载 **java** 及配置环境

下载地址：http://www.java.com/zh_CN/download/manual.jsp

小知识：

java 环境分 **JDK** 和 **JRE**，**JDK** 就是 **Java Development Kit**.简单的说 **JDK** 是面向开发人员使用的 **SDK**，它提供了 **Java** 的开发环境和运行环境。**JRE** 是 **Java Runtime Environment** 是指 **Java** 的运行环境，是面向 **Java** 程序的使用者，而不是开发者。

请选择适合本机的版本进行下载。以 **windows** 为例 **java** 环境为 **exe** 程序，安装默认路径即可。作者

默认安装在 C:\Program Files\Java\jdk1.7.0_45\路径下。下面设置环境变量：

“我的电脑”右键菜单--->属性--->高级--->环境变量--->系统变量：

变量名：JAVA_HOME

变量值：C:\Program Files\Java\jdk1.7.0_45\

变量名：PATH

变量值：%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

变量名：CLASS_PATH

变量值：.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;

在 windows 命令提示符下输入 java 回车，果显示 java 用法及参数，说明环境配置成功。

第二步、下载运行 selenium server

下载地址：<https://code.google.com/p/selenium/>

在页面的左侧列表中找到 selenium-server-standalone-XXX.jar 进行下载。下载完成可以放到任意位置，直接在命令提示符下启动 selenium server：

C:\selenium> java -jar selenium-server-standalone-XXX.jar

```

C:\WINDOWS\system32\cmd.exe - java -jar selenium-server-standalone-2.... -x
C:\Documents and Settings\Administrator>cd ..

C:\Documents and Settings>cd ..

C:\>cd selenium

C:\selenium>java -jar selenium-server-standalone-2.33.0.jar
一月 21, 2014 11:07:26 上午 org.openqa.grid.selenium.GridLauncher main
INFO: Launching a standalone server
11:07:26.703 INFO - Java: Oracle Corporation 23.21-b01
11:07:26.703 INFO - OS: Windows XP 5.1 x86
11:07:26.703 INFO - v2.33.0, with Core v2.33.0. Built from revision 4e90c97
11:07:26.796 INFO - RemoteWebDriver instances should connect to: http://127.0.0.
1:4444/wd/hub
11:07:26.796 INFO - Version Jetty/5.1.x
11:07:26.796 INFO - Started HttpContext[/selenium-server/driver,/selenium-server
/driver]
11:07:26.796 INFO - Started HttpContext[/selenium-server,/selenium-server]
11:07:26.796 INFO - Started HttpContext[/,/]
11:07:26.843 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@f7d41f

11:07:26.843 INFO - Started HttpContext[/wd,/wd]
11:07:26.843 INFO - Started SocketListener on 0.0.0.0:4444
11:07:26.843 INFO - Started org.openqa.jetty.Server@164de59

```

图 9.1

运行脚本：

```
#coding=utf-8

import time

from selenium import webdriver

from selenium.webdriver.common.desired_capabilities import DesiredCapabilities


#指定运行主机与端口号

driver = webdriver.Remote(

    command_executor='http://127.0.0.1:4444/wd/hub',

    desired_capabilities=DesiredCapabilities.CHROME)


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

driver.find_element_by_name("q").send_keys("hello")

driver.find_element_by_id("qb").click()

time.sleep(2)

driver.close()
```

查看 selenium server 日志就发现， selenium server 记录了客户端与服务器端交互的日志信息：

```

C:\WINDOWS\system32\cmd.exe - java -jar selenium-server-standalone-2.... ->
scriptEnabled=true, browserName=chrome, version=>]
Starting ChromeDriver (v2.8.241075) on port 64247
11:39:08.218 INFO - Done: /session
11:39:08.218 INFO - Executing: org.openqa.selenium.remote.server.handler.GetSessionCapabilities@16946d4 at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682>
11:39:08.218 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682
11:39:08.234 INFO - Executing: [get: http://www.youdao.com] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/url
11:39:09.109 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/url
11:39:09.109 INFO - Executing: [find element: By.name: q] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element
11:39:09.140 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element
11:39:09.140 INFO - Executing: [send keys: 0 org.openqa.selenium.support.events.EventFiringWebDriver$EventFiringWebElement@b7c5f9e8, [h, e, l, l, o]] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element/0/value
11:39:09.187 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element/0/value
11:39:09.187 INFO - Executing: [find element: By.id: qb] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element
11:39:09.203 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element
11:39:09.203 INFO - Executing: [click: 1 org.openqa.selenium.support.events.EventFiringWebDriver$EventFiringWebElement@b7c5f9e9] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element/1/click
11:39:10.140 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/element/1/click
11:39:12.140 INFO - Executing: [close window] at URL: /session/d12d6417-6cbf-4b50-8e65-28319326a682/window
11:39:13.843 INFO - Done: /session/d12d6417-6cbf-4b50-8e65-28319326a682/window

```

图 9.2

第三节、selenium Grid 工作原理

到此为止，其实还没有提到 selenium-grid，因为到目前为止我们还没有需求说同时覆盖多个平台和浏览器，而 selenium-grid 在这种情况下就会体现出其作用来。selenium-grid 是用于设计帮助我们进行分布式测试的工具，其整个结构是由一个 hub 节点和若干个代理节点组成。hub 用来管理各个代理节点的注册和状态信息，并且接受远程客户端代码的请求调用，然后把请求的命令再转发给代理节点来执行。使用 selenium-grid 远程执行测试的代码与直接调用 Selenium-Server 是一样的(只是环境启动的方式不一样，需要同时启动一个 hub 和至少一个 node):

```

> java -jar selenium-server-standalone-x.xx.x.jar -role hub
> java -jar selenium-server-standalone-x.xx.x.jar -role node

```

上面是启动一个 hub 和一个 node，若是同一台机器要启动多个 node 则要注意端口分配问题，可以这

样来启动多个 node:

```
> java -jar selenium-server-standalone-x.xx.x.jar -role node -port 5555
> java -jar selenium-server-standalone-x.xx.x.jar -role node -port 5556
> java -jar selenium-server-standalone-x.xx.x.jar -role node -port 5557
```

调用 Selenium-Grid 的基本结构图如下:

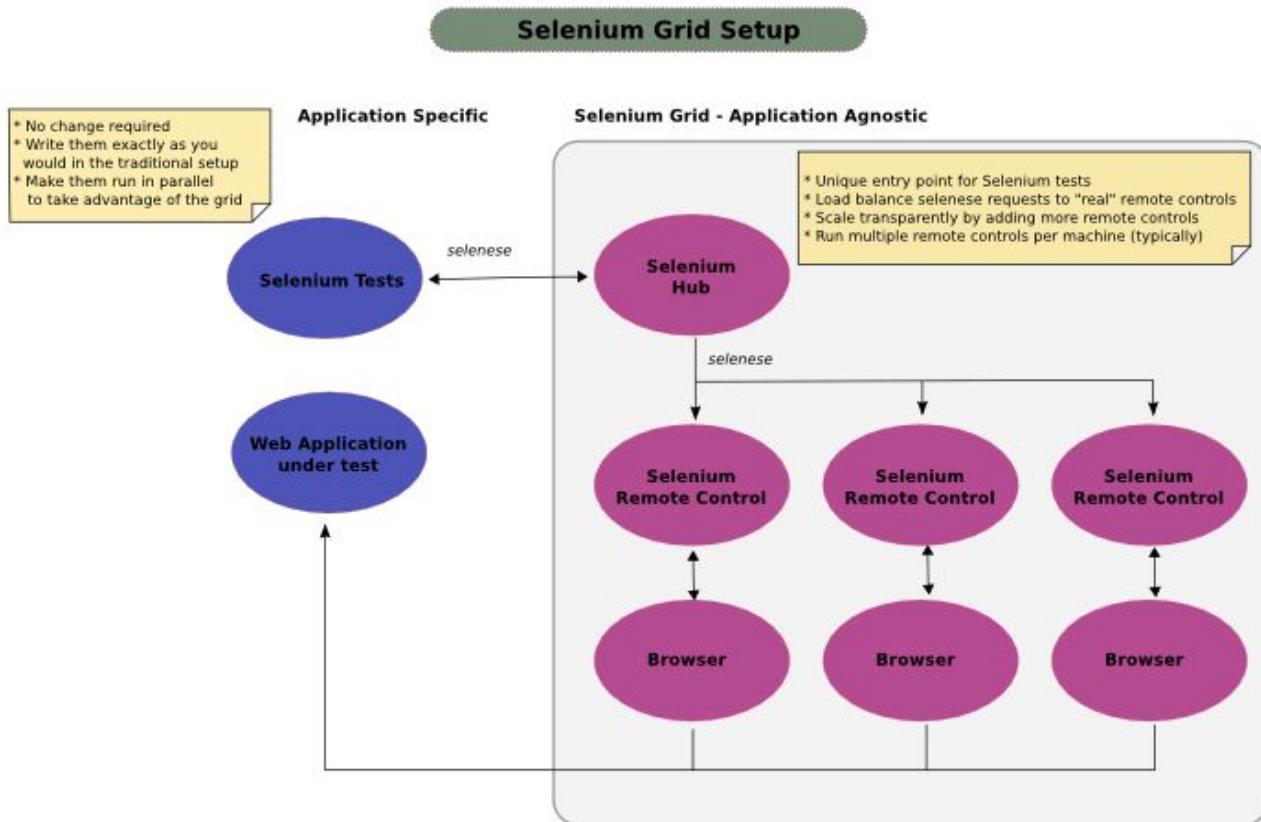


图 9.3

上面是使用 **selenium-grid** 的一种普通方式，仅仅使用了其支持的分布式执行的功能，即当你同时需要测试用例比较多时，可以平行的执行这些用例进而缩短测试总耗时；除此之外，**selenium-grid** 还支持一种更友好的功能，即可以根据你用例中启动测试的类型来相应的把用例转发给符合匹配要求的测试代理。例如你的用例中指定了要在 Linux 上 FireFox 的 17 版本进行测试，那么 **selenium-grid** 会自动匹配注册信息为 Linux、且安装了 FireFox17 的代理节点，如果匹配成功则转发测试请求，如果失败则拒绝请求。使用 **selenium-grid** 的远程兼容性测试的代码同上。其调用的基本结构图如下：

Selenium Grid : Requesting a Specific Environment

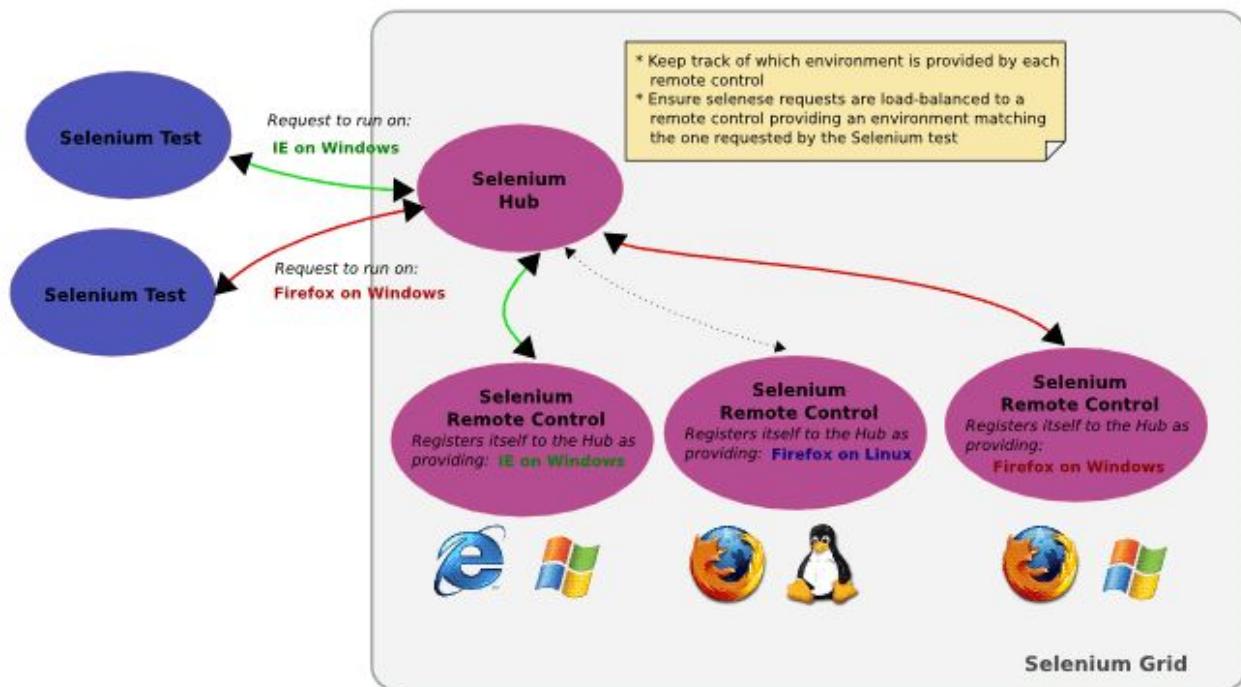


图 9.4

第四节、selenium Grid 应用

9.4.1、多浏览器执行用例

相信读者一定隐藏到心里一个遗留问题，我们在前面通过 python+webdriver 编写的测试脚本，只是在固定的某一款浏览器下运行，也许你已经尝试对浏览器进行参数化 (webdriver.Firefox、webdriver.Chrome)，浏览器驱动并不是我们普通的数据，我们不能像对待普通数据一样对其进行参数化。

学到 selenium server 之后，我们似乎找到了一些眉目，下面就尝试将一段脚本在不同的浏览器下运行。

首先我们先来了解 webdriver 提供的 Remote 的格式。

```
driver = webdriver.Remote(  
    command_executor=' http://127.0.0.1:4444/wd/hub' ,
```

```
desired_capabilities=DesiredCapabilities.CHROME)
```

' http://127.0.0.1:4444/wd/hub' 可以看作一个字符串，对其进行参数化没有什么困难。

那么 DesiredCapabilities.CHROME 里面包含了什么东西呢？

```
>>> from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
>>> p=DesiredCapabilities.CHROME
>>> print p
{'platform': 'ANY', 'browserName': 'chrome', 'version': '', 'javascriptEnabled': True}
```

我们将 DesiredCapabilities.CHROME 的内容打印输出，发现其本身是一个字典。

'platform': 'ANY' 平台默认可以是任何（window, MAC, android）。

'browserName': 'chrome' 浏览器名字是 chrome 。

'version': '' 浏览器的版本默认为空。

'javascriptEnabled': True javascript 启动状态为 True

python webdriver API 提供了不同平台及浏览器的参数：

```
ANDROID = {'platform': 'ANDROID', 'browserName': 'android', 'version': '',
'javascriptEnabled': True}
CHROME = {'platform': 'ANY', 'browserName': 'chrome', 'version': '',
'javascriptEnabled': True}
FIREFOX = {'platform': 'ANY', 'browserName': 'firefox', 'version': '',
'javascriptEnabled': True}
HTMLUNIT = {'platform': 'ANY', 'browserName': 'htmlunit', 'version': ''}
HTMLUNITWITHJS = {'platform': 'ANY', 'browserName': 'htmlunit', 'version': 'firefox',
'javascriptEnabled': True}
INTERNETEXPLORER = {'platform': 'WINDOWS', 'browserName': 'internet explorer',
'version': '', 'javascriptEnabled': True}
IPAD = {'platform': 'MAC', 'browserName': 'iPad', 'version': '',
'javascriptEnabled': True}
IPHONE = {'platform': 'MAC', 'browserName': 'iPhone', 'version': '',
'javascriptEnabled': True}
SAFARI = {'platform': 'ANY', 'browserName': 'safari', 'version': '',
'javascriptEnabled': True}
PHANTOMJS = {'platform': 'ANY', 'browserName': 'phantomjs', 'version': '',
'javascriptEnabled': True}
OPERA = {'platform': 'ANY', 'browserName': 'opera', 'version': '',
'javascriptEnabled': True}
```

我们现在要做的是对 browserName 参数进行参数化，传入不同的浏览器，使脚本在不同的浏览器下运行。要想运行脚本之前我们需要先启动 selenium server：

```
C:\selenium> java -jar selenium-server-standalone-2.39.0.jar
```

实现脚本如下：

```
#coding=utf-8

import time

from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities

#浏览器数组
lists=['chrome','internet explorer']

#通过不同的浏览器执行脚本
for browser in lists:
    print browser
    driver = webdriver.Remote(
        command_executor='http://127.0.0.1:4444/wd/hub',
        desired_capabilities={'platform': 'ANY',
                             'browserName': browser,
                             'version': '',
                             'javascriptEnabled': True
        })

    driver.get("http://www.youdao.com")
    driver.find_element_by_name("q").send_keys("hello")
    driver.find_element_by_id("qb").click()
    time.sleep(2)

    driver.close()
```

在脚本的执行过程中，我们将看到，第一次启动 chrome 浏览器执行有道搜索的用例，第二次启动 IE 浏览器执行有道搜索的用例。

因为我们在读取 lists 每一条数据时打印，如果会看到如下的运行结果：

```
>>> ===== RESTART =====  
>>>  
chrome  
internet explorer
```

在实际的过程中浏览器的数据会写在一个单独的文件中。请参考本书第四章第三节数据驱动（参数化）的方式对浏览器进行参数化。

9.4.2、多节点执行用例

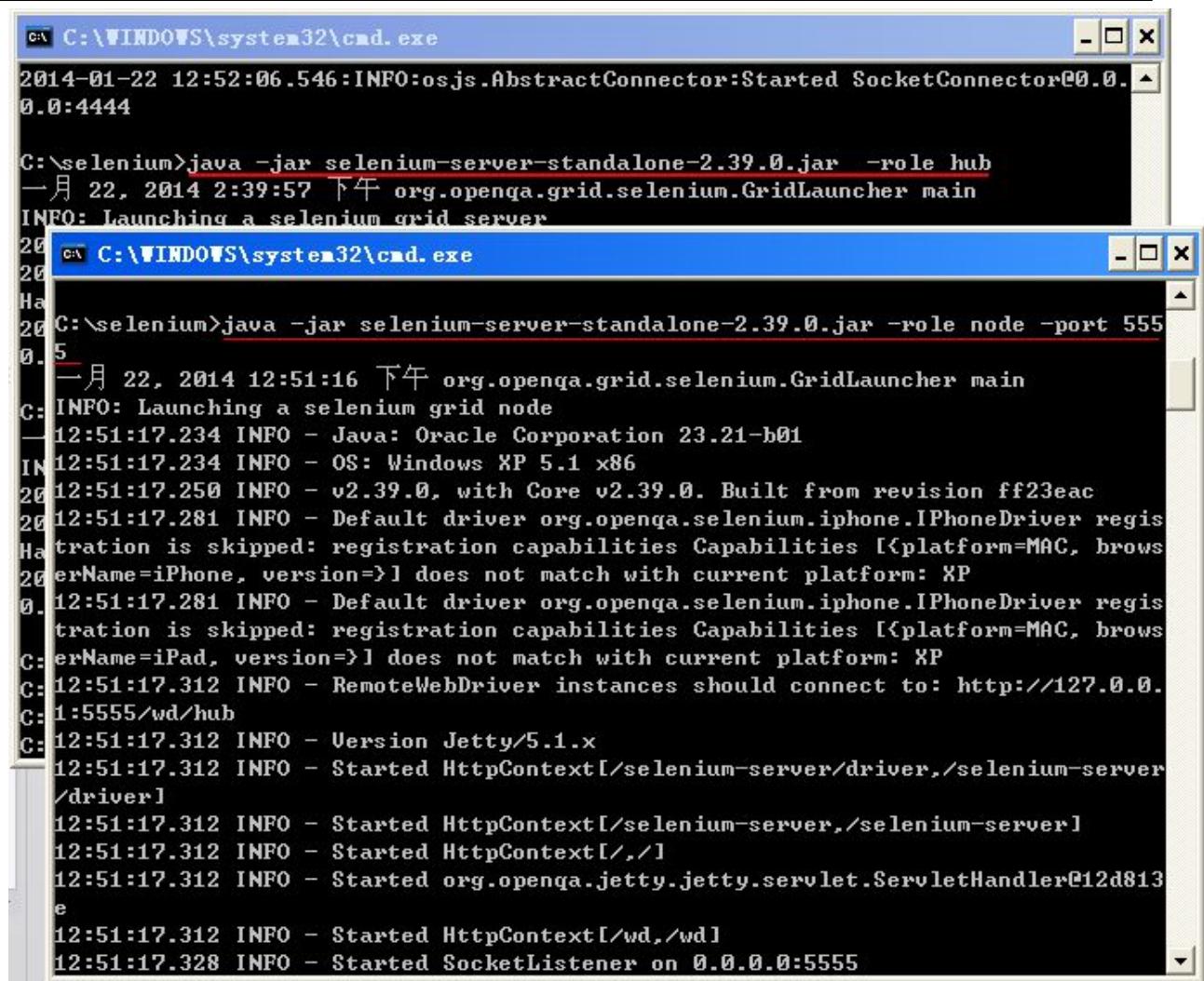
我们来先通过一台电脑演示启动多个节点，运行测试用例。通过 selneium Grid 工作原理一节我们了解到要想启动多节点，前提是必须启动一个 Hub。

在本机打开两个命令提示符窗口分别启动一个 hub 和一个 node（节点）

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
```

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
```

如图 9.5



The screenshot shows two separate command-line windows, both titled 'C:\WINDOWS\system32\cmd.exe'. The top window displays the log for starting a Selenium hub server on port 4444. The bottom window displays the log for starting a Selenium node server on port 5555, which includes detailed logs about driver registrations and Jetty startup.

```

C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
2014-01-22 12:52:06.546:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.0.0:4444
一月 22, 2014 2:39:57 下午 org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid server

C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5555
2014-01-22 12:51:16 下午 org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid node
12:51:17.234 INFO - Java: Oracle Corporation 23.21-b01
12:51:17.234 INFO - OS: Windows XP 5.1 x86
12:51:17.250 INFO - v2.39.0, with Core v2.39.0. Built from revision ff23eac
12:51:17.281 INFO - Default driver org.openqa.selenium.iphone.IPhoneDriver registration is skipped: registration capabilities Capabilities [{platform=MAC, browserName=iPhone, version=>}] does not match with current platform: XP
12:51:17.281 INFO - Default driver org.openqa.selenium.iphone.IPhoneDriver registration is skipped: registration capabilities Capabilities [{platform=MAC, browserName=iPad, version=>}] does not match with current platform: XP
12:51:17.312 INFO - RemoteWebDriver instances should connect to: http://127.0.0.1:5555/wd/hub
12:51:17.312 INFO - Version Jetty/5.1.x
12:51:17.312 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
12:51:17.312 INFO - Started HttpContext[/selenium-server,/selenium-server]
12:51:17.312 INFO - Started HttpContext[/,/]
12:51:17.312 INFO - Started org.openqa.jetty.servlet.ServletHandler@12d813e
12:51:17.312 INFO - Started HttpContext[/wd,/wd]
12:51:17.328 INFO - Started SocketListener on 0.0.0.0:5555

```

图 9.5

通过浏览器访问 grid 的控制台：

<http://127.0.0.1:4444/grid/console>

我们将在浏览器中看到，启动的节信息，如图 10.x；从信息中可看到当前节点的 IP 和 OS 信息。



图 9.6

再次添加新的 node：

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role node -port 5556
```

重新刷新 grid 的控制台。我们发现控制台又多出监控 5556 端口的 node：



图 9.7

下面修改脚本使其在不同的节点与浏览器上运行：

config.py

```
#coding=utf-8
```

```

def getconfig():

    d={'http://127.0.0.1:5556/wd/hub':'chrome',
       'http://127.0.0.1:4444/wd/hub':'internet explorer',
       }

    print "success read computer and browser!!"

    return d

```

se_server.py

```

import time

from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import DesiredCapabilities
import config #导入 config.py 文件

#通过 host,browser 来参数化脚本

for host,browser in config.getconfig().items():

    print host

    print browser

    driver = webdriver.Remote(
        command_executor=host,
        desired_capabilities={'platform': 'ANY',
                             'browserName': browser,
                             'version': '',
                             'javascriptEnabled': True
        })

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

    driver.find_element_by_name("q").send_keys("hello")

    driver.find_element_by_id("qb").click()

    time.sleep(2)

    driver.close()

```

运行结果：

```
>>> ===== RESTART =====
>>>
success read computer and browser!!
http://127.0.0.1:5555/wd/hub
internet explorer
http://127.0.0.1:4444/wd/hub
chrome
```

启动远程 node

我们目前启动的 Hub 与 node 都是在一台主机。那么要在其它主机启动 node 必须满足以下几个要求：

- 本地 hub 主机与远程 node 主机之间可以相互 ping 通。
- 远程主机必须安装运行脚本的浏览器及驱动（如，chrome 浏览器及 chromedriver.exe 驱动）
- 远程主机必须安装 java 环境
- 远程主机必须安装 selenium server

操作步骤如下：

启动本地 hub 主机（**本地主机 IP 为： 172.16.10.66**）：

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar -role hub
```

启动远程主机（**操作系统： ubuntu ， IP 地址： 172.16.10.34**）：

远程主机启动 node 方式如下：

```
fnngj@fnngj-VirtualBox:~/selenium$ java -jar selenium-server-standalone-2.39.0ar -role node -port 5555
-hub http://172.16.10.66:4444/grid/register
```

（设置的端口号为： 5555 ， 指向的 hub ip 为 **172.16.10.66**）

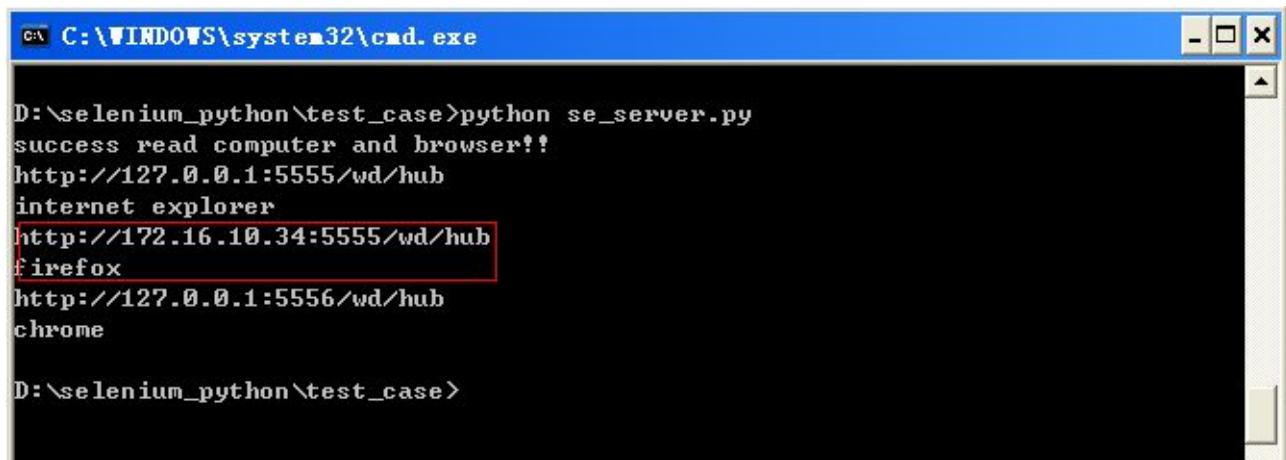
修改 config.py 文件，添加远程主机的操作。

```
#coding=utf-8
...
...
```

```
'http://172.16.10.34:5555/wd/hub':'firefox',
添加远程主机信息，确保 ip 与端口号正确
'''

def getconfig():
    d={'http://127.0.0.1:5556/wd/hub':'chrome',
       'http://127.0.0.1:5555/wd/hub':'internet explorer',
       'http://172.16.10.34:5555/wd/hub':'firefox',
    }
    print "success read computer and browser!!"
    return d
```

现在程序不能在编辑器下运行，需要打开命令提示符来运行 se_server.py 程序。



从运行结果看到，我们调用远程主机运行程序是 OK 的，能过远程主机也可以看到浏览器被成功启动并执行相关操作。

9.4.3、分布式并行运行脚本

Selenium Grid 只是提供多系统、多浏览器的执行环境，Selenium Grid 本身并不提供并行的执行策略。也就是说我们不可能简单地丢给 selenium grid 一个 test case 它就能并行地在不同的平台及浏览器下运行。如果您希望利用 Selenium Grid 的优势，那么您需要编写以并行模式运行的 Selenium 测试。

这里我们需要再次借助 python 的多线程技术来并行的运行脚本。

启动 selenium server

```
C:\selenium>java -jar selenium-server-standalone-2.39.0.jar
```

se_thread.py

```
#coding=utf-8

from threading import Thread

from selenium import webdriver

import time


#配置的 selenium server

def get_browser(caps):

    return webdriver.Remote(

        desired_capabilities=caps,

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

    )

#各平台配置信息

browsers = [

    { "platform": "WINDOWS", "browserName" : "firefox", "version" : '', "name" : "FF" },

    { "platform": "WINDOWS", "browserName" : "internet explorer", "version" : '', "name" : "IE" },

    { "platform": "WINDOWS", "browserName" : "chrome", "name" : "chrome" },]

#执行的脚本

browsers_waiting = []

def get_browser_and_wait(browser_data):

    print "starting %s" % browser_data["name"]

    browser = get_browser(browser_data)

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

    browsers_waiting.append({ "data" : browser_data, "driver" : browser })

    print "%s ready" % browser_data["name"]

    while len(browsers_waiting) < len(browsers):
```

```

print "browser %s sending heartbeat while waiting" % browser_data["name"]

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

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

browser.find_element_by_id("su").click()

time.sleep(3)

#使用多线程运行脚本

thread_list = []

for i, browser in enumerate(browsers):

    t = Thread(target=get_browser_and_wait, args=[browser])

    thread_list.append(t)

    t.start()

for t in thread_list:

    t.join()

print "all browsers ready"

#循环关闭浏览器

for i, b in enumerate(browsers_waiting):

    print "browser %s's title: %s" % (b["data"]["name"], b["driver"].title)

    b["driver"].quit()

```

运行结果：

```

>>>

starting FF
starting IE
starting chrome

chrome ready

browser chrome sending heartbeat while waiting

IE ready

browser IE sending heartbeat while waiting

```

```
browser chrome sending heartbeat while waiting
FF ready
all browsers ready
browser chrome's title: selenium_百度搜索
browser internet explorer's title: selenium_百度搜索
browser fielfox's title: selenium_百度搜索
```

如果想使脚本在不同节点上运行需要对结果做配置，相关代码如下：

```
API_KEY = "KEY"
API_SECRET = "SECRET"

def get_browser(caps):
    return webdriver.Remote(
        desired_capabilities=caps,
        command_executor="http://%s:%s@hub.testingbot.com/wd/hub" % (API_KEY, API_SECRET)
    )
```

总结：

本章介绍 selenium grid 版本问题，以及如何使用 selenium server 启动 hub 多个节点，如何操作脚本在不同平台，不同浏览器下运行。selenium grid 本身并没我们想象的那么神奇，要想顺利的使用的使用 selenium grid 的优势还需要读者做许多功课。

第十章 行为驱动开发 BDD 框架 lettuce 入门

看到 TDD/BDD 一定会感觉高端大气上档次，不是我们普通吊民玩的，最近一直在摸索自动化测试。也想体验并引入 BDD 低调奢华的内涵。于是，在网络上搜索资料；话说这玩艺儿真的不太好理解，尤其对于没有丰富编程经验的同学。

学习 BDD ruby 的 cucumber 是个不错的选择，但我是 python 流的，自然找了来它的兄弟 lettuce，从官方版本（v0.1rc11）来看确实够年轻的，不过由 ruby 的 cucumber 在前面开路，lettuce 应该会发展的很顺利。

lettuce 除了官方文档外，几乎找不到其它资料，为了理解 lettuce，我们不妨多去看看 cucumber 的资料。

lettuce 是一个非常有用的和迷人的 BDD（行为驱动开发）工具。Python 项目的自动化测试，它可以执行纯文本的功能描述，就像 ruby 语言的 cucumber。

lettuce，使开发和测试过程变得很容易，可扩展性，可读性和-什么是最好的-它允许我们用自然语言去描述一个系统的 behavior，你不能想象这些描述可以自动测试你的系统。

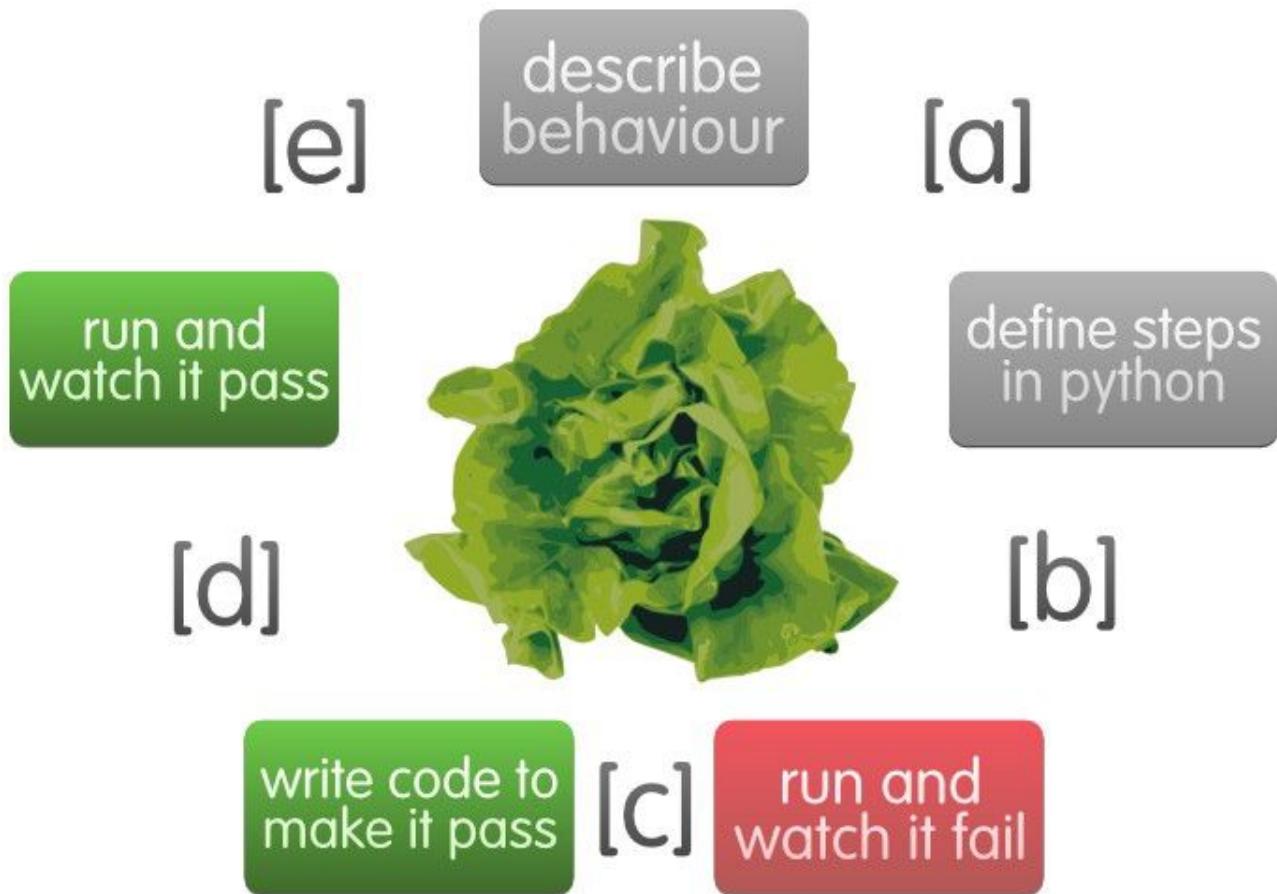


图 10.1

第一节、安装与例子

lettuce 官方网址: <http://lettuce.it/>

安装

请确认你已经安装了 python 以及 pip 安装包管理工具。

不管是 windows 还是 linux 环境，进入 pip 目录，只需下面一个命令就可以安装 lettuce .

```
user@machine:~$ [sudo] pip install lettuce
```

安装好 lettuce 后，如果在终端直接执行 lettuce 命令，得到以下输出：

```
D:\selenium_use_case\lettuce>lettuce
Oops!
could not find features at \features

D:\selenium_use_case\lettuce>
```

图 10.2

哎呀！不能发现 features，lettuce 期望在当前目录下创建 features 子目录

例子（阶乘）

下面就通过官网的例子来领略 lettuce 的风骚。

什么阶乘？

$0! = 1$

$1! = 1$

$2! = 2 \times 1 = 2$

$3! = 3 \times 2 \times 1 = 6$

....

$10! = 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 3628800$

.....

下面是用 python 语言的两种阶乘实现：

```
#coding=utf-8

#循环实现
def f(n):
    c = 1
    for i in range(n):
        i=i+1
        c=c*i
    return c

#递归实现
def f2(n):
    if n > 1:
```

```

    return n*f2(n-1)
else:
    return 1

#调用方法
print f(10)
print f2(10)

```

OK！介于我们理解上面阶乘的基础上，来看看 BDD 是如何实现的。

第二节、lettuce 解析

创建以下目录结构：

`.../tests/features/zero.feature`

`/steps.py`

现在我们来编写 `zero.feature` 文件的内容

```

Feature: Compute factorial
  In order to play with Lettuce
  As beginners
  We'll implement factorial

  Scenario: Factorial of 0
    Given I have the number 0
    When I compute its factorial
    Then I see the number 1

```

我这里对上面的内容做个翻译：

功能：计算阶乘

为了使用 lettuce
作为初学者
我们将实现阶乘

场景：0的阶乘

```
假定我有数字0
当我计算它的阶乘
然后，我看到了1
```

是不是很自然的描述？！第一段功能介绍，我要实现什么功能；第二段场景，也可以看做是一条测试用例，当我输入什么数据时，程序应该返回给我什么数据。

虽然是自然语言，但还是有语法规则的，不然一千个人有一千中描述，程序以什么样的规则去匹配呢？其实它的语法规则非常简单就几个关键字，记住他们的含义及作用即可。

-
- Feature (功能)
 - Scenario (情景)
 - Given (给定)
 - And (和)
 - When (当)
 - Then (则)
-

他们的含义与原有自动化测试框架的概念相似，类比如下：

Lettuce	Unit Test
Feature (功能)	test suite (测试用例集)
Scenario (情景)	test case (测试用例)
Given (给定)	setup (创建测试所需环境)
When (当)	test (触发被测事件)
Then (则)	assert(断言，验证结果)

图 10.3

关于 **feature** 文件的作用，执行以及语法规则将在下一节中详细介绍，这一节主要先来体验 lettuce 的风骚。

有了上面 **zero.feature** 文件的行为做指导，下面打开 **steps.py** 文件来编写我们的程序。

```
from lettuce import *
@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)

@step('I compute its factorial')
```

```

def compute_its_factorial(step):
    world.number = factorial(world.number)

@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected, \
        "Got %d" % world.number

def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number

```

乍一看怎么跟我上面实现阶乘的代码相差甚远呀！不知道你和你的小伙伴有没有惊呆！？好吧，以我拙劣的 **python** 语言水平试着来分析一下，这是啥？这是啥？这又是啥？

```
from lettuce import *
```

引入 lettuce 下面的所有包

```
@step('I have the number (\d+)')
```

@step 字面意思是步骤

I have the number (\d+) 对应的就是 zero.feature 文件中的第六句： Given I have the number 0

(\d+) 是一个正则表达式， \d 表示匹配一个数字， + 表示匹配的数字至少有一个或多个。关于这个可以参考其他 **python** 正则表达式的资料。

第一步：

```

@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)

```

定义一个方法 **have_the_number**，把假设的输入（0）转换成整型放入 **world.number** 中。

第二步：

```
@step('I compute its factorial')
def compute_its_factorial(step):
    world.number = factorial(world.number)
```

把 have_the_number 方法中 world.number 的变量值 (0) 放入 factorial() 方法中，并把结果返再赋值给 world.number 变量。

I compute its factorial 对应的就是 zero.feature 文件中的第七句: When I compute its factorial

第三步:

```
def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number
```

这个是 factorial()方法被调用时的处理过程，对参数的内容转换成整数，判断如果等于0或1的话就直接返回1，否则返回具体的数。(处理结果给了第三步的 world.number)

第四步:

```
@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected,
           "Got %d" % world.number
```

expected 获取的是 zero.feature 文件中预期结果，与第三步处理的实际结果 (world.number) 进行比较；assert 函数进行断言结果是否正确。

I see the number (\d+)对应的也是 zero.feature 文件中的第八句: Then I see the number 1

运行

切换到 tests 目录下，运行 lettuce 命令：

```
fnngj@fnngj-H24X:~/python/lettuce/tests$ lettuce
```

运行结果如下：

```
fnngj@fnngj-H24X:~/python/lettuce/tests$ lettuce
Feature: Compute factorial      # features/zero.feature:1
  In order to play with Lettuce # features/zero.feature:2
  As beginners                 # features/zero.feature:3
  We'll implement factorial    # features/zero.feature:4

  Scenario: Factorial of 0      # features/zero.feature:5
    Given I have the number 0   # features/steps.py:5
    Given I have the number 0   # features/steps.py:5
    When I compute its factorial # features/steps.py:10
    When I compute its factorial # features/steps.py:10
    Then I see the number 1     # features/steps.py:15

1 feature (1 passed)
1 scenario (1 passed)
3 steps (3 passed)
```

图 10.4

运行结果很清晰，首先是 zero.feature 文件里功能描述（feature），然后场景（scenario）每一步所对应 steps.py 文件里的哪一行代码。

最后给出运行结果：

```
Feature(1 passed) 一个功能通过
Scenario(1 passed) 一个场景通过
Steps(3 passed) 三个步骤通过
```

第三节、添加测试场景

下面我们可以在 zero.feature 中多加几个场景（测试用例）：

```
Feature: Compute factorial
  In order to play with Lettuce
  As beginners
  We'll implement factorial

  Scenario: Factorial of 0
    Given I have the number 0
    When I compute its factorial
```

```
Then I see the number 1
```

Scenario: Factorial of 1

```
Given I have the number 1
```

```
When I compute its factorial
```

```
Then I see the number 1
```

Scenario: Factorial of 2

```
Given I have the number 2
```

```
When I compute its factorial
```

```
Then I see the number 2
```

Scenario: Factorial of 3

```
Given I have the number 3
```

```
When I compute its factorial
```

```
Then I see the number 6
```

运行结果：

```

fnngj@fnngj-H24X: ~/python/lettuce/tests
Given I have the number 0      # features/steps.py:5
Given I have the number 0      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 1       # features/steps.py:15

Scenario: Factorial of 1      # features/zero.feature:11
Given I have the number 1      # features/steps.py:5
Given I have the number 1      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 1       # features/steps.py:15

Scenario: Factorial of 2      # features/zero.feature:16
Given I have the number 2      # features/steps.py:5
Given I have the number 2      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 2       # features/steps.py:15

Scenario: Factorial of 3      # features/zero.feature:21
Given I have the number 3      # features/steps.py:5
Given I have the number 3      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 6       # features/steps.py:15
Traceback (most recent call last):
  File "/usr/local/lib/python2.7/dist-packages/lettuce/core.py", line 144, in __call__
    ret = self.function(self.step, *args, **kw)
  File "/home/fnngj/python/lettuce/tests/features/steps.py", line 18, in check_number
    "Got %d" % world.number
AssertionError: Got 3

1 feature (0 passed)
4 scenarios (3 passed)
12 steps (1 failed, 11 passed)
fnngj@fnngj-H24X:~/python/lettuce/tests$ 

```

图 10.5

嗯？第四场景没通过， $3! = 3*2*1=6$ 这个预期结果肯定是正确的，那就是代码的逻辑有问题吧！如果你细心的话一定发现了 setup.py 中的代码并未真正实现阶乘，我们需要对它进行修改：

```

#coding=utf-8
from lettuce import *
@step('I have the number (\d+)')
def have_the_number(step, number):
    world.number = int(number)
    print world.number

```

```

@step('I compute its factorial')
def compute_its_factorial(step):
    world.number = factorial(world.number)
    print world.number

@step('I see the number (\d+)')
def check_number(step, expected):
    expected = int(expected)
    assert world.number == expected, \
        "Got %d" % world.number

def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number*factorial(number-1)

```

代码修改部分：

```

def factorial(number):
    number = int(number)
    if (number == 0) or (number == 1):
        return 1
    else:
        return number*factorial(number-1)

```

参照本文开头，通过递归的方式实现阶乘的代码，现在才算完整的实现阶乘。OK！再来运行以下脚本吧！全绿了有木有！！

```

fnngj@fnngj-H24X: ~/python/lettuce/tests
Given I have the number 1      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 1       # features/steps.py:15

Scenario: Factorial of 2      # features/zero.feature:16
Given I have the number 2      # features/steps.py:5
Given I have the number 2      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 2       # features/steps.py:15

Scenario: Factorial of 3      # features/zero.feature:21
Given I have the number 3      # features/steps.py:5
Given I have the number 3      # features/steps.py:5
When I compute its factorial # features/steps.py:10
When I compute its factorial # features/steps.py:10
Then I see the number 6       # features/steps.py:15

1 feature (1 passed)
4 scenarios (4 passed)
12 steps (12 passed)
fnngj@fnngj-H24X:~/python/lettuce/tests$ 

```

图 10.6

第四节、lettuce 的目录结构与执行过程

Lettuce 是 python 世界的 BDD 框架, 开发人员主要与两类文件打交到, Feature 文件和相应的 Step 文件。Feature 文件是以 `feature` 为后缀名的文件, 以 Given-When-Then 的方式描述了系统的场景 (scenarios) 行为; Step 文件为普通的 python 文件, Feature 文件中的每个 Given/When/Then 步骤在 Step 文件中都有对应的 Ruby 执行代码, 两类文件通过正则表达式相关联。下面笔者大家简单对 lettuce 工程的目录结构和执行过程进行分析。

目前大多数教程都建议采用以下目录结构, 所有的文件 (夹) 都位于 `features` 文件夹下。

`.../tests/features/test.feature`

`/step_definitions/test.py`

`/support/env.py`

Feature 文件(如 `test.feature`)直接位于 `features` 文件夹下, 可以为每个应用场景创建一个 Feature 文件; 与 Feature 文件对应的 Step 文件(如 `test.py`)位于 `step_definitions` 子文件夹下; 同时, 存在

support 子文件夹，其下的 **env.py** 文件为环境配置文件。在这样的目录结构条件下执行 **lettuce** 命令，会首先执行 **env.py** 做前期准备工作，比如可以用 **webdriver** 新建浏览器窗口，然后 **lettuce** 将 **test.py** 文件读入内存，最后执行 **test.feature** 文件，当遇到 **Given/When/Then** 步骤时，**lettuce** 将在 **test.py** 中搜索是否有相应的 **step**，如果有，则执行相应的 **python** 脚本。

这样的目录结构只是推荐的目录结构：对于 **lettuce** 而言，除了顶层的 **features** 文件夹是强制性的之外，其它目录结构都不是强制性的，**lettuce** 将对 **features** 文件夹下的所有内容进行扁平化（**flatten**）处理和首字母排序。具体来说，**lettuce** 在运行时，首先将递归的执行 **features** 文件夹下的所有 **python** 文件(其中则包括 **Step** 文件)，然后通过相同的方式执行 **Feature** 文件。但是，如果 **features** 文件夹下存在 **support** 子文件夹，并且 **support** 下有名为 **env.py** 的文件，**lettuce** 将首先执行该文件，然后执行 **support** 下的其它文件，再递归执行 **features** 下的其它文件。

比如有如下 **lettuce** 目录结构：

```
.../tests/features/a.feature
    /a.py
    /b.feature
    /b.py
    /other/c.feature
    /other/f.py
    /other/g.py
    /setup_definitions/e.py
    /support/c.py
    /support/d.py
    /support/env.py
```

此时执行 **lettuce** 命令，得到以下输出（部分）结果：

```
env.py
c.py
d.py
a.py
b.py
f.py
g.py
e.py
```

上面结果即为 **python** 文件的执行顺序，可以看出，**support** 文件夹下 **env.py** 文件首先被执行，其

次按照字母排序执行 `c.py` 和 `d.py`; 接下来, `lettuce` 将 `features` 文件夹下的所用文件(夹)扁平化, 并按字母顺序排序, 从而先执行 `a.py` 和 `b.py`, 而由于 `other` 文件夹排在 `step_definitions` 文件夹的前面, 所以先执行 `other` 文件夹下的 Ruby 文件(也是按字母顺序执行: 先 `f.py`, 然后 `g.py`), 最后执行 `step_definitions` 下的 `e.py`。

当执行完所有 `python` 文件后, `lettuce` 开始依次读取 `Feature` 文件, 执行顺序也和前述一样, 即:
`a.feature` --> `b.feature` --> `c.feature`

笔者还发现, 这些 `python` 文件甚至可以位于 `features` 文件夹之外的任何地方, 只是需要在位于 `features` 文件夹之内的 `python` 文件中 `require` 一下, 比如在 `env.py` 中。

第五节、lettuce webdriver 自动化测试

下面向读者介绍如何通过 `lettuce` 和 `webdriver` 结合来编写自动化脚本。

第一步, 安装 `lettuce` (在本章第一节已经安装)

```
user@machine:~$ [sudo] pip install lettuce
```

第二步, 安装 `lettuce.webdriver`

https://pypi.python.org/pypi/lettuce_webdriver

```
user@machine:~$ [sudo] pip install lettuce.webdriver
```

第三步, 安装 `nose` (`nose` 继承自 `unittest`, 为 `python` 自动化测试框架, 且更容易使用)

<https://pypi.python.org/pypi/nose/>

```
user@machine:~$ [sudo] pip install lettuce
```

`.../tests/features/google.feature`

`/step_definitions/setps.py`

`/support/terrain.py`

`google.feature`

Feature: Go to google

Scenario: Visit Google

Given I go to "http://www.google.com/"

When I fill in field with class "gsfi" with "selenium"

Then I should see "seleniumhq.org" within 2 second

sets.py

```
from lettuce import *
from lettuce.webdriver.util import assert_false
from lettuce.webdriver.util import AssertContextManager

def find_field_by_class(browser, attribute):
    xpath = "//input[@class='%s']" % attribute
    elems = browser.find_elements_by_xpath(xpath)
    return elems[0] if elems else False

@step('I fill in field with class "(.*?)" with "(.*?)"')
def fill_in_textfield_by_class(step, field_name, value):
    with AssertContextManager(step):
        text_field = find_field_by_class(world.browser, field_name)
        text_field.clear()
        text_field.send_keys(value)
```

terrain.py

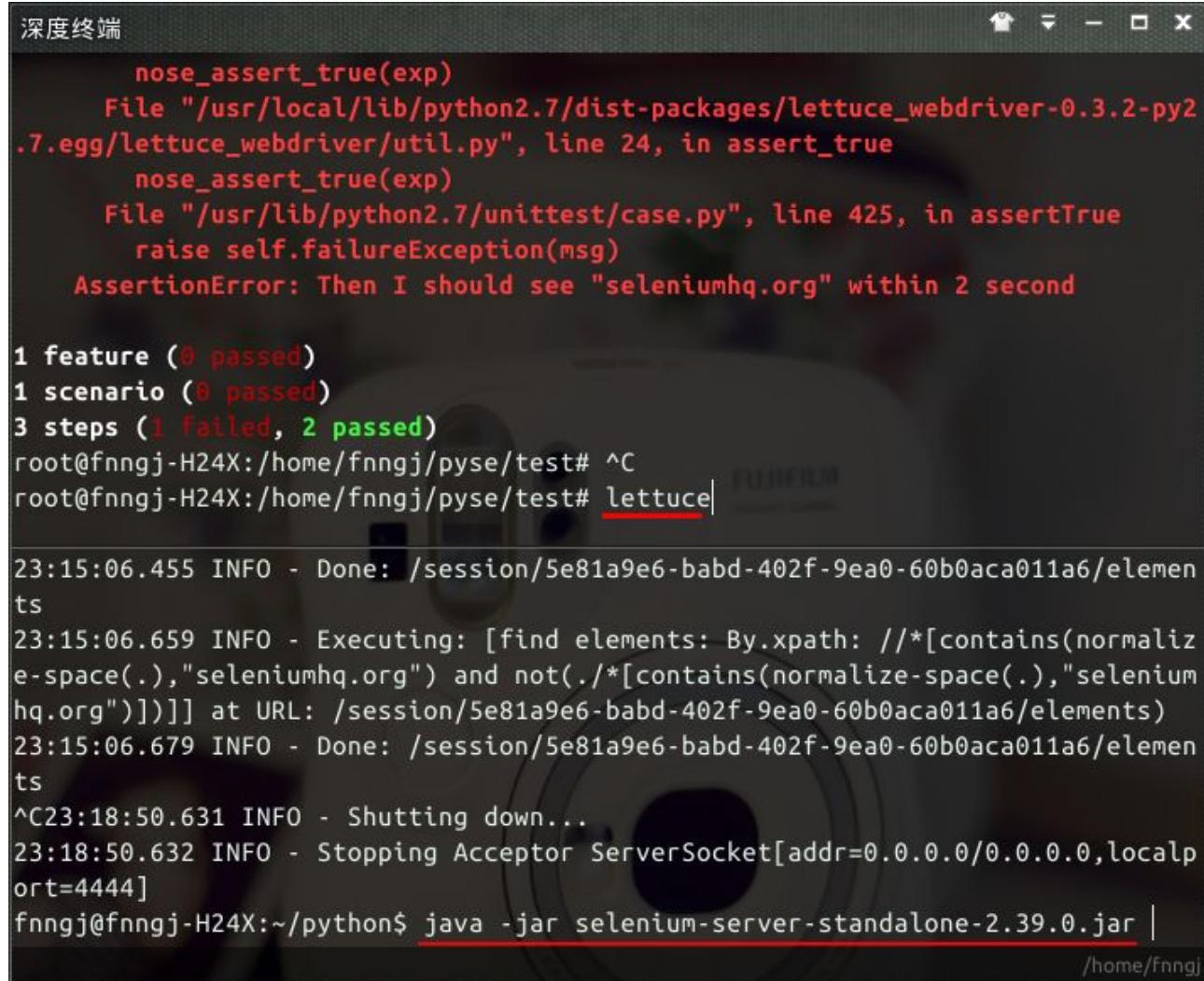
```
from lettuce import before, world
from selenium import webdriver
import lettuce.webdriver.webdriver

@Before.all
def setup_browser():
    desired_capabilities = webdriver.DesiredCapabilities.FIREFOX
    desired_capabilities['version'] =
    desired_capabilities['platform'] = 'ANY'
    desired_capabilities['browserName'] = 'firefox'
    desired_capabilities['avascriptEnabled'] = True

    world.browser = webdriver.Remote(
        desired_capabilities=desired_capabilities,
        command_executor="http://127.0.0.1:4444/wd/hub"
```

)

本例子人运行需要启动 selenium server :



```

深度终端

nose_assert_true(exp)
  File "/usr/local/lib/python2.7/dist-packages/lettuce_webdriver-0.3.2-py2
.7.egg/lettuce_webdriver/util.py", line 24, in assert_true
    nose_assert_true(exp)
  File "/usr/lib/python2.7/unittest/case.py", line 425, in assertTrue
    raise self.failureException(msg)
AssertionError: Then I should see "seleniumhq.org" within 2 second

1 feature (0 passed)
1 scenario (0 passed)
3 steps (1 Failed, 2 passed)
root@fnngj-H24X:/home/fnngj/pyse/test# ^C
root@fnngj-H24X:/home/fnngj/pyse/test# lettuce

23:15:06.455 INFO - Done: /session/5e81a9e6-babd-402f-9ea0-60b0aca011a6/elemen
ts
23:15:06.659 INFO - Executing: [find elements: By.xpath: //*[contains(normaliz
e-space(.),"seleniumhq.org") and not(./*[contains(normalize-space(.),"selenium
hq.org"))])] at URL: /session/5e81a9e6-babd-402f-9ea0-60b0aca011a6/elements)
23:15:06.679 INFO - Done: /session/5e81a9e6-babd-402f-9ea0-60b0aca011a6/elemen
ts
^C23:18:50.631 INFO - Shutting down...
23:18:50.632 INFO - Stopping Acceptor ServerSocket[addr=0.0.0.0/0.0.0.0,localp
ort=4444]
fnngj@fnngj-H24X:~/python$ java -jar selenium-server-standalone-2.39.0.jar |
/home/fnngj

```

参考 selenium grid 一章的学习，启动 selenium server ,执行 lettuce 运行脚本，提示 2passed 和 1failed ，失败语句为。

Then I should see "seleniumhq.org" within 2 second

请读者根据所学习的 lettuce 知识以及前面的内容排查错误的原因并修正。

第十一章 git/getcafe 管理自动化测试项目

当我们的自动化框架在项目的实践中逐渐成熟并形成一定规模之后，自动化脚本开发与维护也不是由一人所能完成的，那么必定会有新人员参与脚本的开发与维护工作。那么多人协同开发必定需要引入版本控制工具来对项目进行控制和管理。

Git 是一个开源的分布式版本控制系统，用以有效、高速的处理从很小到非常大的项目版本管理。Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。相比 CVS 、 SVN 等版本控制工具， Git 无疑更大优秀，功能更大强大，在项目版本管理中被越来越多的广泛的使用。但 Git 相对来说比较难学。

使用 git 来管理项目有两种方式，一种是本地部署 git 版本管理系统，另一种是通过在线的代码托管。本地部署 git 版本管理系统，需要自己来搭建环境，但项目的提交与更新速度快，更适合比较封闭项目；使用在线托管最大的好处是在有网络的情况下可以随时随地的提交自己的代码，但项目是公开的，当然也可以创建私有项目，大多属于付费服务。

在代码托管服务器， github 无疑是最优秀的，其优秀稳定的服务吸引了大批开发者和开源团队贡献自己的代码和项目。但由于网站没有中文版，对于初学 git 的读者来说又增加了层学习难度，所以，在本例子笔者选用了 gitcafe 来讲解如何创建和提交自己的项目代码。 gitcafe 是国内非常优秀的代码托管服务网站，而且与 github 的使用极为相似。

第一节、Git 搭建

1. 下载及安装 Git

根据你当前使用的系统平台，请下载并安装相应的客户端软件。

MacOSX 用户下载链接：https://github.com/timcharper/git_osx_installer/downloads

Windows 用户下载链接：<http://code.google.com/p/msysgit/downloads/list>

Linux 平台安装方法如下：

```
Debian/Ubuntu $ apt-get install git-core
Fedora $ yum install git
Gentoo $ emerge --ask --verbose dev-vcs/git
Arch Linux $ pacman -S git
```

下载并安装完成后，通常在 Mac OSX 及 Linux 平台下我们用终端工具（Terminal）来使用 Git，而在 Windows 平台下用 Git Bash 工具。



图11.1

```
root@fnngj-H24X:~# apt-get install git-core
```

2. 创建 SSH 秘钥

在你的电脑与 GitCafe 服务器之间保持通信时，我们使用 **SSH key** 认证方式来保证通信安全，所以在使用 GitCafe 前你必须先创建自己的 **SSH key**。

(1) . 进入 SSH 目录

```
root@fnngj-H24X:/home/fnngj/python/pyse# cd ~/.ssh
```

```
root@fnngj-H24X:~/ssh# pwd
```

```
/root/.ssh
```

(2) . 生成新的 SSH 秘钥

如果你已经有了一个秘钥（默认秘钥位置`~/.ssh/id_rsa`文件存在）

```
root@fnngj-H24X:~/ssh# ssh-keygen -t rsa -C "fnngj@126.com"
```

```
Generating public/private rsa key pair.
```

```
Enter file in which to save the key (/root/.ssh/id_rsa): --回车
```

```
Enter passphrase (empty for no passphrase): --回车
```

```
Enter same passphrase again: --回车
```

```
Your identification has been saved in /root/.ssh/id_rsa.
```

```
Your public key has been saved in /root/.ssh/id_rsa.pub.
```

```
The key fingerprint is:
```

```
3e:d5:1c:68:af:72:ef:4d:33:36:f9:84:5d:db:6d:17 fnngj@126.com
```

```
The key's randomart image is:
```

```
+--[ RSA 2048]----+
```

```
|           |
|           |
|           o . |
|           . + . |
|           S . +   E.|
```

```
|           ...    +*|
```

```
|           + o     O.B|
```

```
|           + . + B.|
```

```
| .o . .
+-----+
root@fnngj-H24X:~/ssh# ls
id_rsa  id_rsa.pub
```

查看目录下会生成两个问题，`id_rsa` 是私钥，`id_rsa.pub` 是公钥。记住千万不要把私钥文件 `id_rsa` 透露给任何人。

3. 添加 SSH 公钥到 GitCafe

(1) . 用文本工具打开公钥文件 `~/.ssh/id_rsa.pub`，复制里面的所有内容到剪贴板。



图 11.2

(2) . 进入 GitCafe -->账户设置-->SSH 公钥管理设置项，点击添加新公钥 按钮，在 Title 文本框中输入任意字符，在 Key 文本框粘贴刚才复制的公钥字符串，按保存按钮完成操作。

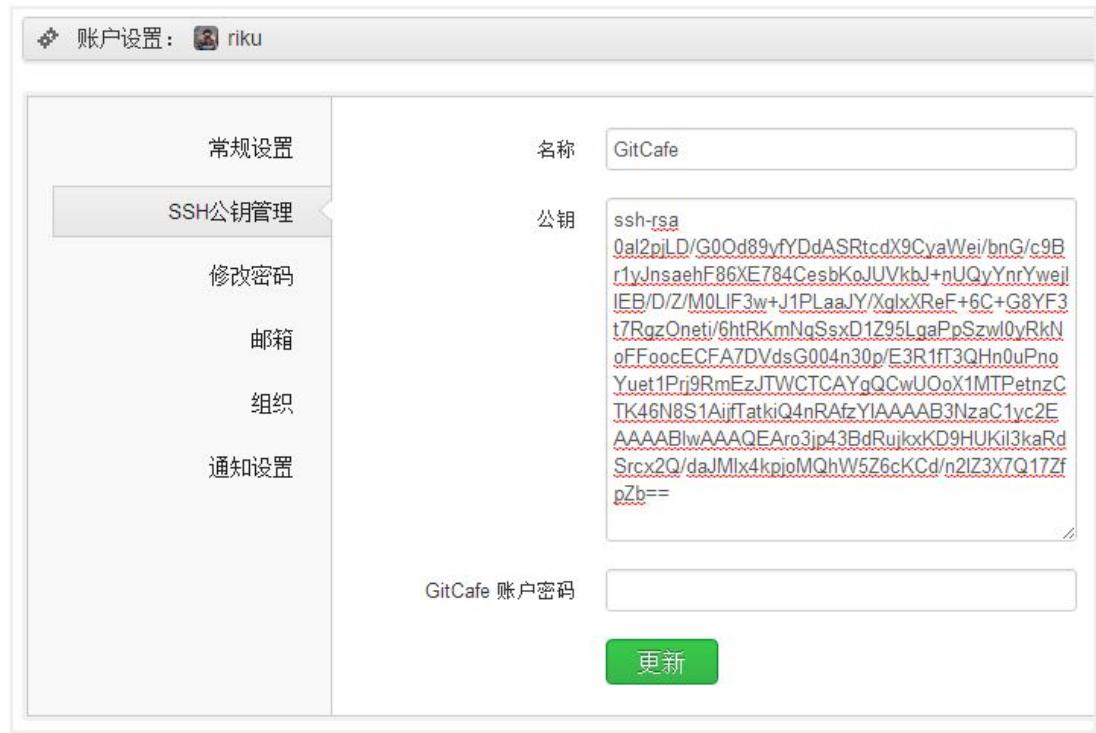


图 11.3

4. 测试连接

以上步骤完成后，你就可以通过以下命令来测试是否可以连接 **GitCafe** 服务器了。

```
root@fnngj-H24X:~/ssh# ssh -T git@gitcafe.com
The authenticity of host 'gitcafe.com (117.79.146.98)' can't be established.
RSA key fingerprint is 84:9e:c9:8e:7f:36:28:08:7e:13:bf:43:12:74:11:4e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'gitcafe.com,117.79.146.98' (RSA) to the list of known hosts.
Hi fnngj! You've successfully authenticated, but GitCafe does not provide shell access.
```

第二节、提交代码

在 **gitcafe** 创建一个项目

注册并登陆 `gitcafe` , 选择创建一个项目。

创建一个新的项目

拥有者	<input type="text" value="fnngj"/>
项目名称	<input type="text" value="pyse"/>
项目中文名(选填)	<input type="text"/>
项目描述(选填)	<input type="text" value="python selenium 实现web自动化测试"/>
项目主页(选填)	<input type="text"/>
项目图标	<div style="border: 1px dashed #ccc; padding: 5px; text-align: center;"> 选择文件 ... 尺寸: 128x128 </div>
初始化项目	<input checked="" type="checkbox"/> README文件 <input type="checkbox"/> 选择.gitignore文件
<input checked="" type="radio"/> 公开项目 <input type="radio"/> 私有项目 (当前账户余额为 0 极特币, 无法创建私有项目, 请充值)	
<input style="width: 100%;" type="button" value="创建"/>	

图11.4

项目创建完成:



图11.5

Git 的基本命令：

```
root@fnngj-H24X:/home/fnngj/python/pyse# git
usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [-c name=value] [--help]
           <command> [<args>]
```

最常用的 git 命令有：

add	添加文件内容至索引
bisect	通过二分查找定位引入 bug 的变更
branch	列出、创建或删除分支
checkout	检出一个分支或路径到工作区
clone	克隆一个版本库到一个新目录
commit	记录变更到版本库
diff	显示提交之间、提交和工作区之间等的差异
fetch	从另外一个版本库下载对象和引用
grep	输出和模式匹配的行
init	创建一个空的 git 版本库或者重新初始化一个
log	显示提交日志
merge	合并两个或更多开发历史
mv	移动或重命名一个文件、目录或符号链接
pull	获取并合并另外的版本库或一个本地分支
push	更新远程引用和相关的对象
rebase	本地提交转移至更新后的上游分支中

```

reset      重置当前 HEAD 到指定状态

rm         从工作区和索引中删除文件

show       显示各种类型的对象

status     显示工作区状态

tag        创建、列出、删除或校验一个 GPG 签名的 tag 对象

```

See 'git help <command>' for more information on a specific command.

全局设置：

设置自己的用户名和密码，和 gitcafe 保持一致：

```

root@fnngj-H24X:/home/fnngj/python/pyse# git config --global user.name 'fnngj'
root@fnngj-H24X:/home/fnngj/python/pyse# git config --global user.email 'fnngj@126.com'

```

(这一步必不可少!)

在本地创建一个项目：

```

root@fnngj-H24X:/home/fnngj/python/pyse# ls
baidu.py  baidu.py~

```

创建 pyse 目录，在目录下创建了一个简单 python 自动化脚本 baidu.py

```

root@fnngj-H24X:/home/fnngj/python/pyse# git init

```

初始化空的 Git 版本库于 /home/fnngj/python/pyse/.git/

Git init 对我们的目录进行初始化。使 pyse 目录交由 git 进行管理。

```

root@fnngj-H24X:/home/fnngj/python/pyse# git status
# 位于分支 master
#
# 初始提交
#

```

```
# Untracked files:
#   (使用 "git add <file>..." 以包含要提交的内容)
#
#       baidu.py
#       baidu.py~
nothing added to commit but untracked files present (use "git add" to track)
```

git status //查看当前项目下所有文的状态

我们看到当前处于 master (主) 分支, 罗列了当前目录下的文件 (baidu.py), 并且提示我未对当前目录下的文件进行跟踪 (跟踪什么? 跟踪文件增、删、改的情况。), 更详细人告诉我可以通过 git add <file> 来对文件进行跟踪。

```
root@fnngj-H24X:/home/fnngj/python/pyse# git add .
root@fnngj-H24X:/home/fnngj/python/pyse# git status
# 位于分支 master
#
# 初始提交
#
# 要提交的变更:
#   (使用 "git rm --cached <file>..." 撤出暂存区)
#
#       新文件:      baidu.py
#       新文件:      baidu.py~
#
```

git add . // (.) 点表示当前目录下的所有内容, 交给 git 管理, 也就是提交到了 git 的本地仓库。

当然我们我也可以通过 **git add baidu.py** 来制定具体跟踪的文件。再来通过 **git status** 命令查看, baidu.py 文件已经交给 git 跟踪了。

Commit 对提交内容做个描述:

```
root@fnngj-H24X:/home/fnngj/python/pyse# git commit -m 'first commit file'

[master (根提交) 06b5780] first commit file

2 files changed, 22 insertions(+)

create mode 100644 baidu.py

create mode 100644 baidu.py~root@fnngj-H24X:~/ssh# ssh-keygen -t rsa -C "fnngj@126.com"

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id_rsa): --回车

Enter passphrase (empty for no passphrase): --回车

Enter same passphrase again: --回车

Your identification has been saved in /root/.ssh/id_rsa.

Your public key has been saved in /root/.ssh/id_rsa.pub.

The key fingerprint is:

3e:d5:1c:68:af:72:ef:4d:33:36:f9:84:5d:db:6d:17 fnngj@126.com

The key's randomart image is:

+--[ RSA 2048]----+
|          |
|          |
|          o . |
|         . + . |
|        S . + E.|
|       ... +*|
|      + o   O.B|
|     + . + B.|
|    .o . .|
+-----+
root@fnngj-H24X:~/ssh# ls
```

```
id_rsa  id_rsa.pub
```

在每次提交项目前都需要进行 commit，对提交的内容加以描述。提示信息告诉我，我是在 master 下提交的2个新创建的文件，分别是 baidu.py 和 baidu.py~

下面将项目添加到 gitcafe 上去，还记得 gitcafe 创建的项目吧！？

```
root@fnngj-H24X:/home/fnngj/python/pyse# git remote add origin 'git@gitcafe.com:fnngj/pyse.git'
```

```
root@fnngj-H24X:/home/fnngj/python/pyse# git push -u origin master
```

Counting objects: 4, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (4/4), 436 bytes, done.

Total 4 (delta 1), reused 0 (delta 0)

To git@gitcafe.com:fnngj/pyse.git

* [new branch] master -> master

Branch master set up to track remote branch master from origin.

git remote add origin 'git@gitcafe.com:fnngj/pyse.git'

/如果你是第一次提交项目，这一句非常重要，这是你本地的当前的项目与远程的哪个仓库建立连接。

git push -u origin master

//将本地的项目提交到远程仓库中。

OK！现在已经可以在 gitcafe 上看到我们创建的项目了！

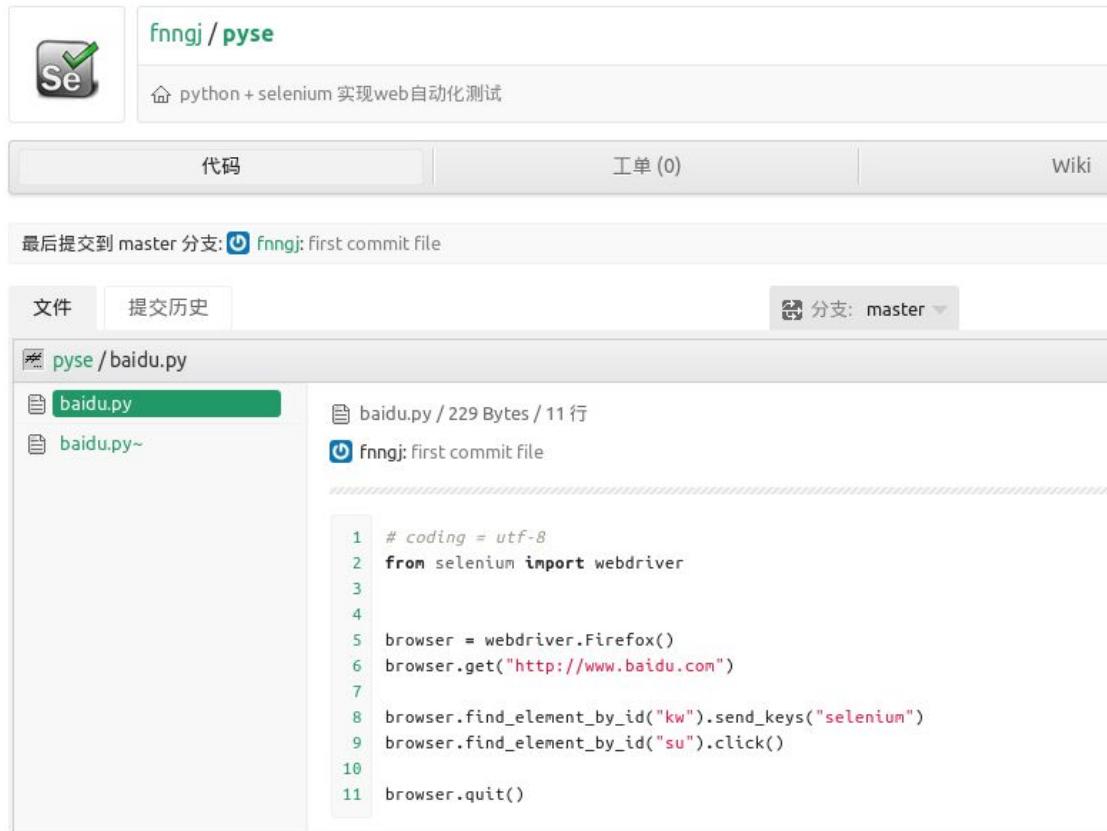


图11.6

第三节、更新代码

克隆代码

上面的代码编写与提交是我在公司完成的，假如我回到了家里想继续编写自己的程序，那么需要将代码克隆下来。

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2
$ git clone git://gitcafe.com/fnngj/pyse.git
Cloning into 'pyse'...
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 7 (delta 1)
```

```
Receiving objects: 100% (7/7), 262.60 KiB | 0 bytes/s, done.
```

```
Resolving deltas: 100% (1/1), done.
```

```
Checking connectivity... done.
```

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2
```

```
$ cd pyse
```

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ ls
```

```
baidu.py  baidu.py~
```

<git://gitcafe.com/fnngj/pyse.git> 就是我们项目在 gitcafe 上的路径。通过 `git clone` 命令就可以把项目克隆到本地。

更新提交

我们克隆文件到本地的目的是进一步地对代码进行修改并提交。

```

MINGW32:/d/selenium_use_case/python_selenium2/pyse
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Gitcafehelpdoc.wps

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    all_tests.py
    all_tests_mail.py
    all_tests_new.py
    data/
    report/
    test_case/
no changes added to commit (use "git add" and/or "git commit -a")
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

```

图11.7

通过 `git status` 你会发现，我在 `pyse` 目录下做了比较大的改动，首先修改了 `Gitcafehelpdoc.wps` 文件，然后新增加了一些文件和文件夹。

去克隆一个项目很简单，提交一个项目比较麻烦。

因为我已经更新了环境(电脑)，所以我需要再次在本机生成 `ssh` 公钥，并把生成的公钥添加到 `gitcafe` 中。具体步骤，参考前面的内容。

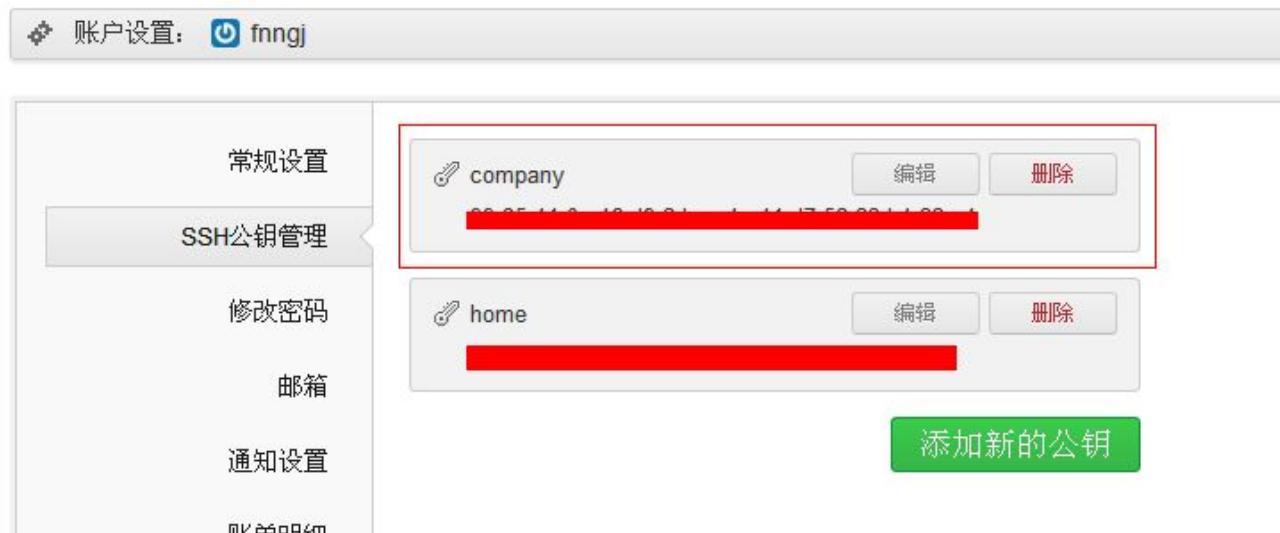


图11.8

测试链接，设置全局变量：

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ ssh -T git@gitcafe.com
```

```
Hi fnngj! You've successfully authenticated, but GitCafe does not provide shell
```

```
access.
```

```
root@fnngj-H24X:/home/fnngj/python/pyse# git config --global user.name 'fnngj'
```

```
root@fnngj-H24X:/home/fnngj/python/pyse# git config --global user.email 'fnngj@126.com'
```

提交代码到 gitcafe

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ git add .    ---添加当前目录下的所文件及文件夹
```

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ git commit -m "add test file and data and report"  ---对提交的内容进行说明
```

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ git push origin master  --提交代码到远程服务器 (gitcafe)
```

```
fatal: remote error: access denied or repository not exported: /fnngj/pyse.git
```

```
Administrator@XP-201210141900 /d/sele
```

```
$ git remote -v
```

```
origin  git://gitcafe.com/fnngj/pyse.git (fetch)
```

```
origin  git://gitcafe.com/fnngj/pyse.git (push)
```

```
fatal: remote error: access denied or repository not exported: /fnngj/pyse.git
```

我们在 git push 的时候出问题了，告诉我不能与远程服务器链接。但是通过 git remote -v 查看你当前项目远程连接的是的仓库地址是 OK 的呀！on，是这个地址有问题，这是一个 git 地址，我们只能用于克隆文件到本地，无法通有这个地址进行 push 项目。

```
git@gitcafe.com:fnngj/pyse.git
```

上面的地址才是一个可以 push 项目的地址。

那么我们就需要把 origin 删除掉，重新以上面的地址建立链接。

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git remote rm origin    --删除 origin

Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git remote add origin 'git@gitcafe.com:fnngj/pyse.git'    --重新链接

Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git remote -v    -- 现在的链接地址是可以 push

origin  git@gitcafe.com:fnngj/pyse.git (fetch)

origin  git@gitcafe.com:fnngj/pyse.git (push)

Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git push -u origin master    --重新 push 项目

Counting objects: 24, done.

Delta compression using up to 8 threads.

Compressing objects: 100% (18/18), done.

Writing objects: 100% (22/22), 45.95 KiB | 0 bytes/s, done.

Total 22 (delta 2), reused 0 (delta 0)

To git@gitcafe.com:fnngj/pyse.git

    427652a..efed4f4  master -> master

Branch master set up to track remote branch master from origin.
```

删除提交

在我 git 管理的目录下，有些文件或目录已经废弃掉了，我需要手动删除这些文件或目录，通过 `git rm` 命令来完成。

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)
```

```
$ git rm baidu.py~ ---删除文件
rm 'baidu.py~'

$ git rm abc/      ---删除目录
rm 'abc'

Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.
```

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: baidu.py~ ---git 提示删除了 baidu.py~文件

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git commit -m "delete baidu.py~"

Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git push origin master
```

pull 最新代码到本地

形成习惯，尽量为了避免冲突，我们在 **push** 代码之后先把服务器端最新的代码 **pull** 到本地。

```
Administrator@XP-201210141900 /d/selenium_use_case/python_selenium2/pyse (master)

$ git pull origin master

From gitcafe.com:fnngj/pyse
 * branch            master       -> FETCH_HEAD
```

Already up-to-date.

小结:

通过本节有学习，我们可以自由的随时随地提交自己的代码到服务器上，在多人开发项目中，我们会涉及到更多的技术，例如 **git** 的分支等。代码版本控制不是本书重点，笔者在这里更多的是起一抛砖引玉的作用，目前 **git** 的相关文档已经相当丰富了，读者可以进一步的学习。

主流的在线托管网站：

<http://www.github.com/>

<http://git.oschina.net/>

<http://gitcd.com/>

附录

UliPad--python 开发利器环境搭建

工欲善其事,必先利其器

有时候往往选择太多,变得无从选择。

如果你在 python 开发中已经找到了趁手的 IDE 这一节可以无视。

其实,python 下面能找到一款不错的开发工具是不太容易的。IDLE 写写单个小程序很好,但一个程序文件与执行信息是两个窗口,程序开多了就分不清哪个了。

notepad++ 是一个小巧的万能编辑器。

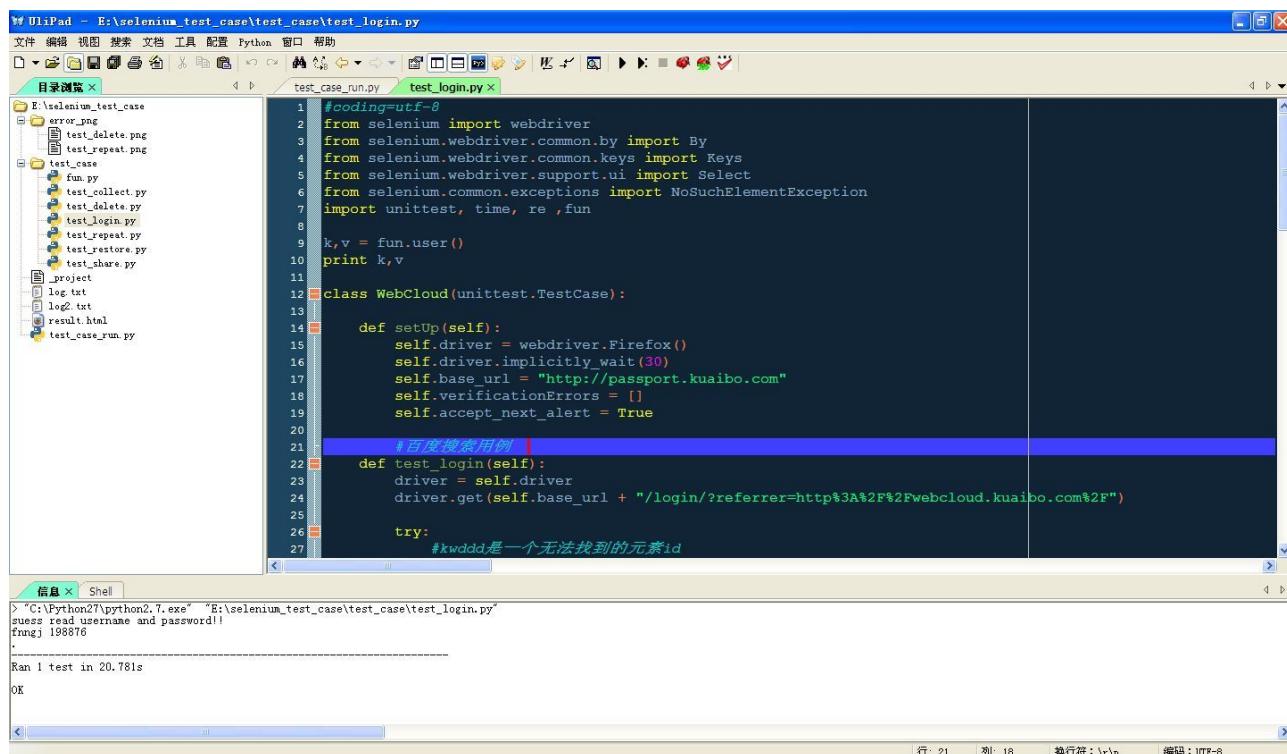
linux 会有一些非常不错的交互式 python IDE ,如 ipython 、 bpython 等。

vim 肯定是开发神器,但一般也只有高手才会运用自如,体会它的奥妙。

UliPad 是找到的写 python 最舒服的一个 IDE 。

地址: <https://code.google.com/p/ulipad/>

- 免费,可以免费获得并使用它的所有功能。
- 支持 windows 、 MAC 、 linux 等平台。
- 小巧,内存占用很少,10MB 左右。



The screenshot shows the UliPad IDE interface. The title bar reads "UliPad - E:\selenium_test_case\test_case\test_login.py". The main window has two tabs: "test_case_run.py" and "test_login.py". The "test_login.py" tab is active, displaying the following Python code:

```

#coding=utf-8
from selenium import webdriver
from selenium.webdriver.common import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import Select
from selenium.common.exceptions import NoSuchElementException
import unittest, time, re,fun
k,v = fun.user()
print k,v
class WebCloud(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(30)
        self.base_url = "http://passport.kuaibo.com"
        selfverificationErrors = []
        self.accept_next_alert = True
    #百度搜索用例
    def test_login(self):
        driver = self.driver
        driver.get(self.base_url + "/login/?referrer=http%3A%2F%2Fwebcloud.kuaibo.com%2F")
        try:
            #kwdd是一个无法找到的元素id

```

The status bar at the bottom shows the command prompt output:

```

> "C:\Python27\python2.7.exe" "E:\selenium_test_case\test_case\test_login.py"
guess read username and password!
running 198876
-----
Ran 1 test in 20.781s
OK

```

图 1

具体的安装使用,这里就不介绍了,不是本文档的主题。有兴趣使用可以参考我的博客:

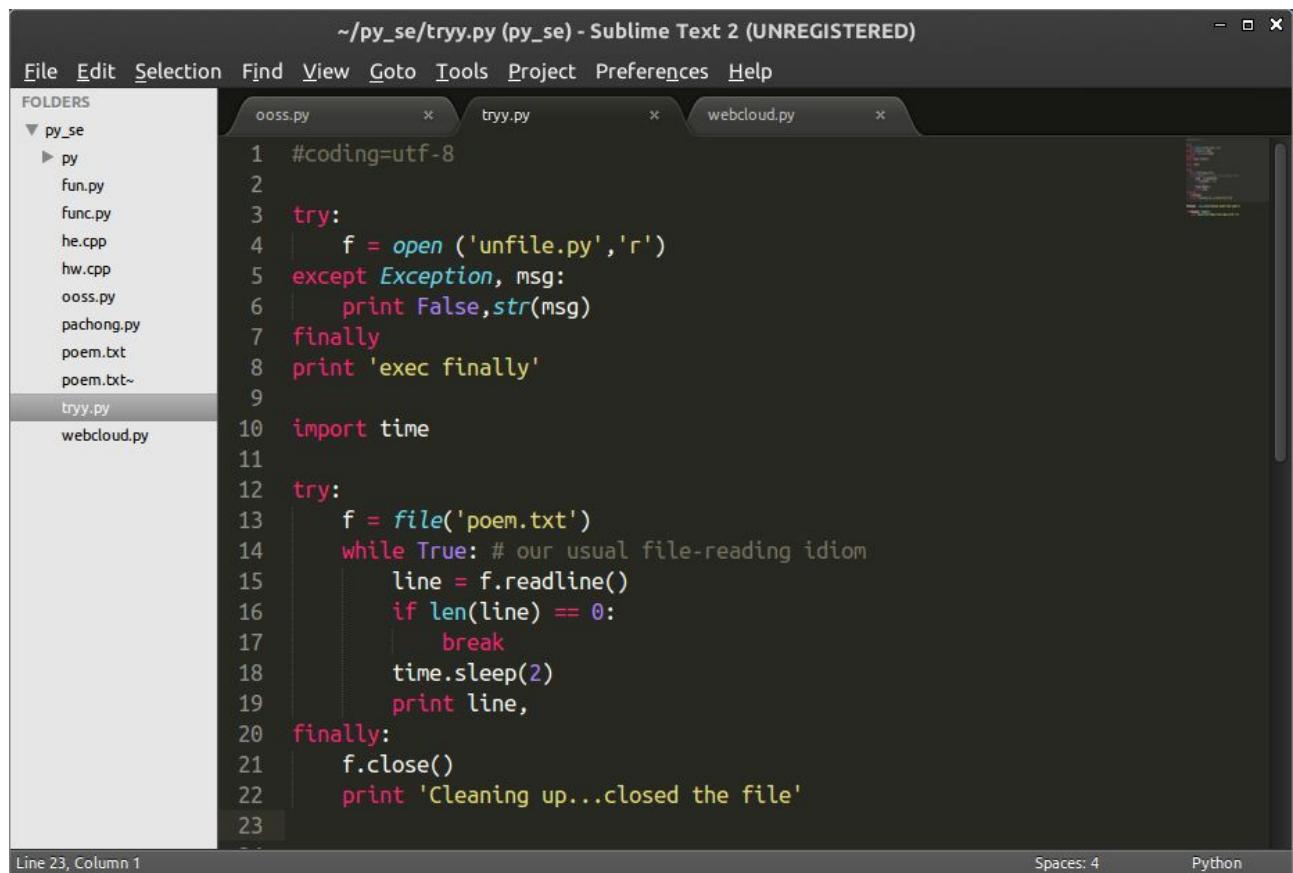
<http://www.cnblogs.com/fnng/p/3393275.html>

Sublime--强大好用的代码编辑器

Sublime Text 是我发现的有一好用的编辑器, 它不单单只支持 python , 几乎支持目前主流的语言, 快捷键丰富, 可以极大的提高代码开发效率。

Sublime Text 网址:

<http://www.sublimetext.com/>



```

~/py_se/tryy.py (py_se) - Sublime Text 2 (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
py_se
  ▶ py
    fun.py
    func.py
    he.cpp
    hw.cpp
    ooss.py
    pachong.py
    poem.txt
    poem.txt~
  tryy.py
  webcloud.py
tryy.py
1 #coding=utf-8
2
3 try:
4     f = open ('unfile.py','r')
5 except Exception, msg:
6     print False,str(msg)
7 finally
8     print 'exec finally'
9
10 import time
11
12 try:
13     f = file('poem.txt')
14     while True: # our usual file-reading idiom
15         line = f.readline()
16         if len(line) == 0:
17             break
18         time.sleep(2)
19         print line,
20     finally:
21         f.close()
22         print 'Cleaning up...closed the file'
23
Line 23, Column 1
Spaces: 4
Python

```

图 2

下面介绍 sublime 的一些使用技巧，相信你掌握这些技巧之后一定会对它爱不释手。

sublime 使用技巧

两个小技巧：选择文字之后，按下 Tab 和 Shift + Tab 可以控制缩进。文件未保存就可以直接退出程序，下次启动会自动恢复。



图 3

一) 在当前项目中，快速搜索文件

1. 搜索文件

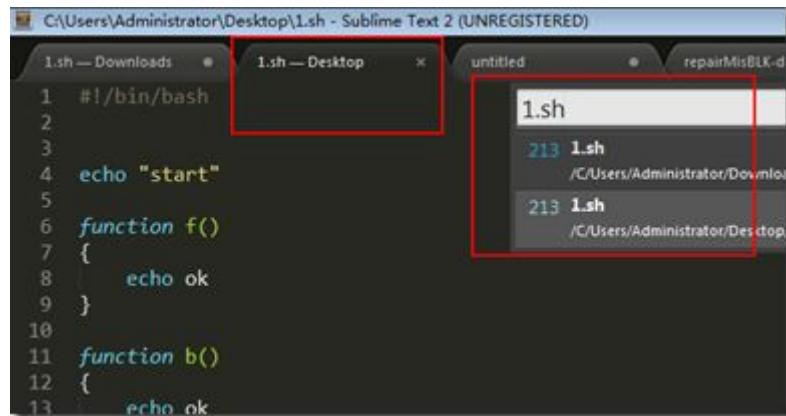


图 4

2. 搜索文件小技巧，在输入文件路径的时候，可以/c/u/a/这样的格式匹配来快速找到文件

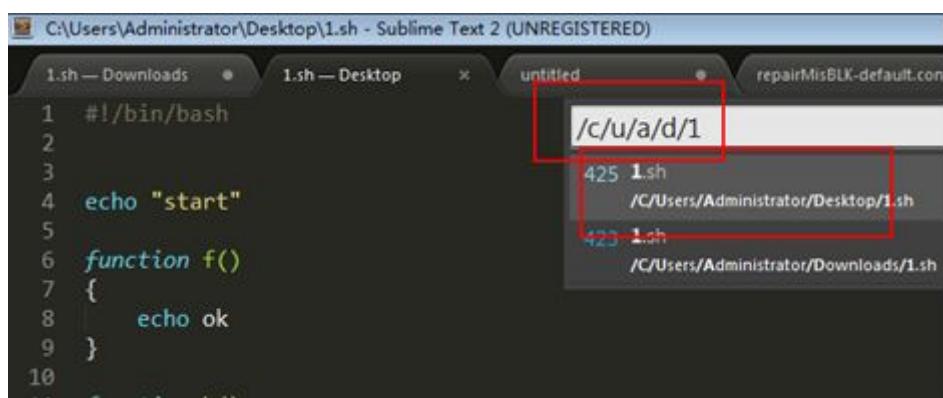


图 5

3. 搜索到了2个结果，可以按上下键来在多个结果中跳转

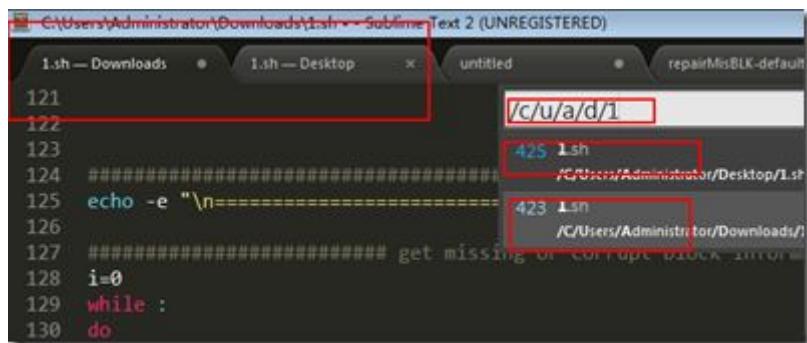


图 6

二) 添加注释

1. 添加块注释，类似于/* */用这种方法来添加的注释一样。

先选择要注释的内容，然后按 `ctrl + /`



图 7

2. 添加行注释，把鼠标移到改行的任意位置，按 `ctrl + /`即可

```

1  #!/bin/bash
2
3
4 echo "start"
5
6 function f()
7 {
8     echo ok
9 }
10
11 # function b()
12 {
13     echo ok

```

图 8

3. 取取消单行注释，鼠标位于已经注释的行的任意位置，执行 $\text{ctrl} + /$ 即可

```

# function b()
{
    echo ok
}

```

```

function b()
{
    echo ok
}

```

图 9

4. 取消块注释

选择要取消的内容，按 $\text{ctrl} + /$ 即可

```

# function b()
# {
#     echo ok
# }

```

图 10

```

function b()
{
    echo ok
}

```

图 11

5. 即取消注释和添加注释是逆操作

三) 快速跳转到指定的行

ctrl + g, 然后输入行号, enter 就行。比如跳转到第五行。 或者 ctrl + p, 再输入 :

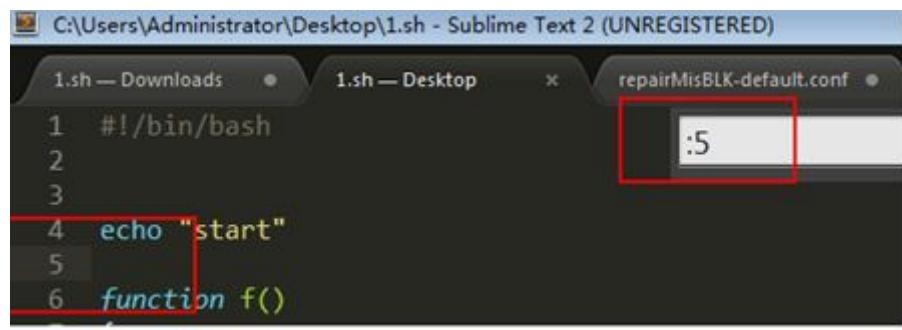


图 12

四) 搜索函数

按 ctrl + r 或 ctrl + p ,在执行@。之后填写要搜索的函数名

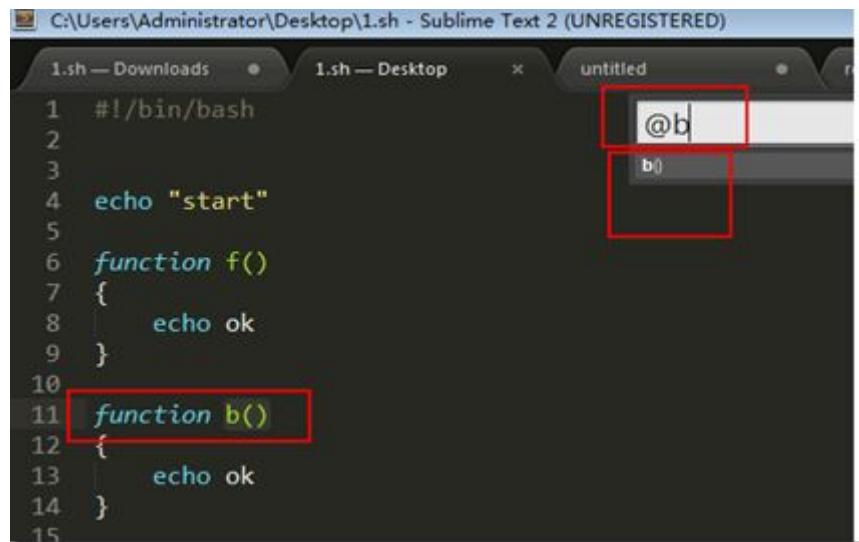


图 13

五) 隐藏菜单和显示菜单栏

1. 隐藏菜单栏: view --> Hide Menu

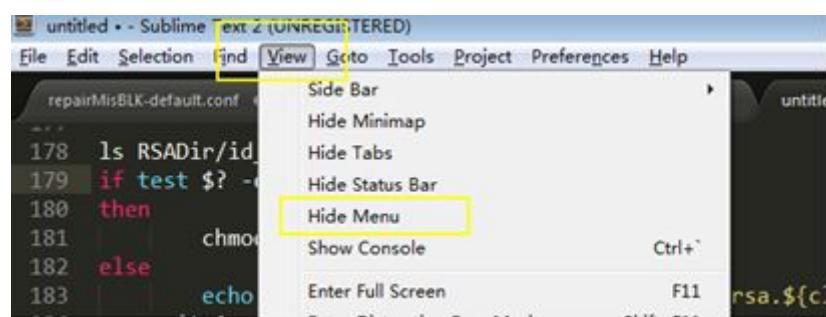


图 14

2. 隐藏菜单栏后，要显示菜单栏：

i. 这是隐藏之后

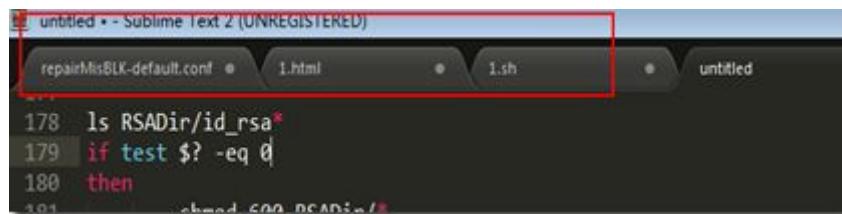


图 15

3. 隐藏之后显示菜单栏按住 Alt 键，菜单栏即会出现。松开后，则菜单栏就会消失。要永久显示则是：按住 Alt 键-->view--> show Menu



图 16

主要快捷键列表

Ctrl+L 选择整行（按住-继续选择下行）

Ctrl+KK 从光标处删除至行尾

Ctrl+Shift+K 删除整行

Ctrl+Shift+D 复制光标所在整行，插入在该行之前

Ctrl+J 合并行（已选择需要合并的多行时）

Ctrl+KU 改为大写

Ctrl+KL 改为小写

Ctrl+D 选词（按住-继续选择下个相同的字符串）

Ctrl+M 光标移动至括号内开始或结束的位置

Ctrl+Shift+M 选择括号内的内容（按住-继续选择父括号）

Ctrl+/ 注释整行（如已选择内容，同“Ctrl+Shift+ /”效果）

Ctrl+Shift+ / 注释已选择内容

Ctrl+Z 撤销

Ctrl+Y 恢复撤销

Ctrl+M 光标跳至对应的括号

Alt+. 闭合当前标签

Ctrl+Shift+A 选择光标位置父标签对儿

Ctrl+Shift+[折叠代码

Ctrl+Shift+] 展开代码
Ctrl+KT 折叠属性
Ctrl+K0 展开所有
Ctrl+U 软撤销
Ctrl+T 词互换
Tab 缩进 自动完成
Shift+Tab 去除缩进
Ctrl+Shift+↑ 与上行互换
Ctrl+Shift+↓ 与下行互换
Ctrl+K Backspace 从光标处删除至行首
Ctrl+Enter 光标后插入行
Ctrl+Shift+Enter 光标前插入行
Ctrl+F2 设置书签
F2 下一个书签
Shift+F2 上一个书签

参考

本文档参考文献书籍:

selenium python API 官方:

<http://selenium.googlecode.com/git/docs/api/py/api.html>

python 单元测试框架:

<http://www.ibm.com/developerworks/cn/linux/l-pyunit/index.html>

乙醇 webdriver 实用指南 (python 版)

https://github.com/easonhan007/webdriver_guide

lettuce 官方例子:

<http://lettuce.it/tutorial/simple.html#tutorial-simple>

参考书籍:

《零成本实现 Web 自动化测试 基于 Selenium 和 Bromine》

《python 核心编程 (第二版)》 18 章: 多线程编程